Эта книга дает исчерпывающее понимание основных концепций нейроэволюции и формирует навыки использования подобных алгоритмов для решения практических задач.

Прочитав книгу, вы:

- откроете для себя самые популярные алгоритмы нейроэволюции – NEAT, HyperNEAT и ES-HyperNEAT;
- узнаете, как реализовать алгоритмы нейроэволюции на языке Python;
- освоите продвинутые инструменты визуализации для исследования сложных графов топологии нейронных сетей;
- научитесь проверять результаты экспериментов и анализировать производительность алгоритмов;
- узнаете, как улучшить производительность существующих методов при помощи нейроэволюции;
- научитесь применять глубокую нейроэволюцию для разработки автономного агента, играющего в классические игры Atari.

Для специалистов по анализу данных и машинному обучению, стремящихся реализовывать алгоритмы нейроэволюции с нуля.

Требуются базовые знания в области глубокого обучения и нейронных сетей, а также навыки программирования на языке Python.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@alians-kniga.ru

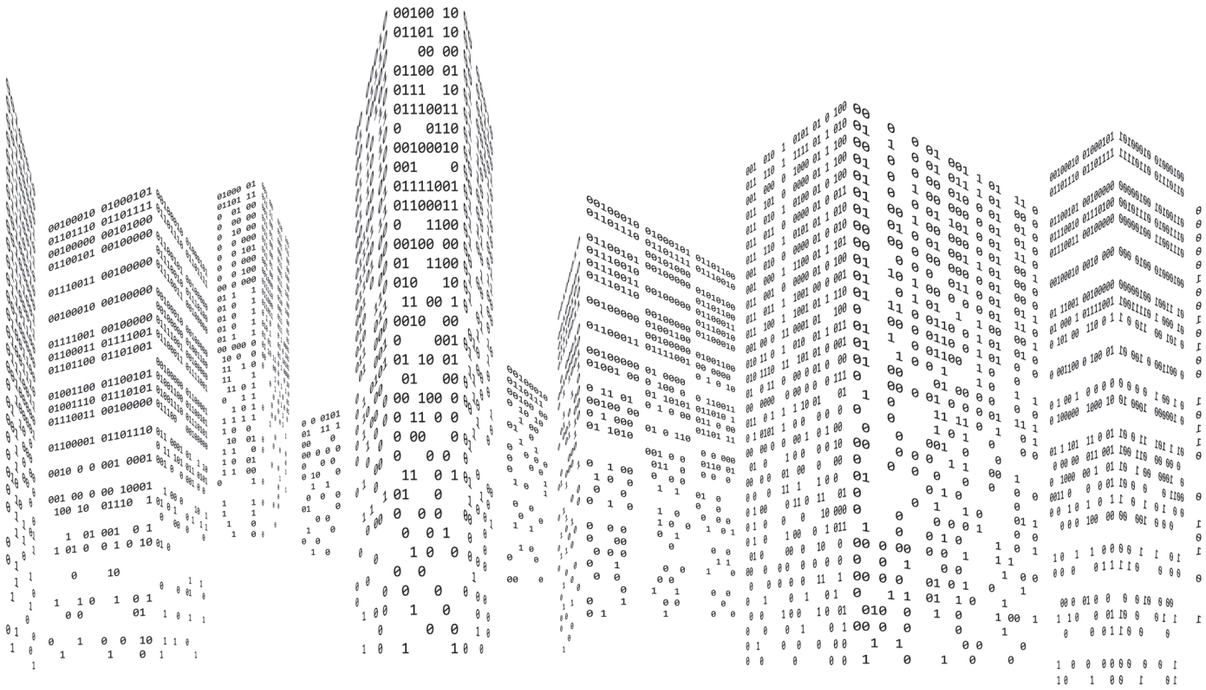
Packt>

ДМК ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-854-8

Эволюционные нейросети на языке Python

Ярослав Омельяненко



Эволюционные нейросети на языке Python

ДМК ИЗДАТЕЛЬСТВО

Ярослав Омельяненко

Эволюционные нейросети на языке Python

Hands-On Neuroevolution with Python

Iaroslav Omelianenko

Эволюционные нейросети на языке Python

Ярослав Омельяненко

Москва, 2020

УДК 004.421
ББК 32.811
О57

Ярослав Омельяненко
О57 Эволюционные нейросети на языке Python / пер. с анг. В. С. Яценкова. – М.: ДМК Пресс, 2020. – 310 с.: ил.

ISBN 978-5-97060-854-8

Эта книга дает всестороннее представление о нейроэволюции – подходе к обучению искусственных нейронных сетей, который использует эволюционные алгоритмы, чтобы упростить процесс решения сложных задач в таких областях, как игры, робототехника и моделирование естественных процессов.

Читатель начнет знакомство с ключевыми концепциями и методами нейроэволюции, написав несложный код на языке Python, а затем получит практический опыт работы с популярными библиотеками Python и научится решать распространенные и нестандартные прикладные задачи, используя алгоритмы на основе нейроэволюции. Речь пойдет о том, как адаптировать методы нейроэволюции к существующим проектам нейронных сетей для повышения эффективности обучения и принятия решений; в завершение будет рассказано о топологиях нейронных сетей и о том, как нейроэволюция позволяет развивать сложную топологию из простейшей базовой структуры.

Издание предназначено для специалистов в области машинного обучения и искусственного интеллекта, которые стремятся реализовать алгоритмы нейроэволюции с нуля. Наличие базовых знаний в области глубокого обучения и нейронных сетей, а также программирования на языке Python обязательно.

УДК 004.421
ББК 32.811

ISBN (анг.) 9781838824914
ISBN (рус.) 978-5-97060-854-8

Моей жене Люси с благодарностью за то,
что она была моим лучшим другом и любящей супругой
на протяжении всей нашей совместной жизни

Оглавление

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Об авторе

Ярослав Омельяненко более десяти лет работает техническим директором и директором по исследованиям. Он является активным членом исследовательского сообщества и опубликовал несколько исследовательских работ на arXiv, ResearchGate, Preprints и др. Он начал заниматься прикладным машинным обучением более десяти лет назад, разрабатывая автономные агенты для мобильных игр. Последние 5 лет активно участвовал в исследованиях, связанных с применением методов глубокого машинного обучения для аутентификации, распознавания личных качеств, групповой робототехники, искусственного интеллекта и многого другого. Он активный разработчик программного обеспечения и создает реализации нейроэволюционного алгоритма с открытым исходным кодом на языке Go.

Я хочу поблагодарить всех исследователей и разработчиков за то, что они поделились результатами своей работы и вдохновляются идеалами открытого кода. Без сообщества open source наш мир был бы другим

О рецензентах

Алан Макинтайр (Alan McIntyre) – главный архитектор программного обеспечения в CodeReclaimers, LLC, где он оказывает услуги по индивидуальному проектированию и разработке программного обеспечения для технических вычислительных приложений, включая вычислительную геометрию, компьютерное зрение и машинное обучение. Ранее он работал инженером-программистом в General Electric, Microsoft и нескольких стартапах.

Унсал Гокдаг (Unsal Gokdag) с 2017 года работает старшим научным сотрудником в области логистики, а в 2013 году работал инженером по исследованиям и разработкам. В настоящее время он готовит докторскую диссертацию по сравнению алгоритмов машинного обучения для удаления пятен на изображениях и классификации поляриметрических изображений SAR. Его прошлый опыт включает работу в области машинного обучения, компьютерного зрения и биоинформатики. Он впервые применил алгоритм NEAT в своей дипломной работе бакалавра и с тех пор интересуется эволюционными алгоритмами. В настоящее время проживает в Германии.

Я хочу поблагодарить мою семью за бескорыстную любовь, которую они мне подарили. Без этой поддержки я бы ничего не добился. Моей маме и сестре: спасибо за все, что вы сделали для меня в самые трудные времена. Папа, я скучаю по тебе

Предисловие от автора

Когда традиционные методы глубокого обучения приблизились к пределу своих возможностей, все больше и больше исследователей начали искать альтернативные подходы к обучению *искусственных нейронных сетей* (artificial neural network, ANN).

Глубокое машинное обучение чрезвычайно эффективно для распознавания образов, но не позволяет выполнять задачи, требующие понимания контекста или незнакомых данных. Многие исследователи, включая Джеффа Хинтона, отца современной концепции глубокого машинного обучения, согласны с тем, что существующий подход к проектированию систем искусственного интеллекта больше не в состоянии справиться с насущными проблемами.

В этой книге мы обсуждаем жизнеспособную альтернативу традиционным методам глубокого машинного обучения – нейроэволюционные алгоритмы. *Нейроэволюция* – это семейство методов машинного обучения, которые используют эволюционные алгоритмы для облегчения решения сложных задач, таких как игры, робототехника и моделирование естественных процессов. Нейроэволюционные алгоритмы имитируют процесс естественного отбора. Очень простые искусственные нейронные сети могут стать очень сложными. Конечным результатом нейроэволюции является оптимальная топология сети, которая делает модель более энергоэффективной и удобной для анализа.

В этой книге вы узнаете о различных нейроэволюционных алгоритмах и получите практические навыки по их использованию для решения различных типовых задач информатики – от классического обучения с подкреплением до создания агентов для автономной навигации по лабиринту. Кроме того, вы узнаете, как можно применить нейроэволюцию для обучения глубоких нейронных сетей при создании агента, способного играть в классические игры Atari.

Цель этой книги – дать вам ясное представление о методах нейроэволюции, проводя различные эксперименты с пошаговым руководством. Она содержит практические примеры в таких областях, как игры, робототехника и моделирование естественных процессов, с использованием реальных наборов данных, чтобы помочь вам лучше понять исследуемые методы. После прочтения этой книги у вас будет все необходимое для применения методов нейроэволюции к другим задачам, по аналогии с представленными экспериментами.

При написании этой книги я стремился дать вам знания о передовых технологиях, которые являются жизненно важной альтернативой традиционному глубокому обучению. Я надеюсь, что применение нейроэволюционных алгоритмов в ваших проектах позволит вам элегантно и эффективно решать проблемы, которые казались неразрешимыми.

Для кого написана эта книга

Эта книга предназначена для практиков машинного обучения, исследователей глубокого обучения и энтузиастов искусственного интеллекта, которые стремятся реализовать нейроэволюционные алгоритмы с нуля. Вы узнаете, как применять эти алгоритмы к различным задачам реального мира, как методы нейроэволюции могут оптимизировать процесс обучения искусственных нейронных сетей. Вы познакомитесь с основными понятиями нейроэволюции и получите необходимые практические навыки, чтобы использовать ее в своей работе и экспериментах. Знание Python, а также основ глубокого обучения и нейронных сетей является обязательным.

О чем эта книга

Глава 1 знакомит с основными понятиями генетических алгоритмов, такими как генетические операторы и схемы кодирования генома.

В *главе 2* обсуждаются практические аспекты методов нейроэволюции. В этой главе рассказано о достоинствах и недостатках популярных библиотек Python, которые предоставляют реализации алгоритма NEAT и его расширений.

Глава 3 – это место, где вы начинаете экспериментировать с алгоритмом NEAT, реализуя решение для классической проблемы информатики.

В *главе 4* продолжаются эксперименты, связанные с классическими проблемами информатики в области обучения с подкреплением.

В *главе 5* вы продолжаете эксперименты с нейроэволюцией, пытаясь создать решатель, способный найти выход из лабиринта. Вы узнаете, как реализовать симуляцию робота с набором датчиков для обнаружения препятствий и контроля его положения в лабиринте.

В *главе 6* вы воспользуетесь практическим опытом, полученным при создании решателя лабиринта в предыдущей главе, чтобы встать на путь создания более совершенного решателя.

Глава 7 знакомит вас с передовыми методами нейроэволюции. Вы узнаете о схеме косвенного кодирования генома, в которой используются *сети, производящие составные паттерны* (Compositional Pattern Producing Network, CPPN) для кодирования топологий крупномасштабных искусственных нейросетей.

В *главе 8* вы узнаете, как выбрать конфигурацию модульной сетчатки, которая лучше всего подходит для конкретной проблемной области.

В *главе 9* мы обсуждаем, как широко распространенная в природе стратегия коэволюции может быть перенесена в область нейроэволюции.

Глава 10 представляет вам концепцию глубокой нейроэволюции, которую можно использовать для обучения *глубоких искусственных нейронных сетей* (Deep Artificial Neural Network, DNN).

В *главе 11* рассказано, как начать работу над прикладной задачей, как настроить гиперпараметры нейроэволюционного алгоритма, как использовать передовые инструменты визуализации и какие показатели можно использовать для анализа выполнения алгоритма.

Глава 12 обобщает все, что вы узнали в этой книге, и дает рекомендации о том, как продолжить свое самообразование.

Как получить максимальную отдачу от этой книги

Для работы с примерами, представленными в этой книге, необходимы практические навыки программирования на языке Python. Для лучшего понимания исходного кода предпочтительно использовать IDE, которая поддерживает подсветку синтаксиса Python и ссылки внутри кода. Если у вас нет специализированной среды разработки, вы можете использовать инструмент разработки Microsoft Visual Studio Code. Он бесплатный и кросс-платформенный, и вы можете свободно скачать его по адресу https://code.visualstudio.com.

Python и большинство библиотек, которые мы обсуждаем в этой книге, являются кросс-платформенными и совместимы с Windows, Linux и macOS. Все описанные в книге эксперименты выполняются из командной строки, поэтому ознакомьтесь с приложением консоли терминала, установленным в выбранной вами ОС.

Для выполнения эксперимента, описанного в главе 10, вам необходим доступ к современному ПК с графическим ускорителем Nvidia GeForce GTX 1080Ti или выше. Этот эксперимент также лучше проводить в среде Ubuntu Linux. Ubuntu – это современная и мощная ОС на базе Linux. Навык работы с ней вам очень пригодится.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** на странице с описанием соответствующей книги.

Скачивание цветных иллюстраций

Мы предоставляем файл PDF с цветными изображениями скриншотов и рисунков, используемых в этой книге. Вы можете скачать его по адресу: **https://static.packt-cdn.com/downloads/ 9781838824914_ColorImages.pdf**

Условные обозначения и соглашения, принятые в книге

В книге используются следующие типографские соглашения.

Курсив используется для смыслового выделения новых терминов, адресов электронной почты, а также имен и расширений файлов.

`Моноширинный шрифт` применяется для листингов программ, а также в обычном тексте для обозначения имен переменных, функций, типов, объектов, баз данных, переменных среды, операторов и ключевых слов.

`Моноширинный полужирный шрифт` используется для обозначения команд или фрагментов текста, которые пользователь должен ввести дословно без изменений.

`Моноширинный курсив` – для обозначения в исходном коде или в командах шаблонных меток-заполнителей, которые должны быть заменены соответствующими контексту реальными значениями.

Часть I

Основы эволюционных вычислительных алгоритмов и методов нейроэволюции

В первой части книги представлены основные понятия эволюционных вычислений и обсуждаются особенности алгоритмов на основе нейроэволюции и перечень библиотек Python, которые могут использоваться для их реализации. Вы познакомитесь с основами методов нейроэволюции и получите практические рекомендации о том, как начать свои эксперименты. Здесь также приведено краткое знакомство с менеджером пакетов Anaconda для Python в рамках настройки вашей среды разработки.

Эта часть книги состоит из следующих глав:

- главы 1 «Обзор методов нейроэволюции»;
- главы 2 «Библиотеки Python и настройка среды разработки».

Глава 1

Обзор методов нейроэволюции

Концепция *искусственной нейронной сети* (artificial neural networks, ANN), которую мы дальше для удобства будем называть просто *нейросетью*, основана на структуре человеческого мозга. Существует устойчивое убеждение, что если суметь достоверно скопировать эту сложнейшую структуру, то можно создать *искусственный интеллект* (artificial intellect, AI). Мы все еще на пути к достижению этой цели. Хотя нам по силам создание специализированных AI-агентов, мы еще далеки от создания универсального искусственного интеллекта.

В этой главе вы познакомитесь с понятием искусственных нейросетей и двумя методами, которые мы можем использовать для их обучения (градиентный спуск с обратным распространением ошибки и нейроэволюция), чтобы нейросеть научилась приближаться к целевой функции. Тем не менее мы сосредоточимся в основном на обсуждении семейства алгоритмов на основе нейроэволюции. Вы узнаете о реализации эволюционного процесса, основанного на естественной эволюции, и познакомитесь с наиболее популярными алгоритмами нейроэволюции: NEAT, HyperNEAT и ES-HyperNEAT. Мы также обсудим методы оптимизации, которые можно использовать для поиска окончательных решений, и проведем сравнение между алгоритмами приближения к цели и алгоритмом поиска новизны. К концу этой главы вы будете иметь полное представление о внутреннем устройстве алгоритмов нейроэволюции и будете готовы применить эти знания на практике.

В этой главе мы рассмотрим следующие темы:

- эволюционные алгоритмы и нейроэволюционные методы;
- обзор алгоритма NEAT;
- NEAT на основе гиперкуба;
- HyperNEAT с развивающимся субстратом;
- метод оптимизации на основе поиска новизны.

1.1 Эволюционные алгоритмы и нейроэволюционные методы

Термин «искусственная нейронная сеть» обозначает граф *узлов*, соединенных *связями*, где каждая из связей имеет определенный *вес*. Узел нейросети является своего рода пороговым оператором, который позволяет сигналу проходить дальше только после срабатывания определенной функции активации. Это отдаленно напоминает принцип, по которому организованы нейроны головного мозга. Как правило, процесс обучения нейросети состоит из выбора подходящих значений веса для всех связей в сети. Таким образом, нейросеть способна аппроксимировать любую функцию и может рассматриваться как *универсальный аппроксиматор*, который определяется теоремой универсальной аппроксимации.

Чтобы познакомиться с доказательством теоремы универсальной аппроксимации, прочтите следующие статьи:

- *Cybenko G.* (1989) Approximations by Superpositions of Sigmoidal Functions, Mathematics of Control, Signals, and Systems, 2 (4), 303–314;
- *Leshno Moshe, Lin Vladimir Ya., Pinkus Allan, Schocken Shimon* (January 1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Networks. 6 (6): 861–867. doi:10.1016/S0893-6080(05)80131-5 (**https://www.sciencedirect.com/science/ article/abs/pii/S0893608005801315?via%3Dihub**);
- *Kurt Hornik* (1991). Approximation Capabilities of Multilayer Feedforward Networks, Neural Networks, 4 (2), 251–257. doi:10.1016/0893-6080(91)90009-T (**https://www. sciencedirect.com/science/article/abs/pii/089360809190009T?via%3Dihub**);
- *Hanin B.* (2018). Approximating Continuous Functions by ReLU Nets of Minimal Width. arXiv preprint arXiv:1710.11278 **(https://arxiv.org/abs/1710.11278**).

За последние 70 лет было придумано много методов обучения нейросетей. Однако наиболее популярная техника, получившая известность в последнем десятилетии, была предложена Джеффри Хинтоном. Она основана на обратном распространении ошибки прогнозирования через сеть с различными методами оптимизации, построенными на основе градиентного спуска функции потерь по отношению к весам связей между узлами сети. Этот метод демонстрирует выдающуюся эффективность обучения глубоких нейронных сетей для задач, связанных в основном с распознаванием образов. Однако, несмотря на присущие ему достоинства, он имеет существенные недостатки. Один из этих недостатков заключается в том, что для усвоения чего-то полезного из определенного набора данных требуется огромное количество обучающих образцов. Другим существенным недостатком является фиксированная архитектура нейросети, созданная экспериментатором вручную, что приводит к неэффективному использованию вычислительных ресурсов. Это связано с тем, что значительное количество сетевых узлов не участвует

в процессе вывода. Кроме того, методы обратного распространения имеют проблемы с передачей полученных знаний в другие смежные области.

Наряду с методами обратного распространения применяются очень многообещающие эволюционные алгоритмы, которые могут решать вышеупомянутые проблемы. Эти основанные на биологии методы черпают вдохновение из теории эволюции Дарвина и используют принципы эволюции видов для создания искусственных нейронных сетей. Основная идея нейроэволюции состоит в том, чтобы создавать нейросеть с помощью стохастических методов поиска, основанных на популяции. Используя эволюционный подход, можно разработать оптимальные архитектуры нейронных сетей, которые точно решают конкретные задачи. В результате могут быть созданы компактные и энергоэффективные сети с умеренными требованиями к вычислительной мощности. Процесс эволюции реализуется путем применения генетических операторов (мутация, кроссовер) к популяции хромосом (генетически кодированные представления нейросетей или решений) на протяжении многих поколений. Большие надежды на этот метод основаны на том, что в природных биологических системах каждое последующее поколение становится все более приспособленным к внешним обстоятельствам, которые можно выразить целевой функцией, то есть они становятся лучшими приближениями целевой функции.

Далее мы обсудим основные понятия генетических алгоритмов. Вам достаточно будет иметь умеренный уровень понимания принципов работы генетических алгоритмов.

1.1.1 Генетические операторы

Генетические операторы находятся в самом сердце каждого эволюционного алгоритма, и от них зависит результативность любого нейроэволюционного алгоритма. Существует два основных генетических оператора: *мутация* и *кроссовер* (*рекомбинация*).

В этой главе вы узнаете об основах генетических алгоритмов и о том, как они отличаются от обычных алгоритмов, в которых для обучения нейросети используются методы обратного распространения ошибок.

Оператор мутации

Оператор мутации выполняет важную роль сохранения генетического разнообразия популяции в процессе эволюции и предотвращает остановку в локальных минимумах, когда хромосомы организмов в популяции становятся слишком похожими. Эта мутация изменяет один или несколько генов в хромосоме в соответствии с вероятностью мутации, определенной экспериментатором. Вводя случайные изменения в хромосому *решателя* (solver), мутация позволяет эволюционному процессу исследовать новые области в пространстве поиска возможных решений и находить все лучшие и лучшие решения на протяжении поколений.

На рис. 1.1 показаны распространенные типы операторов мутации.

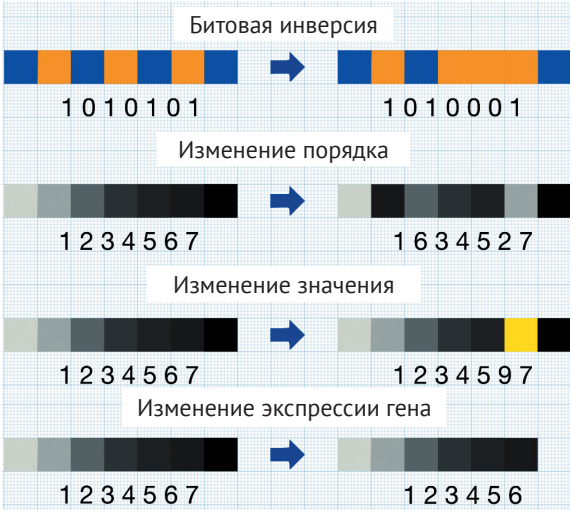


Рис. 1.1. Типы операторов мутации

Точный тип оператора мутации зависит от вида генетического кодирования, используемого конкретным генетическим алгоритмом. Среди различных типов мутаций, с которыми мы сталкиваемся, можно выделить следующие основные варианты:

- **битовая инверсия**: инвертируется случайно выбранный бит (бинарное кодирование);
- **изменение порядка**: изменение положения двух случайно выбранных генов в геноме (кодирование пермутации);
- **изменение значения**: к рабочему гену в случайной позиции добавляется небольшое значение (кодирование значения);
- **изменение экспрессии гена**: выбранный случайным образом ген добавляется в генотип / удаляется из генотипа (структурное кодирование).

Генотипы могут быть закодированы с использованием схем генетического кодирования с фиксированной и переменной длиной хромосомы. Первые три мутации могут быть применены к обоим типам схем кодирования. Последняя мутация может происходить только в генотипах, которые были закодированы с использованием переменной длины.

Оператор кроссовера

Оператор кроссовера (рекомбинации) позволяет нам стохастически генерировать новые поколения (решения) из существующих популяций путем рекомбинации генетической информации от двух родителей для генерации потомка. Таким образом, доли хороших решений от родительских организмов могут быть объединены и потенциально могут привести к лучшему потомству. Как правило, после операции кроссовера полученное потомство подвергается мутации перед добавлением в популяцию следующего поколения.

Различные операторы кроссовера показаны на рис. 1.2.

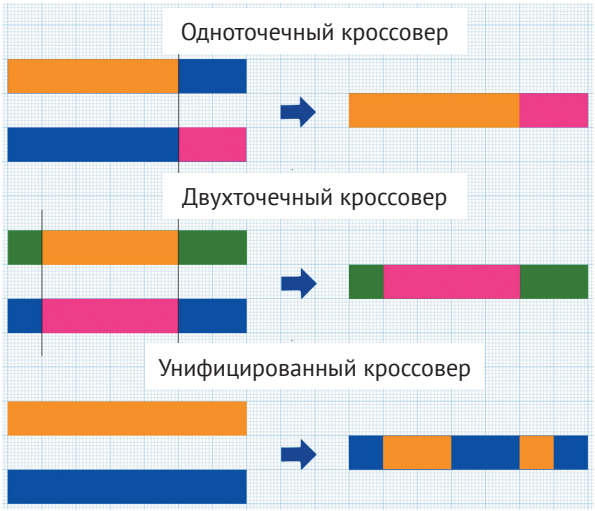


Рис. 1.2. Типы операторов кроссовера (рекомбинации)

Различные типы операторов кроссовера также зависят от схемы кодирования генома, используемого конкретными алгоритмами, но наиболее распространенными являются следующие:

- **одноточечный кроссовер**: выбирается случайная точка пересечения, часть генома от начала до точки пересечения копируется в потомство от одного родителя, а остаток копируется от другого родителя;
- **двухточечный кроссовер**: случайным образом выбираются две точки пересечения, часть генома от начала до первой точки копируется из первого родителя, часть между первой и второй точками пересечения копируется из второго родителя, и остаток копируется из первого родителя;
- **унифицированный кроссовер**: гены случайным образом копируются из первого или второго родителя.

1.1.2 Схемы кодирования генома

Одним из наиболее важных решений при разработке нейроэволюционного алгоритма является выбор генетического представления нейронной сети, которое может быть реализовано следующими способами:

- стандартная мутация (см. раздел «Оператор мутации»);
- операторы комбинирования (см. раздел «Оператор кроссовера»).

В настоящее время существуют две основные схемы кодирования генома: прямая и косвенная. Рассмотрим каждую схему более подробно.

Прямое кодирование генома

Прямое кодирование генома пытались использовать в методах нейроэволюции для создания нейросетей, относящихся к сетям с *фиксированной топологией*, – то есть топология сети определяется только экспериментатором. Здесь гене-

тический код (генотип) реализуется как вектор действительных чисел, представляющих силу (вес) связей между узлами сети.

Эволюционные операторы изменяют значения вектора весовых коэффициентов с помощью оператора мутации и объединяют векторы родительских организмов с помощью оператора кроссовера для получения потомства. Позволяя легко применять эволюционные операторы, упомянутый метод кодирования в то же время имеет некоторые существенные недостатки. Один из его основных недостатков заключается в том, что топология сети определяется экспериментатором с самого начала и остается постоянной для всех поколений за время выполнения эксперимента. Этот подход противоречит естественному эволюционному процессу, при котором в ходе эволюции изменяются не только свойства, но и физическая структура организмов, что позволяет охватывать максимально широкое пространство поиска и находить оптимальные решения. На рис. 1.3 показан эволюционный процесс.

Рис. 1.3. Эволюционный процесс

Чтобы устранить недостатки методов с фиксированной топологией, Кеннет О. Стэнли предложил метод *нейроэволюции с развитием топологии* (neuroevolution of augmenting topologies, NEAT). Основная идея этого алгоритма заключается в том, что эволюционные операторы применяются не только к вектору с весами всех связей, но и к топологии созданной нейронной сети. Таким образом, путем генерации популяций организмов проверяются различные топологии с различными весами связей. Мы обсудим особенности алгоритма NEAT позже в этой главе.

Алгоритм NEAT демонстрирует выдающуюся эффективность в самых разных задачах – от традиционного обучения с подкреплением до управления сложными автономными персонажами в компьютерных играх – и стал одним из самых популярных эволюционных алгоритмов за всю историю машинного обучения. Тем не менее он принадлежит к семейству алгоритмов прямого кодирования, которое ограничивает его использование до развития только нейросетей небольшого размера, где пространство параметров ограничено максимум тысячами связей. Это связано с тем, что каждая связь кодируется непосредственно в генотипе, и при большом количестве закодированных связей вычислительные требования значительно возрастают. Это делает невозможным использование алгоритма для развития больших нейронных сетей.

Косвенное кодирование генома

Чтобы преодолеть проблемы размерности в подходе с прямым кодированием, Кеннет О. Стэнли предложил метод *косвенного кодирования*, который основан на кодировании фенотипа посредством генома в ДНК. Он основан на том факте, что физический мир построен вокруг геометрии и закономерностей (структурных паттернов), где естественные симметрии встречаются повсюду. Таким образом, размер кода любого физического процесса может быть значительно уменьшен путем повторного использования определенного набора блоков кодирования для одной и той же структуры, которая повторяется много раз. Предложенный метод, называемый *нейроэволюцией с развитием топологии на основе гиперкуба* (hypercube-based neuroevolution of augmenting topologies, HyperNEAT), предназначен для построения крупномасштабных нейронных сетей с использованием геометрических закономерностей. HyperNEAT использует *сети, производящие составные паттерны* (compositional pattern producing network, CPPN), для представления связей между узлами как функции декартова пространства. Мы обсудим HyperNEAT более подробно далее в этой главе.

1.1.3 Коэволюция

В природе популяции разных видов часто одновременно развиваются во взаимодействии друг с другом. Этот тип межвидовых отношений называется *коэволюцией*. Коэволюция является мощным инструментом естественной эволюции, и неудивительно, что она привлекла внимание разработчиков нейроэволюционных алгоритмов. Существует три основных типа коэволюции:

- *мутуализм*, когда два или более вида мирно сосуществуют и взаимно выигрывают друг от друга;
- *конкурентная коэволюция*:
  - *хищничество*, когда один организм убивает другой и потребляет его ресурсы;
  - *паразитизм*, когда один организм использует ресурсы другого, но не убивает его;
- *комменсализм*, когда представители одного вида получают выгоды, не причиняя вреда и не принося выгоды другим видам.

Исследователи изучили существующие стратегии коэволюции и выявили их плюсы и минусы. В этой книге мы представим нейроэволюционный алгоритм, который использует принцип комменсализма для поддержания двух одновременно развивающихся групп: совокупности возможных решений и совокупности целевых функций кандидатов. Мы обсудим алгоритм *эволюции решений и приспособленности* (solution and fitness evolution, SAFE) в главе 9.

1.1.4 Модульность и иерархия

Другим важным аспектом организации естественных когнитивных систем является модульность и иерархия. При изучении человеческого мозга нейробиологи обнаружили, что это не монолитная система с однородной структурой, а сложная иерархия модульных структур. Кроме того, из-за ограни-

чений скорости распространения сигнала в биологических тканях структура мозга обеспечивает *принцип локальности*, когда связанные задачи обрабатываются геометрически смежными структурами в мозге. Эти особенности природных систем не остались без внимания исследователей нейроэволюции и реализованы во многих эволюционных алгоритмах. В главе 8 мы обсудим создание модульных нейросетей с использованием алгоритма на основе нейроэволюции.

1.2 Обзор алгоритма NEAT

Метод NEAT предназначен для уменьшения размерности пространства поиска параметров посредством постепенного развития структуры нейросети в процессе эволюции. Эволюционный процесс начинается с популяции маленьких, простых геномов (семян) и постепенно увеличивает их сложность с каждым новым поколением.

Геномы семян имеют очень простую топологию: доступны (экспрессированы) только входные, выходные и смещающие нейроны. На начальном этапе скрытые узлы отсутствуют, чтобы гарантировать, что поиск решения начинается в пространстве параметров (весов связей) с наименьшим числом измерений. С каждым новым поколением вводятся дополнительные гены, расширяющие пространство поиска решения, представляя новое измерение, которое ранее не существовало.

Таким образом, эволюция начинается с поиска в небольшом пространстве, которое можно легко оптимизировать, и при необходимости добавляет новые измерения. При таком подходе сложные фенотипы (решения) могут быть обнаружены постепенно, шаг за шагом, что намного эффективнее, чем запуск поиска непосредственно в обширном пространстве окончательных решений. Естественная эволюция использует похожую стратегию, время от времени добавляя новые гены, которые делают фенотипы более сложными. В биологии этот процесс постепенного усложнения называется *комплексным расширением*.

Основная цель метода NEAT – минимизировать сложность структуры генома – касается не только конечного продукта, но и всех промежуточных поколений организмов. Таким образом, эволюция топологии сети приводит к значительному выигрышу в производительности за счет сокращения общих решений для пространства поиска. Например, многомерное пространство окончательного решения возникает только в конце эволюционного процесса. Еще одна существенная особенность алгоритма заключается в том, что каждая структура, представленная в геноме, является предметом последующих оценок пригодности в будущих поколениях. Кроме того, во время эволюционного процесса выживут только полезные структуры. Другими словами, структурная сложность генома всегда оправдана целями.

1.2.1 Схема кодирования NEAT

Схема генетического кодирования NEAT разработана таким образом, чтобы можно было легко сопоставлять соответствующие гены во время процесса

спаривания, когда к двум родительским геномам применяется оператор кроссовера. Геном NEAT является линейным представлением схемы связей кодированной нейронной сети, как показано на рис. 4.1.

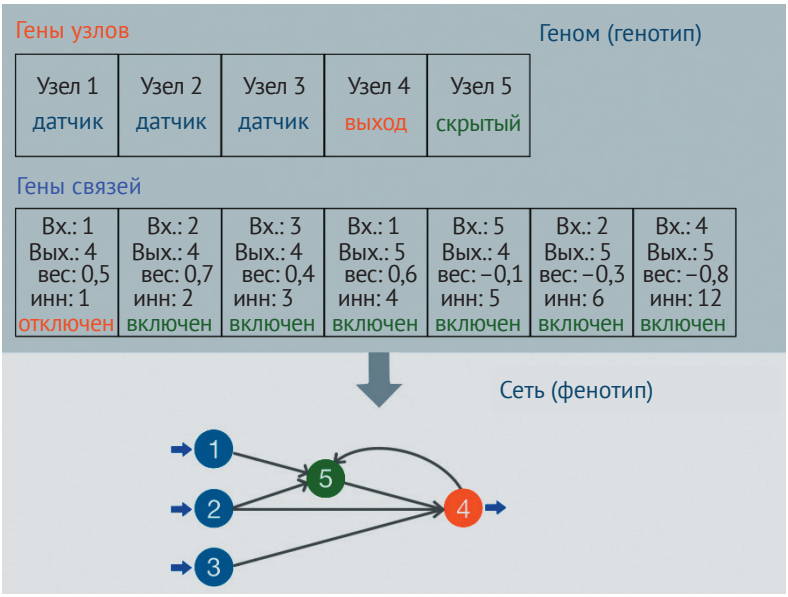


Рис. 1.4. Схема генома NEAT

Каждый геном представлен в виде списка *генов связей*, которые кодируют связи между узлами нейронной сети. Кроме того, существуют *гены узлов*, которые кодируют информацию о сетевых узлах, такую как идентификатор узла, тип узла и тип функции активации. Ген связи кодирует следующие параметры связи между узлами:

- идентификатор входного узла;
- идентификатор выходного узла;
- сила (вес) связи;
- бит, который указывает, включена (экспрессирована) связь или нет;
- номер обновления, который позволяет сопоставлять гены во время рекомбинации.

На нижней части рис. 4.1 представлена схема того же генома в виде ориентированного графа.

1.2.2 Структурные мутации

Специфический для NEAT оператор мутации может изменить силу (вес) связи и структуру сети. Существует два основных типа структурных мутаций:

- добавление новой связи между узлами;
- добавление нового узла в сеть.

Структурные мутации алгоритма NEAT схематически изображены на рис. 1.5.

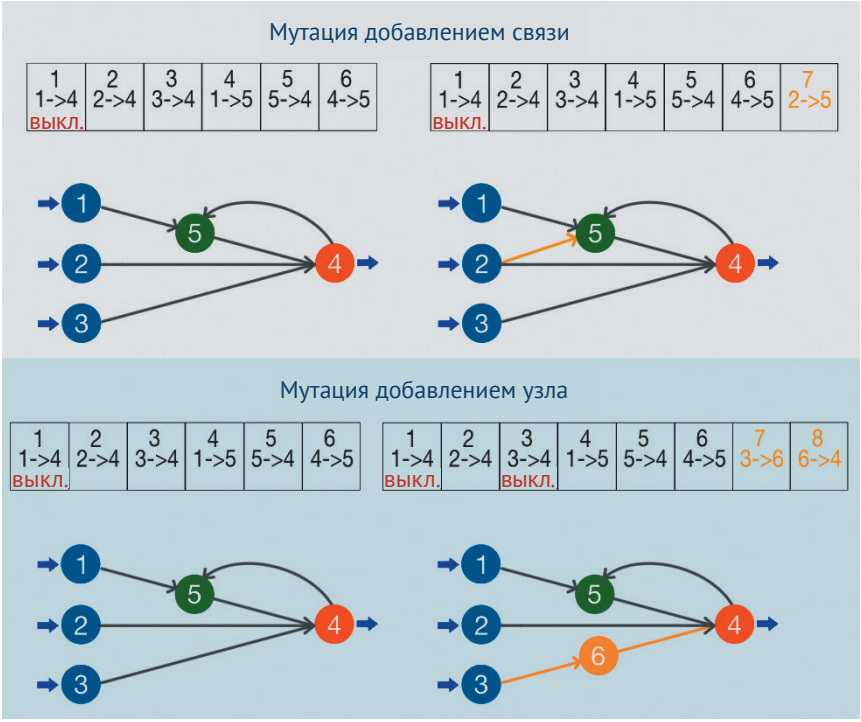


Рис. 1.5. Структурные мутации в алгоритме NEAT

Когда оператор мутации применяется к геному NEAT, вновь добавленному гену (гену связи или гену узла) присваивается очередной номер обновления. В ходе эволюционного процесса геномы организмов в популяции постепенно становятся больше, и образуются геномы различных размеров. Этот процесс приводит к тому, что в одинаковых позициях в геноме находятся разные гены связей, что делает процесс сопоставления между генами одного и того же происхождения чрезвычайно сложным.

1.2.3 Кроссовер с номером обновления

В эволюционном процессе есть неочевидная информация, которая точно говорит нам, какие гены должны совпадать между геномами любого организма в топологически разнообразной популяции. Это информация, при помощи которой каждый ген сообщает нам, от какого предка он был получен.

Гены связей с одним и тем же историческим происхождением представляют одну и ту же структуру, несмотря на то что, возможно, имеют разные значения весов. Историческое происхождение генов в алгоритме NEAT отражено в увеличивающихся номерах обновлений, которые позволяют нам отслеживать хронологию структурных мутаций.

В то же время в процессе кроссовера потомки наследуют номера обновлений генов от родительских геномов. Таким образом, номера обновлений конкретных генов никогда не меняются, что позволяет сопоставить сходные гены из разных геномов в ходе кроссовера. Номера обновлений совпадаю-

щих генов одинаковы. Если номера обновлений не совпадают, ген принадлежит непересекающейся или избыточной части генома, в зависимости от того, находится ли его номер обновления внутри или вне диапазона других родительских номеров. Непересекающиеся или избыточные гены представляют собой структуры, которых нет в геноме другого родителя и которые требуют особой обработки во время фазы кроссовера. Таким образом, потомство наследует гены, которые имеют одинаковые номера обновлений. Они случайным образом выбираются у одного из родителей. Потомство всегда наследует непересекающиеся или избыточные гены от родителей с самой высокой степенью приспособленности. Эта особенность позволяет алгоритму NEAT эффективно выполнять рекомбинацию генов с использованием линейного кодирования генома без необходимости проведения сложного топологического анализа.

На рис. 1.6 показан пример кроссовера (рекомбинации) между двумя родителями с использованием алгоритма NEAT. Геномы обоих родителей выравниваются при помощи номера обновления (число в верхней части ячейки гена связи). После этого производится потомство путем случайного выбора от любого из родителей генов связи с совпадающими номерами обновлений – это гены с номерами от одного до пяти. Наконец, от любого из родителей безусловным образом добавляются непересекающиеся и избыточные гены, которые выстраиваются по нарастанию номера обновления.

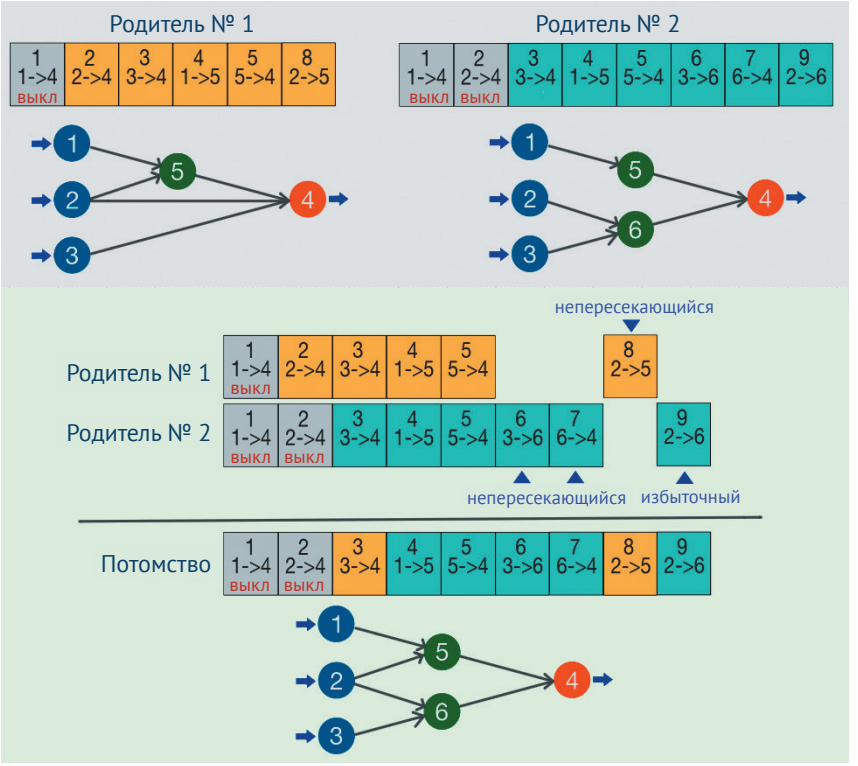


Рис. 1.6. Кроссовер (рекомбинация) алгоритма NEAT

1.2.4 Видообразование

В процессе эволюции организмы могут из поколения в поколение создавать разнообразные топологии, но они не могут производить и поддерживать собственные топологические обновления. Меньшие сетевые структуры оптимизируются быстрее, чем большие, что искусственно уменьшает шансы на выживание потомка генома после добавления нового узла или связи. Таким образом, недавно дополненные топологии испытывают *отрицательное эволюционное давление* из-за временного снижения приспособленности организмов в популяции. В то же время новые топологии могут содержать инновации, которые в конечном итоге приводят к выигрышному решению, и было бы жалко их потерять. Для решения проблемы временного снижения пригодности в алгоритм NEAT была введена концепция видообразования. *Видообразование* ограничивает круг организмов, которые могут спариваться, вводя узкие ниши, когда организмы, принадлежащие к одной и той же нише, в фазе кроссовера конкурируют только друг с другом, а не со всеми организмами в популяции. Видообразование реализуется путем деления популяции так, чтобы организмы с похожей топологией принадлежали к одному и тому же виду, то есть за счет *кластеризации геномов по видам*. Обобщенная форма алгоритма видообразования представлена на рис. 1.7.

Алгоритм 1: Кластеризация геномов по видам

Вход: Популяция организмов (*Population*) и известные виды (*Species*)

Результат: Организмы кластеризованы по видам. При необходимости созданы новые виды.

```
foreach genome ∈ Population do
    foreach S ∈ Species do
        if genome.IsCompatible(S) then
            // Добавляем подходящий геном в текущие виды
            S.AddGenome(genome);
        else if S is the last known species then
            // Создаем новые виды для данного генома
            S_new ⟵ create_new_species(genome) ;
            // Добавляем новые виды в список известных видов
            Species ⟵ Species ∪ S_new ;
```

Рис. 1.7. Обобщенная форма алгоритма видообразования

Метод NEAT позволяет создавать сложные нейросети, способные решать различные задачи оптимизации управления, а также другие проблемы обучения без учителя. Благодаря способности топологии нейросети эволюционировать посредством усложнения и видообразования полученные решения, как правило, имеют оптимальную производительность обучения и логического вывода.

Результирующая топология растет строго в соответствии с проблемой, которая должна быть решена, без каких-либо лишних скрытых слоев, вводимых

обычными методами проектирования топологии нейросетей, обучаемых на основе обратного распространения ошибки.

Подробнее прочитать про алгоритм NEAT можно в оригинальной статье по адресу: **http://nn.cs.utexas.edu/downloads/papers/stanley.phd04.pdf.**

1.3 NEAT на основе гиперкуба

Интеллект является продуктом мозга, а человеческий мозг как структура сам является продуктом естественной эволюции. Такая сложная структура развивалась в течение миллионов лет под давлением суровых условий и в то же время конкурировала за выживание с другими живыми существами. В результате возникла чрезвычайно развитая система со многими слоями, модулями и триллионами связей между нейронами. Структура человеческого мозга является нашей путеводной звездой и помогает нам в создании систем искусственного интеллекта. Однако как мы можем постичь всю сложность человеческого мозга с помощью наших несовершенных инструментов?

Изучая мозг человека, нейробиологи обнаружили, что его пространственная структура играет важную роль во всех задачах восприятия и познания – от зрения до абстрактного мышления. Было найдено много сложных геометрических структур, таких как клетки коры мозга, которые помогают нам в инерциальной навигации, и кортикальные столбцы, которые связаны с сетчаткой глаза для обработки зрительных образов. Было доказано, что структура мозга позволяет нам эффективно реагировать на образы в сигналах, поступающих от сенсориума[1], с помощью обособленных нейронных структур, которые активируются определенными образами на входах. Эта особенность мозга позволяет ему использовать чрезвычайно эффективный способ представления и обработки всего разнообразия входных данных, полученных из окружающей среды. Наш мозг превратился в набор эффективных механизмов распознавания и обработки образов, которые активно применяют повторное использование нейронных модулей, тем самым значительно сокращая количество необходимых нейронных структур. Это стало возможным только благодаря сложной модульной иерархии и пространственной интеграции различных частей.

Иными словами, биологический мозг реализует сложные иерархические и пространственно-ориентированные процедуры обработки данных. Это вдохновило исследователей нейроэволюции на внедрение аналогичных методов обработки данных в области искусственных нейронных сетей. При проектировании подобных систем необходимо учитывать следующие проблемы:

- огромное количество входных признаков и параметров обучения, которые требуют построения крупномасштабных нейросетей;
- эффективное представление естественных геометрических закономерностей и симметрий, которые наблюдаются в физическом мире;

[1] Совокупность органов чувств (источников биологических сигналов) и клеток головного мозга, непосредственно принимающих эти сигналы. – *Прим. перев.*

- эффективная обработка входных данных благодаря внедрению принципа локальности, то есть когда пространственно- и семантически смежные структуры данных обрабатываются модулями взаимосвязанных нейронных блоков, которые занимают одну и ту же компактную область всей сетевой структуры.

В этом разделе вы узнали о методе HyperNEAT на основе гиперкуба, который был предложен Кеннетом О. Стэнли для решения различных задач с использованием многомерных геометрических структур. Далее мы рассмотрим *сети, производящие составные паттерны* (CPPN).

1.3.1 Сети, производящие составные паттерны

Алгоритм HyperNEAT расширяет исходный алгоритм NEAT, вводя новый тип схемы кодирования косвенного генома под названием CPPN. Этот тип кодирования позволяет представлять паттерны связей нейросети фенотипа как функцию его геометрии.

HyperNEAT сохраняет паттерн связей нейронной сети фенотипа в виде четырехмерного гиперкуба, где каждая точка кодирует связь между двумя узлами (то есть координатами исходного и целевого нейронов), а CPPN рисует в нем различные паттерны. Другими словами, CPPN вычисляет четырехмерную функцию, которая определяется следующим образом:

$$w = CPPN(x_1, y_2, x_2, y_2).$$

Здесь исходный узел находится в точке (x_1, y_2), а целевой узел – в точке (x_1, y_2). На этом этапе CPPN возвращает вес для каждой связи между каждым узлом в сети фенотипа, который представлен в виде сетки. По соглашению связь между двумя узлами не *экспрессируется* (не участвует в процессах генома), если величина веса связи, вычисленная с помощью CPPN, меньше минимального порогового значения ($w_{\min}$). Таким образом, паттерн связей, созданный CPPN, может представлять произвольную топологию сети. Паттерн связей может использоваться для кодирования крупномасштабных нейросетей путем обнаружения закономерностей в обучающих данных и может повторно использовать тот же набор генов для кодирования повторений. Паттерн связей, созданный CPPN, принято называть *субстратом* (substrate).

На рис. 1.8 показана интерпретация паттерна геометрической связности на основе гиперкуба.

В отличие от традиционных нейросетевых архитектур, CPPN может использовать различные функции активации для своих скрытых узлов и тем самым отражать различные геометрические закономерности. Например, для представления повторений можно использовать тригонометрический синус, а для обеспечения локальности в определенной части сети (то есть симметрии вдоль оси координат) – гауссову кривую. Таким образом, схема кодирования CPPN может компактно представлять паттерны с различными геометрическими закономерностями, такими как симметрия, повторение, повторение с закономерностями и т. д.

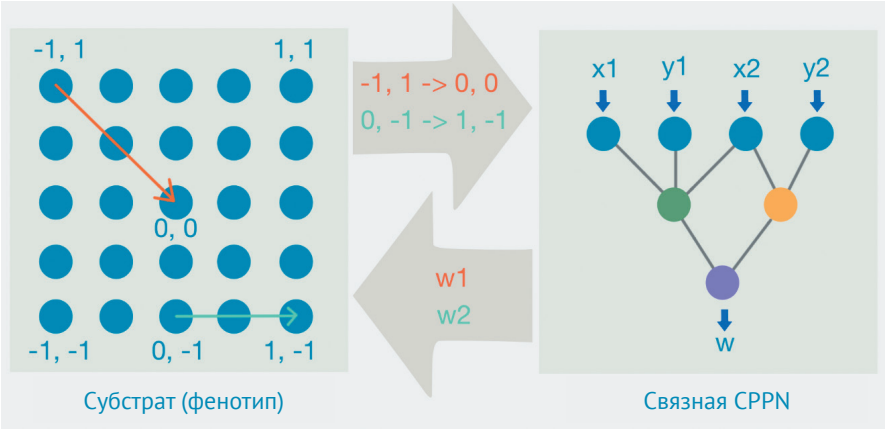


Рис. 1.8. Интерпретация паттерна геометрической связности на основе гиперкуба

1.3.2 Конфигурация субстрата

Компоновка сетевых узлов в субстрате, к которому подключается CPPN, может принимать различные формы, которые лучше всего подходят для решения конкретных задач. Экспериментатор отвечает за выбор подходящей компоновки для достижения оптимальной производительности. Например, выходные узлы, которые управляют радиальным объектом, таким как механизм с шестью шагающими конечностями, удобнее всего располагать в круговой форме, чтобы паттерн связей мог быть представлен в полярных координатах.

Существует несколько распространенных типов конфигурации субстрата, которые обычно используются с HyperNEAT (рис. 1.9):

- **двухмерная сетка**: регулярная сетка узлов сети в двухмерном декартовом пространстве с центром в (0, 0);
- **трехмерная сетка**: регулярная сетка узлов сети в трехмерном декартовом пространстве с центром в (0, 0, 0);
- **сэндвич**: две двухмерные плоские сетки с узлами входа и выхода, где один слой может выстраивать связи в направлении другого;
- **круговая**: упорядоченная радиальная структура, которая подходит для определения закономерностей в полярных координатах.

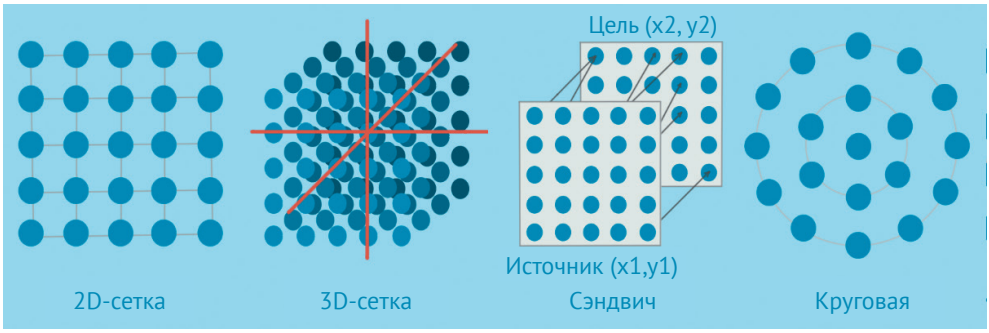


Рис. 1.9. Примеры конфигурации слоев субстрата

1.3.3 CPPN с развивающимися связями и алгоритм HyperNEAT

Метод называется HyperNEAT, потому что он использует модифицированный алгоритм NEAT для развития CPPN, представляющих объемные структуры в гиперпространстве. Каждая экспрессированная (доступная для генетических операций) точка паттерна, находящаяся в *гиперкубе* (hypercube), представляет собой связь между двумя узлами графа нижнего измерения (субстрата). Таким образом, размерность гиперпространства в два раза больше размерности нижележащего графа наименьшего измерения. Далее в главе 8 мы рассмотрим некоторые примеры, в которых используются двумерные паттерны связей.

Обобщенная форма алгоритма HyperNEAT представлена на рис. 1.10.

Алгоритм 2: Общая форма алгоритма HyperNEAT

```
begin
    1. Выбор нужной конфигурации субстрата (слои узлов и сопоставление входов/выходов);
    2. Инициализация популяции минимальной CPPN со случайными весами связей;
    repeat
        foreach организм ∈ популяция do
            3. Запросить CPPN организма о весе каждой возможной связи в субстрате,
               представляющем его фенотип. Если абсолютное значение выхода превышает
               пороговую величину, создать связь с весом, масштабированным
               пропорционально выходному значению;
            4. Использовать субстрат как фенотип ANN в целевой области, чтобы оценить
               пригодность найденного решения;
        5. Воспроизвести CPPN организмов в популяции при помощи NEAT;
    until решение найдено;
```

Рис. 1.10. Обобщенная форма алгоритма HyperNEAT

Любой ген связи или ген узла, который был добавлен к CPPN во время ее эволюции, приводит к открытию нового глобального измерения вариаций паттернов связей через субстрат фенотипа (новые признаки). Каждая модификация генома CPPN представляет новый способ изменения всего паттерна связей. Кроме того, ранее разработанные CPPN могут быть использованы в качестве основы для создания паттернов связей для субстрата с более высоким разрешением, чем то, которое использовалось для его обучения. Это позволяет нам получить рабочее решение прежней проблемы при любом разрешении, возможно, без ограничения верхнего предела. Вышеупомянутые свойства сделали HyperNEAT мощным инструментом для развития крупномасштабных искусственных нейронных сетей, имитирующих биологические объекты.

Больше информации о методе HyperNEAT вы можете найти по адресу **https://eplex.cs.ucf.edu/papers/stanley_alife09.pdf**.

1.4 HyperNEAT с развиваемым субстратом

Метод HyperNEAT основан на идее, что геометрические структуры живого мозга могут быть полноценно представлены искусственными нейронными сетями с узлами, расположенными в определенных пространственных местоположениях. Таким образом, нейроэволюция получает значительные преимущества и позволяет обучать крупномасштабные нейросети для сложных задач, что было невозможно с обычным алгоритмом NEAT. В то же время хотя подход HyperNEAT и основан на структуре естественного мозга, ему по-прежнему не хватает пластичности процесса естественной эволюции. Позволяя эволюционному процессу разрабатывать различные паттерны связей между узлами сети, метод HyperNEAT накладывает жесткое ограничение на то, где размещаются узлы. Экспериментатор должен определить расположение узлов сети с самого начала, и любое неверное предположение, сделанное исследователем, снизит эффективность эволюционного процесса.

Размещая сетевой узел в определенном месте на субстрате, экспериментатор создает непреднамеренное ограничение на паттерн весов, создаваемый CPPN. Это ограничение затем мешает CPPN, когда она пытается закодировать геометрические закономерности мира природы в топографию решающей нейросети (фенотип). В данном случае паттерн связей, создаваемый CPPN, должен идеально совпадать со слоем субстрата, который определен экспериментатором; возможны только связи между сетевыми узлами, которые определены заранее. Такое ограничение приводит к ненужным ошибкам аппроксимации, которые портят результат. Если позволить CPPN разрабатывать паттерны связей для узлов, которые находятся в немного других местах, это может оказаться более эффективным подходом.

1.4.1 Плотность информации в гиперкубе

Почему мы должны начинать с наложения подобных ограничений на расположение узлов? Разве не было бы лучше, если бы неявные подсказки, полученные из паттернов связей, стали бы указаниями к тому, где разместить следующий узел, чтобы лучше представить естественные закономерности физического мира?

Области с одинаковыми весами связей кодируют небольшое количество информации и, следовательно, имеют небольшое функциональное значение. В то же время области с огромными градиентами значений веса являются чрезвычайно информационными. Такие области могут получить выгоду от размещения дополнительных узлов сети для более точного кодирования природного процесса. Как вы помните из нашего обсуждения алгоритма HyperNEAT, связь между двумя узлами в субстрате можно представить в виде точки в четырехмерном гиперкубе. Главная идея предложенного алгоритма ES-HyperNEAT состоит в том, чтобы разместить больше гиперточек в тех областях гиперкуба, где обнаруживаются бóльшие вариации весов связей. В то же время в областях с меньшим разбросом весов связей размещается меньшее количество гиперточек.

Размещение узлов и образование связей между ними может быть продиктовано изменением веса связей, которые создаются эволюционирующей CPPN для данной области субстрата. Другими словами, для принятия решения о размещении следующего узла в субстрате нет необходимости в дополнительной информации, кроме той, что мы уже получаем от CPPN, кодирующей паттерны связей сети. Основным руководящим принципом для алгоритма определения топографии субстрата становится плотность информации.

Размещение узла в фенотипе нейросети отражает информацию, которая закодирована в паттернах связей, созданных CPPN.

1.4.2 Квадродерево как эффективный экстрактор информации

Для представления гиперточек, которые кодируют веса связей внутри гиперкуба, алгоритм ES-HyperNEAT использует квадродерево. *Квадродерево* – это древовидная структура данных, в которой каждый внутренний узел имеет ровно четыре дочерних узла. Эта структура данных была выбрана благодаря присущим ей свойствам, что позволяет ей представлять двумерные области на разных уровнях детализации. С помощью квадродерева можно организовать эффективный поиск в двухмерном пространстве, разбив любую интересующую область на четыре подобласти, и каждая из них становится листом дерева, а корневой (родительский) узел представляет исходную (декомпозированную) область.

Используя метод извлечения информации на основе квадродерева, метод ES-HyperNEAT итеративно ищет новые связи между узлами в двумерном пространстве субстрата нейросети, начиная с узлов ввода и вывода, которые были предварительно определены экспериментатором. Этот метод намного эффективнее в вычислительном отношении, чем поиск непосредственно в четырехмерном пространстве гиперкуба.

На рис. 1.11 показан пример извлечения информации с использованием квадродерева.

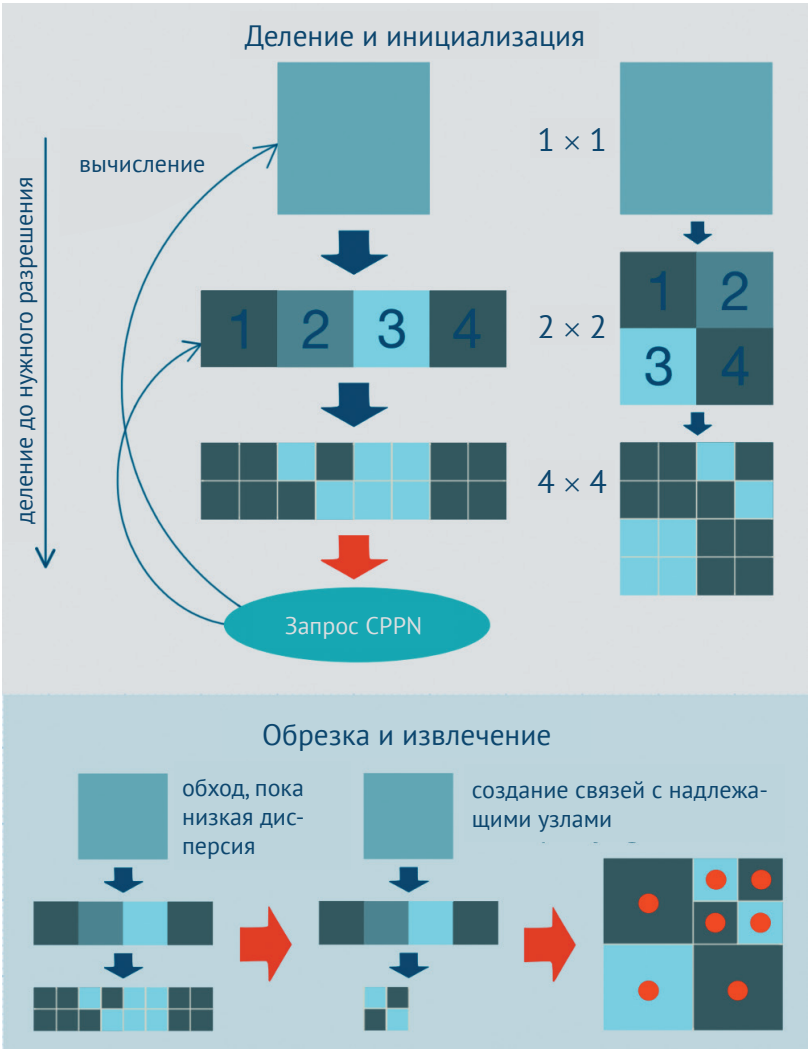


Рис. 1.11. Пример извлечения информации с использованием квадродерева

Алгоритм поиска на основе квадродерева работает в два основных этапа:

1. **Деление и инициализация**. На этом этапе квадродерево создается путем рекурсивного деления исходного пространства субстрата, занимающего область от (−1, −1) до (1, 1). Деление останавливается, когда достигается желаемая глубина дерева. От этого неявно зависит, сколько подпространств помещается в начальное пространство субстрата (разрешение инициализации). После этого для каждого узла дерева с центром в (a, b) запрашивается CPPN с аргументами (a, b, x_i, y_i) для определения весов связей. Когда веса связей для k конечных узлов конкретного узла p квадродерева найдены, дисперсию этого узла можно рассчитать по следующей формуле:

$$\sigma^2 = \sum_{i=1}^{k} \left(\bar{\omega} - \omega_i\right)^2.$$

Здесь $\bar{\omega}$ – средний вес соединения между k конечных узлов и ω_i – вес соединения с конкретным конечным узлом. Рассчитанное значение дисперсии является эвристическим индикатором наличия информации в конкретной области субстрата. Если это значение выше, чем конкретный порог деления (определяющий желаемую плотность информации), то для соответствующего квадрата можно повторить стадию деления. Таким образом алгоритм может обеспечить требуемую плотность информации. Посмотрите на верхнюю часть рис. 1.11, чтобы увидеть, как выполняются деление и инициализация с использованием квадродерева.

2. **Обрезка и извлечение**. Чтобы гарантировать, что в областях с высокой плотностью информации (большой разброс весов) будет экспрессировано большее количество связей (и узлов в субстрате), для созданного на предыдущем этапе квадродерева выполняется процедура обрезки и извлечения. Для квадродерева выполняют поиск в глубину, пока дисперсия текущего узла не станет меньше порога дисперсии σ_t^2 или пока не окажется, что у узла нет дочерних элементов (нулевая дисперсия). Для каждого отвечающего этим условиям узла экспрессирована связь с соответствующим центром (x, y), и каждый родительский узел уже определен либо экспериментатором, либо найден при предыдущем запуске этих двух этапов (то есть среди скрытых узлов, которые уже были созданы методом ES-HyperNEAT). Фаза обрезки и извлечения наглядно представлена на нижней части рис. 1.11.

1.4.3 Алгоритм ES-HyperNEAT

Алгоритм ES-HyperNEAT начинается с определяемых пользователем входных узлов и подробно исследует исходящие из них связи, направленные ко вновь экспрессированным скрытым узлам. Экспрессия паттернов исходящих связей и размещения скрытых узлов в пространстве субстрата выполняется с использованием метода извлечения информации из квадродерева, который мы описали ранее. Процесс извлечения информации применяется итеративно до тех пор, пока не будет достигнут желаемый уровень плотности представления информации или пока в гиперкубе не перестанет обнаруживаться новая информация. После этого результирующая сеть соединяется с определенными пользователем выходными узлами путем экспрессии паттернов входящих подключений, направленных к выходам. Для этого мы также используем извлечение информации из квадродерева. В конечной (экспрессированной) сети остаются только те скрытые узлы, которые имеют путь как к входному, так и к выходному узлу.

Теперь у нас определено множество узлов и связей внутри субстрата фенотипа нейросети. Может быть выгодно удалить некоторые узлы из сети, введя дополнительную стадию обрезки. На этой стадии мы сохраняем только точки в пределах определенной полосы и удаляем точки на краю полосы. Делая полосы шире или уже, CPPN может управлять плотностью закодированной информации. Для получения более подробной информации о стадии обрезки,

пожалуйста, прочтите статью про ES-HyperNEAT (**https://eplex.cs.ucf.edu/papers/risi_alife12.pdf**).

Обобщенная форма алгоритма ES-HyperNEAT представлена на рис. 1.12.

Алгоритм 3: ES-HyperNEAT

```
Параметры: initialDepth, maxDepth, varianceThreshold, bandThreshold, iterationLevel,
           divisionThreshold
Вход     : CPPN, InputPositions, OutputPositions
Выход    : Connections, HiddenNodes

begin
    // Связи вход -> скрытый узел
    foreach input ∈ InputPositios do
        // Анализ паттерна исходящих связей
        root ⟵ DivisionAndInitialization(input.x, input.y, true);
        // Построение квадродерева и сохранение подходящих связей
        PruningAndExtraction(input.x, input.y, inputConnections, root, true);
        foreach c ∈ inputConnections do
            node ⟵ Node(c.x2, c.y2);
            if node ∉ HiddenNodes then
                HiddenNodes ⟵ HiddenNodes ∪ node;
    // Связи скрытый узел -> скрытый узел
    UnexploredHiddenNodes ⟵ HiddenNodes;
    for i ← 1 to iterationLevel do
        foreach hidden ∈ UnexploredHiddenNodes do
            root ⟵ DivisionAndInitialization(hidden.x, hidden.y, true);
            PruningAndExtraction(hidden.x, hidden.y, hidnConnections, root, true);
            foreach c ∈ hidnConnections do
                node ⟵ Node(c.x2, c.y2);
                if node ∉ HiddenNodes then
                    HiddenNodes ⟵ HiddenNodes ∪ node;
            // Удаление исследованных узлов
            UnexploredHiddenNodes ⟵ HiddenNodes − UnexploredHiddenNodes;
    // Связи скрытый узел -> выход
    foreach ouput ∈ OutputPositions do
        // Анализ входящих связей по отношению к выходу (output)
        root ⟵ DivisionAndInitialization(output.x, output.y, false);
        PruningAndExtraction(output.x, output.y, outputConnections, root, false);
        /* Здесь не создаются новые узлы, потому что все скрытые узлы, подключенные
           ко входу/выходу, уже экспрессированы */
    connections ⟵ inputConnections ∪ hidnConnections ∪ outputConnections;
    Удаление узлов и связей, не занятых в построении путей от входов к выходам.
```

Рис. 1.12. Обобщенная форма алгоритма ES-HyperNEAT

Алгоритм ES-HyperNEAT использует все преимущества методов NEAT и HyperNEAT и предоставляет еще более мощные новые функции, в том числе:

- автоматическое размещение скрытых узлов внутри субстрата для точного соответствия паттернам связи, которые выражены развитыми CPPN;
- позволяет нам гораздо проще создавать модульные нейросети фенотипа благодаря специфической способности сразу начать эволюционный поиск с уклоном в сторону локальности (благодаря особому устройству исходных архитектур CPPN);
- с помощью ES-HyperNEAT можно уточнить существующую структуру фенотипа нейросети, увеличив число узлов и связей в субстрате в процессе эволюции – в противоположность методу HyperNEAT, где количество узлов субстрата предопределено.

Алгоритм ES-HyperNEAT позволяет нам использовать исходную архитектуру HyperNEAT без изменения генетической структуры части NEAT. Это позволяет нам решать проблемы, которые трудно решить с помощью алгоритма HyperNEAT из-за сложностей с предварительным созданием соответствующей конфигурации субстрата.

Подробнее ознакомиться с алгоритмом ES-HyperNEAT и механизмами, лежащими в его основе, можно в статье по адресу **https://eplex.cs.ucf.edu/papers/risi_alife12.pdf**

1.5 Метод оптимизации поиском новизны

Большинство методов машинного обучения, включая эволюционные алгоритмы, основывают свое обучение на оптимизации целевой функции. Основная идея, лежащая в основе методов оптимизации *целевой функции* (objective function), заключается в том, что лучший способ улучшить производительность *решателя* (solver) – вознаграждать его за приближение к цели. В большинстве эволюционных алгоритмов близость к цели измеряется *приспособленностью* (fitness) решателя. Мера полезности организма определяется *функцией приспособленности* (fitness function), которая является метафорой эволюционного давления на организм для адаптации к окружающей среде. Согласно этой парадигме, наиболее приспособленный организм лучше приспособлен к окружающей среде и лучше всего подходит для поиска решения.

Хотя методы прямой оптимизации функций приспособленности хорошо работают во многих простых случаях, при решении более сложных задач они часто становятся жертвами *ловушки локального оптимума*. Сходимость к локальному оптимуму означает, что ни один локальный шаг в пространстве поиска не обеспечивает каких-либо улучшений в процессе оптимизации функции приспособленности. Традиционные генетические алгоритмы используют мутационные и островные механизмы, чтобы вырваться из таких локальных оптимумов. Однако, как мы выясним позже, выполняя эксперименты в данной книге, эти подходы не всегда работают при решении обманчивых проблем, или может потребоваться слишком много времени, чтобы найти успешное решение.

Многие реальные проблемы имеют такие обманчивые ландшафты функций приспособленности, которые не могут быть успешно пройдены с помощью процесса оптимизации, основанного исключительно на измерении близости

текущего решения к цели. В качестве примера мы можем рассмотреть задачу навигации по неизвестному городу с нерегулярным рисунком улиц. В такой задаче движение к месту назначения часто означает путешествие по обманчивым дорогам, которые уводят вас еще дальше, только чтобы привести вас к месту назначения после нескольких поворотов. Но если вы решите начать с дорог, которые лучше всего направлены на пункт назначения, они могут завести вас в тупик, в то время как пункт назначения находится прямо за стеной, но недоступен.

1.5.1 Поиск новизны и естественная эволюция

Рассматривая, как естественный отбор работает в физическом мире, мы видим, что движущей силой эволюционного разнообразия является *поиск новизны* (novelty search, NS). Другими словами, любой развивающийся вид получает непосредственные эволюционные преимущества по сравнению с конкурентами, находя новые модели поведения. Это позволяет ему более эффективно использовать окружающую среду. У естественной эволюции нет определенных целей, и она расширяет пространство поиска решений, вознаграждая за исследование и использование новых форм поведения. Новизну можно рассматривать как точку приложения усилий для многих скрытых творческих сил в мире природы, что позволяет эволюции разрабатывать еще более сложные модели поведения и биологические структуры.

Вдохновившись естественной эволюцией, Джоэл Леман предложил новый метод поисковой оптимизации для искусственного эволюционного процесса, названный поиском новизны. При использовании этого метода для поиска решения не определяется и не используется никакая конкретная функция приспособленности; вместо этого новизна каждого найденного решения вознаграждается непосредственно в процессе нейроэволюции.

Таким образом, новизна найденных решений направляет нейроэволюцию к конечной цели. Такой подход дает нам возможность использовать творческие силы эволюции, не зависящие от адаптивного давления, стремящегося приспособить решение к определенной нише.

Эффективность поиска новизны может быть продемонстрирована с помощью эксперимента по навигации в лабиринте, где поиск на основе близости к цели находит решение для простого лабиринта за гораздо большее количество шагов (поколений), чем поиск новизны. Кроме того, в случае сложного лабиринта с дезориентирующей конфигурацией поиск на основе близости к цели вообще не может найти решение. Мы обсудим эксперименты по навигации в лабиринте в главе 5.

1.5.2 Метрика новизны

Алгоритм поиска новизны использует метрику новизны для отслеживания уникальности поведения каждого нового организма. То есть *метрика новизны* – это мера того, насколько далеко новый организм находится от остальной части популяции в *пространстве поведения* (behavior space). Эффективная реализация метрики новизны должна позволить нам вычислить *разреженность* (sparseness) в любой точке пространства поведения. Любая область с более плотным скоплением посещенных точек менее нова и дает меньшее эволюционное вознаграждение.

Наиболее простой мерой разреженности в точке является среднее расстояние до *k*-ближайших соседей этой точки в пространстве поведения. Когда это расстояние велико, интересующий объект находится в разреженной области. В то же время более плотным областям присущи меньшие значения расстояний. Таким образом, разреженность в точке вычисляется по следующей формуле:

$$\rho(x) = \frac{1}{k}\sum_{i=0}^{k} dist(x, \mu_i).$$

Здесь μ_i – это *i*-й ближайший сосед *x*, определенный по метрике расстояния dist(x_i, μ_i). Метрика расстояния является предметно-ориентированной мерой *поведенческой разницы* между двумя индивидуумами.

Кандидаты из разреженных областей получают более высокие оценки новизны. Когда эта оценка превышает некоторый минимальный порог ρ_{min}, индивидуум в этой точке добавляется в архив лучших исполнителей, характеризующий распределение предыдущих решений в пространстве поведения. Нынешнее поколение населения в сочетании с архивом определяет, где был поиск раньше и где он будет происходить сейчас. Таким образом, градиент поиска направляется на новое поведение без какой-либо явной цели, просто максимизируя показатель новизны. Тем не менее поиск новизны по-прежнему основывается на прикладной информации, потому что изучение нового поведения требует всестороннего исследования области поиска.

Обобщенная форма алгоритма поиска новизны показана на рис. 1.13.

Алгоритм 4: Метод поиска новизны

Параметры : noveltyThreshold
Вход : Population, NoveltyArchive
Результаты : Каждый организм в популяции оценивается по метрике новизны, исходя из данных в *NoveltyArchive* и других организмов в *Population*. Если его оценка превышает noveltyThreshold, он добавляется в *NoveltyArchive*. Значение *Fitness* каждого организма обновляется на основе оценки новизны. Далее приспособленность (*Fitness*) может повлиять на решение о том, какой из организмов заслуживает участия в репродукции.

```
begin
    foreach organism ∈ Population do
        // Вычисляем оценку новизны для организма (organism) на основе
        // сочетания популяции (population) и архива новинок (NoveltyArchive)
        novelty ⟵ AvgKnnDistance(organism, Population, NoveltyArchive);
        if novelty > noveltyThreshold then
            // Добавляем organism в NoveltyArchive
            NoveltyArchive ⟵ NoveltyArchive ∪ organism;
            // Удаляем из архива новинок записи с наименьшим значением
            // оценки новизны, чтобы сохранить размер архива.
            PurgeNoveltyArchive(NoveltyArchive)
        /* Оценка новизны, сохраненная в качестве меры приспособленности, пригодится
           на следующем шаге эволюционного алгоритма, таком как репродукция, когда
           более приспособленные организмы имеют больше шансов оставить потомство. */
        organism.Fitness ⟵ novelty
```

Рис. 1.13. Обобщенная форма алгоритма поиска новизны

Метод оптимизации поиска новизны позволяет эволюции искать решения в любом дезориентирующем пространстве и находить оптимальные решения. С помощью этого метода возможно реализовать дивергентную эволюцию, когда популяция вынуждена не останавливаться на конкретном нишевом решении (локальный оптимум) и должна исследовать все пространство решений. Как показали эксперименты, это очень эффективный метод поисковой оптимизации, несмотря на его нелогичный подход, который полностью игнорирует явную цель во время поиска. Более того, он может найти окончательное решение в большинстве случаев даже быстрее, чем традиционный поиск на основе целей, измеряющий приспособленность как расстояние от окончательного решения.

Подробное описание алгоритма можно найти по адресу **http://joellehman.com/lehman-dissertation.pdf**.

1.6 Заключение

В этой главе мы начали с обсуждения различных методов, которые используются для обучения искусственных нейронных сетей. Мы рассмотрели, чем традиционные методы градиентного спуска отличаются от методов, основанных на нейроэволюции. Затем мы представили один из самых популярных алгоритмов нейроэволюции (NEAT) и два способа его расширения (HyperNEAT и ES-HyperNEAT). Наконец, вы познакомились с методом поисковой оптимизации (поиск по новизне), способным найти решение для множества дезориентирующих проблем, которые не могут быть решены с помощью традиционных методов поиска на основе целей. Теперь вы готовы применить эти знания на практике после настройки необходимой среды моделирования, которую мы обсудим в следующей главе.

В следующей главе мы также рассмотрим доступные библиотеки, позволяющие проводить эксперименты с нейроэволюцией в виде кода на языке Python. Вы узнаете, как настроить рабочую среду и какие инструменты пригодятся для управления зависимостями в экосистеме Python.

1.7 Дополнительное чтение

Для более глубокого понимания тем, которые мы обсуждали в этой главе, изучите дополнительные материалы по следующим ссылкам:

- **NEAT**: **http://nn.cs.utexas.edu/downloads/papers/stanley.phd04.pdf**;
- **HyperNEAT**: **https://eplex.cs.ucf.edu/papers/stanley_alife09.pdf**;
- **ES-HyperNEAT**: **https://eplex.cs.ucf.edu/papers/risi_alife12.pdf**;
- **Novelty Search**: **http://joellehman.com/lehman-dissertation.pdf**.

Глава 2

Библиотеки Python и настройка среды разработки

В этой главе вы познакомитесь с библиотеками Python, которые мы будем использовать для реализации алгоритмов нейроэволюции, упомянутых в предыдущей главе. Мы также обсудим сильные и слабые стороны каждой представленной библиотеки. В дополнение к этому рассмотрим обзорные примеры использования. Затем вы узнаете, как настроить среду для экспериментов, которые мы будем выполнять позже в этой книге. Наконец, вы познакомитесь с настройкой рабочей среды при помощи Anaconda Distribution – популярного у исследователей данных инструмента для управления зависимостями Python и виртуальными средами. В данной главе вы узнаете, как начать использовать Python для экспериментов с алгоритмами нейроэволюции, которые будут рассмотрены в этой книге.

В этой главе рассмотрены следующие темы:

- библиотеки Python для экспериментов с нейроэволюцией;
- настройка среды разработки.

2.1 Библиотеки Python для экспериментов с нейроэволюцией

Язык программирования Python является одним из самых популярных языков для деятельности, связанной с машинным обучением, а также исследованиями и разработками в области искусственного интеллекта. Наиболее известные фреймворки либо написаны на Python, либо предоставляют соответствующие интерфейсы. Такую популярность можно объяснить коротким циклом обучения программированию на Python и тем, что это язык написания сценариев, которые позволяют быстро и просто проводить эксперименты. В соответствии с тенденциями в сообществе машинного обучения на Python было написано несколько библиотек с поддержкой нейроэволюции, и число подобных библиотек со временем продолжает расти. Далее мы рассмотрим

наиболее стабильные библиотеки Python для экспериментов в области эволюционных алгоритмов.

2.1.1 Библиотека NEAT-Python

Как следует из названия, это реализация алгоритма NEAT на языке программирования Python. Библиотека NEAT-Python обеспечивает реализацию стандартных методов NEAT для моделирования генетической эволюции геномов организмов в популяции. Она содержит утилиты для преобразования генотипа организма в его фенотип (искусственную нейронную сеть) и предоставляет удобные методы для загрузки и сохранения конфигураций генома вместе с параметрами NEAT. Кроме того, она предлагает исследователям полезные подпрограммы, помогающие собирать статистику о ходе эволюционного процесса и сохранять/загружать промежуточные контрольные точки. Контрольные точки позволяют нам периодически сохранять состояние эволюционного процесса и позже возобновлять выполнение процесса из сохраненных контрольных точек.

Достоинства библиотеки NEAT-Python:

- стабильная реализация;
- всесторонняя документация;
- доступность для легкой установки при помощи менеджера пакетов PIP;
- наличие встроенных инструментов сбора статистики и поддержка сохранения контрольных точек выполнения, а также возобновление выполнения с заданной контрольной точки;
- наличие нескольких типов функций активации;
- поддержка фенотипов рекуррентных нейронных сетей непрерывного времени;
- легко расширяется для поддержки различных модификаций NEAT.

Недостатки библиотеки NEAT-Python:

- по умолчанию реализован только алгоритм NEAT;
- активная разработка библиотеки прекращена, и сейчас она находится в состоянии текущей технической поддержки и мелких доработок.

Пример использования NEAT-Python

Ниже приведен общий пример использования библиотеки NEAT-Python без прицела на какую-либо конкретную проблему. Он описывает типичные шаги, которые необходимо предпринять, чтобы получить необходимые результаты. Мы будем широко использовать эту библиотеку в данной книге. В принципе, вы можете сразу перейти к следующей главе и приступить к изучению более сложного примера применения, но лучше дочитать эту главу до конца, чтобы узнать больше об альтернативных библиотеках. Давайте начнем.

1. Загрузим настройки NEAT и начальную конфигурацию генома:

```
config = neat.Config (neat.DefaultGenome, neat.DefaultReproduction, neat.
DefaultSpeciesSet, neat.DefaultStagnation, config_file)
```

Здесь параметр `config_file` указывает на файл, который содержит настройки библиотеки NEAT-Python и конфигурацию исходного генома по умолчанию.

2. Создадим популяцию организмов из данных конфигурации:

```
p = neat.Population(config)
```

3. Добавим репóртер статистики и сборщик контрольных точек:

```
# Вывод текущего состояния в консоль.
p.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
p.add_reporter(stats)
p.add_reporter(neat.Checkpointer(5))
```

4. Запустим процесс эволюции на определенное количество поколений (в нашем случае 300):

```
winner = p.run(eval_genomes, 300)
```

Здесь `eval_genomes` – это функция, которая используется для оценки геномов всех организмов в популяции с учетом конкретной функции приспособленности, а `winner` – это лучший генотип.

5. Фенотип нейросети может быть создан из генома следующим образом:

```
winner_ann = neat.nn.FeedForwardNetwork.create(winner, config)
```

6. После этого мы можем передать в нейросеть входные данные и запросить результаты:

```
for xi in xor_inputs:
    output = winner_ann.activate(xi)
    print(xi, output) # Печать результатов.
```

Библиотеку можно скачать по адресу **https://github.com/CodeReclaimers/neat-python**.

Предыдущий пример кода должен дать вам лишь общее представление о библиотеке. Более развернутые примеры кода будут предоставлены в следующих главах.

2.1.2 Библиотека PyTorch NEAT

Эта библиотека построена как оболочка вокруг библиотеки NEAT-Python. Она обеспечивает простую интеграцию сущностей, созданных библиотекой NEAT-Python с платформой PyTorch. В результате появляется возможность преобразовать геном NEAT в фенотип нейросети, который основан на реализации рекуррентных нейронных сетей посредством PyTorch. Кроме того, это позволяет нам представлять сети CPPN в виде структур PyTorch, которые являются основными строительными блоками метода HyperNEAT. Благодаря интеграции с PyTorch мы получаем возможность использовать графические процессоры для вычислений, потенциально ускоряя эволюционный процесс благодаря повышенной скорости оценки геномов организмов в развивающейся популяции.

Достоинства библиотеки PyTorch NEAT:

- построена на основе стабильной библиотеки NEAT-Python, что позволяет нам использовать все ее преимущества;

- интеграция с платформой PyTorch;
- применение GPU ускоряет оценку геномов NEAT;
- включает в себя реализацию CPPN, которая является строительным блоком алгоритма HyperNEAT;
- интеграция со средой OpenAI GYM.

Недостатки PyTorch NEAT заключаются в следующем:

- полностью реализован только алгоритм NEAT;
- обеспечивает лишь частичную поддержку реализации алгоритма HyperNEAT.

Про OpenAI GYM можно больше узнать по адресу **https://gym.openai.com**.

Пример использования PyTorch NEAT

Ниже приведен пример использования библиотеки PyTorch NEAT для реализации контроллера балансировки обратного маятника. Пока это лишь обзорный пример. Позже в этой книге мы более подробно рассмотрим проблему балансировки маятника на тележке. Давайте начнем.

1. Загрузим настройки NEAT и конфигурацию начального генома:

```
config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
         neat.DefaultSpeciesSet, neat.DefaultStagnation, config_file)
```

Здесь файл `config_file` хранит настройки алгоритма NEAT, а также конфигурацию генома по умолчанию.

2. Создадим популяцию организмов на основе конфигурации:

```
pop = neat.Population(config)
```

3. Подготовим оценщик генома в мультисредах на основе PyTorch и OpenAI GYM:

```
def make_env():
    return gym.make("CartPole-v0")

def make_net(genome, config, bs):
    return RecurrentNet.create(genome, config, bs)

def activate_net(net, states):
    outputs = net.activate(states).numpy()
    return outputs[:, 0] > 0.5

evaluator = MultiEnvEvaluator(
    make_net, activate_net, make_env=make_env,
    max_env_steps=max_env_steps
)

def eval_genomes(genomes, config):
    for _, genome in genomes:
        genome.fitness = evaluator.eval_genome(genome, config)
```

Здесь вызов функции `gym.make("CartPole-v0")` – это обращение к фреймворку OpenAI GYM для создания среды балансировки одиночного маятника.

4. Добавим сбор статистики и ведение отчета:

```
stats = neat.StatisticsReporter()
pop.add_reporter(stats)
reporter = neat.StdOutReporter(True)
pop.add_reporter(reporter)
logger = LogReporter("neat.log", evaluator.eval_genome)
pop.add_reporter(logger)
```

5. Запустим процесс эволюции на определенное количество поколений (в нашем случае 100):

```
winner = pop.run(eval_genomes, 100)
```

Здесь `eval_genomes` – это функция для оценки геномов всех организмов в популяции по определенной функции пригодности, и победителем является наиболее эффективный найденный генотип.

6. Создадим из генома фенотип нейросети:

```
winner_ann = RecurrentNet.create(genome, config, bs)
```

Здесь `genome` – это конфигурация генома NEAT, `config` – объект, который инкапсулирует настройки NEAT, а `bs` – параметр, указывающий желаемый размер пакета.

7. Теперь мы можем передать в нейросеть входные данные и запросить результат:

```
action = winner_ann.activate(states).numpy()
```

Здесь `action` – это спецификатор действия, который будет использоваться в симуляции, а `states` – это тензор, который включает текущее состояние среды, полученное из симулятора.

Исходный код библиотеки доступен по адресу **https://github.com/uber-research/PyTorch-NEAT**.

Предыдущий исходный код должен дать вам общее представление о библиотеке. Полные примеры кода будут предоставлены в следующих главах.

2.1.3 Библиотека MultiNEAT

Библиотека MultiNEAT является самой универсальной среди библиотек, которые мы обсудим в этой книге, поскольку она поддерживает стандартный алгоритм NEAT и два важнейших расширения: HyperNEAT и ES-HyperNEAT. Кроме того, библиотека MultiNEAT обеспечивает реализацию метода оптимизации поиском новизны. Библиотека написана на языке программирования C++, но предоставляет полный интерфейс в среде Python. Установочный пакет MultiNEAT Python также доступен посредством менеджера пакетов Anaconda, который упрощает установку и использование в любой ОС.

Достоинства библиотеки MultiNEAT:

- стабильная реализация;
- реализует множество алгоритмов из семейства NEAT, таких как:
  - NEAT;
  - HyperNEAT;
  - ES-HyperNEAT;
- обеспечивает реализацию метода оптимизации поиском новизны;
- поддерживает пластичные нейронные сети Хебба;
- обеспечивает визуализацию генотипов и фенотипов при помощи OpenCV в Python;
- интеграция со средой OpenAI GYM;
- комплексная документация.

Недостатки библиотеки MultiNEAT:

- нет поддержки графического процессора;
- не поддерживает контрольные точки.

Пример использования MultiNEAT

Ниже приведен пример использования библиотеки MultiNEAT для реализации решателя XOR с использованием нейроэволюции. Это всего лишь обзорный пример без реализации оценщика приспособленности XOR (`evaluate_xor`), который будет обсуждаться в следующей главе. Давайте начнем.

1. Зададим настройки конфигурации NEAT:

```
params = NEAT.Parameters()
params.PopulationSize = 100
# Остальные настройки опущены для краткости.
```

2. Создадим минимальную конфигурацию генома и породим популяцию организмов из этого генома:

```
g = NEAT.Genome(0, 3, 0, 1, False,
      NEAT.ActivationFunction.UNSIGNED_SIGMOID,
      NEAT.ActivationFunction.UNSIGNED_SIGMOID, 0, params, 0)
pop = NEAT.Population(g, params, True, 1.0, i)
```

3. Запустим процесс эволюции на 1000 поколений или пока не будет найден победитель:

```
for generation in range(1000):
    # Оценка геномов.
    genome_list = NEAT.GetGenomeList(pop)
    fitnesses = EvaluateGenomeList_Serial(genome_list,
                            evaluate_xor, display=False)
    [genome.SetFitness(fitness) for genome, fitness in
zip(genome_list, fitnesses)]
    # Оценка значения пригодности относительно заданного порога.
    best = max(fitness_list)
    if best > 15.0:
        # Получаем фенотип наилучшего организма.
        net = NEAT.NeuralNetwork()
        pop.Species[0].GetLeader().BuildPhenotype(net)
```

```
        # Возвращаем приспособленность и фенотип нейросети победителя.
        return (best, net)

    # Следующая эпоха.
    pop.Epoch()
```

4. Ниже приводится запрос фенотипа нейросети победителя, а также входные данные:

```
net.Input( [ 1.0, 0.0, 1.0 ] )
net.Activate()
output = net.Output()
```

Библиотеку можно скачать по адресу **https://github.com/peter-ch/MultiNEAT**.

Предыдущий исходный код дает вам общее представление о библиотеке. Развернутые примеры кода будут предоставлены в следующих главах.

2.1.4 Библиотека Deep Neuroevolution

Глубокие нейронные сети (deep neural networks, DNN) демонстрируют выдающийся прирост производительности в задачах, связанных с распознаванием образов и обучением с подкреплением, используя возможности параллельной обработки современных графических процессоров. В контексте нейроэволюции особенно интересно исследовать, как обычные методы *глубокого обучения с подкреплением* (deep reinforcement learning, RL) можно сопоставить с методами, основанными на глубокой нейроэволюции. Чтобы ответить на этот вопрос, исследовательская группа из лаборатории UberAI разработала и опубликовала библиотеку Deep Neuroevolution на языке программирования Python, которая использует платформу TensorFlow для вычислений, связанных с обучением нейронной сети на устройствах с графическим процессором.

Библиотека обеспечивает реализацию простого *генетического алгоритма* (genetic algorithm, GA) и метода оптимизации поиском новизны. Она также реализует метод эволюционных стратегий, который представляет собой еще один вид эволюционного алгоритма.

Вы можете найти более подробную информацию о методе эволюционных стратегий в статье: *Hans-Georg Beyer*. The Theory of Evolution Strategies. Springer April 27, 2001.

Достоинства библиотеки Deep Neuroevolution заключаются в следующем:

- стабильная реализация;
- поддержка графического процессора через интеграцию с TensorFlow;
- способность работать напрямую с многомерными задачами, например учиться действовать прямо на уровне пикселей;
- наличие метода оптимизации поиском новизны;
- безградиентный метод для оптимизации DNN;
- обеспечивает визуализацию процесса обучения с помощью визуального инспектора для нейроэволюции (visual inspector for neuroevolution, VINE);

- обеспечивает интеграцию со средой OpenAI GYM;
- обеспечивает интеграцию со средой игр Atari.

Недостаток библиотеки Deep Neuroevolution заключается в том, что она не обеспечивает реализацию алгоритмов нейроэволюции семейства NEAT, то есть NEAT, HyperNEAT и ES-HyperNEAT.

Генетический алгоритм, который реализован в библиотеке Deep Neuroevolution, контролирует эволюцию популяции организмов с геномами, кодирующими вектор параметров обучения (веса соединений) для глубокой нейронной сети. В каждом поколении каждый генотип оценивается и дает оценку пригодности. После этого определенное количество организмов выбирается случайным образом из числа наиболее подходящих людей, чтобы стать родителями следующего поколения. Генотип каждого выбранного родительского организма затем мутирует путем добавления гауссова шума. Кроме того, в алгоритме используется понятие элитарности, при котором конкретное количество организмов, наиболее подходящих по форме, из предыдущего поколения добавляется к следующему без каких-либо изменений. Оператор кроссовера не применяется во время эволюционного процесса для упрощения алгоритма. Топология DNN, используемая в этом алгоритме, фиксирована и установлена экспериментаторами вручную.

Рассмотрим следующий простой генетический алгоритм, представленный в общей форме на рис. 2.1.

Алгоритм 5: Простой генетический алгоритм

Параметры: оператор мутации ψ, размер популяции N, количество выбранных организмов T, подпрограмма инициализации φ, функция приспособленности F.

Выход : Наилучшие организмы популяции (*Elite*) через G поколений.

```
begin
    for g ← 1, 2 ..., G поколений do
        for i ← 1, ..., N в популяции следующего поколения do
            if g = 1 then
                // Инициализация случайной ДНК с гауссианом
                P_i^{g=1} ← φ(N(0, I))
            else
                // Выбор родителей для следующего поколения
                k ← uniformRandom(1, T)
            // Вычисление приспособленности организма
            F_i ← F(P_i^g)
        Сортировка P_i^g в порядке убывания F_i
        // Находим лучших кандидатов (elite) для следующего поколения
        if g = 1 then
            // Берем десять верхних организмов из списка популяции
            // (список отсортирован по убыванию приспособленности)
            C ← P_{1...10}^{g-1}
        else
            // Получаем первые девять организмов популяции, объединенных
            // с текущим элитным организмом (Elite)
            C ← P_{1...9}^g ∪ Elite
        /* Отобранные элитные кандидаты C затем оцениваются на 30 дополнительных
           эпизодах, чтобы окончательно подтвердить их элитарность */
```

$Elite \leftarrow \arg\max_{\Theta \in C} \frac{1}{30} \sum_{j=1}^{30} F(\Theta)$
// Копируем $Elite$ в популяцию следующего поколения
$\mathcal{P}^g \leftarrow Elite \cup (\mathcal{P}^g - Elite)$ // Добавляем $Elite$ только один раз
Return : $Elite$

Рис. 2.1. Пример простого генетического алгоритма в общей форме

Больше информации о реализации библиотеки Deep Neuroevolution можно найти по адресу **https://github.com/uber-research/deep-neuroevolution**.

2.1.5 Сравнение библиотек Python, реализующих нейроэволюцию

В табл. 2.1 приведена сравнительная информация о библиотеках Python, которые мы обсуждали в этой главе.

Таблица 2.1 Сравнение библиотек Python, реализующих алгоритмы нейроэволюции

	NEAT-Python	PyTorch NEAT	MultiNEAT	Deep Neuroevolution
NEAT	Да	Да	Да	Нет
HyperNEAT	Нет	Частично (только CPPN)	Да	Нет
ES-HyperNEAT	Нет	Нет	Да	Нет
Поиск новизны	Нет	Нет	Да	Да
OpenAI GYM	Нет	Да	Да	Да
Визуализация	Нет	Нет	Да	Да
Поддержка GPU	Нет	Да	Нет	Да
PIP	Да	Нет	Нет	Нет
Anaconda	Нет	Нет	Да	Нет
Контрольные точки	Да	Да	Нет	Да

Библиотека NEAT-Python обеспечивает отличную интеграцию визуализации и проста в использовании. Однако ее существенный недостаток в том, что она реализована исключительно на языке Python и, как следствие, имеет очень низкую скорость выполнения. Подходит только для простых задач.

Библиотека MultiNEAT Python имеет ядро, реализованное на C ++, что дает ей немного лучшую производительность по сравнению с библиотекой NEAT-Python. Она может быть использована для решения более сложных задач, требующих создания более крупных фенотипов нейросети. Кроме того, она обеспечивает реализацию методов HyperNEAT и ES-HyperNEAT, что делает ее правильным выбором для задач, связанных с обучением крупномасштабных нейросетей.

Библиотека Deep Neuroevolution является самой продвинутой реализацией нейроэволюции и позволяет нам использовать возможности графических процессоров для выполнения задач машинного обучения с миллионами обучаемых параметров. Ей можно найти хорошее применение в области обработки визуальных данных.

Позже в этой книге мы познакомимся с каждой из упомянутых библиотек Python поближе и применим их на практике.

2.2 Настройка среды

При работе с библиотеками Python важно правильно настроить рабочую среду. Существует много зависимостей, включая версию на языке Python и двоичные файлы, доступные в системе; все они должны быть собраны в одном месте и иметь совместимые версии. В процессе подбора и установки зависимостей легко могут возникнуть конфликтующие конфигурации библиотек и языковых версий, что огорчит исследователя и заставит его потратить много часов на поиск и исправление ошибок. Чтобы решить эту проблему, в языке программирования Python была введена концепция виртуальной среды. *Виртуальная среда* позволяет нам создавать изолированные среды Python, которые содержат все необходимые зависимости и исполняемые файлы, которые используются в конкретном проекте Python. Такая виртуальная среда может быть легко создана и удалена после того, как она больше не нужна, не оставляя никаких следов в системе.

Среди наиболее популярных инструментов для работы с виртуальными средами Python можно выделить следующие:

- Pipenv;
- Virtualenv;
- Anaconda.

2.2.1 Pipenv

Pipenv – это инструмент, который объединяет менеджер пакетов с менеджером виртуальных сред. Основная цель – упростить для разработчиков настройку уникальной рабочей среды для конкретного проекта со всеми необходимыми зависимостями.

Pipenv можно установить с помощью PIP (установщик пакета для Python), введя в терминале следующую команду:

```
$ pip install --user pipenv
```

Эта команда устанавливает инструмент `pipenv` в пространство пользователя, чтобы не допустить вмешательства в любые общесистемные пакеты.

Чтобы установить все зависимости и создать новую виртуальную среду (если она отсутствует) для вашего проекта, перейдите в каталог проекта и запустите процесс установки следующим образом:

```
$ cd my_project_folder
$ pipenv install <package>
```

Эта команда создает новую виртуальную среду в `my_project_folder` и устанавливает в нее пакет `<package>`. Вот и все.

Вы можете предоставить файл конфигурации (Pipfile), который указывает установщику, какие пакеты должны быть установлены, а также другую информацию, относящуюся к процессу построения среды. Когда вы запускаете `install` в первый раз, файл Pipfile будет создан автоматически, если он еще не существует.

Больше информации об этом инструменте можно найти по адресу **https://pipenv.kennethreitz.org/en/latest/**.

2.2.2 Virtualenv

Virtualenv – это инструмент, который используется для создания изолированных сред Python, начиная с Python v3.3, и частично интегрирован в стандартную библиотеку в модуле `venv`. Основная проблема, которая решается с помощью этого инструмента, заключается в поддержании уникального набора зависимостей, версий и разрешений для каждого проекта Python по отдельности. Virtualenv решает эту задачу, создавая отдельную среду с собственными установочными каталогами для каждого проекта. Это не позволяет нам смешивать какие-либо зависимости и библиотеки с другими проектами. Также можно заблокировать доступ к глобально установленным библиотекам.

Virtualenv – это чистый менеджер виртуальных сред, который не предоставляет никаких процедур менеджера пакетов. Поэтому для управления зависимостями проекта он обычно нуждается в менеджере пакетов, таком как PIP. Давайте коротко продемонстрируем работу с Virtualenv.

Установите Virtualenv при помощи менеджера пакетов PIP:

```
$ pip install virtualenv
```

1. Убедитесь, что установка прошла успешно:

```
$ virtualenv --version
```

2. Создайте виртуальную среду для вашего проекта с помощью следующих команд:

```
$ cd my_project_folder
$ virtualenv venv
```

Эти команды создают новую виртуальную среду в каталоге вашего проекта `my_project_folder`. Свежая среда включает в себя папку с исполняемыми файлами Python внутри нее, а также копию библиотеки PIP, которая является менеджером пакетов, позволяющим устанавливать другие зависимости.

3. Перед началом использования виртуальную среду необходимо активировать с помощью следующей команды, которую необходимо ввести в окне терминала:

```
$ source /path/to/ENV/bin/activate
```

После выполнения этой команды все необходимые переменные среды будут установлены в правильные значения, характерные для вашего проекта, и текущий сеанс терминала будет использовать их для любых последующих введенных команд.

4. С помощью PIP в активную среду можно легко установить дополнительные пакеты:

```
$ pip install sqlite
```

Данная команда устанавливает пакет SQLite в текущей активной среде.

Если после команды `pip install` не указано имя пакета, менеджер PIP будет искать в текущем каталоге файл `requirements.txt` для определения перечня устанавливаемых пакетов.

Дополнительная информация доступна по адресу **https://virtualenv.pypa.io/en/latest/**.

2.2.3 Anaconda Distribution

Anaconda Distribution – это установочный пакет и менеджер виртуальной среды, который популярен среди специалистов по обработке данных и машинному обучению, поскольку он обеспечивает легкий доступ к обширной коллекции специализированных научных библиотек (более 1500) и полезных инструментов. Кроме того, он позволяет вам писать исходный код и выполнять скрипты на Python и R из одного места. С помощью Anaconda можно легко создавать, сохранять, загружать и переключаться между виртуальными средами, а также устанавливать из хранилища в каждую виртуальную среду тысячи пакетов, которые были проверены и подготовлены группой Anaconda.

Для установки Anaconda необходимо скачать установочный файл, соответствующий вашей операционной системе, по адресу **https://www.anaconda.com/distribution/**.

После установки Anaconda вы можете создать новую среду для вашего проекта с помощью следующих команд:

```
$ cd my_project_folder
$ conda create --name ENV_NAME <package>
```

Эти команды создают новую виртуальную среду для вашего проекта и устанавливают в нее указанный пакет или несколько пакетов. Дополнительные пакеты могут быть легко установлены в новую среду позже, после ее активации.

Список всех доступных в системе сред можно получить с помощью следующей команды:

```
$ conda env list
```

Любая существующая среда может быть активирована следующим образом:

```
$ conda activate ENV_NAME
```

Чтобы деактивировать текущую активную среду, используйте команду

```
$ conda deactivate
```

Дополнительные библиотеки могут быть установлены в текущей среде либо через стандартный менеджер PIP, либо с помощью команды `conda install`:

```
$ conda install sqlite
```

После выполнения предыдущей команды в текущую активную среду будет установлен пакет SQLite.

В этой книге мы будем использовать Anaconda для управления зависимостями и средами для большинства наших проектов.

Если вы хотите больше узнать о командах Anaconda, обратитесь по адресу **https://docs.anaconda.com/anaconda-cloud/commandreference/**.

2.3 Заключение

В этой главе вы узнали о четырех популярных библиотеках Python, которые можно использовать для экспериментов в области нейроэволюции. Мы обсудили сильные и слабые стороны каждой представленной библиотеки и рассмотрели обзорные примеры использования этих библиотек в Python. После этого рассмотрели, как настроить рабочую среду для экспериментов в окружении Python, чтобы избежать побочных эффектов наличия нескольких версий одной и той же библиотеки в пути Python. Мы обнаружили, что лучший способ сделать это – создать изолированные виртуальные среды для каждого проекта Python, и разобрали несколько популярных решений этой задачи, созданных сообществом открытого исходного кода. Наконец, вы познакомились с инструментом Anaconda Distribution, который включает, помимо прочего, менеджер пакетов и менеджер среды. В оставшейся части этой книги мы будем использовать Anaconda для правильной настройки среды в наших экспериментах.

В следующей главе мы обсудим, как использовать алгоритм NEAT для решения классической проблемы информатики. Вы создадите решатель проблемы XOR, используя библиотеку NEAT-Python, которую мы обсуждали в этой главе. Мы также обсудим гиперпараметры, которые используются для настройки алгоритма NEAT, и способы их подбора для повышения производительности процесса нейроэволюции.

Часть II

Применение методов нейроэволюции для решения классических задач информатики

В этой части книги речь пойдет о том, как применять нейроэволюционные алгоритмы для решения классических проблем информатики. Вы изучите основные приемы и навыки, необходимые для использования нейроэволюционных алгоритмов при решении классических задач информатики, а также подготовитесь к работе с более продвинутыми методами, описанными в третьей части книги.

Эта часть состоит из следующих глав:

- главы 3 «Использование NEAT для оптимизации решения задачи XOR»;
- главы 4 «Эксперименты по балансировке маятника»;
- главы 5 «Автономное прохождение лабиринта»;
- главы 6 «Метод оптимизации поиском новизны».

Глава 3

Использование NEAT для оптимизации решения задачи XOR

В этой главе вы узнаете об одном из классических компьютерных экспериментов, который демонстрирует, что алгоритм NEAT работоспособен и может создать правильную топологию сети. Вы узнаете из первых рук о написании целевой функции для решения задачи XOR. Также узнаете, как выбрать правильные гиперпараметры алгоритма NEAT, оптимальные для решения задачи XOR. Цель этой главы – познакомить вас с основными приемами применения алгоритма NEAT для решения классических задач информатики.

После завершения эксперимента и выполнения упражнений, описанных в этой главе, вы получите четкое представление об особенностях эксперимента XOR и получите практические навыки, необходимые для написания соответствующего исходного кода Python с использованием библиотеки NEAT-Python. Вы также получите опыт настройки гиперпараметров библиотеки NEAT-Python и использования специальных утилит для визуализации результатов эксперимента. После этого вы будете готовы приступить к экспериментам с более сложными задачами, о которых узнаете немного позже.

В этой главе мы рассмотрим следующие темы:

- суть задачи XOR;
- как определить целевую функцию для решателя задачи XOR;
- выбор гиперпараметров для эксперимента XOR;
- запуск эксперимента XOR.

3.1 Технические требования

Для проведения экспериментов, описанных в этой главе, ваш компьютер должен удовлетворять следующим техническим требованиям:

- Windows 8/10, macOS 10.13 или новее, или современный Linux;
- Anaconda Distribution версия 2019.03 или новее.

Исходный код примеров этой главы можно найти в каталоге `Chapter3` в файловом архиве книги.

3.2 Суть задачи XOR

Классический *многослойный персептрон* (multilayer perceptron, MLP) или *искусственная нейронная сеть* (artificial neural network, ANN), не обладающие *скрытыми узлами* топологии, способны правильно решать только задачи с линейно разделяемым пространством решений. В результате такие конфигурации не могут использоваться в качестве шаблона решения задач распознавания или управления. Однако, используя более сложные архитектуры MLP, включающие в себя скрытые элементы с некоторой нелинейной функцией активации (например, сигмоидальной), можно аппроксимировать любую функцию с заданной точностью. Поэтому мы можем взять нелинейно разделяемую задачу и проверить, способен ли процесс нейроэволюции вырастить произвольное количество скрытых элементов в фенотипе нейросети решателя задачи.

Решатель задачи XOR – это классический компьютерный эксперимент в области обучения с подкреплением, который не может быть решен без введения нелинейного этапа в алгоритм решателя. Пространство поиска решения задачи имеет минимальный размер и подходит для демонстрации того, что алгоритм NEAT способен развивать топологию нейросети, начиная с очень простой и постепенно увеличивая сложность, чтобы найти подходящую структуру сети, в которой все связи организованы надлежащим образом. Демонстрируя способность алгоритма NEAT последовательно наращивать соответствующую топологию, эксперимент XOR также демонстрирует, что NEAT может избежать локальных максимумов ландшафта приспособленности. *Локальный максимум* – это ловушка, в которой решатель может застрять, создав локального лидера с неправильной схемой связей. После этого локальный чемпион может доминировать над популяцией настолько, что решатель не сможет решить задачу.

Таблица истинности функции XOR выглядит следующим образом:

Таблица 2.1. Таблица истинности функции XOR (исключающее ИЛИ)

Вход 1	Вход 2	Выход
1	1	0
1	0	1
0	1	1
0	0	0

XOR (*исключающее ИЛИ*) – это двоичный логический оператор, который возвращает логическую единицу (истину), если только на одном из входов присутствует единица. Для получения правильного выходного сигнала два входных сигнала должны быть объединены нелинейным скрытым элементом. Не существует линейной функции для комбинации входов XOR, которая могла бы правильно разделить их на классы.

Алгоритм NEAT начинается с начальной популяции, которая кодирует очень простой фенотип, и постепенно развивает топологию фенотипа, пока не будет создана структура успешной нейросети. Начальная структура фенотипа нейросети не содержит никаких скрытых узлов и состоит из двух входных узлов, одного выходного узла и одного узла смещения. Два входных узла и узел смещения связаны с выходным узлом, то есть исходный генотип имеет три гена связи и четыре гена узла. Узел смещения – это особый тип ввода, который всегда инициализируется определенным значением, большим 0 (обычно это 1,0 или 0,5). Узел смещения необходим, если мы хотим привести активацию нейрона (выходного или скрытого) – которая рассчитывается соответствующей функцией активации, применяемой к сумме входов и смещения, – к определенному ненулевому значению, если оба входа имеют значение 0.

Начальный и оптимальный (наименьший возможный) фенотипы решателя XOR показаны на рис. 2.1.

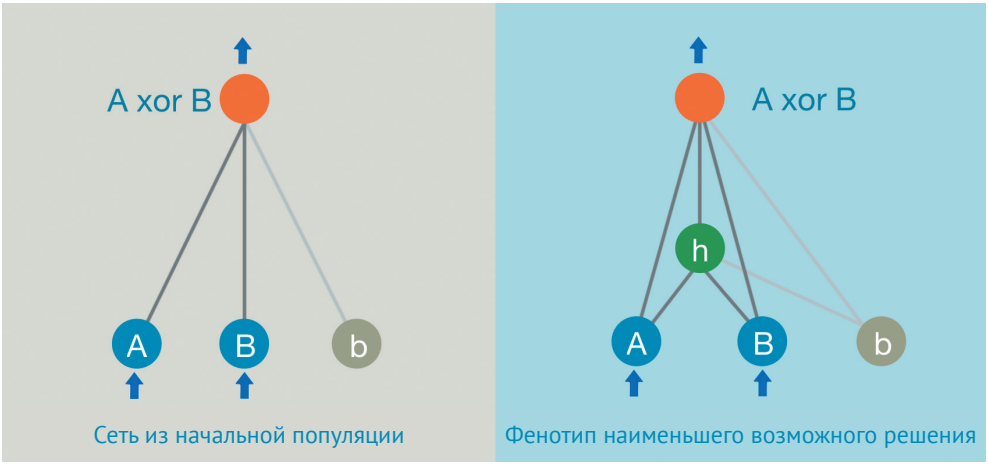


Рис. 3.1. Начальный и оптимальный фенотипы решателя XOR

Фенотип становится все сложнее, пока не будет найдено окончательное решение путем добавления одного или нескольких дополнительных скрытых узлов. Наименьший возможный решатель включает в себя только один скрытый узел, а метод NEAT демонстрирует свою мощь, находя оптимальную конфигурацию решателя среди более сложных.

3.3 Целевая функция для эксперимента XOR

В эксперименте XOR приспособленность организма в популяции определяется как квадрат расстояния между правильным ответом и суммой выходных данных, которые генерируются для всех четырех входных состояний XOR. Она рассчитывается следующим образом:

1. Фенотип нейросети активируется всеми четырьмя входными состояниями XOR.
2. Выходные значения вычитаются из правильных ответов для каждого состояния, а затем суммируются абсолютные значения результатов.

3. Значение ошибки, найденное на предыдущем шаге, вычитается из максимального значения приспособленности (4) для расчета приспособленности текущего организма. Наивысшая приспособленность означает наилучшую точность решателя.
4. Затем рассчитанная приспособленность возводится в квадрат, чтобы пропорционально повысить приспособленность организмов, тем самым получая решающие нейросети, которые дают более близкие к правильному ответу решения. Такой подход делает эволюционное давление более интенсивным.

Исходя из сказанного, целевую функцию можно определить следующим образом:

$$f = \left(4 - \sum_{i=1}^{4} \left| y_i - ANN\left(x1_i, x2_i\right) \right| \right)^2.$$

Соответствующий исходный код, основанный на библиотеке NEAT-Python, выглядит так:

```
# Входы XOR и ожидаемые выходные значения.
xor_inputs = [(0.0, 0.0), (0.0, 1.0), (1.0, 0.0), (1.0, 1.0)]
xor_outputs = [ (0.0,), (1.0,), (1.0,), (0.0,)]

def eval_fitness(net):
    error_sum = 0.0
    for xi, xo in zip(xor_inputs, xor_outputs):
        output = net.activate(xi)
        error_sum += abs(output[0] - xo[0])
    # Вычисляем усиленную приспособленность.
    fitness = (4 - error_sum) ** 2
    return fitness
```

Обратите внимание, что нет необходимости нормализовать значение приспособленности для соответствия диапазону [0,1] (как это происходит с методами на основе обратного распространения), потому что в процессе обучения не используются вычисления обратного градиента. Показатели приспособленности организмов сравниваются напрямую на основе их абсолютных значений. Таким образом, диапазон значений не важен.

Вы можете попробовать собственные варианты расчета баллов приспособленности. Например, вы можете реализовать функцию, напоминающую среднеквадратичную ошибку, и сравнить точность алгоритма для различных реализаций целевой функции. Единственное требование состоит в том, что целевая функция должна давать более высокие оценки приспособленности для лучших решателей.

3.4 Настройка гиперпараметров

Эксперимент XOR, который мы обсудим в этой главе, в качестве фреймворка использует библиотеку NEAT-Python. Библиотека NEAT-Python использует набор гиперпараметров, которые влияют на выполнение и точность алгоритма NEAT. Файл конфигурации хранится в формате, аналогичном файлам .ini

Windows, – каждая секция начинается с имени в квадратных скобках [секция], за которым следуют пары ключ–значение, разделенные знаком равенства (=).

В этом разделе мы обсудим некоторые гиперпараметры библиотеки NEAT-Python, которые можно найти в разделах файла конфигурации.

Полный перечень гиперпараметров библиотеки NEAT-Python можно найти по адресу **https://neat-python.readthedocs.io/en/latest/config_file.html**.

3.4.1 Секция NEAT

В этой секции указываются параметры, специфичные для алгоритма NEAT. Раздел включает в себя следующие параметры:

- `fitness_criterion` – функция, которая вычисляет критерий завершения из набора значений приспособленности всех геномов в популяции. Значения параметров – это имена стандартных агрегатных функций, таких как `min`, `max` и `mean`. Значения `min` и `max` используются для завершения процесса эволюции, если минимальная или максимальная приспособленность популяции превышает заданный порог `fitness_threshold`. Когда значение задано как `mean`, в качестве критерия завершения используется средняя приспособленность популяции;
- `fitness_threshold` – пороговое значение, которое сравнивается с приспособленностью, вычисленной с помощью функции `fitness_criterion` для проверки необходимости завершения эволюции;
- `no_fitness_termination` – флаг, который запрещает завершение эволюционного процесса на основе приспособленности, определенной предыдущими параметрами. Если установлено значение `True`, эволюция будет прекращена только после того, как будет оценено максимальное количество поколений;
- `pop_size` – количество отдельных организмов в каждом поколении;
- `reset_on_extinction` – флаг, который определяет, следует ли создавать новую случайную популяцию, если все виды в текущем поколении вымерли из-за стагнации. Если установлено значение `False`, после полного исчезновения популяции будет выброшен флаг `CompleteExtinctionException`.

3.4.2 Секция DefaultStagnation

Эта секция определяет параметры, которые являются специфическими для подпрограмм стагнации видов, реализованных в классе `DefaultStagnation`. Секция включает в себя следующие параметры:

- `species_fitness_func` – вид функции, которая используется для вычисления приспособленности видов, то есть для расчета совокупного значения приспособленности всех организмов, принадлежащих к определенному виду. Допустимые значения: `max`, `min` и `mean`;
- `max_stagnation` – виды, которые не показали улучшение значения приспособленности, рассчитываемого функцией `species_fitness_func` в течение более чем `max_stagnation` числа поколений, считаются застойными и подвержены вымиранию;

- `species_elitism` – количество элитных видов, которые безоговорочно защищены от стагнации. Они предназначены для предотвращения полного вымирания популяции до появления новых видов. Указанное количество видов с наивысшей приспособленностью всегда выживает в популяции, несмотря на то что дальнейшее улучшение приспособленности не наблюдается.

3.4.3 Секция DefaultReproduction

В этой секции представлена конфигурация для подпрограмм воспроизведения, которые реализованы встроенным классом `DefaultReproduction`. Секция включает в себя следующие параметры:

- `elitism` – количество наиболее приспособленных организмов каждого вида, которые копируются без изменений в следующее поколение. Этот фактор позволяет нам сохранить любые полезные мутации, которые были обнаружены в предыдущих поколениях;
- `survival_threshold` – доля организмов каждого вида, которые могут быть родителями следующего поколения, то есть имеют право на половое размножение (кроссовер). Регулируя это значение, можно определить самый низкий показатель приспособленности организма, который позволяет ему участвовать в процессе воспроизводства. Это возможно благодаря тому, что фракция `survivial_threshold` берется из отсортированного списка организмов, упорядоченных по приспособленности в порядке убывания;
- `min_species_size` – минимальное количество организмов каждого вида, которое необходимо сохранить после цикла размножения.

3.4.4 Секция DefaultSpeciesSet

В этой секции представлена конфигурация процесса видообразования, который реализован встроенным классом `DefaultSpeciesSet` и включает в себя следующий единственный параметр:

- `compatibility_threshold` – пороговое значение, от которого зависит, принадлежат ли организмы к одному и тому же виду (геномное расстояние меньше этого значения) или другому виду. Более высокие значения порога уменьшают склонность эволюционного процесса к видообразованию.

3.4.5 Секция DefaultGenome

В этой секции определяются параметры конфигурации, которые задействованы в создании и поддержке генома, реализованных классом `DefaultGenome`. Секция включает в себя следующие параметры:

- `activ_default` – имя функции активации для использования в генах узлов;
- `activation_mutate_rate` – если геном поддерживает несколько функций активации (например, для генома CPPN), то этот параметр определяет вероятность того, что мутация заменит функцию активации текущего узла новой, полученной из списка поддерживаемых функций (см. `activation_options`);

- activation_options – разделенный пробелами список функций активации, которые могут использоваться генами узла;
- aggregation_default – имя агрегатной функции по умолчанию, которая будет использоваться сетевым узлом для любых агрегированных входных сигналов, полученных от других узлов до активации;
- aggregation_mutate_rate – если геном поддерживает несколько агрегатных функций, то этот параметр определяет вероятность мутации, которая заменяет агрегатную функцию текущего узла новой из списка агрегатных функций (см. aggregation_options);
- aggregation_options – разделенный пробелами список агрегатных функций, которые могут использоваться генами узлов. Поддерживаемые значения: sum, min, max, mean, median и maxabs;
- compatibility_threshold – пороговое значение для принятия решения о том, принадлежат ли организмы одному и тому же виду (геномное расстояние меньше этого значения) или разным видам. Более высокие значения порога означают, что эволюционный процесс обладает меньшей склонностью к видообразованию;
- compatibility_disjoint_coefficient – коэффициент, который используется при вычислении геномного расстояния для выяснения того, как непересекающиеся или избыточные гены влияют на результат вычисления. Более высокие значения этого параметра усиливают значимость присутствия непересекающихся или избыточных генов при расчете геномного расстояния;
- compatibility_weight_coefficient – коэффициент, от которого зависит то, как вычисленные значения разницы между атрибутами смещения и отклика генов узлов и весовыми атрибутами генов связей влияют на результаты;
- conn_add_prob – вероятность мутации, которая вводит новый ген связи между существующими генами узлов;
- conn_delete_prob – вероятность мутации, которая удаляет существующий ген связи из генома;
- enabled_default – значение по умолчанию для атрибута состояния (включено/выключено) вновь созданных генов связей;
- enabled_mutate_rate – вероятность мутации, которая переключает атрибут состояния гена связи;
- feed_forward – управляет типом фенотипических сетей, которые будут генерироваться во время генеза. Если установлено значение True, то повторяющиеся соединения не допускаются;
- initial_connection – указывает начальный шаблон связей для вновь созданных геномов. Допустимые значения включают unconnected, fs_neat_nohidden, fs_neat_hidden, full_direct, full_nodirect, partial_direct и partial_nodirect;
- node_add_prob – вероятность мутации, которая добавляет новый ген узла;
- node_delete_prob – вероятность мутации, которая удаляет из генома существующий ген узла и все связи с ним;
- num_hidden, num_inputs, num_outputs – количество скрытых, входных и выходных узлов в геномах исходной популяции;

- `single_structural_mutation` – если установлено значение `True`, то в процессе эволюции разрешены только структурные мутации, то есть лишь добавление или удаление узлов либо связей.

3.4.6 Гиперпараметры эксперимента XOR

Эксперимент XOR начинается с очень простой начальной конфигурации генома, которая имеет только два входных узла, один выходной узел и один специальный вход – узел смещения. В исходном геноме скрытый узел отсутствует:

```
[DefaultGenome]
# Параметры сети
num_hidden = 0
num_inputs = 2
num_outputs = 1

# Параметры узла смещения
bias_init_mean = 0.0
bias_init_stdev = 1.0
```

Функция активации всех узлов сети является сигмоидальной, а входы узлов агрегируются функцией суммы (`sum`):

```
[DefaultGenome]
# Параметры активации узла
activation_default = sigmoid

# Параметры агрегации узла
aggregation_default = sum
```

Тип закодированной сети – полносвязная сеть прямого распространения:

```
[DefaultGenome]
feed_forward = True
initial_connection = full_direct
```

В ходе эволюции новые сетевые узлы и связи добавляются и/или удаляются с определенной вероятностью:

```
[DefaultGenome]
# Вероятность добавления/удаления узла
node_add_prob = 0.2
node_delete_prob = 0.2

# Вероятность добавления/удаления связи
conn_add_prob = 0.5
conn_delete_prob = 0.5
```

Все связи включены по умолчанию, с очень низкой вероятностью отключения из-за мутации:

```
[DefaultGenome]
# Параметры состояния связи
enabled_default = True
enabled_mutate_rate = 0.01
```

Чтобы стимулировать разнообразие видов, мы зададим сильное влияние избыточных/непересекающихся частей родительских геномов на расстояние между геномами:

```
[DefaultGenome]
# Параметры расстояния между геномами
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient = 0.5
```

Стагнация видов может длиться до 20 поколений, а уникальные виды частично защищены от вымирания:

```
[DefaultStagnation]
species_fitness_func = max
max_stagnation = 20
species_elitism = 2
```

Порог выживания организмов в пределах вида установлен достаточно низким, чтобы сузить эволюционный процесс, позволяя размножаться только наиболее приспособленным организмам (верхние 20 % списка организмов, упорядоченные по приспособленности). В то же время вводится элитарность, чтобы безоговорочно копировать двух наиболее приспособленных особей в следующее поколение каждого вида. Минимальный размер вида также влияет на видообразование, и мы оставляем для него значение по умолчанию:

```
[DefaultReproduction]
elitism = 2
survival_threshold = 0.2
min_species_size = 2
```

Порог совместимости видов определяет разнообразие видов в популяции. Более высокие значения этого параметра приводят к более разнообразной популяции. Разнообразие видов должно быть сбалансировано, чтобы поддерживать эволюционный процесс в желаемом направлении, избегая исследования слишком большого количества векторов поиска, но в то же время позволяя исследовать новинки:

```
[DefaultSpeciesSet]
compatibility_threshold = 3.0
```

Численность популяции установлена в 150, что довольно умеренно, но достаточно для такой простой проблемы, как XOR. Критерий завершения (`fitness_threshold`) установлен на `15.5`, чтобы гарантировать, что эволюция завершается, когда найденное решение максимально близко к цели (в соответствии с нашей функцией `fitness` максимальная оценка приспособленности составляет `16.0`).

В этой задаче мы заинтересованы в поиске лидера эволюции, способного решить проблему XOR, поэтому наша функция завершения (`fitness_criterion`) – это функция `max`, которая выбирает максимальную приспособленность среди всех организмов в популяции:

```
[NEAT]
fitness_criterion = max
```

```
fitness_threshold = 15.5
pop_size = 150
reset_on_extinction = False
```

Полный конфигурационный файл `xor_config.ini` можно найти в файловом архиве книги.

Мы представили только основные гиперпараметры, которые сильно влияют на производительность алгоритма NEAT. Значения гиперпараметров были протестированы при создании работающего решателя XOR, но вы можете изменить их по своему усмотрению и посмотреть, что произойдет.

3.5 Выполнение эксперимента XOR

Прежде чем мы начнем работать над экспериментом XOR, нам нужно правильно настроить нашу среду Python в соответствии с требованиями библиотеки NEAT-Python, которую мы выбрали в качестве основы для написания нашего кода. Библиотека NEAT-Python доступна из PyPI, поэтому мы можем использовать команду `pip` для ее установки в виртуальную среду эксперимента XOR.

3.5.1 Настройка среды

Прежде чем мы начнем писать код, относящийся к эксперименту XOR, необходимо создать соответствующую среду Python, и в нее должны быть установлены все зависимости. Выполните следующие шаги, чтобы правильно настроить рабочую среду.

1. Виртуальная среда Python 3.5 для эксперимента XOR создается с помощью команды `conda` из Anaconda Distribution:

```
$ conda create --name XOR_neat python=3.5
```

Убедитесь, что пакет Anaconda Distribution установлен на ваш компьютер, как сказано в главе 2.

2. Чтобы использовать вновь созданную виртуальную среду, вы должны активировать ее:

```
$ conda activate XOR_neat
```

3. После этого библиотеку NEAT-Python можно установить в активную среду с помощью следующей команды:

```
$ pip install neat-python == 0.92
```

Мы используем версию 0.92, которая была самой новой на момент работы над книгой.

4. Наконец, нам нужно установить дополнительные зависимости, которые используются утилитами визуализации. Это можно сделать с помощью команды `conda`:

```
$ conda install matplotlib
$ conda install graphviz
$ conda install python-graphviz
```

Теперь мы готовы к написанию исходного кода.

3.5.2 Исходный код эксперимента XOR

Чтобы начать эксперимент, нам нужно создать каталог с именем `Chapter3`, используя команду `mkdir` (для Linux и macOS) или `md` (для Windows):

```
$ mkdir Chapter3
```

В этом каталоге будут храниться все файлы исходных кодов, имеющих отношение к эксперименту, упомянутому в главе 3.

Затем нам нужно скопировать файл `xor_config.ini` из файлового архива исходного кода, связанного с этой главой, во вновь созданный каталог. Этот файл содержит полную конфигурацию гиперпараметров для эксперимента XOR, которую мы обсуждали ранее. Эксперименты, которые будут рассмотрены в данной книге, используют различные утилиты для визуализации результатов, чтобы наглядно продемонстрировать внутренние аспекты процесса нейроэволюции. Эксперимент XOR также зависит от конкретных утилит визуализации, которые реализованы в файле `visualize.py` из файлового архива этой книги. Вам также необходимо скопировать этот файл в каталог `Chapter3`.

Установочный пакет Anaconda Distribution включает в себя бесплатный кросс-платформенный редактор кода VS Code. Он достаточно прост с точки зрения функциональности, но обеспечивает отличную поддержку Python и позволяет легко переключаться между виртуальными средами. Вы можете использовать его для написания исходного кода для экспериментов, описанных в этой книге.

Наконец, создайте файл под названием `xor_experiment.py` в каталоге `Chapter3` и используйте ваш любимый редактор исходного кода Python для написания кода.

1. Сначала определим импорт библиотек, которые понадобятся позже:

```
# Импорт стандартных библиотек Python.
import os
# Импорт библиотек NEAT-Python.
import neat
# Импорт утилиты для визуализации результатов.
import visualize
```

2. Далее напишем код оценки приспособленности в соответствии с тем, что мы говорили ранее:

```
# Входы и ожидаемые выходы XOR для оценки приспособленности.
xor_inputs = [(0.0, 0.0), (0.0, 1.0), (1.0, 0.0), (1.0, 1.0)]
xor_outputs = [ (0.0,), (1.0,), (1.0,), (0.0,)]
```

```
def eval_fitness(net):
    """
    Оценивает приспособленность генома, который был использован
    для создания предоставленной сети
    Аргументы:
        net: нейросеть прямого распространения, созданная из генома
    Возвращает:
        Оценка приспособленности - чем выше балл, тем лучше
        приспособленность организма. Максимальная оценка: 16.0
    """
    error_sum = 0.0
    for xi, xo in zip(xor_inputs, xor_outputs):
        output = net.activate(xi)
        error_sum += abs(xo[0] - output[0])
    # Вычисление усиленного значения приспособленности
    fitness = (4 - error_sum) ** 2
    return fitness
```

Никогда не упускайте возможность оставлять в исходном коде комментарии, которые описывают назначение функции, ее входные параметры и результаты выполнения. Также полезно комментировать некоторые интересные или необычные части исходного кода, чтобы их было проще понять человеку, который увидит код позже (это можете оказаться вы!).

3. С помощью функции оценки приспособленности вы можете написать функцию для оценки всех организмов в текущем поколении и соответственно обновить приспособленность каждого генома:

```
def eval_genomes(genomes, config):
    """
    Функция для оценки приспособленности каждого генома в списке геномов.
    Предоставленная конфигурация используется для создания нейронной
    сети прямого распространения из каждого генома, и после этого нейронная
    сеть оценивается по ее способности решать задачу XOR.
    В результате выполнения этой функции показатель приспособленности
    каждого генома обновляется до новой оценки.
    Аргументы:
        genomes: Список геномов из популяции в текущем поколении

        config: Файл конфигурации, содержащий гиперпараметры
    """
    for genome_id, genome in genomes:
    genome.fitness = 4.0
    net = neat.nn.FeedForwardNetwork.create(genome, config)
    genome.fitness = eval_fitness(net)
```

4. Теперь, когда реализована функция оценки приспособленности отдельного генома и определена целевая функция, пришло время реализовать функцию для запуска эксперимента. Функция `run_experiment` загружает

конфигурацию гиперпараметра из файла конфигурации и создает начальную популяцию генома:

```
# Загрузка конфигурации.
config = neat.Config(neat.DefaultGenome,
         neat.DefaultReproduction, neat.DefaultSpeciesSet,
         neat.DefaultStagnation, config_file)


# Создает популяцию, которая является объектом верхнего уровня.
p = neat.Population(config)
```

5. Нам пригодится сбор статистики для оценки эксперимента и наблюдения за процессом в реальном времени. Также важно сохранить контрольные точки, что позволяет восстановить выполнение с заданной контрольной точки в случае сбоя. Два типа репóртеров (стандартный вывод и сборщик статистики) и сборщик контрольных точек могут быть зарегистрированы следующим образом:

```
# Добавление репортера stdout для отображения хода процесса в терминале.
p.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
p.add_reporter(stats)
p.add_reporter(neat.Checkpointer(5,
               filename_prefix='out/neat-checkpoint-'))
```

6. Теперь мы готовы запустить нейроэволюцию в течение `300` поколений, предоставив в качестве аргумента функцию `eval_genome`, которая служит для оценки показателей приспособленности каждого генома в популяции каждого поколения, пока не будет найдено решение или процесс не достигнет максимального числа поколений:

```
# Выполнение в течение 300 поколений.
best_genome = p.run(eval_genomes, 300)
```

7. Когда выполнение алгоритма NEAT останавливается по достижении успеха или максимального числа поколений, возвращается наиболее подходящий геном. Можно проверить, является ли этот геном победителем, то есть способен ли решить задачу XOR с заданной точностью:

```
# Проверяем, является ли лучший геном адекватным решателем задачи XOR
best_genome_fitness = eval_fitness(net)
if best_genome_fitness > config.fitness_threshold:
    print("\n\nSUCCESS: The XOR problem solver found!!!")
else:
    print("\n\nFAILURE: Failed to find XOR problem solver!!!")
```

8. Наконец, собранные статистические данные и наиболее подходящий геном можно визуализировать, чтобы изучить результаты процесса нейроэволюции и увидеть, как геном развивался от нуля до максимального числа поколений:

```
# Визуализация результатов эксперимента
node_names = {-1:'A', -2: 'B', 0:'A XOR B'}
visualize.draw_net(config, best_genome, True,
```

```
    node_names=node_names, directory=out_dir)
visualize.plot_stats(stats, ylog=False, view=True,
    filename=os.path.join(out_dir, 'avg_fitness.svg'))
visualize.plot_species(stats, view=True,
    filename=os.path.join(out_dir, 'speciation.svg'))
```

Полный исходный код эксперимента XOR можно найти в файле `xor_experiment.py` в файловом архиве книги.

Для визуализации графов и статистики, полученных в результате выполнения кода эксперимента, будет использоваться библиотека Matplotlib. Также будет представлен сетевой граф наиболее подходящего генома.

3.5.3 Запуск эксперимента и анализ результатов

Для запуска эксперимента, находясь в каталоге `Chapter3`, введите в окне терминала следующую команду:

```
$ python xor_experiment.py
```

Не забудьте активировать виртуальную среду `XOR_neat` с помощью команды `$ conda activate XOR_neat`. В противном случае будут возникать ошибки, связанные с отсутствием пакета `neat`.

Вы можете использовать любое приложение терминала командной строки на свое усмотрение. После ввода команды алгоритм NEAT начинает выполнение, и окно терминала выводит промежуточные результаты в режиме реального времени. Для каждого поколения вывод выглядит следующим образом:

```
****** Running generation 43 ******
Population's average fitness: 6.01675 stdev: 2.53269
Best fitness: 14.54383 - size: (4, 7) - species 2 - id 5368
Average adjusted fitness: 0.238
Mean genetic distance 2.482, standard deviation 0.991
Population of 151 members in 5 species:
   ID  age size fitness adj fit stag
  ==== === ==== ======= ======= ====
     1  43   28     9.0   0.241    0
     2  33   42    14.5   0.274    7
     3  20   39     9.0   0.306    0
     4   4   34     9.0   0.221    0
     5   1    8     8.4   0.149    0
Total extinctions: 0
Generation time: 0.045 sec (0.038 average)
```

Средняя приспособленность популяции в поколении `43` составляет всего `6.01675`, что довольно мало по сравнению с критерием завершения, установленным в файле конфигурации (`fitness_threshold=15.5`). Тем не менее похоже, что у нас есть потенциальный вид-чемпион (`ID:2`), который находится на пути к достижению целевого порога приспособленности путем развития организма-чемпиона с показателем приспособленности `14.54383`, который

кодирует фенотип нейросети, состоящий из четырех узлов и семи соединений (размер (4,7)).

Популяция включает 151 особь и разделена на пять видов со следующими свойствами:

- id – идентификатор вида;
- age – возраст видов как количество поколений от их создания до настоящего времени;
- size – количество особей, принадлежащих к данному виду;
- fitness – показатель приспособленности вида, рассчитанный по его особям (в нашем случае max);
- adj fit – приспособленность определенного вида, которая была скорректирована с учетом показателей приспособленности всей популяции;
- stag – возраст стагнации конкретного вида как число поколений с момента последнего улучшения приспособленности вида.

Когда с помощью алгоритма NEAT найден подходящий решатель задачи XOR, в окне терминала отображается результат. Он начинается с общей статистики об окончательной популяции генома и победителе (успешном решателе XOR):

```
****** Running generation 44 ******

Population's average fitness: 6.04705 stdev: 2.67702
Best fitness: 15.74620 - size: (3, 7) - species 2 - id 6531

Best individual in generation 44 meets fitness threshold - complexity: (3, 7)
```

Из предыдущего вывода мы можем заключить, что в поколении 44 процесс эволюции создает геном, который кодирует фенотип нейросети, способный решить проблему XOR с заданной точностью. Этот геном принадлежит организму из вида с ID:2, и этот вид уже отвечал за эволюционный процесс в течение последних семи поколений. Организм-чемпион (ID:6531) поколения 44 является мутацией организма (ID:5368) в виде с ID:2 из предыдущего поколения, который потерял один скрытый узел и теперь имеет три узла с семью связями (size: (3, 7)).

Затем следует секция лучшего генома:

```
Best genome:
Key: 6531
Fitness: 15.74619841601669
Nodes:
    0 DefaultNodeGene(key=0, bias=-3.175506745721987, response=1.0,
activation=sigmoid, aggregation=sum)
    224 DefaultNodeGene(key=224, bias=-2.5796785460461154, response=1.0,
activation=sigmoid, aggregation=sum)
    612 DefaultNodeGene(key=612, bias=-1.626648521448398, response=1.0,
activation=sigmoid, aggregation=sum)
Connections:
    DefaultConnectionGene(key=(-2, 224), weight=1.9454770276940339,
```

```
enabled=True)
    DefaultConnectionGene(key=(-2, 612), weight=2.1447044917213383,
enabled=True)
    DefaultConnectionGene(key=(-1, 0), weight=-2.048078253002224,
enabled=True)
    DefaultConnectionGene(key=(-1, 224), weight=3.6675667680178328,
enabled=True)
    DefaultConnectionGene(key=(224, 0), weight=6.1133731818187655,
enabled=True)
    DefaultConnectionGene(key=(612, 0), weight=-2.1334321035742474,
enabled=True)
    DefaultConnectionGene(key=(612, 224), weight=1.5435290073038443,
enabled=True)
```

Секция вывода лучшего генома представляет статистику точности чемпиона популяции, а также конфигурацию его генома. Входные узлы имеют идентификаторы `-1` и `-2` и не показаны, потому что они относительно просты, и всего лишь предоставляют нам способ для ввода значений в сетевой граф. Выходной узел и два скрытых узла имеют идентификаторы `0`, `224` и `612` соответственно. Кроме того, `DefaultNodeGene` содержит значения смещения, имя функции активации и имя функции, которая используется для агрегирования входных данных на каждом узле. Гены связей (`DefaultConnectionGene`), которые будут представлены позже, предоставляют идентификаторы исходного и целевого узлов вместе с соответствующим весом соединения.

Наконец, давайте взглянем на раздел `Output`:

```
Output:
input (0.0, 0.0), expected output (0.0,), got [1.268084297765355e-07]
input (0.0, 1.0), expected output (1.0,), got [0.9855287279878023]
input (1.0, 0.0), expected output (1.0,), got [0.9867962503269723]
input (1.0, 1.0), expected output (0.0,), got [0.004176868376596405]
```

В секции `Output` представлены выходные значения, которые выданы нейросетью, построенной по фенотипу лидера популяции при получении четырех пар входных данных. Как мы видим, результат близок к ожидаемым значениям в пределах указанной точности.

Каталог `Output` также содержит схему графа нейросети успешного решателя XOR, которая выглядит следующим образом (рис. 3.2):

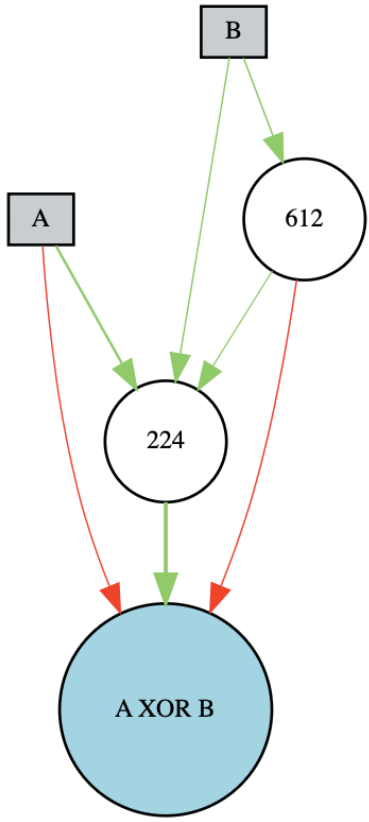


Рис. 3.2. Граф нейросети успешного решателя XOR

Нейросеть фенотипа-победителя близка к оптимальной конфигурации, которую мы описали ранее, но у нее есть еще один скрытый узел (`ID:612`). Узел смещения не показан на графе, поскольку библиотека NEAT-Python не выделяет смещение в отдельный узел; вместо этого она присваивает значение смещения каждому сетевому узлу в качестве атрибута, который можно увидеть в списке вывода (каждый `DefaultNodeGene` имеет атрибут смещения). График изменения приспособленности по поколениям эволюции также сохраняется в каталог `Output` (рис. 3.3).

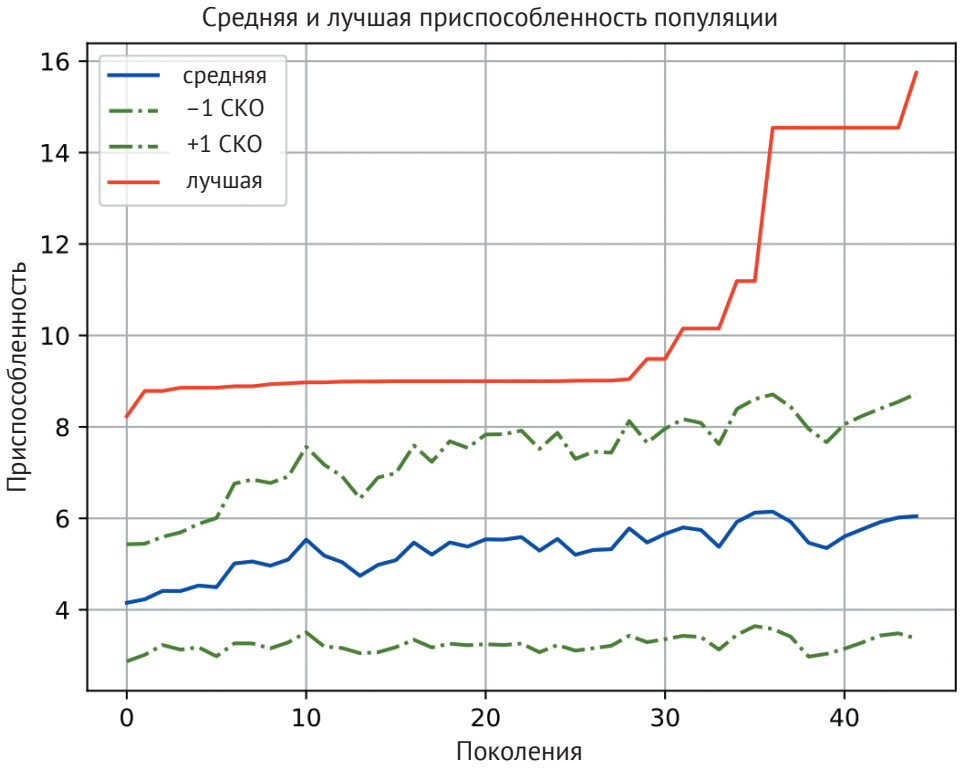


Рис. 3.3. График изменения приспособленности по поколениям эволюции

Этот график визуализирует изменения лучших и средних показателей приспособленности популяции за поколения эволюции. Средняя приспособленность популяции увеличилась незначительно. Тем не менее благодаря функции видообразования, которая была введена в алгоритм NEAT, некоторые виды продемонстрировали выдающуюся эффективность с самых ранних поколений (#10), и благодаря сохранению полезной мутации им наконец удалось создать организм-чемпион, который решает задачу XOR с заданной точностью.

Выходной каталог также содержит график видообразования, показанный на рис. 3.4.

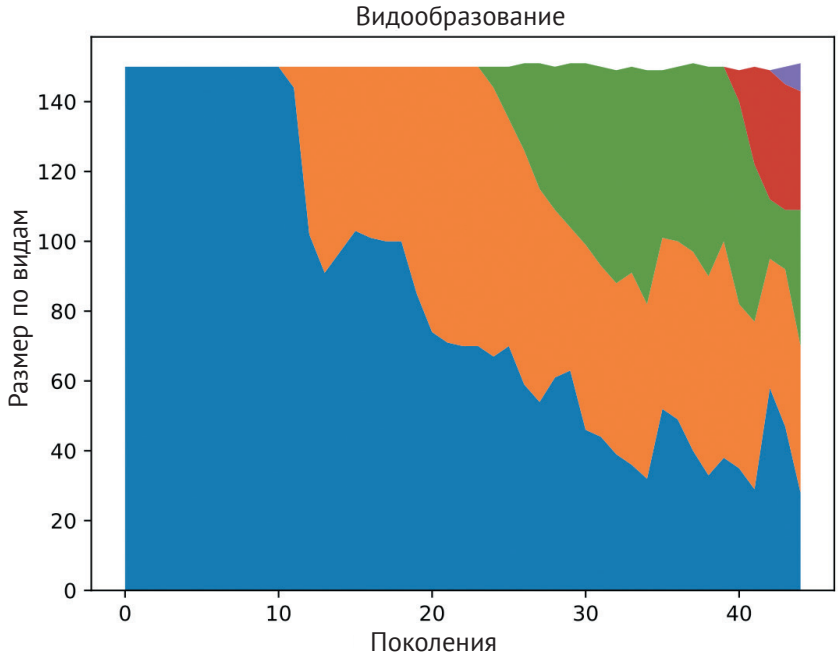


Рис. 3.4. График видообразования по поколениям эволюции

График видообразования демонстрирует, как на протяжении поколений популяции организмов развивался процесс видообразования. Каждый отдельный вид отмечен своим цветом. Эволюция началась с одного вида (`ID:1`), который включает в себя всю популяцию. Затем около 10-го поколения зародился второй вид (`ID:2`), и в конечном итоге он произвел организм-чемпион. Кроме того, на более поздних этапах эволюции популяция разветвлялась на еще три вида в поколениях `23`, `39` и `42`.

3.6 Упражнения

Теперь, когда у нас есть исходный код нейроэволюционного решателя XOR, попробуйте поэкспериментировать, изменив гиперпараметры NEAT, влияющие на эволюционный процесс.

Одним из параметров, представляющих особый интерес, является `compatibility_threshold`, который можно найти в разделе файла конфигурации `DefaultSpeciesSet`:

- попробуйте увеличить значение этого параметра и следите за видообразованием популяции. Сравните производительность алгоритма с новым значением и значением по умолчанию (`3.0`). Она становится лучше?
- что произойдет, если вы уменьшите значение этого параметра? Сравните новую производительность со значением по умолчанию.

Другим важным параметром, который управляет эволюционным процессом, является `min_species_size`, который можно найти в разделе `DefaultReproduction`. Изменяя значения этого параметра, вы можете напрямую контролировать минимальное количество особей на вид и неявно контролировать разнообразие видов:

1. Установите значение параметра `compatibility_threshold` в значение по умолчанию (`3.0`) и попытайтесь увеличить значение параметра `min_species_size` в диапазоне `[2, 8]`. Сравните производительность алгоритма со значением по умолчанию. Посмотрите, как разнообразие видов меняется от поколения к поколению. Просмотрите выходные данные алгоритма и проверьте, не случалась ли стагнация каких-либо видов и удаление вида из-за превышения возраста стагнации.
2. Установите чрезвычайно высокое значение параметра `min_species_size` для нашей популяции (`32`) и найдите взрыв видового разнообразия в конце процесса эволюции на графике видообразования. Почему это происходит? Изучите граф, изображающий конфигурацию фенотипа нейросети в файле `Digraph.gv.svg`. Является ли этот граф оптимальным?

Увеличение минимального размера видов делает эволюционный процесс более сложным и позволяет ему сохранять более полезные мутации. В результате у нас увеличивается вероятность получения оптимального генома, который кодирует нейросеть фенотипа минимального решателя XOR.

Граф нейросети минимального решателя XOR показан на рис. 3.5.

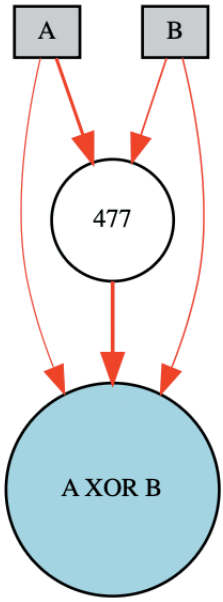


Рис. 3.5. Граф нейросети минимального решателя XOR

Как мы уже упоминали, нейросеть минимального решателя XOR имеет только один скрытый узел, как показано на рис. 3.5. Попробуйте написать модифицированный код для решения тройной задачи XOR (A xor B xor C).

Можно ли решить эту проблему с помощью тех же гиперпараметров, которые мы использовали в эксперименте, описанном в этой главе?

3.7 Заключение

В этой главе мы представили классическую задачу информатики, связанную с созданием оптимального решателя XOR. Мы обсудили основы задачи XOR и продемонстрировали ее важность в качестве первого эксперимента с нейроэволюцией – она позволяет проверить, может ли алгоритм NEAT развить более сложную топологию нейросети, начиная с самой простой конфигурации. Затем мы определили целевую функцию для оптимального решателя XOR и рассмотрели подробное описание гиперпараметров NEAT. После этого использовали библиотеку NEAT-Python для написания исходного кода решателя XOR с применением определенной целевой функции и провели эксперименты.

Результаты проведенного нами эксперимента позволили нам сделать вывод о взаимосвязи между количеством видов в популяции, минимальным размером каждого вида и производительностью алгоритма, а также полученными топологиями нейросетей.

В следующей главе вы узнаете о классических экспериментах по обучению с подкреплением, которые часто используются в качестве эталонов для реализации стратегии управления. Вы узнаете, как писать точные симуляции реальных физических устройств и как использовать такие симуляции при определении целевых функций для алгоритма NEAT. Вы получите собственный опыт написания стратегий управления для различных контроллеров балансировки обратного маятника на тележке с использованием библиотеки NEAT-Python.

Глава 4

Балансировка тележки с обратным маятником

В этой главе вы узнаете о классическом эксперименте по обучению с подкреплением, который также является общепринятым эталоном для тестирования различных реализаций стратегий управления. Мы рассмотрим три модификации эксперимента по балансировке тележки с обратным маятником и разработаем стратегии управления, которые можно использовать для стабилизации аппаратов с обратным маятником заданных конфигураций. Вы узнаете, как писать точные симуляции реальных физических систем и как использовать их при определении целевой функции для алгоритма NEAT. После прочтения этой главы вы будете готовы применить алгоритм NEAT для реализации контроллеров, способных напрямую управлять физическими устройствами.

В этой главе мы рассмотрим следующие темы:

- задача балансировки обратного маятника в обучении с подкреплением;
- реализация симулятора устройства с тележкой в Python;
- определение целевой функции балансировочного контроллера с помощью симулятора;
- особенности задачи балансировки двух маятников;
- реализация симулятора тележки с двумя маятниками в Python;
- определение целевой функции для контроллера балансировки двух маятников.

4.1 Технические требования

Для проведения экспериментов, описанных в этой главе, ваш компьютер должен удовлетворять следующим техническим требованиям:

Windows 8/10, macOS 10.13 или новее, или современный Linux;

Anaconda Distribution версия 2019.03 или новее.

Исходный код примеров этой главы можно найти в каталоге `Chapter4` в файловом архиве книги.

4.2 Задача балансировки обратного маятника

Обратный маятник представляет собой неустойчивую механическую систему, центр масс которой находится выше точки вращения. Его можно стабилизировать, прикладывая внешние силы под управлением специализированной системы, которая контролирует угол маятника и перемещает точку вращения горизонтально назад и вперед под центром масс, когда маятник начинает падать. *Балансировщик обратного маятника* – это классическая задача в теории динамики и управления, которая используется в качестве эталона для тестирования стратегий управления, в том числе основанных на методах обучения с подкреплением. Мы особенно заинтересованы в реализации специального алгоритма управления, основанного на нейроэволюции, для стабилизации обратного маятника в течение заданного периода времени.

Эксперимент, описанный в этой главе, предусматривает моделирование обратного маятника, реализованного в виде тележки, которая может перемещаться горизонтально, с установленными на ней сверху шарниром и маятником в виде вертикального стержня[1]. Конструкция системы показана на рис. 4.1.

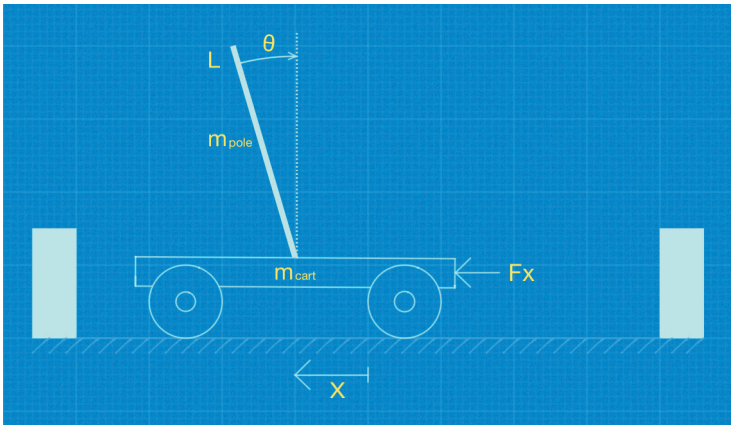


Рис. 4.1. Система из тележки и обратного стержневого маятника

Прежде чем мы начнем писать исходный код симулятора, нам нужно определить *уравнение движения*, которое можно использовать для оценки значений переменных состояния балансировщика маятника в любой момент времени.

4.2.1 Уравнения движения балансировщика

Задача контроллера состоит в том, чтобы приложить к центру масс тележки последовательность сил F_x таким образом, чтобы маятник уравновешивался в течение определенного (или бесконечного) промежутка времени, а тележка оставалась в пределах дорожки, то есть не ударялась о левую или правую стену. Принимая во внимание упомянутую механику, мы можем квалифицировать задачу балансировки маятника как проблему *управляемого избегания*, потому

[1] Такой маятник принято называть обратным, потому что его опорный шарнир расположен снизу. – *Прим. перев.*

что состояние системы тележка–маятник должно поддерживаться таким образом, чтобы избегать определенных областей пространства состояний. Для достижения соответствующего состояния не существует единственного решения, поэтому приемлемым является любое решение уравнений движения, позволяющее избежать определенных областей.

Для обучения контроллера балансировки маятника алгоритм обучения должен получать из среды минимальное количество знаний о задаче. Такие знания должны отражать близость нашего контроллера к цели. Задача уравновешивания маятника состоит в том, чтобы стабилизировать нестабильную систему и удерживать контроль над ней как можно дольше. Таким образом, подкрепляющий сигнал r_t, полученный из среды моделирования, должен отражать возникновение отказа (потери контроля). Отказ может быть вызван отклонением маятника больше допустимого угла или столкновением тележки с одной из стен. Подкрепляющий сигнал r_t можно определить следующим образом:

$$r_t = \begin{cases} 0 \text{ если} -0,21 < \theta_t < 0,21 \text{ радиан; и} -2,4 < x_t < 2,4 \text{ м} \\ -1 \text{ в остальных случаях} \end{cases}.$$

В этом уравнении θ_t – угол между маятником и вертикалью, положительный по часовой стрелке; x_t – горизонтальная позиция тележки относительно центра дорожки.

Обратите внимание, что подкрепляющий сигнал r_t не зависит ни от угловой скорости маятника $\dot{\theta}$, ни от горизонтальной скорости тележки $\dot{x}$. Он лишь предоставляет информацию о том, находится ли динамическая система тележка–маятник в заранее установленных пределах.

Уравнения движения системы тележка–маятник без учета трения выглядят следующим образом:

$$\ddot{\Theta} = \frac{g \sin\theta + \cos\theta \left(\frac{-F_x - m_p L \dot{\theta}^2 \sin\theta}{m_c + m_p} \right)}{L \left(\frac{4}{3} - \frac{m_p \cos^2 \theta_t}{m_c + m_p} \right)}.$$

Здесь $\dot{\theta}$ – угловая скорость маятника, $\ddot{\theta}$ – угловое ускорение маятника, $\dot{x}$ – горизонтальная скорость тележки, $\ddot{x}$ – ускорение тележки вдоль оси x.

В нашем эксперименте мы будем использовать следующие параметры системы:

- m_c = 1,0 кг – масса тележки;
- m_p = 0,1 кг – масса маятника;
- L = 0,5 м – расстояние от центра масс маятника до шарнира;
- g = 9,8 м/с^2 – ускорение свободного падения.

4.2.2 Уравнения состояния и управляющие воздействия

Экспериментальная система тележка–маятник моделируется путем численной аппроксимации уравнений движения с использованием метода Эйлера с шагом τ = 0,02 с. Уравнения состояния можно определить следующим образом:

$$x_{t+\tau} = x_t + \tau \dot{x}_t,$$

$$\dot{x}_{t+\tau} = \dot{x}_t + \tau \ddot{x}_t;$$

$$\theta_{t+\tau} = \theta_t + \tau \dot{\theta}_t,$$

$$\dot{\theta}_{t+\tau} = \dot{\theta}_t + \tau \ddot{\theta}_t.$$

Для небольшого диапазона углов маятника, который применяется в нашем эксперименте, мы можем использовать линейную аппроксимацию поверхности, которая делит пространство всех возможных состояний системы, требующих различных действий (поверхность переключения). Таким образом, пространство действия состоит из левого и правого толкающих действий. Контроллер тележки, который мы используем в нашем эксперименте, не предназначен для создания нулевой силы. Вместо этого на каждом шаге по времени *t* он прикладывает силу к центру масс тележки с одинаковой амплитудой, но в одном из двух противоположных направлений. Такая система управления называется *двухпозиционным регулятором* (bang-bang controller) и может быть определена с помощью следующего уравнения:

$$F_x = \begin{cases} 10\text{H}, & \text{если } a[t] = 1 \\ -10\text{H}, & \text{если } a[t] = 0 \end{cases}.$$

Здесь $a[t]$ – сигнал действия, полученный от решателя. Исходя из значения этого сигнала, двухпозиционный регулятор прикладывает силу F_x с одинаковой амплитудой (10 Н), но в одном из двух противоположных направлений, в зависимости от того, какое действие выбрал решатель.

4.2.3 Взаимодействие между решателем и симулятором

В каждый данный момент времени *t* решатель получает масштабированные значения упомянутых ранее переменных состояния. Эти значения служат входными данными для нейросети, созданной на основе фенотипа, и определяются следующим образом:

$$x_0[t] = 1.0,$$

$$x_1[t] = \frac{1}{4.8}\left(x[t] + 2.4\right),$$

$$x_2[t] = \frac{1}{3}\left(\dot{x}[t] + 1.5\right),$$

$$x_3[t] = \frac{1}{0.42}\left(\theta[t] + 0.21\right),$$

$$x_4[t] = \frac{1}{4}\left(\dot{\theta}[t] + 2\right).$$

В первом уравнении x_0 – это постоянное смещение, а $x_1 \ldots x_4$ относятся к горизонтальной позиции тележки, горизонтальной скорости, углу отклонения маятника от вертикали и угловой скорости соответственно.

Принимая во внимание установленные ранее ограничения системы (см. r_t), масштабированные значения x_1 и x_3 гарантированно лежат в интервале [0, 1],

в то время как масштабированные значения x_2 и x_4 в основном попадают в интервал [0, 1], но, в конце концов, могут выйти за его пределы. Переменные состояния масштабируются для достижения двух основных целей:

- устранить предвзятость в обучении, которая может возникнуть, когда члены с доминирующе большим размахом значений оказывают более значительное влияние на ученика из-за эффектов округления;
- поскольку значения переменных состояния сосредоточены вокруг нуля, для этой конкретной задачи можно найти нейросеть решателя, которая не нуждается в скрытых узлах. Тем не менее мы заинтересованы в развитии топологии нейронных сетей с помощью алгоритма NEAT. Введенная схема масштабирования гарантирует, что процесс нейроэволюции в конечном итоге приводит к фенотипам, которые кодируют скрытые узлы.

Контроллер балансировки маятника принимает масштабированные входные значения и выдает выходной сигнал, являющийся двоичным значением, определяющим действие, которое должно быть применено в момент времени t, как обсуждалось ранее. Частота дискретизации переменных состояния системы тележка–маятник и частота, с которой прикладывается управляющее усилие, такие же, как частота симуляции $1/\tau = 50$ Гц.

Таким образом, начальная конфигурация нейросети контроллера может быть представлена в виде схемы на рис. 4.2.

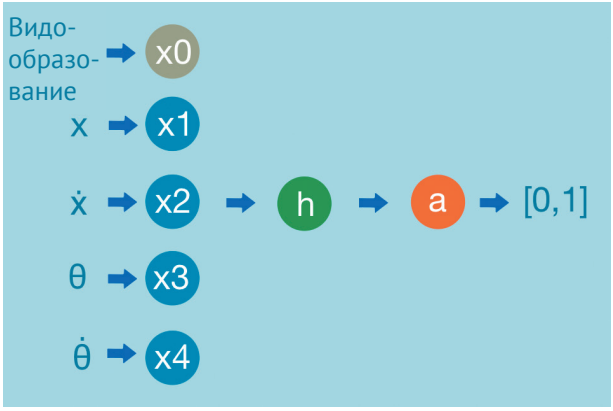


Рис. 4.2. Начальная конфигурация нейросети балансировщика обратного маятника

Начальная конфигурация нейросети балансировщика одиночного обратного маятника включает в себя пять входных узлов: для горизонтального положения тележки (**x1**) и его скорости (**x2**), для вертикального угла маятника (**x3**) и его угловой скорости (**x4**) и дополнительный входной узел для смещения (**x0**) (*которое может быть необязательным в зависимости от конкретной используемой библиотеки NEAT*). Выходной узел (**a**) является двоичным узлом, выдающим управляющий сигнал [0 или 1]. Скрытый узел (***h***) является необязательным и может быть пропущен.

4.3 Целевая функция для эксперимента по балансировке одиночного маятника

Наша цель – создать контроллер балансировки одиночного обратного маятника, который сможет поддерживать систему в стабильном состоянии в рамках определенных ограничений как можно дольше, но, по крайней мере, в течение ожидаемого количества временных шагов, указанных в конфигурации эксперимента (500 000). Таким образом, целевая функция должна оптимизировать продолжительность стабильной балансировки маятника и может быть определена как логарифмическая разница между ожидаемым количеством шагов и фактическим количеством шагов, полученных во время оценки фенотипа нейросети. Функция ошибки задается следующим образом:

$$\mathcal{L} = \frac{\log t_{max} - \log t_{eval}}{\log t_{max}}.$$

В данном эксперименте t_{max} – это ожидаемое количество шагов от конфигурации эксперимента, а t_{eval} – фактическое количество шагов, в течение которых контроллер смог поддерживать стабильное состояние балансировщика в допустимых пределах (см. определение сигнала подкрепления r_t для информации о допустимых пределах).

4.3.1 Моделирование тележки

Приведенное выше определение целевой функции предполагает, что мы можем измерить количество шагов, в течение которых балансировщик маятника находился в стабильном состоянии. Чтобы выполнить такие измерения, нам нужно реализовать симулятор системы с тележкой, используя уравнения движения и ограничения, определенные ранее.

Исходный код примеров этой главы можно найти в каталоге `Chapter4` файлового архива книги.

Во-первых, нам нужно создать в рабочем каталоге файл с именем `cart_pole.py`. Этот файл содержит исходный код уравнений движения и функцию для оценки приспособленности балансировщика одиночного маятника.

1. Начнем с определения констант, описывающих физику системы с тележкой:

```
GRAVITY = 9.8 # m/s^2
MASSCART = 1.0 # kg
MASSPOLE = 0.5 # kg
TOTAL_MASS = (MASSPOLE + MASSCART)
# Расстояние от центра масс маятника до шарнира
# (обычно половина длины стержневого маятника).
LENGTH = 0.5 # m
POLEMASS_LENGTH = (MASSPOLE * LENGTH) # kg * m
FORCE_MAG = 10.0 # N
FOURTHIRDS = 4.0/3.0
# Количество секунд между обновлениями состояния.
TAU = 0.02 # sec
```

2. После этого мы готовы реализовать уравнения движения, используя следующие константы:

```
force = -FORCE_MAG if action <= 0 else FORCE_MAG
cos_theta = math.cos(theta)
sin_theta = math.sin(theta)
temp = (force + POLEMASS_LENGTH * theta_dot * theta_dot * \
        sin_theta) / TOTAL_MASS
# Угловое ускорение маятника.
theta_acc = (GRAVITY * sin_theta - cos_theta * temp) /\
            (LENGTH * (FOURTHIRDS - MASSPOLE * \
             cos_theta * cos_theta / TOTAL_MASS))
# Линейное ускорение тележки.
x_acc = temp - POLEMASS_LENGTH * theta_acc * \
        cos_theta / TOTAL_MASS
# Обновление четырех переменных состояния по методу Эйлера.
x_ret = x + TAU * x_dot
x_dot_ret = x_dot + TAU * x_acc
theta_ret = theta + TAU * theta_dot
theta_dot_ret = theta_dot + TAU * theta_acc
```

Чтобы разобраться с назначением констант, изучите реализацию функции `do_step(action, x, x_dot, theta, theta_dot)` в исходном коде примера этой главы.

Приведенный выше фрагмент кода в качестве входных данных использует текущее состояние системы (`x`, `x_dot`, `theta`, `theta_dot`) вместе с управляющим действием и применяет уравнения движения, описанные ранее, для обновления состояния системы перед следующим шагом. Обновленное состояние системы затем возвращается, чтобы обновить симулятор и проверить нарушения ограничений. В целом получается цикл моделирования, организованный в соответствии с описанием в следующем разделе.

4.3.2 Цикл моделирования

Итак, мы полностью реализовали уравнения движения и числовую аппроксимацию переменных состояния для одного шага моделирования устройства с тележкой. После этого мы готовы начать реализацию полного цикла моделирования, в котором используется нейросеть контроллера, чтобы оценить текущее состояние системы и выбрать соответствующее действие (усилие, которое будет приложено к тележке) для следующего шага. Упомянутая ранее нейросеть создается для каждого генома популяции на определенном поколении эволюции, что позволяет нам оценивать эффективность всех геномов.

Изучите реализацию функции `run_cart_pole_simulation(net, max_bal_steps,random_start=True)` для более глубокого понимания работы алгоритма.

Полный цикл моделирования может состоять из следующих шагов:

1. Во-первых, нам нужно инициализировать переменные начального состояния либо нулевыми значениями, либо случайными значениями в рамках ограничений, описанных ранее и сосредоточенных вокруг нуля. Случайные значения состояния могут быть созданы следующим образом:

```
# -1.4 < x < 1.4
x = (random.random() * 4.8 - 2.4) / 2.0
# -0.375 < x_dot < 0.375
x_dot = (random.random() * 3 - 1.5) / 4.0
# -0.105 < theta < 0.105
theta = (random.random() * 0.42 - 0.21) / 2.0
# -0.5 < theta_dot < 0.5
theta_dot = (random.random() * 4 - 2) / 4.0
```

Мы намеренно сократили диапазон всех значений по сравнению с соответствующими масштабированными ограничениями, чтобы убедиться, что алгоритм не запускается в критическом состоянии, то есть когда стабилизация изначально невозможна.

2. После этого мы готовы начать цикл моделирования через определенное количество шагов, которые задаются параметром `max_bal_steps`. Следующий код выполняется *внутри цикла моделирования*.
3. Переменные состояния необходимо масштабировать, чтобы они соответствовали диапазону [0,1], прежде чем загружать их в качестве входных данных в нейросеть контроллера. Эта процедура дает вычислительные и эволюционные преимущества, о чем мы говорили ранее. Здесь мы не указываем значение смещения в явном виде, поскольку фреймворк NEAT-Python обрабатывает его внутренне, поэтому входные данные нейросети можно определить следующим образом:

```
input[0] = (x + 2.4) / 4.8
input[1] = (x_dot + 1.5) / 3
input[2] = (theta + 0.21) / .42
input[3] = (theta_dot + 2.0) / 4.0
```

4. Затем масштабированные входные данные можно использовать для активации нейросети, а выходные данные нейросети применяются для получения дискретного значения действия:

```
# Активация NET
output = net.activate(input)
# Получаем значение, определяющее действие.
action = 0 if output[0] < 0.5 else 1
```

5. Получив значение действия и текущие значения переменных состояния, вы можете запустить один шаг моделирования тележки с маятником. После шага моделирования возвращенные переменные состояния проверяются на соответствие ограничениям, чтобы проверить, находится ли состояние системы в допустимых границах.

В случае обнаружения неудачи возвращается текущее количество шагов моделирования, и это значение будет использоваться для оценки приспособленности фенотипа:

```
# Применяем действие к симулятору тележки
x, x_dot, theta, theta_dot = do_step(action = action,
                  x = x, x_dot = x_dot,
                  theta = theta, theta_dot = theta_dot )

# Проверяем на соответствие допустимым границам.
# В случае выхода за границы возвращаем число шагов.
if x < -2.4 or x > 2.4 or theta < -0.21 or theta > 0.21:
    return steps
```

Если нейросеть контроллера смогла поддерживать стабильное состояние балансировки системы тележка–маятник для всех шагов моделирования, функция `run_cart_pole_simulation` возвращает значение с максимальным количеством шагов моделирования.

4.3.3 Оценка приспособленности генома

Используя количество успешных шагов моделирования, возвращенных функцией `run_cart_pole_simulation`, описанной ранее, мы готовы реализовать функцию оценки приспособленности генома.

1. Сначала мы запускаем цикл симуляции тележки с маятником, который возвращает количество успешных шагов симуляции:

```
steps = run_cart_pole_simulation(net, max_bal_steps)
```

2. После этого мы готовы оценить приспособленность конкретного генома, как описано ранее:

```
log_steps = math.log(steps)
log_max_steps = math.log(max_bal_steps)
# Значение ошибки в интервале [0, 1]
error = (log_max_steps - log_steps) / log_max_steps
# Приспособленность вычисляется как разность единицы и ошибки
fitness = 1.0 - error
```

Для углубленного понимания работы кода изучите реализацию функции `eval_fitness(net, max_bal_steps=500000)`.

Мы используем логарифмическую шкалу, потому что большинство прогонов симуляции дают сбой примерно за 100 шагов, но мы проверяем 500 000 шагов балансировки.

4.4 Эксперимент по балансировке одиночного маятника

Теперь, когда у нас есть целевая функция, определенная и реализованная вместе с симулятором динамической системы тележка–маятник, мы готовы начать писать исходный код для запуска нейроэволюционного процесса с помощью

алгоритма NEAT. Мы будем использовать ту же библиотеку NEAT-Python, что и в эксперименте XOR в предыдущей главе, но с соответствующим образом настроенными гиперпараметрами NEAT. Гиперпараметры хранятся в файле `single_pole_config.ini`, который можно найти в каталоге исходного кода, относящегося к этой главе. Вам нужно скопировать этот файл в локальный каталог `Chapter4`, в котором у вас уже должен быть скрипт Python с симулятором тележки, который мы создали ранее.

4.4.1 Выбор гиперпараметров

В секции NEAT файла конфигурации мы определяем популяцию из `150` отдельных организмов и порог приспособленности со значением `1.0` в качестве критерия прекращения.

Приспособленность `fitness_criterion` установлена в значение `max`, что означает завершение эволюционного процесса, когда любая особь достигает значения приспособленности, равного значению `fitness_threshold`:

```
[NEAT]
fitness_criterion = max
fitness_threshold = 1.0
pop_size = 150
reset_on_extinction = False
```

Кроме того, мы значительно снизили вероятность добавления нового узла, чтобы сместить эволюционный процесс в разработку большего количества паттернов связей с минимальным количеством узлов в контроллере. Таким образом мы стремимся снизить энергопотребление нейросети развитого контроллера и сократить вычислительные затраты на обучение.

Соответствующие параметры в файле конфигурации следующие:

```
# Вероятность добавления/удаления узла
node_add_prob = 0.02
node_delete_prob = 0.02
```

Параметры, описывающие нашу начальную конфигурацию сети по количеству скрытых, входных и выходных узлов, задаются следующим образом:

```
# Параметры сети
num_hidden = 0
num_inputs = 4
num_outputs = 1
```

Мы увеличили порог совместимости видов, чтобы сместить эволюционный процесс в сторону меньшего количества видов. Кроме того, мы увеличили минимальный размер вида, чтобы указать, что мы заинтересованы в гораздо более густонаселенных видах, которые имеют больше шансов на сохранение полезных мутаций. В то же время мы уменьшили максимальный возраст стагнации, чтобы интенсифицировать эволюционный процесс, усилив раннее вымирание стагнирующих видов, которые не показывают каких-либо улучшений приспособленности.

Соответствующие параметры в файле конфигурации:

```
[DefaultSpeciesSet]
compatibility_threshold = 4.0
[DefaultStagnation]
species_fitness_func = max
max_stagnation = 15
species_elitism = 2
[DefaultReproduction]
elitism = 2
survival_threshold = 0.2
min_species_size = 8
```

Изучите файл конфигурации `single_pole_config.ini`.

Исходя из этих параметров конфигурации, в ходе эволюционного процесса будут использоваться более многочисленные виды, однако количество уникальных видов будет оставаться небольшим.

4.4.2 Настройка рабочей среды

Прежде чем вы начнете писать исходный код движка эксперимента, вы должны настроить виртуальную среду Python и установить все необходимые зависимости. Вы можете сделать это с помощью Anaconda, выполнив следующие команды в командной строке:

```
$ conda create --name single_pole_neat python=3.5
$ conda activate single_pole_neat
$ pip install neat-python==0.92
$ conda install matplotlib
$ conda install graphviz
$ conda install python-graphviz
```

Сначала эти команды создают и активируют виртуальную среду `single_pole_neat` с Python 3.5. После этого устанавливается библиотека NEAT-Python версии 0.92, а также другие зависимости, необходимые для утилит визуализации.

4.4.3 Исходный код эксперимента

Во-первых, вам нужно создать файл `single_pole_experiment.py` в рабочем каталоге `Chapter4`. В этом файле будет сохранен исходный код эксперимента по балансировке одиночного маятника. Также вам необходимо скопировать файл `visualize.py` из файлового архива главы в рабочий каталог. Мы будем использовать утилиты из этого файла для визуализации результатов эксперимента.

Исходный код движка эксперимента включает две основные функции.

Функция оценки приспособленности всех геномов в популяции

Первая функция получает список всех геномов в популяции и назначает оценку приспособленности для каждого из них. Эта функция передается по ссылке в движок нейроэволюции библиотеки NEAT-Python. Исходный код этой функции выглядит следующим образом:

```
def eval_genomes(genomes, config):
    for genome_id, genome in genomes:
        genome.fitness = 0.0
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        fitness = cart.eval_fitness(net)
        if fitness >= config.fitness_threshold:
            # Выполняем дополнительные шаги оценки со случайными начальными
состояниями,
            # чтобы гарантировать, что мы нашли стабильную стратегию управления,
            # а не какой-то особый случай начального состояния.
            success_runs = evaluate_best_net(net, config,
                                             additional_num_runs)
            # Нормируем приспособленность
            fitness = 1.0 - (additional_num_runs - success_runs) / \
                      additional_num_runs

        genome.fitness = fitness
```

Обратите внимание, что мы вводим дополнительные прогоны симуляции для победившего генома, чтобы убедиться, что его стратегия управления стабильна при запуске из множества случайных начальных состояний. Эта дополнительная проверка гарантирует, что мы нашли настоящего победителя, а не особый случай, специфичный для конкретного исходного состояния.

Предыдущая функция получает список всех геномов в популяции и параметры конфигурации NEAT. Для каждого конкретного генома она создает фенотип нейросети и использует его в качестве контроллера для запуска моделирования тележки с обратным маятником, как задано в следующей строке кода:

```
fitness = cart.eval_fitness(net)
```

Полученное значение приспособленности затем сравнивается с пороговым значением, которое мы определили в параметрах конфигурации. Если приспособленность превышает пороговое значение, мы можем предположить, что был найден успешный контроллер. Чтобы дополнительно проверить успешность найденного контроллера, сначала будут запущены дополнительные прогоны моделирования, а затем будет рассчитан окончательный показатель приспособленности. Это делает следующий фрагмент приведенного выше кода:

```
success_runs = evaluate_best_net(net, config, additional_num_runs)
fitness = 1.0 - (additional_num_runs - success_runs) / additional_num_runs
```

Дополнительные прогоны моделирования используют различные начальные значения для генератора случайных чисел, чтобы покрыть большинство возможных начальных состояний тележки с обратным маятником.

Функция выполнения эксперимента

Вторая функция конфигурирует, выполняет и выводит результаты процесса нейроэволюции. Здесь мы наметим некоторые критические места в реализации функции выполнения эксперимента.

1. Функция начинается с загрузки гиперпараметров из файла конфигурации и порождает начальную популяцию, используя загруженную конфигурацию:

```
# Загружаем конфигурацию.
config = neat.Config(neat.DefaultGenome,
                     neat.DefaultReproduction,
                     neat.DefaultSpeciesSet,
                     neat.DefaultStagnation,
                     config_file)


# Создаем популяцию, которая является объектом верхнего уровня NEAT.
p = neat.Population(config)
```

2. После этого функция настраивает репо́ртеры для сбора статистики, касающейся выполнения эволюционного процесса. Также добавляются выходные репортеры для вывода результатов выполнения на консоль в режиме реального времени. Сборщик контрольных точек настроен на сохранение промежуточных этапов выполнения, что может пригодиться, если позже вам потребуется восстановить процесс обучения:

```
# Добавляем репортер вывода хода выполнения в консоль.
p.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
p.add_reporter(stats)
p.add_reporter(neat.Checkpointer(5,
             filename_prefix='out/spb-neat-checkpoint-'))
```

3. Наконец, процесс эволюции выполняется в течение указанного числа поколений, а результаты сохраняются в каталоге `Output`:

```
# Запуск для N поколений.
best_genome = p.run(eval_genomes, n=n_generations)


# Отображение лучшего генома из поколений.
print('\nBest genome:\n{!s}'.format(best_genome))
# Проверяем, дает ли нам лучший геном контроллер-победитель
# для балансировки тележки с одним обратным маятником.
net = neat.nn.FeedForwardNetwork.create(best_genome, config)
best_genome_fitness = cart.eval_fitness(net)
if best_genome_fitness >= config.fitness_threshold:
    print("\n\nSUCCESS: The Single-Pole balancing controller
           has been found!!!")
else:
    print("\n\nFAILURE: Failed to find Single-Pole balancing
           controller!!!")
```

Полная реализация представлена в коде функции `run_experiment(config_file, n_generations=100)` в файловом архиве книги.

После того как в ходе эволюционного процесса найден лучший геном, проверяем, действительно ли он соответствует критериям порога приспособленности, которые мы установили в файле конфигурации. Возможно, во время этого процесса не будет найдено никакого рабочего решения, но тем не менее библиотека NEATPython вернет геном с формально наилучшим соответствием. Вот почему мы нуждаемся в этой дополнительной проверке, чтобы гарантировать, что наиболее подходящий геном может действительно решить проблему на практике.

4.4.4 Запуск эксперимента по балансировке одиночного маятника

Вам необходимо перейти в каталог, содержащий файл `single_pole_experiment.py`, и выполнить следующую команду:

```
$ python single_pole_experiment.py
```

Не забудьте активировать соответствующую виртуальную среду с помощью команды `conda activate single_pole_neat`.

Во время выполнения скрипта эксперимента на консоль будут выводиться следующие выходные данные для каждого поколения эволюции:

```
****** Running generation 13 ******

Population's average fitness: 0.26673 stdev: 0.12027
Best fitness: 0.70923 - size: (1, 2) - species 1 - id 2003
Average adjusted fitness: 0.161
Mean genetic distance 1.233, standard deviation 0.518
Population of 150 members in 1 species:
   ID age size fitness adj fit stag
 ==== === ==== ======= ======= ====
    1  13  150     0.7   0.161    7
Total extinctions: 0
Generation time: 4.635 sec (0.589 average)
```

В выходных данных вы можете видеть, что средняя приспособленность населения в поколении 14 низкая, но приспособленность организма с лучшими показателями (`0.70923`) уже близка к нашему пороговому значению завершения (`fitness_threshold = 1.0`), которое было установлено в файле конфигурации. Организм-чемпион кодирует фенотип нейросети, состоящий из одного нелинейного узла (выход) и только двух соединений (`size: (1, 2)`). Также интересно отметить, что в популяции существует только один вид.

После того как победитель найден, вывод консоли имеет следующие строки:

```
****** Running generation 14 ******

Population's average fitness: 0.26776 stdev: 0.13359
Best fitness: 1.00000 - size: (1, 3) - species 1 - id 2110

Best individual in generation 14 meets fitness threshold - complexity: (1,
3)

Best genome:
Key: 2110
Fitness: 1.0
Nodes:
  0 DefaultNodeGene(key=0, bias=-3.328545880116371, response=1.0,
activation=sigmoid, aggregation=sum)
Connections:
  DefaultConnectionGene(key=(-4, 0), weight=2.7587300138861037,
enabled=True)
  DefaultConnectionGene(key=(-3, 0), weight=2.951449584136504,
enabled=True)
  DefaultConnectionGene(key=(-1, 0), weight=0.9448711043565166,
enabled=True)

Evaluating the best genome in random runs
Runs successful/expected: 100/100
SUCCESS: The stable Single-Pole balancing controller has been found!!!
```

Лучший геном, который является победителем эволюции, кодирует фенотип нейросети, состоящий только из одного нелинейного узла (выход) и трех соединений из входных узлов (`size: (1, 3)`). Интересно отметить, что эволюция смогла выработать надежную стратегию управления, которая полностью игнорирует линейную скорость тележки и использует только три других входа: x, θ и θ. Этот факт является еще одним признаком правильности эволюционного отбора, потому что мы решили игнорировать трение тележки, что фактически исключает линейную скорость тележки из уравнений движения.

Граф нейросети победившего контроллера балансировки одиночного обратного маятника представлен на рис. 4.3.

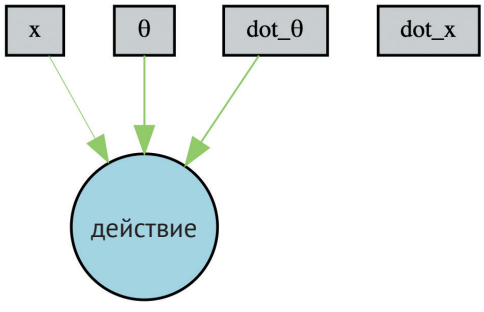


Рис. 4.3. Граф оптимального контроллера балансировки обратного маятника, найденный алгоритмом NEAT

График изменения значений приспособленности на протяжении поколений эволюции показан на рис. 4.4.

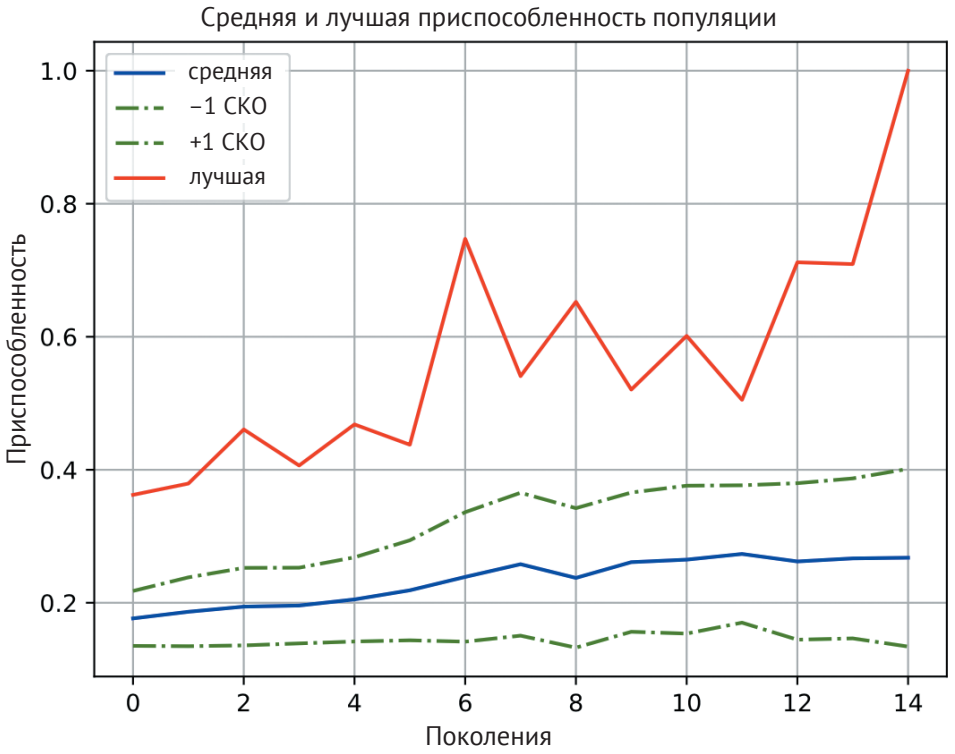


Рис. 4.4. Среднее и лучшее значения функции приспособленности в эксперименте с обратным маятником

Средняя приспособленность популяции во всех поколениях была низкой, но с самого начала произошла полезная мутация, которая породила определенную линию организмов. Из поколения в поколение одаренные особи из этой линии смогли не только сохранить свои полезные черты, но и улучшить их, что в конечном итоге привело к появлению победителя эволюции.

4.5 Упражнения

1. Попробуйте увеличить значение параметра `node_add_prob` и посмотреть, что произойдет. Производит ли алгоритм какое-то количество скрытых узлов, и если да, то сколько?
2. Попробуйте уменьшить/увеличить значение `compatibility_threshold`. Что произойдет, если вы установите его на `2.0` или `6.0`? Сможет ли алгоритм найти решение в каждом случае?
3. Попробуйте установить значение `elitism` в разделе `DefaultReproduction` равным нулю. Посмотрите, что получится. Сколько времени понадобилось эволюционному процессу, чтобы найти приемлемое решение на этот раз?

4. Задайте для параметра `survival_threshold` в разделе `DefaultReproduction` значение `0.5`. Посмотрите, как это влияет на видообразование в процессе эволюции. Почему так происходит?
5. Увеличьте значения `additional_num_runs` и `additional_steps` на порядок величины, чтобы дополнительно изучить, насколько хорошо обобщена найденная стратегия управления. Алгоритм все еще в состоянии найти выигрышное решение?

Последнее упражнение приведет к увеличению времени выполнения алгоритма.

4.6 Задача балансировки двойного маятника

Задача балансировки одиночного маятника не вызывает затруднений у алгоритма NEAT, способного быстро найти оптимальную стратегию управления для поддержания стабильного состояния системы. Чтобы усложнить эксперимент, мы представляем более продвинутую версию задачи балансировки. В этой версии на тележке установлены два обратных маятника на шарнире.

Схема новой системы с двумя маятниками показана на рис. 4.5.

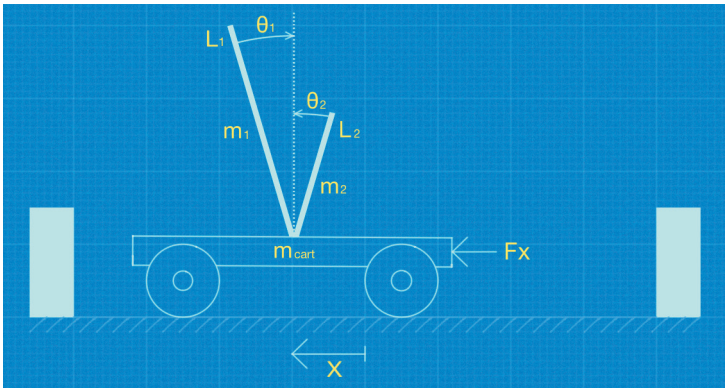


Рис. 4.5. Система из тележки с двумя маятниками

Прежде чем перейти к деталям реализации эксперимента, нам необходимо определить переменные состояния и уравнения движения для моделирования тележки с двумя обратными маятниками.

4.6.1 Переменные состояния системы и уравнения движения

Задача контроллера – прикладывать усилие к тележке таким образом, чтобы как можно дольше удерживать в равновесии два перевернутых маятника. В то же время тележка должна оставаться в определенных границах. Как и в случае с задачей балансировки одиночного маятника, рассмотренной ранее, стратегию управления можно определить как задачу управляемого избегания. Это означает, что контроллер должен поддерживать стабильное состояние системы, избегая опасных зон, когда тележка выходит за границы пути или любой

из маятников отклоняется от вертикали больше допустимого. Единственного решения этой задачи не существует, но можно найти подходящую стратегию управления, учитывая, что маятники имеют разную длину и массу. Поэтому они по-разному реагируют на управляющее воздействие.

Текущее состояние балансировщика двух маятников может быть определено следующими переменными:

- горизонтальная позиция тележки x;
- скорость тележки $\dot{x}$;
- угол отклонения первого маятника от вертикали θ_1;
- угловая скорость первого маятника $\dot{\theta}_1$;
- угол отклонения второго маятника θ_2;
- угловая скорость второго маятника $\dot{\theta}_2$.

Уравнения движения для двух несвязанных маятников, сбалансированных на одной тележке, которая игнорирует трение между колесами и дорожкой, имеют следующий вид:

$$\ddot{x} = \frac{F_x - \sum_{i=1}^{2} \tilde{F}_i}{M + \sum_{i=1}^{2} \tilde{m}_i};$$

$$\dot{\theta}_i = -\frac{3}{4L_i}\left(\ddot{x}\cos\theta_i + g\sin\theta_i + \frac{\mu_{pi}\dot{\theta}_i}{m_i L_i} \right).$$

В этом уравнении $\tilde{F}_i$ – сила реакции от i-го маятника на тележке:

$$\tilde{F}_i = m_i L_i \dot{\theta}_i^2 \sin\theta_i + \frac{3}{4} m_i \cos\theta_i \left(\frac{\mu_{pi}\dot{\theta}_i}{m_i L_i} + g\sin\theta_i \right).$$

Далее, $\tilde{m}_i$ – эффективная масса i-го маятника:

$$\tilde{m}_i = m_i \left(1 - \frac{3}{4}\cos^2\theta_i \right).$$

В модели двух маятников на тележке используются параметры, перечисленные в табл. 4.1.

Таблица 4.1. Параметры модели тележки с двумя обратными маятниками

Символ	Описание	Значения
x	Позиция тележки на дорожке	$\in$ [–2.4, 2.4] м
θ	Отклонение маятника от вертикали	$\in$ [–36, 36] градусов
F_x	Сила, приложенная к тележке	$\mp$10Н
L_i	Расстояние от центра масс маятника до шарнира	L_1 = 0,5 м, L_2 = 0,05 м
M	Масса тележки	1.0 кг
m_i	Масса i-го маятника	m_1 = 0,1 кг, m_2 = 0,01 кг
μ_p	Коэффициент трения в шарнире i-го маятника	0.000002
g	Ускорение свободного падения	–9,8 м/с^2

Следующий фрагмент кода определяет эти параметры системы как константы:

```
GRAVITY = -9.8 # m/s^2 – уравнения движения системы из двух маятников
               # подразумевают отрицательное значение
MASS_CART = 1.0 # kg
FORCE_MAG = 10.0 # N
# Первый маятник
MASS_POLE_1 = 1.0 # kg
LENGTH_1 = 0.5 # m – обычно половина длины первого маятника
# Второй маятник
MASS_POLE_2 = 0.1 # kg
LENGTH_2 = 0.05 # m – обычно половина длины второго маятника
# Коэффициент трения шарнира
MUP = 0.000002
```

Реализация уравнений движения в коде Python выглядит следующим образом:

```
# Находим направление приложенной силы.
force = (action - 0.5) * FORCE_MAG * 2.0 # действие имеет бинарное значение
# Вычисляем проекцию сил, действующих на маятники
cos_theta_1 = math.cos(theta1)
sin_theta_1 = math.sin(theta1)
g_sin_theta_1 = GRAVITY * sin_theta_1
cos_theta_2 = math.cos(theta2)
sin_theta_2 = math.sin(theta2)
g_sin_theta_2 = GRAVITY * sin_theta_2
# Вычисляем промежуточные значения
ml_1 = LENGTH_1 * MASS_POLE_1
ml_2 = LENGTH_2 * MASS_POLE_2
temp_1 = MUP * theta1_dot / ml_1
temp_2 = MUP * theta2_dot / ml_2
fi_1 = (ml_1 * theta1_dot * theta1_dot * sin_theta_1) + \
       (0.75 * MASS_POLE_1 * cos_theta_1 * (temp_1 + g_sin_theta_1))
fi_2 = (ml_2 * theta2_dot * theta2_dot * sin_theta_2) + \
       (0.75 * MASS_POLE_2 * cos_theta_2 * (temp_2 + g_sin_theta_2))
mi_1 = MASS_POLE_1 * (1 - (0.75 * cos_theta_1 * cos_theta_1))
mi_2 = MASS_POLE_2 * (1 - (0.75 * cos_theta_2 * cos_theta_2))
# Вычисляем результаты: ускорение тележки и угловые ускорения маятников
x_ddot = (force + fi_1 + fi_2) / (mi_1 + mi_2 + MASS_CART)
theta_1_ddot = -0.75 * (x_ddot * cos_theta_1 + \
                        g_sin_theta_1 + temp_1) / LENGTH_1
theta_2_ddot = -0.75 * (x_ddot * cos_theta_2 + \
                        g_sin_theta_2 + temp_2) / LENGTH_2
```

Полный исходный код представлен в файле `cart_two_pole.py`, который хранится в файловом архиве книги в каталоге `Chapter4`. Ознакомьтесь с функцией `calc_step(action, x, x_dot, theta1, theta1_dot, theta2, theta2_dot)`.

Приведенный выше код получает текущее состояние системы (`x`, `x_dot`, `theta1`, `theta1_dot`, `theta2`, `theta2_dot`) наряду с управляющим сигналом и вычисляет производные (ускорение тележки и угловое ускорение обоих маятников).

4.6.2 Подкрепляющий сигнал

После выполнения действий среда моделирования должна возвращать минимальную информацию о состоянии системы в виде сигнала подкрепления r_t. Сигнал подкрепления показывает, нарушает ли контроллер балансировки двух маятников установленные ограничения после применения действия. Его можно определить следующим образом:

$$r_t = \begin{cases} 0 \text{ если } -0{,}63 < \theta_i^t < 0{,}63 \text{ радиан; и } -2{,}4 < x_t < 2{,}4 \text{ м} \\ 1 \text{ в остальных случаях} \end{cases}.$$

Реализация сигнала подкрепления в Python выглядит так:

```
res = x < -2.4 or x > 2.4 or \
    theta1 < -THIRTY_SIX_DEG_IN_RAD or theta1 > THIRTY_SIX_DEG_IN_RAD or \
    theta2 < -THIRTY_SIX_DEG_IN_RAD or theta2 > THIRTY_SIX_DEG_IN_RAD
```

Условие проверяет, что угол отклонения каждого маятника от вертикали находится в диапазоне ±36 градусов (0,63 радиан) и что расстояние от тележки до центра дорожки лежит в диапазоне ±2,4 м.

4.6.3 Начальные условия и обновление состояния

В эксперименте по балансировке одиночного маятника мы использовали случайные условия начального состояния, но в эксперименте с двумя маятниками начальные условия немного проще. Эксперимент запускается с нулевыми значениями для всех скоростей тележки и маятников. Начальное отклонение длинного маятника составляет один градус от вертикали, а короткий маятник находится строго в вертикальном положении.

Мы устанавливаем следующие начальные условия:

$$\begin{aligned} x &= 0; \\ \dot{x} &= 0; \\ \theta_1 &= \frac{\pi}{180}; \\ \dot{\theta}_1 &= 0; \\ \theta_2 &= 0; \\ \dot{\theta}_2 &= 0. \end{aligned}$$

Состояние системы тележка–маятники обновляется на каждом этапе моделирования путем численной аппроксимации уравнений движения с использованием метода Рунге–Кутты четвертого порядка с размером шага 0,01 секунды. Метод аппроксимации Рунге–Кутты четвертого порядка позволяет рассчитать отклик системы с учетом переменных состояния текущего временно́го шага. Новые управляющие входы генерируются каждую секунду.

Таким образом, частота управления составляет 50 Гц, а частота обновления состояния системы – 100 Гц.

Реализация метода Рунге–Кутты четвертого порядка в Python выглядит следующим образом.

1. Используем текущие переменные состояния системы тележка–маятники, чтобы обновить промежуточное состояние для следующего шага полупериода и выполнить первый шаг моделирования:

```
hh = tau / 2.0
yt = [None] * 6

# Обновляем промежуточное состояние.
for i in range(6):
    yt[i] = y[i] + hh * dydx[i]

# Выполняем шаг моделирования.
x_ddot, theta_1_ddot, theta_2_ddot = calc_step(action = f, yt[0],
yt[1], yt[2], yt[3], yt[4], yt[5])

# Сохраняем производные.
dyt = [yt[1], x_ddot, yt[3], theta_1_ddot, yt[5], theta_2_ddot]
```

2. Обновляем промежуточное состояние, используя производные, полученные на первом этапе моделирования, и выполняем второй этап моделирования:

```
# Обновляем промежуточное состояние.
for i in range(6):
    yt[i] = y[i] + hh * dyt[i]

# Выполняем один шаг моделирования.
x_ddot, theta_1_ddot, theta_2_ddot = calc_step(action = f, yt[0],
yt[1], yt[2], yt[3], yt[4], yt[5])

# Сохраняем производные.
dym = [yt[1], x_ddot, yt[3], theta_1_ddot, yt[5], theta_2_ddot]
```

3. Обновляем промежуточное состояние, используя производные, полученные на первом и втором этапах моделирования, и выполняем третий этап моделирования, используя обновленное состояние:

```
# Обновляем промежуточное состояние.
for i in range(6):
    yt[i] = y[i] + tau * dym[i]
    dym[i] += dyt[i]

# Выполняем один шаг моделирования.
x_ddot, theta_1_ddot, theta_2_ddot = calc_step(action = f, yt[0],
yt[1], yt[2], yt[3], yt[4], yt[5])

# store derivatives
dyt = [yt[1], x_ddot, yt[3], theta_1_ddot, yt[5], theta_2_ddot]
```

Наконец, воспользуемся производными от первых трех этапов моделирования, чтобы аппроксимировать конечное состояние системы, которое будет использоваться в дальнейшем моделировании:

```
# Находим состояние системы после аппроксимации.
yout = [None] * 6 # Аппроксимированное состояние системы.
h6 = tau / 6.0
for i in range(6):
    yout[i] = y[i] + h6 * (dydx[i] + dyt[i] + 2.0 * dym[i])
```

Давайте рассмотрим элементы предыдущего уравнения:

- `f` – управляющее действие, применяемое во время моделирования (0 или 1);
- `y` – список с текущими значениями переменных состояния (x, $\dot{x}$, θ_1, $\dot{\theta}_1$, θ_2, $\dot{\theta}_2$);
- `dydx` – это список производных переменных состояния ($\dot{x}$, $\ddot{x}$, $\dot{\theta}_1$, $\ddot{\theta}_1$, $\dot{\theta}_2$, $\ddot{\theta}_2$);
- `tau` – размер шага по времени для приближения.

Для более подробного знакомства с тонкостями реализации изучите код функции `rk4(f, y, dydx, tau)` в файле `cart_two_pole.py`.

Реализация метода Рунге–Кутты четвертого порядка получает текущее состояние системы (`x, x_dot, theta1, theta1_dot, theta2, theta2_dot`) вместе с их производными и аппроксимирует состояние системы на следующем шаге.

4.6.4 Управляющие действия

Как и в случае эксперимента с балансировкой одиночного маятника, который обсуждался ранее в этой главе, контроллер балансировки двойного маятника генерирует только два управляющих сигнала: толчок влево и толчок вправо с постоянной силой. Таким образом, действующая сила в момент t может быть определена следующим образом:

$$F_t = \begin{cases} 10\text{Н}, \text{ если } a[t] = 1 \\ -10\text{Н}, \text{ если } a[t] = 0 \end{cases}.$$

В данном уравнении $a[t]$ – управляющий сигнал, полученный от контроллера в момент t.

4.6.5 Взаимодействие между решателем и симулятором

Прежде чем применить переменные состояния в качестве входных данных для нейросети контроллера, их следует масштабировать, чтобы привести к интервалу [0,1]. Уравнения для предварительной обработки входных переменных имеют следующий вид:

$$x_0[t] = \frac{1}{4.8}\left(x[t] + 2.4\right),$$
$$x_1[t] = \frac{1}{3}\left(\dot{x}[t] + 1.5\right),$$
$$x_2[t] = \frac{1}{1.256}\left(\theta_1[t] + 0.628\right),$$
$$x_3[t] = \frac{1}{4}\left(\dot{\theta}_1[t] + 2\right),$$
$$x_4[t] = \frac{1}{1.256}\left(\theta_2[t] + 0.628\right),$$
$$x_5[t] = \frac{1}{4}\left(\dot{\theta}_2[t] + 2\right).$$

В этих уравнениях переменные $x_0 \ldots x_5$ соответствуют горизонтальному положению тележки, ее горизонтальной скорости, углу первого маятника относительно вертикали, его угловой скорости, а также углу и угловой скорости второго маятника соответственно.

Принимая во внимание системные ограничения, определенные ранее (см. определение r_t), масштабированные значения x_0, x_2 и x_4 гарантированно находятся в интервале [0, 1], в то время как масштабированные значения x_1, x_3 и x_5 в основном попадают в интервал 0...1, но могут рано или поздно выйти за эти пределы.

Соответствующий код для масштабирования входных данных выглядит так:

```
input[0] = (state[0] + 2.4) / 4.8
input[1] = (state[1] + 1.5) / 3.0
input[2] = (state[2] + THIRTY_SIX_DEG_IN_RAD) / (THIRTY_SIX_DEG_IN_RAD * 2.0)
input[3] = (state[3] + 2.0) / 4.0
input[4] = (state[4] + THIRTY_SIX_DEG_IN_RAD) / (THIRTY_SIX_DEG_IN_RAD * 2.0)
input[5] = (state[5] + 2.0) / 4.0
```

Список состояний содержит переменные текущего состояния в следующем порядке: $x, \dot{x}_1, \theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2$.

4.7 Целевая функция для эксперимента по балансировке двойного маятника

Целевая функция для этой задачи аналогична целевой функции, определенной ранее для задачи балансировки одиночного маятника и задается следующими уравнениями:

$$\mathcal{L} = \frac{log\, t_{max} - log\, t_{eval}}{log\, t_{max}},$$
$$\mathcal{F} = 1.0 - \mathcal{L}.$$

В этих уравнениях $t_{\max}$ – это ожидаемое количество шагов, указанных в конфигурации эксперимента (100 000), и t_{eval} – фактическое количество шагов, в течение которых контроллер смог поддерживать стабильное состояние балансировщика в допустимых пределах.

Мы используем логарифмические шкалы, потому что большинство испытаний завешаются неудачей в первые несколько сотен шагов, но мы тестируем кандидатов на протяжении 100 000 шагов. С логарифмической шкалой мы имеем лучшее распределение показателей приспособленности даже по сравнению с небольшим количеством шагов в неудачных испытаниях.

Первое из предыдущих уравнений определяет ошибку, которая находится в диапазоне [0, 1], а второе – это показатель приспособленности, который дополняет значение ошибки. Таким образом, значения показателя приспособленности находятся в диапазоне [0, 1], и чем выше значение, тем лучше результат.

Исходный код Python аналогичен определению целевой функции в эксперименте с балансировкой одиночного маятника, но он использует другие вызовы симулятора для получения количества шагов балансировки:

```
# Сначала мы запускаем цикл моделирования, возвращающий
# количество успешных шагов.
steps = cart.run_markov_simulation(net, max_bal_steps)

if steps == max_bal_steps:
     # Максимальная приспособленность.
     return 1.0
elif steps == 0: # Нужно избежать ошибки при попытке взять log(0)
     # Минимальная приспособленность.
     return 0.0
else:
     log_steps = math.log(steps)
     log_max_steps = math.log(max_bal_steps)
     # Значение ошибки в интервале [0, 1]
     error = (log_max_steps - log_steps) / log_max_steps
     # Приспособленность является дополнением значения ошибки
     return 1.0 - error
```

Здесь мы используем логарифмическую шкалу, потому что большинство прогонов дают сбой слишком рано, в пределах 100 шагов или около того, но мы запускаем моделирование на 100 000 шагов.

4.8 Эксперимент по балансировке

В этом эксперименте используется версия задачи балансировки двойного маятника, которая предполагает полное знание текущего состояния системы, включая угловые скорости маятников и скорость тележки. Критерием успеха в этом эксперименте является поддержание баланса обоих маятников в течение 100 000 шагов или приблизительно 33 минут условного смоделированного времени. Маятник считается сбалансированным, когда он отклоняется не более чем на ±36 градусов от вертикали, а тележка остается в пределах ±2,4 метра от центра дорожки.

4.8.1 Выбор гиперпараметров

По сравнению с предыдущим экспериментом, описанным в этой главе, задача балансировки двух маятников решается гораздо труднее из-за сложной дина-

мики движения. Таким образом, пространство поиска для успешной стратегии контроля шире и требует более разнообразной популяции. Чтобы увеличить разнообразие популяции, мы задаем ее размер в 10 раз больше, чем в эксперименте с балансировкой одиночного маятника.

Пороговое значение приспособленности осталось прежним:

```
[NEAT]
fitness_criterion = max
fitness_threshold = 1.0
pop_size = 1000
reset_on_extinction = False
```

Чтобы еще больше усилить эволюционное разнообразие, мы увеличиваем вероятность добавления новых узлов и связей, а также меняем схему конфигурации исходных связей. Кроме того, значение параметра `initial_connection` содержит вероятность создания связи, что вносит дополнительную неопределенность в процесс формирования графа связей:

```
# Вероятность добавления/удаления связей.
conn_add_prob = 0.5
conn_delete_prob = 0.2

initial_connection = partial_direct 0.5

# Вероятность добавления/удаления узлов.
node_add_prob = 0.2
node_delete_prob = 0.2
```

Наконец, принимая во внимание размер популяции и возможный размер вида, мы сократили долю особей, которым разрешено размножаться (`survival_threshold`). Эта настройка ограничивает пространство поиска решения, позволяя участвовать в процессе рекомбинации только самым подходящим организмам:

```
[DefaultReproduction]
elitism = 2
survival_threshold = 0.1
min_species_size = 2
```

Последний параметр противоречив и может снизить эффективность эволюционного процесса в целом. Но с большой популяцией он часто работает хорошо, уменьшая количество возможных рекомбинаций. Таким образом, как правило, большие значения порога выживания используются для небольших популяций, а небольшие значения – для больших популяций.

Из-за повышенной сложности этого эксперимента один из гиперпараметров становится чрезвычайно важным для конечного результата. Процесс нейроэволюции строится вокруг вероятности возникновения мутаций, а вероятность мутации зависит от значений, выданных генератором случайных чисел.

Как вы знаете, в обычных компьютерах нет истинного источника случайности. Вместо этого случайность генерируется псевдослучайным алгоритмом, который сильно зависит от некого начального числа, с которого начинается генерирование последовательности случайных чисел. На самом деле начальное значение точно определяет последовательность всех псевдослучайных чисел, которые будут выданы данным генератором.

Таким образом, мы можем рассматривать начальное число как существенный параметр, определяющий начальные условия. Этот параметр устанавливает свойства случайного аттрактора, который будет усиливать крошечные изменения в числовом пространстве поиска алгоритма. Эффект усиления в конечном итоге определяет, сможет ли алгоритм найти победителя и сколько времени это займет.

Начальное значение генератора случайных чисел определяется в строке с номером 100 файла `two_pole_markov_experiment.py`:

```
# Начальное значение генератора случайных чисел
seed = 1559231616
random.seed(seed)
```

Полный список гиперпараметров, задействованных в эксперименте с двумя маятниками, можно найти в файле `two_pole_markov_config.ini`.

Данный код устанавливает начальное значение стандартного генератора случайных чисел, поставляемого со средой Python.

4.8.2 Настройка рабочей среды

Рабочую среду для эксперимента по балансировке двух маятников можно настроить с помощью следующих команд, введенных в любом выбранном вами приложении терминала:

```
$ conda create --name double_pole_neat python=3.5
$ conda activate double_pole_neat
$ pip install neat-python==0.92
$ conda install matplotlib
$ conda install graphviz
$ conda install python-graphviz
```

Эти команды создают и активируют виртуальную среду `double_pole_neat` с Python 3.5. После этого устанавливается библиотека NEAT-Python версии 0.92, а также другие зависимости, используемые нашими утилитами визуализации.

4.8.3 Реализация эксперимента

Исходный код, реализующий оценку приспособленности генома, аналогичен тому, который используется для эксперимента по балансированию одного маятника. Основное отличие состоит в том, что он будет ссылаться на другую среду моделирования, чтобы получить количество сбалансированных шагов. Вы можете обратиться к исходному коду функций `eval_fitness(net, max_bal_`

steps=100000) и eval_genomes(genomes, config) в файле two_pole_markov_experiment.py для изучения деталей реализации.

В этом эксперименте мы ввели адаптивное обучение, которое попытается найти правильную длину короткого маятника в процессе эволюции. Длина короткого маятника меняет динамику движения системы. Не все комбинации гиперпараметров в сочетании с определенной длиной короткого маятника могут привести к успешной стратегии управления. В данном случае мы реализуем последовательное увеличение длины короткого маятника, пока не будет найдено решение:

```
# Запуск эксперимента
pole_length = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]
num_runs = len(pole_length)
for i in range(num_runs):
    cart.LENGTH_2 = pole_length[i] / 2.0
    solved = run_experiment(config_path, n_generations=100, silent=False)
    print("run: %d, solved: %s, length: %f" %
                                        (i + 1, solved, cart.LENGTH_2))
if solved:
    print("Solution found in: %d run, short pole length: %f" %
                                        (i + 1, cart.LENGTH_2))
break
```

Полный код примера доступен для детального ознакомления в файле two_pole_markov_experiment.py.

Этот код запускает моделирование с использованием различных значений длины короткого маятника, пока не будет найдено решение.

4.8.4 Запуск эксперимента с двумя маятниками

Реализовав симулятор балансировки тележки с двумя маятниками, анализатор приспособленности генома и код движка эксперимента, мы готовы приступить к экспериментам. Войдите в каталог, содержащий файл two_pole_markov_experiment.py, и выполните в окне терминала команду

```
$ python two_pole_markov_experiment.py
```

Не забудьте активировать соответствующую виртуальную среду с помощью команды conda activate double_pole_neat.

Эта команда запустит эволюционный процесс под управлением алгоритма NEAT, используя гиперпараметры, указанные в файле two_pole_markov_config.ini, а также симулятор тележки с двумя маятниками, который мы реализовали раньше.

Спустя 96 поколений мы можем найти победителя в поколении 97. Вывод в консоль для последнего поколения выглядит примерно так:

```
****** Running generation 97 ******

Population's average fitness: 0.27393 stdev: 0.10514
Best fitness: 1.00000 - size: (1, 6) - species 26 - id 95605

Best individual in generation 97 meets fitness threshold - complexity: (1, 6)

Best genome:
Key: 95605
Fitness: 1.0
Nodes:
  0 DefaultNodeGene(key=0, bias=7.879760594997953, response=1.0,
activation=sigmoid, aggregation=sum)
Connections:
  DefaultConnectionGene(key=(-6, 0), weight=1.9934757746640883,
enabled=True)
  DefaultConnectionGene(key=(-5, 0), weight=3.703109977745863,
enabled=True)
  DefaultConnectionGene(key=(-4, 0), weight=-11.923951805881497,
enabled=True)
  DefaultConnectionGene(key=(-3, 0), weight=-4.152166115226511,
enabled=True)
  DefaultConnectionGene(key=(-2, 0), weight=-3.101569479910728,
enabled=True)
  DefaultConnectionGene(key=(-1, 0), weight=-1.379602358542496,
enabled=True)

Evaluating the best genome in random runs
Runs successful/expected: 1/1
SUCCESS: The stable Double-Pole-Markov balancing controller found!!!
Random seed: 1559231616
run: 1, solved: True, half-length: 0.050000
Solution found in: 1 run, short pole length: 0.100000
```

В выводе консоли мы видим, что у победившего генома размерность (1, 6), что означает, что у него есть только один нелинейный узел – выход – и полный набор соединений от шести входов до выходных узлов. Мы можем предположить, что была найдена минимально возможная конфигурация нейросети контроллера, поскольку она не включает в себя никаких скрытых узлов, а вместо этого кодирует поведение контроллера при помощи найденных весов связей. Кроме того, интересно отметить, что решение было найдено для наименьшего значения из списка всех возможных значений длины короткого маятника.

Конфигурация нейросети контроллера, способного осуществлять стратегию надежного управления, показана на следующем графе рис. 4.6.

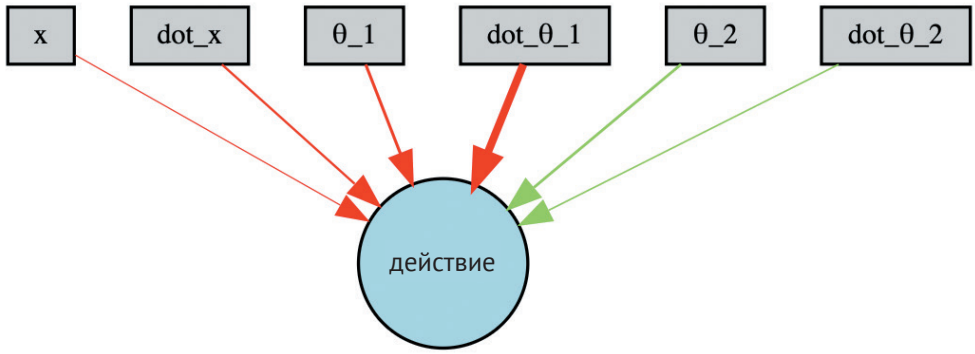


Рис. 4.6. Граф нейросети оптимального балансировщика двух маятников

Показатели приспособленности варьируются в зависимости от поколения, как показано на графике (рис. 4.7).

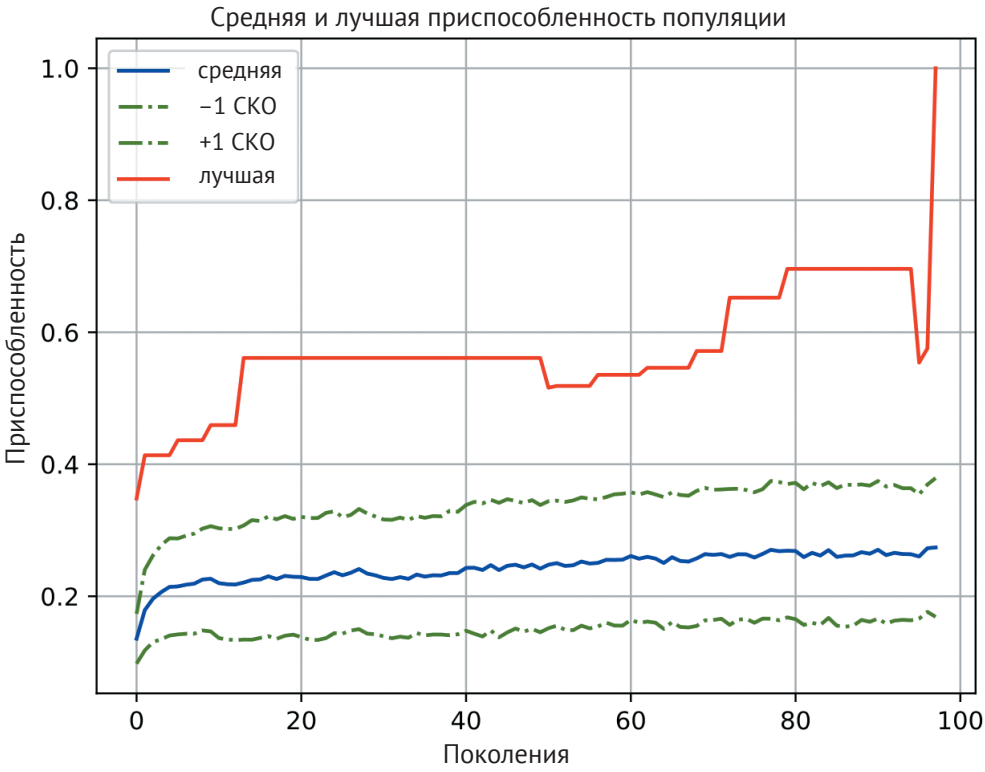


Рис. 4.7. Оценка приспособленности по поколениям в эксперименте с двумя маятниками

Последний график заслуживает особого внимания, если вы хотите знать, как работает эволюция. Вы можете видеть, что, прежде чем выявить победителя, показатель приспособленности резко падает. Это связано с выми-

ранием стагнирующих видов, которые достигли плато со средневысокими показателями приспособленности, но не показали улучшения за последние 15 поколений. После этого вакантное место занимают свежие виды, наделенные генетическими знаниями, унаследованными от вымерших видов. Эти новорожденные виды также вносят полезную мутацию, которая объединяет наследственные достижения с новыми уловками и в конечном итоге дает победителя.

В этом эксперименте мы решили усилить разнообразие видов путем значительного увеличения размера популяции и внесения других изменений в гиперпараметры. На следующем графике вы можете видеть, что мы достигли нашей цели и что процесс нейроэволюции проходит через множество видов, пока не будет найдено решение (рис. 4.8).

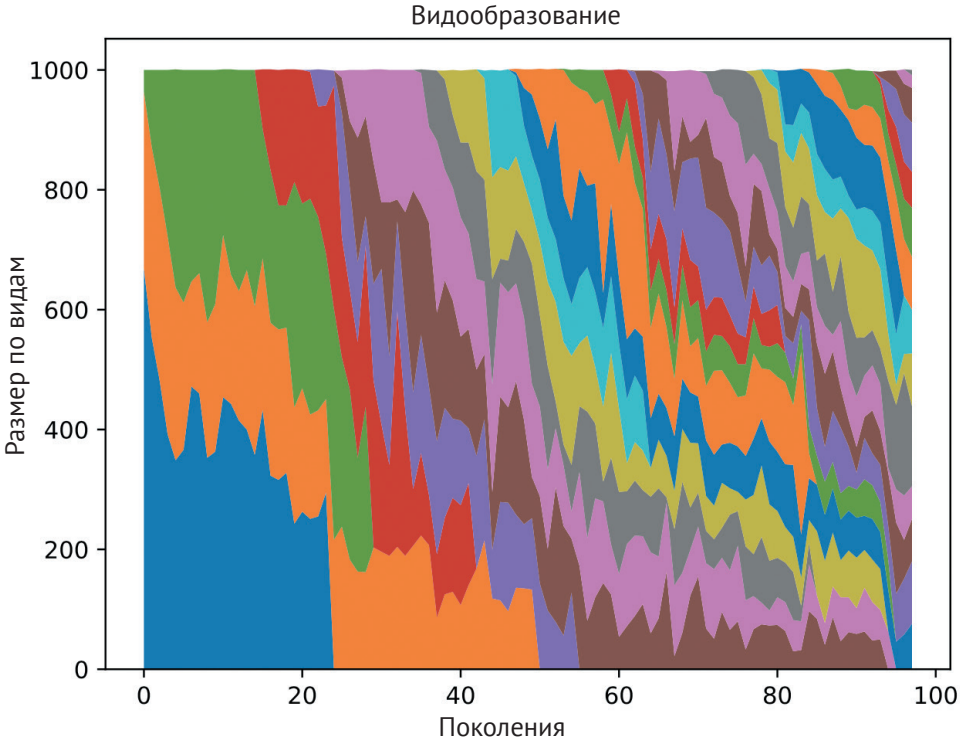


Рис. 4.8. Видовое разнообразие в эксперименте с двумя маятниками

Далее мы хотели бы узнать, как изменение начального значения генератора случайных чисел влияет на алгоритм NEAT. Для начала мы увеличили значение начального числа только на единицу (все остальное не изменилось). С этим новым условием алгоритм NEAT все еще смог найти стабильную стратегию управления, но создал другую, причудливую конфигурацию нейросети контроллера (рис. 4.9) вместо оптимальной конфигурации, показанной ранее на рис. 4.6.

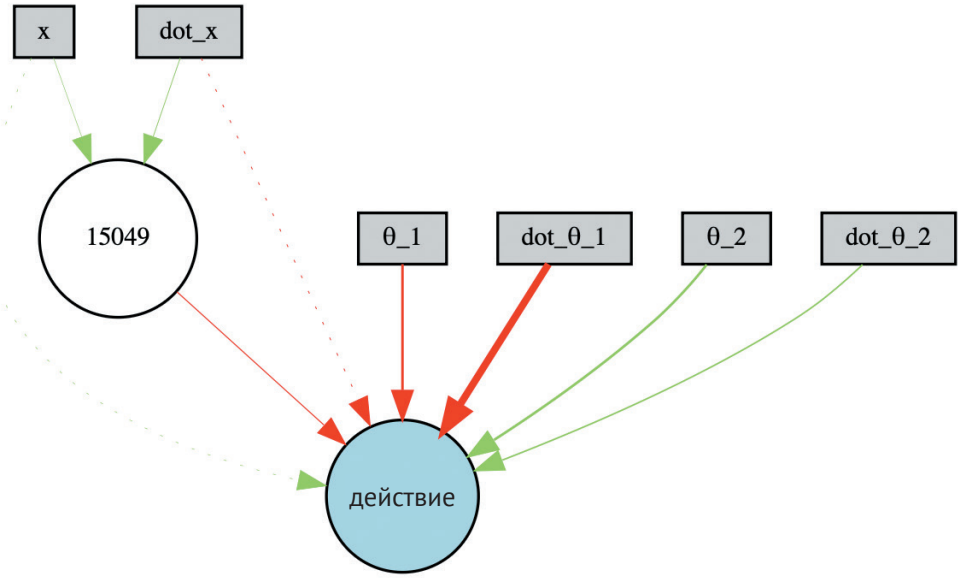


Рис. 4.9. Нейросеть контроллера балансировки двух маятников после увеличения начального числа на единицу (все остальное не изменяется)

Когда значение начального числа генератора случайных чисел увеличили еще больше, например на 10, процесс нейроэволюции вообще не смог найти какое-либо решение задачи балансировки.

Этот эксперимент выявил еще один важный аспект методов, основанных на нейроэволюции, – влияние начальных условий, определяемых значением начального числа генератора случайных чисел. Случайное начальное число определяет свойства случайного аттрактора, который усиливает эффекты процесса эволюции, как хорошие, так и плохие. Следовательно, в этом эксперименте крайне важно найти подходящее значение начального случайного числа, чтобы успешно пройти процесс нейроэволюции. Мы обсудим методы поиска подходящих значений начальных случайных чисел в конце этой книги.

4.9 Упражнения

1. Попробуйте установить в файле конфигурации значение параметра `node_add = 0.02` и посмотрите, что произойдет.
2. Измените начальное значение генератора случайных чисел и посмотрите, что произойдет. Было ли найдено решение с новым значением? Чем оно отличается от того, что мы представили в этой главе?

4.10 Заключение

В данной главе вы узнали, как реализовать стратегии управления для контроллеров, которые могут поддерживать стабильное состояние тележки с одним или двумя обратными маятниками, установленными вертикально. Вы улучшили навыки работы с Python и расширили свои знания о библиотеке NEAT-Python, написав точную модель физического устройства, которая использовалась для определения целевых функций при проведении экспериментов. Кроме того, вы узнали о двух методах численной аппроксимации дифференциальных уравнений – Эйлера и Рунге–Кутты и реализовали их в Python.

Мы обнаружили, что начальные условия, которые определяют нейроэволюционный процесс, такие как начальное случайное число, оказывают значительное влияние на результативность алгоритма. Эти значения определяют всю последовательность числового ряда, который будет сгенерирован генератором случайных чисел. Они служат случайным аттрактором, который может усиливать или ослаблять эффекты эволюции.

В следующей главе мы обсудим, как использовать нейроэволюцию для создания агентов-навигаторов, способных проходить через лабиринт. Вы узнаете, как определить целевую функцию для решения задачи прохождения лабиринта и как написать точную симуляцию робота-агента, который может перемещаться по лабиринту. Мы рассмотрим два типа лабиринтов и продемонстрируем, как может оплошать целенаправленная функция приспособленности, которая пытается найти решение в обманчивой среде сложного лабиринта.

Глава 5

Автономное прохождение лабиринта

Автономное прохождение лабиринта является классической задачей информатики, относящейся к области автономной навигации. В этой главе вы узнаете, как можно использовать методы нейроэволюции для решения задач прохождения лабиринта. Также мы объясним, как определить функцию приспособленности с использованием оценки приспособленности агента-навигатора, рассчитанной как производная от расстояния агента до конечной цели. К концу главы вы усвоите основы обучения автономного навигационного агента с использованием методов нейроэволюции и сможете создать более продвинутый решатель лабиринтов, который будет представлен в следующей главе. Вы познакомитесь с новыми методами визуализации, которые облегчат понимание результатов выполнения алгоритма. Кроме того, получите практический опыт написания симуляторов роботов, способных ориентироваться в лабиринте, и связанных с ними сред лабиринта на языке Python.

В этой главе вы познакомитесь со следующими темами:

- обманчивый характер проблемы навигации в лабиринте;
- написание симулятора робота-решателя лабиринта, оснащенного массивом датчиков и исполнительных механизмов;
- определение целеориентированной функции приспособленности для управления обучением решателя лабиринта на основе нейроэволюции;
- проведение экспериментов с простыми и сложными конфигурациями лабиринта.

5.1 Технические требования

Для проведения экспериментов, описанных в этой главе, ваш компьютер должен удовлетворять следующим техническим требованиям:

- Windows 8/10, macOS 10.13 или новее, или современный Linux;
- Anaconda Distribution версия 2019.03 или новее.

Исходный код примеров этой главы можно найти в каталоге `Chapter5` в файловом архиве книги.

5.2 Задача навигации в лабиринте

Задача прохождения лабиринта является классической проблемой информатики, которая тесно связана с созданием автономных агентов навигации, способных найти путь в неоднозначных средах. Окружающая среда в виде лабиринта – классическая иллюстрация целого класса проблем, которые имеют обманчивый ландшафт приспособленности. Это означает, что *целеориентированная функция приспособленности* (goal-oriented fitness function) может иметь крутые градиенты показателей приспособленности в тупиках лабиринта, которые близки к конечной цели. Такие области лабиринта становятся *локальными оптимумами* для алгоритмов поиска на основе близости к цели, которые могут сходиться в этих областях. Когда алгоритм поиска застревает в таком обманчивом локальном оптимуме, он не может найти адекватного агента, способного пройти лабиринт.

На рис. 5.1 изображен двухмерный лабиринт, в котором затемнены *локально оптимальные тупики*.

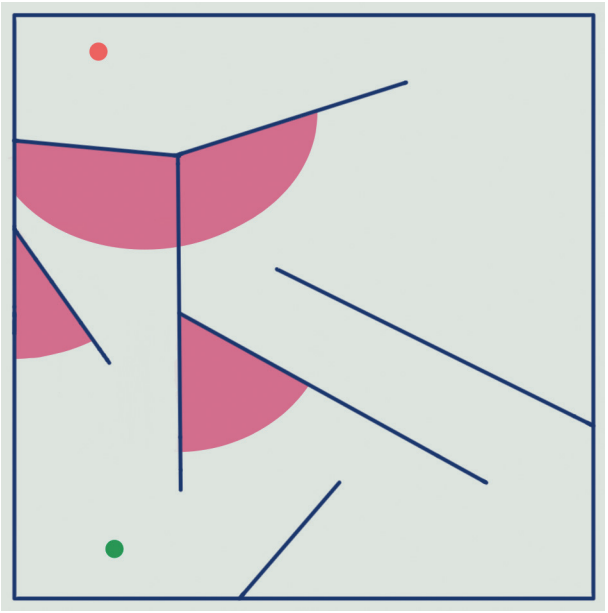


Рис. 5.1. Двухмерный лабиринт с локально оптимальными тупиками (затемнены)

Конфигурация лабиринта на этом рисунке демонстрирует ландшафт обманчивых показателей приспособленности, сосредоточенных в локально оптимальных тупиках (помеченных как закрашенные сегменты). Агент-решатель лабиринта, перемещающийся от начальной точки (нижний круг) к точке выхода (верхний круг) и обученный на основе критерия близости к цели, будет склонен застревать в локальных тупиках. Кроме того, подобная обманчивая оценка приспособленности может воспрепятствовать алгоритму обучения на основе близости к цели найти успешный решатель лабиринтов.

Агент, перемещающийся по лабиринту, представляет собой робота, оборудованного набором датчиков, позволяющих ему обнаруживать близлежащие препятствия и определять направление к выходу из лабиринта. Перемещение робота осуществляется двумя приводами, влияющими на линейное и угловое движение корпуса робота. Приводы робота управляются нейросетью, которая получает данные от датчиков и выдает два управляющих сигнала на приводы.

5.3 Среда моделирования лабиринта

Среда моделирования лабиринта состоит из трех основных компонентов, которые реализованы в виде отдельных классов Python:

- `Agent` – класс, который хранит информацию, связанную с агентом навигатора лабиринта, задействованного в симуляции (см. подробности реализации в файле `agent.py`);
- `AgentRecordStore` – класс, который управляет хранением записей, относящихся к оценкам всех решающих агентов в ходе эволюционного процесса. Собранные записи можно использовать для анализа эволюционного процесса после его завершения (см. подробности реализации в файле `agent.py`);
- `MazeEnvironment` – класс, который содержит информацию о среде моделирования лабиринта. Этот класс также предоставляет методы, которые управляют средой моделирования, управляют положением решающего агента, выполняют обнаружение столкновений и генерируют входные данные для датчиков агента (см. подробности реализации в файле `maze_environment.py`).

В следующих разделах мы рассмотрим каждый компонент среды моделирования лабиринта более подробно.

5.3.1 Агент-решатель лабиринта

В этой главе мы рассмотрим задачу прохождения двухмерного лабиринта. Эту задачу легко визуализировать, и относительно легко написать симулятор робота-навигатора для двухмерного лабиринта. Основная цель робота – пройти по лабиринту к определенной цели за указанное количество шагов симуляции. Роботом управляет нейросеть, которая развивается в процессе нейроэволюции.

Алгоритм нейроэволюции начинается с очень простой начальной конфигурации нейросети, которая имеет только входные узлы для датчиков и выходные узлы для приводов и постепенно становится все более сложной, пока не будет найден успешный решатель лабиринтов. Эта задача усложняется особой конфигурацией лабиринта, в которой есть несколько тупиков, мешающих найти путь к цели и заманивающих агента в локальные оптимумы ландшафта приспособленности, как упоминалось ранее.

На рис. 5.2 представлено схематическое изображение агента-решателя, выполненного в виде робота и задействованного в моделировании решения задачи лабиринта.

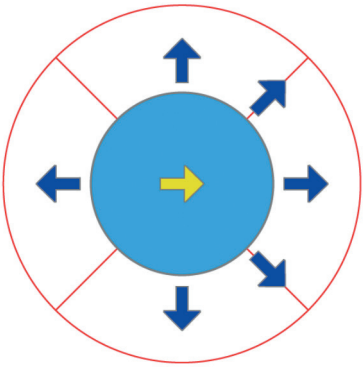


Рис. 5.2. Схематическое изображение навигационного агента

На этой схеме закрашенный круг обозначает твердый корпус робота. Стрелка внутри закрашенного круга показывает направление движения робота. Шесть стрелок вокруг закрашенного круга представляют шесть дальномерных датчиков, которые указывают расстояние до ближайшего препятствия в заданном направлении. Четыре сегмента внешнего круга обозначают четыре радарных датчика с секторным обзором, которые действуют как компас, указывающий направление к цели (выходу из лабиринта).

Специальный *радарный датчик* активируется, когда линия от точки цели до центра робота попадает в его *поле обзора* (field of view, FOV). Дальность обнаружения радарного датчика ограничена областью лабиринта, которая попадает в его поле зрения. Таким образом, в любой момент времени активирован один из четырех радарных датчиков, указывающий направление выхода из лабиринта.

Радарные датчики имеют следующие зоны обзора относительно курса робота:

Датчик	Поле обзора, градусы
Передний	315,0 ~ 405,0
Левый	45,0 ~ 135,0
Задний	135,0 ~ 225,0
Правый	225,0 ~ 315,0

Дальномерный датчик – это следящий луч, направленный от центра робота в определенном направлении. Активируется при пересечении с любым препятствием и возвращает расстояние до обнаруженного препятствия. Дальность обнаружения этого датчика определяется конкретным параметром конфигурации.

Дальномерные датчики робота отслеживают следующие направления относительно направления движения:

Датчик	Направление, градусы
Правый	–90,0
Передний правый	–45,0
Передний	0,0
Передний левый	45,0
Левый	90,0
Задний	–180,0

Движение робота контролируется двумя приводами, прикладывающими силы, которые поворачивают и/или приводят в движение корпус робота, то есть изменяют его линейную и/или угловую скорость.

В реализации робота-навигатора на языке Python есть несколько полей для хранения его текущего состояния и для поддержания состояний активности его датчиков:

```
def __init__(self, location, heading=0, speed=0,
             angular_vel=0, radius=8.0, range_finder_range=100.0):
    self.heading = heading
    self.speed = speed
    self.angular_vel = angular_vel
    self.radius = radius
    self.range_finder_range = range_finder_range
    self.location = location
    # Определяем датчики дальномера.
    self.range_finder_angles = [-90.0, -45.0, 0.0, 45.0, 90.0, -180.0]
    # Определяем радарные датчики
    self.radar_angles = [(315.0, 405.0), (45.0, 135.0),
                         (135.0, 225.0), (225.0, 315.0)]
    # Список состояний активности дальномеров
    self.range_finders = [None] * len(self.range_finder_angles)
    # Список состояний активности секторных радаров.
    self.radar = [None] * len(self.radar_angles)
```

Для изучения подробностей реализации кода рассмотрите файл `agent.py` в файловом архиве книги.

В предыдущем блоке кода показан конструктор по умолчанию класса `Agent`, в котором инициализированы все поля агента. Симулятор среды лабиринта будет использовать эти поля для хранения текущего состояния агента на каждом шаге симуляции.

5.3.2 Реализация среды моделирования лабиринта

Чтобы смоделировать поведение решающего агента, исследующего лабиринт, нам нужно определить среду, которая управляет конфигурацией лабиринта, отслеживает положение агента, исследующего лабиринт, и обеспечивает входные данные для массивов данных датчика навигационного робота.

Все эти функции помещаются в один логический блок, который инкапсулирован в класс Python MazeEnvironment, имеющий следующие поля:

```
def __init__(self, agent, walls, exit_point, exit_range=5.0):
    self.walls = walls
    self.exit_point = exit_point
    self.exit_range = exit_range
    # Агент-решатель лабиринта.
    self.agent = agent
    # Флаг индикации о том, что выход найден.
    self.exit_found = False
    # Начальное расстояние от агента до выхода.
    self.initial_distance = self.agent_distance_to_exit()
```

В предыдущем коде показан конструктор по умолчанию класса MazeEnvironment с инициализацией всех его полей:

- конфигурация лабиринта определяется списком стен и точки выхода. Стены – это списки отрезков; каждый отрезок линии представляет определенную стену в лабиринте, а точка выхода exit_point – это местоположение выхода из лабиринта;
- в поле exit_range хранится значение расстояния до точки выхода, определяющее зону выхода. Мы считаем, что агент успешно прошел лабиринт, когда его позиция находится в зоне выхода;
- поле agent содержит ссылку на инициализированный класс агента, описанный в предыдущем разделе, который определяет начальное местоположение агента в лабиринте и прочие поля агента;
- поле initial_distance хранит расстояние от начальной позиции агента до точки выхода из лабиринта. Это значение будет позже использовано для расчета показателя приспособленности агента.

Генерация данных датчиков

Агент, проходящий лабиринт, управляется нейросетью, которой необходимо иметь данные датчиков на входе для формирования соответствующих управляющих сигналов на выходе. Как мы уже упоминали, робот-навигатор оснащен массивом датчиков двух типов:

- шесть *дальномерных датчиков* для предотвращения столкновений со стенами лабиринта, которые показывают расстояние до ближайшего препятствия в определенном направлении;
- четыре *секторных радарных датчика*, которые указывают направление к точке выхода из лабиринта из любого места в лабиринте.

Показания датчиков необходимо обновлять на каждом шаге моделирования, а класс MazeEnvironment предоставляет два ссылочных метода, которые обновляют датчики обоих типов.

Массив дальномерных датчиков обновляется следующим образом (см. функцию update_rangefinder_sensors):

```
for i, angle in enumerate(self.agent.range_finder_angles):
    rad = geometry.deg_to_rad(angle)
    projection_point = geometry.Point(
```

```
            x = self.agent.location.x + math.cos(rad) * \
                self.agent.range_finder_range,
            y = self.agent.location.y + math.sin(rad) * \
                self.agent.range_finder_range
        )
        projection_point.rotate(self.agent.heading,
                                self.agent.location)
        projection_line = geometry.Line(a = self.agent.location,
                                        b = projection_point)
        min_range = self.agent.range_finder_range
        for wall in self.walls:
            found, intersection = wall.intersection(projection_line)
            if found:
                found_range = intersection.distance(
                                             self.agent.location)
                if found_range < min_range:
                    min_range = found_range
        # Сохраняем расстояние до ближайшего препятствия.
        self.agent.range_finders[i] = min_range
```

Этот код перечисляет все направления дальномерных датчиков, которые определяются углами направления (см. инициализацию поля `range_finder_angles` в конструкторе класса `Agent`). Затем для каждого направления выстраивается линия проекции, начиная с текущей позиции робота и с длиной, равной дальности обнаружения дальномера. После этого линия проверяется на предмет того, пересекает ли она стены лабиринта. Если обнаружено несколько пересечений, расстояние до ближайшей стены сохраняется как текущее значение для конкретного дальномера. В противном случае в качестве текущего значения будет сохранена величина максимальной дальности обнаружения.

Массив секторных радарных датчиков обновляется при помощи кода в классе `MazeEnvironment`:

```
def update_radars(self):
    target = geometry.Point(self.exit_point.x, self.exit_point.y)
    target.rotate(self.agent.heading, self.agent.location)
    target.x -= self.agent.location.x
    target.y -= self.agent.location.y
    angle = target.angle()
    for i, r_angles in enumerate(self.agent.radar_angles):
        self.agent.radar[i] = 0.0 # reset specific radar
        if (angle >= r_angles[0] and angle < r_angles[1]) or
           (angle + 360 >= r_angles[0] and angle + 360 < r_angles[1]):
           # Запуск радара.
           self.agent.radar[i] = 1.0
```

Этот код создает копию точки выхода из лабиринта и поворачивает ее относительно курса и положения агента в глобальной системе координат. Затем целевая точка транслируется в локальную систему координат агента, исследующего лабиринт; агент находится в начале координат. После этого мы вычисляем угол вектора, образованного от начала координат до целевой точки

в локальной системе координат агента. Этот угол является азимутом к точке выхода из лабиринта из текущей позиции агента. Когда азимутальный угол найден, мы перечисляем зарегистрированные радарные датчики, чтобы найти тот, у которого текущий азимутальный угол попадает в поле зрения. Соответствующий радарный датчик активируется путем установки его значения в 1, в то время как другие радарные датчики деактивируются путем обнуления их значений.

Обновление позиции агента

Положение агента в лабиринте необходимо обновлять на каждом этапе моделирования после получения соответствующих сигналов управления от нейросети контроллера. Для обновления позиции агента выполняется следующий код:

```
def update(self, control_signals):
    if self.exit_found:
        return True # Maze exit already found
    self.apply_control_signals(control_signals)
    vx = math.cos(geometry.deg_to_rad(self.agent.heading)) * \
                self.agent.speed
    vy = math.sin(geometry.deg_to_rad(self.agent.heading)) * \
                self.agent.speed
    self.agent.heading += self.agent.angular_vel
    if self.agent.heading > 360:
        self.agent.heading -= 360
    elif self.agent.heading < 0:
        self.agent.heading += 360
    new_loc = geometry.Point(
        x = self.agent.location.x + vx,
        y = self.agent.location.y + vy
    )
    if not self.test_wall_collision(new_loc):
        self.agent.location = new_loc
    self.update_rangefinder_sensors()
    self.update_radars()
    distance = self.agent_distance_to_exit()
    self.exit_found = (distance < self.exit_range)
    return self.exit_found
```

Функция `update(self, control_signals)` определена в классе `MazeEnvironment` и вызывается на каждом шаге моделирования. Она получает список с управляющими сигналами в качестве входных данных и возвращает логическое значение, указывающее, достиг ли агент зоны выхода после обновления своей позиции.

Код в начале этой функции применяет полученные управляющие сигналы к текущим значениям угловой и линейной скоростей агента следующим образом (см. функцию `apply_control_signals(self, control_signals)`):

```
self.agent.angular_vel += (control_signals[0] - 0.5)
self.agent.speed += (control_signals[1] - 0.5)
```

Затем вычисляются компоненты скорости x и y вместе с направлением условной «передней части» агента и используются для оценки его нового положения в лабиринте. Если эта новая позиция не сталкивается ни с одной из стен лабиринта, то она назначается агенту и становится его текущей позицией:

```
vx = math.cos(geometry.deg_to_rad(self.agent.heading)) * \
             self.agent.speed
vy = math.sin(geometry.deg_to_rad(self.agent.heading)) * \
             self.agent.speed
self.agent.heading += self.agent.angular_vel
if self.agent.heading > 360:
    self.agent.heading -= 360
elif self.agent.heading < 0:
    self.agent.heading += 360
new_loc = geometry.Point(
    x = self.agent.location.x + vx,
    y = self.agent.location.y + vy
)
if not self.test_wall_collision(new_loc):
    self.agent.location = new_loc
```

Далее новая позиция агента используется в функциях, которые обновляют дальномерные и радарные датчики, получая значения новых входов датчиков для следующего временного шага:

```
self.update_rangefinder_sensors()
self.update_radars()
```

Наконец, следующая функция проверяет, достиг ли агент выхода из лабиринта, который определяется круглой областью вокруг точки выхода с радиусом, равным значению поля exit_range:

```
distance = self.agent_distance_to_exit()
self.exit_found = (distance < self.exit_range)
return self.exit_found
```

Если выход из лабиринта был достигнут, значение поля exit_found устанавливается равным True, чтобы сообщить об успешном решении задачи, и это значение возвращается из вызова функции.

Для подробного изучения деталей реализации обратитесь к файлу maze_environment.py в файловом архиве книги.

5.3.3 Хранение записей агента

После завершения эксперимента нам понадобится оценка и визуализация того, как каждый отдельный решающий агент работал в течение эволюционного процесса на протяжении всех поколений. Для этого мы собираем дополнительные статистические данные о каждом агенте после запуска модели прохождения лабиринта в течение определенного количества временных шагов.

Коллекция записей агента опосредуется двумя классами Python: `AgentRecord` и `AgentRecordStore`.

Класс `AgentRecord` состоит из нескольких полей данных, как видно из конструктора класса:

```
def __init__(self, generation, agent_id):
    self.generation = generation
    self.agent_id = agent_id
    self.x = -1
    self.y = -1
    self.fitness = -1
    self.hit_exit = False
    self.species_id = -1
    self.species_age = -1
```

Поля имеют следующее назначение:

- `generation` содержит идентификатор поколения, когда была создана запись агента;
- `agent_id` – уникальный идентификатор агента;
- `x` и `y` – позиция агента в лабиринте после завершения симуляции;
- `fitness` – итоговая оценка приспособленности агента;
- `hit_exit` – это флаг, который указывает, достиг ли агент области выхода из лабиринта;
- `species_id` и `species_age` – идентификатор и возраст вида, к которому относится агент.

Класс `AgentRecordStore` содержит список записей агента и предоставляет функции для сохранения собранных записей в определенный файл и чтения из него.

Для изучения деталей реализации рассмотрите файл `agent.py` из файлового архива книги.

Новые экземпляры `AgentRecord` добавляются в хранилище после оценки приспособленности генома, при помощи функции `eval_fitness(genome_id, genome, config, time_steps=400)`, реализованной в файле `maze_experiment.py`. Это делается с помощью следующего кода:

```
def eval_fitness(genome_id, genome, config, time_steps=400):
    maze_env = copy.deepcopy(trialSim.orig_maze_environment)
    control_net = neat.nn.FeedForwardNetwork.create(genome, config)
    fitness = maze.maze_simulation_evaluate(
              env=maze_env, net=control_net, time_steps=time_steps)
    record = agent.AgentRecord(
        generation=trialSim.population.generation,
        agent_id=genome_id)
    record.fitness = fitness
    record.x = maze_env.agent.location.x
    record.y = maze_env.agent.location.y
    record.hit_exit = maze_env.exit_found
```

```
record.species_id = trialSim.population.species.\
                                    get_species_id(genome_id)
record.species_age = record.generation - \
    trialSim.population.species.get_species(genome_id).created
trialSim.record_store.add_record(record)
return fitness
```

Этот код сначала создает полную копию исходной среды лабиринта, чтобы избежать взаимного влияния между оценочными прогонами. После этого он создает контрольную нейросеть из заданного генома, используя конфигурацию, предоставленную NEAT, и начинает оценочное моделирование лабиринта для заданного количества временных шагов. Возвращенная оценка приспособленности агента вместе с другой статистикой затем сохраняется в конкретном экземпляре `AgentRecord` и добавляется в хранилище записей.

Записи, собранные во время одной пробной версии эксперимента, будут сохранены в файле `data.pickle` в каталоге `output` и использованы для визуализации работоспособности всех оцененных агентов.

Для изучения деталей реализации рассмотрите файл `maze_experiment.py` в файловом архиве книги.

5.3.4 Визуализация записей агента

После того как в ходе нейроэволюционного процесса будут собраны все оценочные записи всех агентов, нам будет полезно визуализировать записанные данные, чтобы получить представление о положении дел в нашей эволюции. Визуализация должна включать конечные позиции всех решающих агентов и позволять устанавливать пороговое значение приспособленности вида для контроля за тем, какие виды будут добавлены к соответствующему участку. Мы решили представить собранные записи агентов на двух графиках, нарисованных один над другим. Верхний график предназначен для записей агентов, которые относятся к видам, у которых показатель приспособленности больше или равен указанному пороговому значению, а нижний график – для остальных записей.

Визуализация записей агента реализована в новых методах в скрипте `visualize.py`. Вы уже должны быть знакомы с этим скриптом по предыдущим экспериментам, описанным в этой книге.

Рассмотрите определение функции `draw_maze_records(maze_env, records, best_threshold=0.8, filename=None, view=False, show_axes=False, width=400, height=400)` в файле `visualize.py` в файловом архиве книги.

5.4 Определение целевой функции с использованием показателя приспособленности

В этом разделе вы узнаете о создании успешных агентов, которые проходят лабиринт, используя *целеориентированную целевую функцию* (goal-oriented objective function) для руководства эволюционным процессом. Целевая функция

этого типа основана на оценке показателя приспособленности решателя лабиринта путем измерения расстояния между его конечным положением и целью (выходом из лабиринта) после выполнения 400 этапов моделирования. Иными словами, целенаправленная целевая функция ориентирована на конкретную цель и зависит исключительно от конечной цели эксперимента – достижения области выхода из лабиринта.

В следующей главе мы рассмотрим другой подход к оптимизации поиска решения, который основан на методе оптимизации *поиском новизны* (novelty search, NS). Метод оптимизации поиском новизны основан на изучении новых конфигураций решающего агента в процессе эволюции и не включает определение конечной цели (в данном случае выхода из лабиринта) в определение целевой функции. Мы покажем, что подход поиска новизны может превзойти традиционную целенаправленную целевую функцию, которую мы используем в этой главе.

Целенаправленная целевая функция, задействованная в этом эксперименте, определяется следующим образом. Во-первых, нам нужно определить функцию ошибки как евклидово расстояние между конечной позицией агента в конце симуляции и позицией выхода из лабиринта:

$$\mathcal{L} = \sqrt{\sum_{i=1}^{2} \left(a_i - b_i\right)^2}.$$

Здесь $\mathcal{L}$ – функция ошибки, a – координаты конечной позиции агента и b – координаты выхода из лабиринта. В этом эксперименте мы используем двумерный лабиринт, поэтому у координат есть два значения, по одному для каждого измерения.

Имея функцию ошибки, мы можем определить функцию приспособленности:

$$\mathcal{F} = \begin{cases} 1.0 & L \le R_{exit} \\ \mathcal{F}_n & \textit{остальные} \end{cases}.$$

Здесь R_{exit} – радиус зоны выхода вокруг точки выхода из лабиринта, а $\mathcal{F}_n$ – нормализованный показатель приспособленности, который определяется следующим образом:

$$\mathcal{F}_n = \frac{\mathcal{L} - D_{init}}{D_{init}}.$$

D_{init} – это начальное расстояние от решающего агента до выхода из лабиринта в начале навигационного моделирования.

Уравнение нормализует показатель приспособленности, чтобы он располагался в диапазоне (0, 1], но может вернуть отрицательные значения в тех редких случаях, когда конечная позиция агента находится далеко и от его начальной позиции, и от выхода из лабиринта. Чтобы избежать отрицательных значений, мы будем применять к нормализованному показателю приспособленности следующие поправки:

$$\mathcal{F}_n = \begin{cases} 0{,}01 & \mathcal{F}_n \le 0 \\ \mathcal{F}_n & остальное \end{cases}.$$

Когда показатель приспособленности меньше или равен `0`, ему будет присвоено минимальное поддерживаемое значение `0,01`; в противном случае он останется как есть. Мы выбрали минимальный показатель приспособленности выше нуля, чтобы дать каждому геному шанс на размножение.

Следующий код в Python реализует целенаправленную целевую функцию:

```
# Вычисляем показатель приспособленности на основе расстояния до выхода.
fitness = env.agent_distance_to_exit()
if fitness <= self.exit_range:
    fitness = 1.0
else:
    # Нормализуем показатель приспособленности к интервалу (0,1].
    fitness = (env.initial_distance - fitness) / \
              env.initial_distance
    if fitness <= 0.01:
        fitness = 0.01
```

Сначала код вызывает функцию `agent_distance_to_exit()`, которая вычисляет евклидово расстояние от текущей позиции агента до выхода из лабиринта и использует полученное значение в качестве первого приближения оценки приспособленности. После этого оценка приспособленности (расстояние до выхода из лабиринта) сравнивается со значением диапазона выхода. Если оценка приспособленности меньше или равна значению `self.exit_range`, мы присваиваем ей окончательное значение `1.0`. В противном случае нормализованный показатель приспособленности рассчитывается путем деления разницы между конечным и начальным расстояниями от агента до выхода из лабиринта на начальное расстояние. Иногда это может привести к отрицательному значению нормализованного значения приспособленности, которое корректируется путем сравнения значения приспособленности с `0.01` и внесения необходимых поправок.

С деталями реализации можно ознакомиться в файле `maze_environment.py`.

5.5 Проведение эксперимента с простой конфигурацией лабиринта

Итак, мы начинаем наш эксперимент по созданию успешного агента навигации, который исследует простую конфигурацию лабиринта. Подобная конфигурация лабиринта, несмотря на упомянутые ранее обманчивые локальные тупики оптимальности, подразумевает относительно прямой путь от начальной точки к конечной точке.

На рис. 5.3 изображена конфигурация простого лабиринта, задействованного в текущем эксперименте.

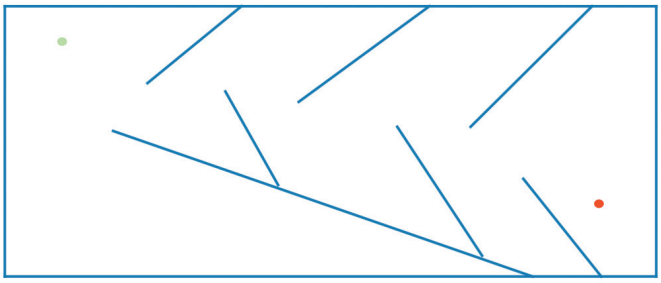


Рис. 5.3. Конфигурация простого лабиринта

Лабиринт имеет две фиксированные позиции, отмеченные закрашенными кружками. Верхний левый кружок обозначает начальную позицию агента (решателя лабиринта). Нижний правый кружок отмечает точное местоположение выхода из лабиринта, который должен быть найден решателем. Задача решателя лабиринта – добраться до окрестности точки выхода из лабиринта, то есть оказаться достаточно близко от нее, в пределах расстояния, заданного в конфигурации.

5.5.1 Выбор гиперпараметров

Согласно определению целевой функции, максимальное значение показателя приспособленности агента, которое может быть получено при достижении заданной окрестности выхода из лабиринта, составляет `1.0`. Мы также ожидаем, что начальная конфигурация нейросети контроллера является более сложной, чем в предыдущих экспериментах, описанных ранее в книге, и это повлияет на скорость выполнения алгоритма. Из-за этого на среднем ПК потребуется слишком много времени, чтобы завершить алгоритм нейроэволюции с заметно большей популяцией генома. Но в то же время стоящая перед нами задача намного сложнее, чем в предыдущих экспериментах, и для успешного поиска решения необходимо использовать более широкую область поиска. Методом проб и ошибок мы обнаружили, что численность популяции может быть задана равной `250`.

Следующий раздел файла конфигурации содержит определение параметров, которые мы только что обсудили:

```
[NEAT]
fitness_criterion = max
fitness_threshold = 1.0
pop_size = 250
reset_on_extinction = False
```

Начальная конфигурация фенотипа нейросети включает 10 входных узлов, 2 выходных узла и 1 скрытый узел. Узлы входа и выхода соответствуют входным датчикам и выходам управляющего сигнала. Скрытый узел пред-

назначен для введения нелинейности с самого начала нейроэволюционного процесса и экономии времени эволюции на добавление этого узла. Конфигурация нейросети выглядит следующим образом:

```
num_hidden = 1
num_inputs = 10
num_outputs = 2
```

Для расширения области поиска решений нам нужно увеличить видовое разнообразие популяции, чтобы попробовать разные конфигурации генома в течение ограниченного числа поколений. Это может быть сделано либо путем уменьшения порога совместимости, либо путем увеличения значений коэффициентов, которые используются для расчета показателей совместимости генома.

В этом эксперименте мы использовали обе поправки, потому что ландшафт функции приспособленности обманчив и нам нужно подчеркнуть даже крошечные изменения в конфигурации генома, чтобы создать новые виды. На это влияют следующие параметры конфигурации:

```
[NEAT]
compatibility_disjoint_coefficient = 1.1
[DefaultSpeciesSet]
compatibility_threshold = 3.0
```

Мы особенно заинтересованы в создании оптимальной конфигурации нейросети, которая имеет минимальное количество скрытых узлов и связей. Оптимальная конфигурация менее затратна в вычислительном отношении во время обучения нейроэволюционным методом, а также во время фазы вывода в симуляторе прохождения лабиринта. Оптимальную конфигурацию нейросети можно получить, уменьшив вероятность добавления новых узлов, как показано в следующем фрагменте из файла конфигурации NEAT:

```
node_add_prob           = 0.1
node_delete_prob        = 0.1
```

Наконец, мы позволяем нейроэволюционному процессу использовать не только нейросеть с прямой связью, но и рекуррентные нейросети. Разрешая рекуррентные соединения, мы даем возможность нейросети иметь память и стать конечным автоматом. Это полезно для эволюционного процесса. Следующий гиперпараметр конфигурации влияет на этот фактор:

```
feed_forward            = False
```

Гиперпараметры, описанные в этом разделе, оказались полезными для алгоритма NEAT, который используется в эксперименте для обучения в течение ограниченного числа поколений успешного агента, проходящего лабиринт.

Полный список гиперпараметров, задействованных в эксперименте с простым лабиринтом, можно найти в файле `maze_config.ini` в файловом архиве книги.

5.5.2 Файл конфигурации лабиринта

Конфигурация лабиринта для нашего эксперимента представлена в виде простого текста. Этот файл загружается в среду моделирования, и в ней создается лабиринт с соответствующей конфигурацией. Файл конфигурации имеет содержимое следующего вида:

```
11
30 22
0
270 100
5 5 295 5
295 5 295 135
295 135 5 135
...
```

Формат файла конфигурации лабиринта следующий:

- первая строка содержит количество стен в лабиринте;
- вторая строка определяет начальную позицию агента (*x, y*);
- третья строка обозначает начальный курс агента в градусах;
- четвертая строка содержит позицию выхода из лабиринта (*x*, *y*);
- следующие строки определяют стены лабиринта.

Стена лабиринта представлена в виде отрезка, где первые два числа определяют координаты начальной точки, а последние два числа – координаты конечной точки. Начальная позиция агента и выход из лабиринта представлены в виде двух чисел, обозначающих координаты *x* и *y* точки в двумерном пространстве.

5.5.3 Настройка рабочей среды

Рабочую среду эксперимента по прохождению простого лабиринта можно настроить с помощью следующих команд, введенных в выбранном вами приложении терминала:

```
$ conda create --name maze_objective_neat python=3.5
$ conda activate maze_objective_neat
$ pip install neat-python==0.92
$ conda install matplotlib
$ conda install graphviz
$ conda install python-graphviz
```

Эти команды создают и активируют виртуальную среду `maze_objective_neat` с Python 3.5. После этого устанавливается библиотека NEAT-Python версии 0.92, а также другие зависимости, используемые нашими утилитами визуализации.

Теперь мы готовы приступить к реализации движка эксперимента.

5.5.4 Реализация движка эксперимента

Движок эксперимента представлен в файле `maze_experiment.py`, к которому вы должны обратиться для получения полной информации о реализации. Этот скрипт Python предоставляет функции для чтения аргументов команд-

ной строки, для настройки и запуска процесса нейроэволюции, а также для отображения результатов эксперимента после его завершения. Кроме того, он включает в себя реализацию функций обратного вызова для оценки приспособленности геномов, принадлежащих к определенной популяции. Эти функции обратного вызова будут предоставлены окружению библиотеки NEAT-Python во время инициализации.

Далее мы обсудим основные части реализации эксперимента, которые ранее не рассматривались в этой главе.

1. Начнем с инициализации среды моделирования лабиринта следующими строками:

```
maze_env_config = os.path.join(local_dir, '%s_maze.txt' %
                               args.maze)
maze_env = maze.read_environment(maze_env_config)
```

Параметр `args.maze` ссылается на аргумент командной строки, предоставленный пользователем при запуске скрипта Python, и относится к типу среды лабиринта, с которой мы хотели бы поэкспериментировать. Он может иметь два значения: `medium` (средний) и `hard` (сложный). Первый относится к простой конфигурации лабиринта, которую мы используем в этом эксперименте.

2. После этого мы задаем конкретный начальный номер для генератора случайных чисел, создаем объект конфигурации NEAT, а затем объект `neat.Population`, используя только что созданный объект конфигурации:

```
seed = 1559231616
random.seed(seed)
config = neat.Config(neat.DefaultGenome,
                     neat.DefaultReproduction,
                     neat.DefaultSpeciesSet,
                     neat.DefaultStagnation,
                     config_file)
p = neat.Population(config)
```

Бывает, что случайное начальное значение, найденное в эксперименте по балансировке двойного маятника, также подходит для этого эксперимента. Можно предположить, что мы нашли случайный аттрактор, специфичный для стохастического процесса, реализуемого библиотекой NEAT-Python. Позже в книге мы проверим, верно ли это и для других экспериментов.

3. Теперь мы готовы создать соответствующую среду моделирования лабиринта и сохранить ее как глобальную переменную, чтобы упростить доступ к ней из функции обратного вызова оценки приспособленности:

```
global trialSim
trialSim = MazeSimulationTrial(maze_env=maze_env, population=p)
```

Объект `MazeSimulationTrial` содержит поля, которые обеспечивают доступ к исходной среде моделирования лабиринта и хранилищу записей, используемому для сохранения результатов оценки агентов, решающих лабиринт.

При каждом вызове функции обратного вызова для оценки приспособленности `eval_fitness(genome_id, genome, config, time_steps=400)` исходная среда моделирования лабиринта дублируется и используется для моделирования прохождения лабиринта конкретным агентом в течение 400 временных шагов. После этого полная статистика об агенте, проходящем лабиринт, включая его окончательное положение в лабиринте, будет собрана из среды и добавлена в хранилище записей.

4. Следующий код стал стандартным для наших экспериментов и связан с добавлением различных репóртеров статистики:

```
p.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
p.add_reporter(stats)
p.add_reporter(neat.Checkpointer(5,
               filename_prefix='%s/maze-neat-checkpoint-' %
               trial_out_dir))
```

Репортеры используются для вывода промежуточных результатов процесса нейроэволюции в консоль, а также для сбора более подробной статистики, которая будет отображаться после завершения процесса.

5. Наконец, мы запускаем процесс нейроэволюции для указанного числа поколений и проверяем, было ли найдено решение:

```
start_time = time.time()
best_genome = p.run(eval_genomes, n=n_generations)
elapsed_time = time.time() - start_time
solution_found = (best_genome.fitness >= \
                  config.fitness_threshold)
if solution_found:
    print("SUCCESS: The stable maze solver controller was found!!!")
else:
    print("FAILURE: Failed to find the stable maze solver controller!!!")
```

Мы предполагаем, что решение было найдено, если лучший геном, возвращаемый библиотекой NEATPython, имеет показатель приспособленности, который больше или равен пороговому значению приспособленности, установленному в файле конфигурации (`1.0`). Чтобы показать в выводе консоли, сколько времени потребовалось для завершения процесса, производится подсчет затраченного времени.

Оценка приспособленности генома

Функция обратного вызова для оценки приспособленности всех геномов, принадлежащих к определенной популяции организмов, реализована следующим образом:

```
def eval_genomes(genomes, config):
    for genome_id, genome in genomes:
        genome.fitness = eval_fitness(genome_id, genome, config)
```

5.5.5 Проведение эксперимента по навигации в простом лабиринте

Имея симулятор решателя лабиринта, движок эксперимента и функцию оценки приспособленности, мы готовы начать эксперимент по поиску решателя лабиринта. Убедитесь, что вы скопировали все нужные скрипты Python и файлы конфигурации (`maze_config.ini` и `medium_maze.txt`) в рабочий каталог.

После этого войдите в рабочий каталог и выполните следующую команду в выбранном вами приложении терминала:

```
$ python maze_experiment.py -m medium -g 150
```

Не забудьте активировать соответствующую виртуальную среду при помощи команды

```
conda activate maze_objective_neat.
```

Предыдущая команда загружает простую конфигурацию лабиринта из файла `medium_maze.txt` и создает соответствующую среду моделирования лабиринта. После этого она запускает нейроэволюционный процесс под управлением алгоритма NEAT, используя гиперпараметры, указанные в файле `maze_config.ini`. Алгоритм NEAT применяет среду моделирования лабиринта для оценки приспособленности каждого генома, созданного в ходе нейроэволюции в течение `150` поколений (`-g` в аргументах командной строки).

После `144` поколений эволюции успешный агент-решатель лабиринтов был найден в поколении `145`. Вывод консоли для последнего поколения выглядит следующим образом.

1. Сначала идет общая статистика по геному популяции:

```
****** Running generation 145 ******

Maze solved in 388 steps
Population's average fitness: 0.24758 stdev: 0.25627
Best fitness: 1.00000 - size: (3, 11) - species 7 - id 35400

Best individual in generation 145 meets fitness threshold -
complexity: (3, 11)
```

2. Далее следует конфигурация генома, кодирующего нейросеть успешного решателя:

```
Best genome:
Key: 35400
Fitness: 1.0
Nodes:
0 DefaultNodeGene(key=0, bias=5.534849614521037, response=1.0,
activation=sigmoid, aggregation=sum)
1 DefaultNodeGene(key=1, bias=1.8031133229851957, response=1.0,
activation=sigmoid, aggregation=sum)
```

```
158 DefaultNodeGene(key=158, bias=-1.3550878188609456,
response=1.0, activation=sigmoid, aggregation=sum)
Connections:
DefaultConnectionGene(key=(-10, 158), weight=-1.6144052085440168,
enabled=True)
DefaultConnectionGene(key=(-8, 158), weight=-1.1842193888036392,
enabled=True)
DefaultConnectionGene(key=(-7, 0), weight=-0.3263706518456319,
enabled=True)
DefaultConnectionGene(key=(-7, 1), weight=1.3186165993348418,
enabled=True)
DefaultConnectionGene(key=(-6, 0), weight=2.0778575294986945,
enabled=True)
DefaultConnectionGene(key=(-6, 1), weight=-2.9478037554862824,
enabled=True)
DefaultConnectionGene(key=(-6, 158), weight=0.6930281879212032,
enabled=True)
DefaultConnectionGene(key=(-4, 1), weight=-1.9583885391583729,
enabled=True)
DefaultConnectionGene(key=(-3, 1), weight=5.5239054588484775,
enabled=True)
DefaultConnectionGene(key=(-1, 0), weight=0.04865917999517305,
enabled=True)
DefaultConnectionGene(key=(158, 0), weight=0.6973191076874032,
enabled=True)
SUCCESS: The stable maze solver controller was found!!!
Record store file: out/maze_objective/medium/data.pickle
```

В выводе консоли вы можете видеть, что во время эволюции был найден успешный решатель лабиринта, который смог достичь области выхода из лабиринта за 388 шагов из отведенных 400. Конфигурация нейросети успешного решателя содержит 2 выходных узла и 1 скрытый узел, с 11 связями между этими узлами и входами. Окончательная конфигурация нейросети показана на рис. 5.4.

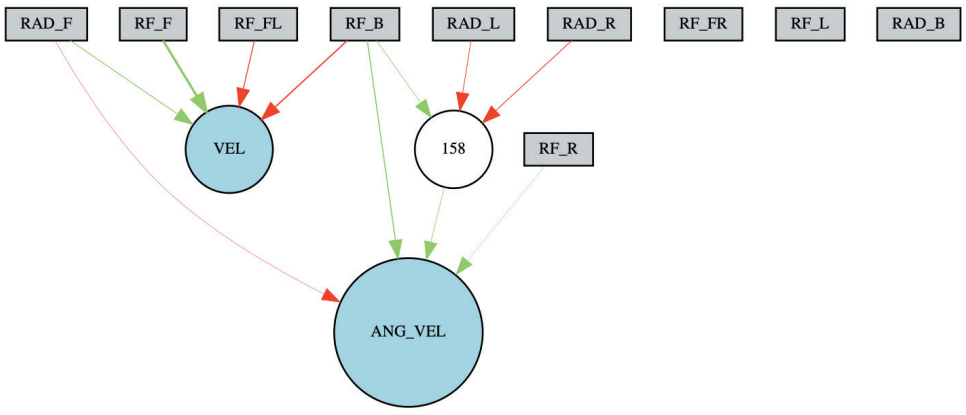


Рис. 5.4. Конфигурация нейросети, соответствующей успешному решателю простого лабиринта

Интересно взглянуть на граф нейросети с точки зрения влияния различных входов датчиков на выходные управляющие сигналы. Мы можем видеть, что конфигурация нейросети полностью игнорирует входные сигналы от переднего и левого дальномерных датчиков (RF_FR и RF_L) и от секторного радарного датчика RAD_B робота. В то же время линейные и угловые скорости робота зависят от уникальных комбинаций других датчиков.

Кроме того, мы можем видеть агрегацию левого и правого радарных датчиков (RAD_L и RAD_R) с дальномером RF_B через скрытый узел, который затем ретранслирует агрегированный сигнал узлу, управляющему угловой скоростью. Если мы посмотрим на изображение простой конфигурации лабиринта (рис. 5.3), то подобная агрегация выглядит довольно естественной. Она позволяет роботу развернуться и продолжить исследовать лабиринт, когда он застрял в одном из тупиков, где расположены локальные оптимумы.

Оценка приспособленности агентов по поколениям показана на рис. 5.5.

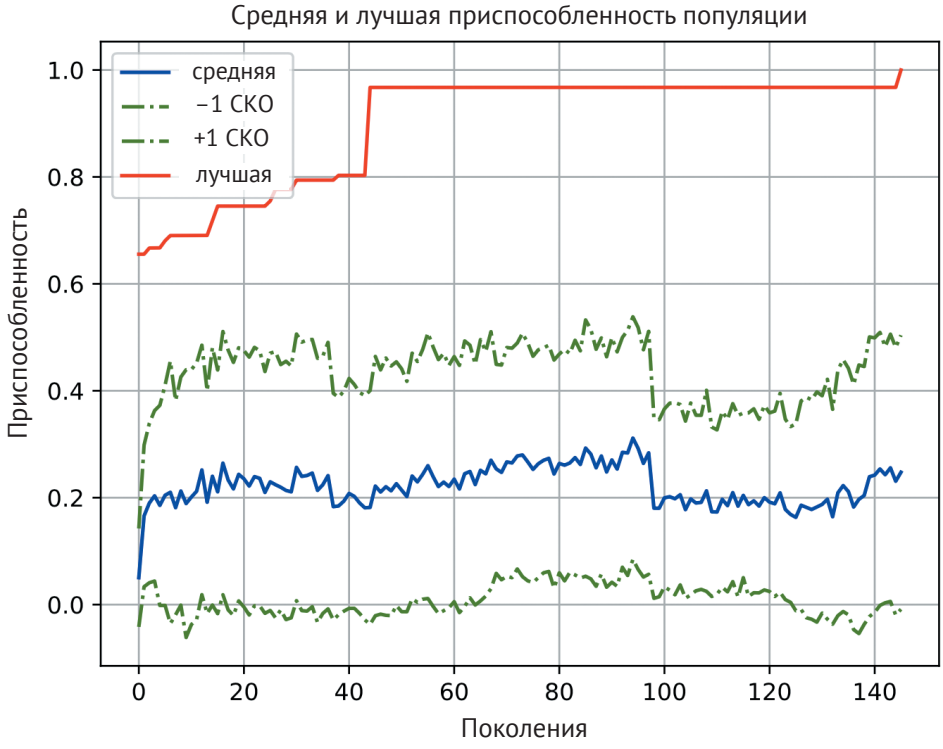


Рис. 5.5. Средние оценки приспособленности агентов по поколениям

На этом графике мы можем видеть, что эволюционный процесс смог произвести довольно успешных агентов, способных пройти лабиринт, в поколении 44 с оценкой приспособленности 0,96738. Но потребовалось еще 100 поколений, чтобы развить геном, который кодирует нейросеть успешного агента.

Кроме того, интересно отметить, что повышение производительности в поколении 44 генерируется видами с ID 1, но геном успешного решателя принад-

лежит виду с ID 7, который даже не был известен во время первого всплеска. Виды, породившие чемпиона, появились спустя 12 поколений и оставались в популяции до конца, сохраняя полезную мутацию и совершенствуясь на ее основе.

Видообразование в течение нескольких поколений показано на рис. 5.6.

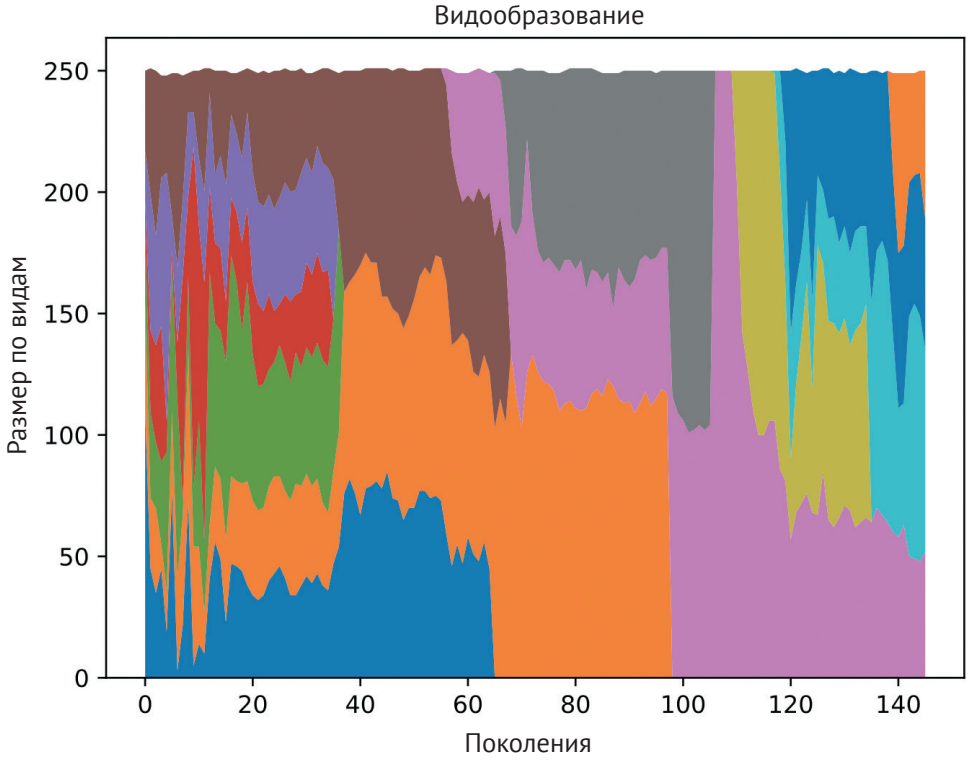


Рис. 5.6. Видообразование по поколениям

На графике видообразования мы наблюдаем вид с ID 7, отмеченный розовым. Этот вид в конечном итоге произвел геном успешного решателя в ходе эволюционного процесса. Размер вида 7 значительно варьируется на протяжении всей его жизни, и в свое время он был единственным видом во всей популяции в течение нескольких поколений (от 105 до 108).

Визуализация записи агента

В этом эксперименте мы представили новый метод визуализации, который позволяет нам визуально различать эффективность различных видов в эволюционном процессе. Визуализация может быть выполнена с помощью следующей команды, выполненной из рабочего каталога эксперимента:

```
$ python visualize.py -m medium -r out/maze_objective/medium/data.pickle --width 300
--height 150
```

Команда загружает записи об оценке приспособленности каждого агента, проходящего лабиринт во время эволюции, которые хранятся в файле `data`.

pickle. После этого визуализатор рисует конечные позиции агентов на карте лабиринта в конце симуляции прохождения. Конечная позиция каждого агента представлена в виде цветного кружка. Цвет кружка обозначает вид, к которому принадлежит конкретный агент. Каждый вид, полученный в ходе эволюции, имеет уникальный цветовой код. Результаты этой визуализации можно увидеть на рис. 5.7.

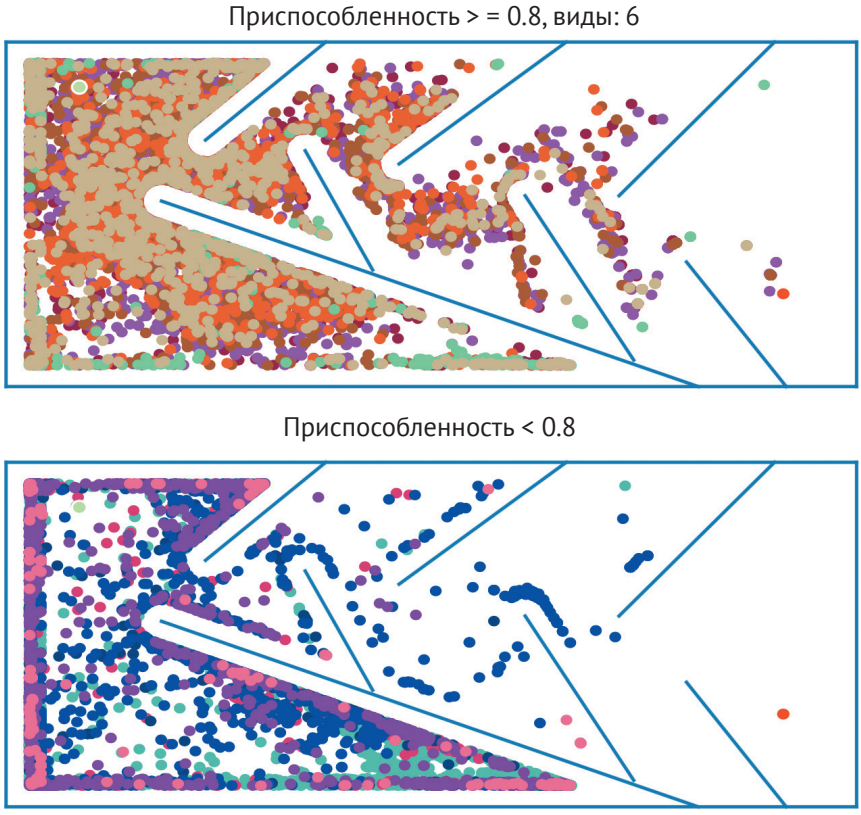


Рис. 5.7. Визуализация процесса развития решающего агента

Чтобы сделать визуализацию более информативной, мы ввели порог приспособленности для фильтрации наиболее эффективных видов. Верхняя половина рисунка показывает конечные позиции решающих агентов, принадлежащих к видам-чемпионам (показатель приспособленности выше 0,8). Как вы можете видеть, организмы, принадлежащие к этим шести видам, являются активными исследователями, у которых есть гены, провоцирующие поиск в неизвестных местах в лабиринте. Их конечные местоположения почти равномерно распределены по области лабиринта вокруг начальной точки и имеют низкую плотность в тупиках локальных оптимумов.

В то же время на нижней половине рисунка вы можете увидеть, что эволюционные неудачники демонстрируют более консервативное поведение, концентрируясь в основном возле стен в стартовой зоне и в самой выраженной

области локального оптимума – самом большом тупике, который находится в нижней части лабиринта.

5.6 Упражнения

1. Попробуйте увеличить параметр `compatibility_disjoint_coefficient` в файле `maze_config.ini` и запустить эксперимент с новыми настройками. Какое влияние эта модификация оказывает на количество видов, произведенных в ходе эволюции? Способен ли процесс нейроэволюции найти успешный решатель лабиринтов?
2. Увеличить численность населения на 200 % (параметр `pop_size`). Был ли процесс нейроэволюции способен найти решение в этом случае, и если да, сколько поколений это заняло?
3. Измените начальное значение генератора случайных чисел (см. строку 118 файла `maze_experiment.py`). Успешен ли процесс нейроэволюции с этим новым значением?

5.7 Эксперимент со сложной конфигурацией лабиринта

Следующий эксперимент в этой главе представляет собой запуск процесса нейроэволюции, чтобы обучить агента, способного пройти лабиринт с более сложной конфигурацией стен. Этот лабиринт содержит мощные локальные ловушки наилучшей приспособленности и не имеет прямого пути от начальной позиции агента к зоне выхода из лабиринта. Конфигурация лабиринта изображена на рис. 5.8.

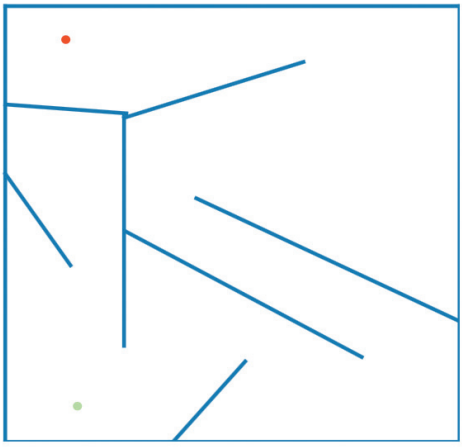


Рис. 5.8. Лабиринт со сложной конфигурацией

Начальная позиция агента находится в левом нижнем углу и отмечена зеленым кружком, а целевая позиция выхода из лабиринта – в верхнем левом углу и отмечена красным кружком. Чтобы пройти лабиринт, агент должен разработать сложную стратегию управления, которая позволяет ему избегать локальных ловушек наилучшей приспособленности вокруг начальной точки. Стратегия

управления должна провести агента от начальной точки до выхода по сложной траектории, имеющей несколько поворотов и больше локальных ловушек.

5.7.1 Настройка гиперпараметров

Для этого эксперимента мы будем использовать те же гиперпараметры, которые использовали в простом эксперименте по решению лабиринтов. Идея заключается в том, чтобы взять прежние начальные условия для нейроэволюционного алгоритма и посмотреть, сможет ли он обучить успешного решателя для другой, более сложной конфигурации лабиринта. Это покажет, насколько хорошо алгоритм обобщается с использованием гиперпараметров, заданных для иной конфигурации лабиринта.

5.7.2 Настройка рабочей среды и движок эксперимента

Настройка рабочей среды остается такой же, как для эксперимента по навигации в простом лабиринте. Движок эксперимента также остается прежним. Мы изменяем только файл, описывающий конфигурацию среды лабиринта.

5.7.3 Выполнение эксперимента по прохождению сложного лабиринта

Как мы уже упоминали, мы будем использовать ту же реализацию движка эксперимента и те же настройки гиперпараметров NEAT, что и в предыдущем эксперименте. Но мы настроим другую среду лабиринта следующим образом:

```
$ python maze_experiment.py -m hard -g 500
```

Дождавшись завершения эксперимента, мы обнаружили, что даже после 500 поколений эволюции успешный решатель не был найден. Лучший геном, полученный с использованием алгоритма нейроэволюции, кодирует причудливую и непрактичную конфигурацию нейросети контроллера, которая показана на рис. 5.9.

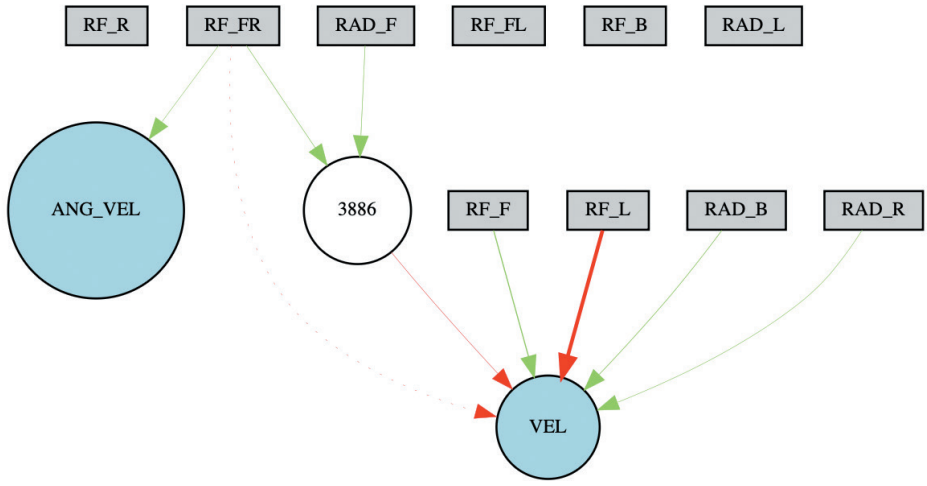


Рис. 5.9. Конфигурация нейросети решателя сложного лабиринта

На графе видно, что вращение робота зависит только от датчика фронтального дальномера (RF_FR), а линейное движение контролируется комбинацией нескольких дальномеров и радарных датчиков. Такая конфигурация управления приводит к упрощенным линейным перемещениям робота до тех пор, пока перед ним не окажется стена. Наше предположение о шаблонах движения подтверждается, когда мы смотрим на визуализацию журнала развития агента (рис. 5.10).

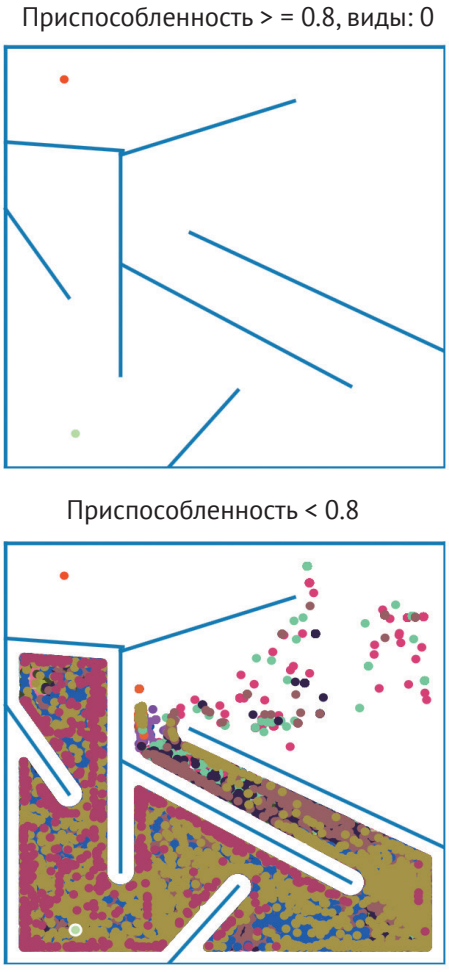


Рис. 5.10. Визуализация журнала развития агента

Визуализация конечных положений агентов демонстрирует, что большинство видов поймано в ловушку вокруг начальной позиции, где расположены некоторые области локальных оптимальных показателей приспособленности. Ни один из видов не смог даже преодолеть заданный порог приспособленности (0,8). Так же, как мы упоминали ранее, существуют четко различимые вертикальные линии, образованные конечными положениями агентов (серые

точки, создающие вертикальные столбцы). Это подтверждает наше предположение о неправильной конфигурации нейросети контроллера, которая была закодирована лучшим геномом, найденным в ходе эволюционного процесса.

Средние показатели приспособленности по поколениям показаны на рис. 5.11.

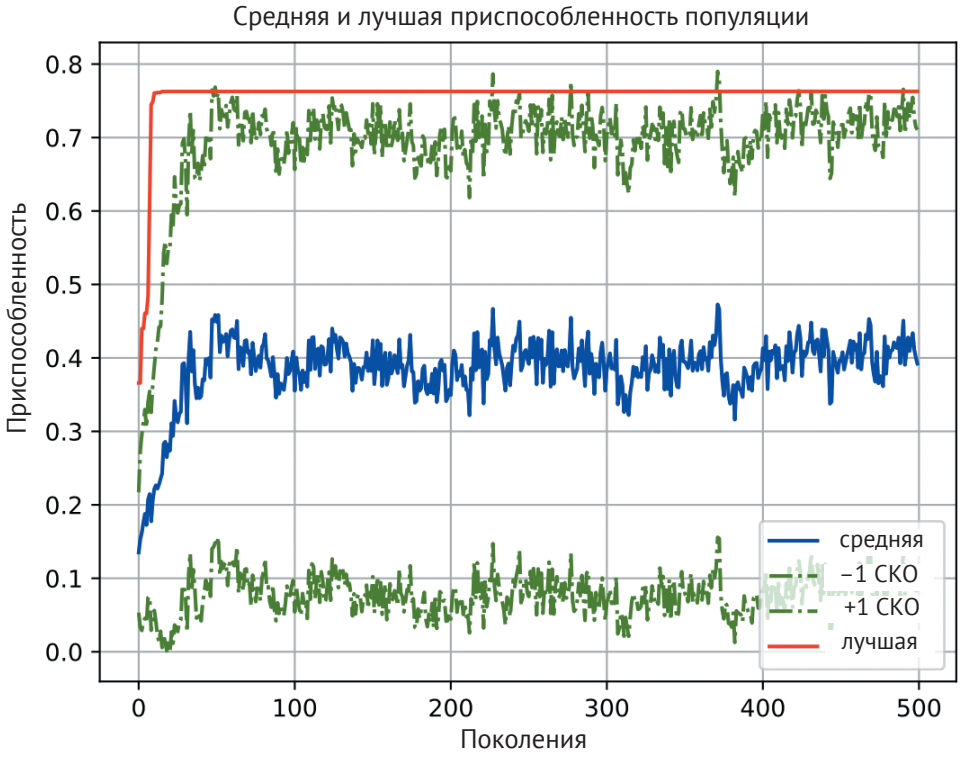


Рис. 5.11. Средние показатели приспособленности по поколениям

На графике средних показателей приспособленности мы можем видеть, что нейроэволюционный процесс смог значительно увеличить показатели приспособленности агентов в самых первых поколениях, но после этого он остановился на плато, не показав улучшений. Это означает, что дальнейшее увеличение числа поколений эволюции не имеет никакого смысла, и необходимо принять другие меры для улучшения характеристик нейроэволюционного процесса.

5.8 Упражнения

Попробуйте увеличить размер популяции, изменив параметр `pop_size` в файле `maze_config.ini`. Помогло ли это нейроэволюционному процессу развить успешный лабиринт?

Выполнение может занять длительное время.

5.9 Заключение

В этой главе вы узнали о классе задач планирования и управления, в которых используются целенаправленные функции приспособленности, имеющие обманчивый ландшафт. В этом ландшафте есть несколько ловушек, созданных локальными областями оптимума функции приспособленности, которые вводят в заблуждение процесс поиска решения, если он основан только на оценке приспособленности, рассчитанной как производная от расстояния от агента до цели. Вы узнали, что обычная целенаправленная функция приспособленности помогла процессу эволюции создать агента для прохождения простой конфигурации лабиринта, но потерпела неудачу с более сложным лабиринтом из-за ловушек локальных оптимумов.

Вы познакомились с полезным методом, который позволяет визуализировать конечные позиции всех оцененных агентов на карте лабиринта. С помощью этой визуализации вы можете делать предположения о характеристиках эволюционного процесса. Затем можете принять решение об изменениях параметров конфигурации, которые могут привести к дальнейшему повышению качества обучения агента.

Кроме того, вы узнали, что, когда существует более высокая вероятность конвергенции функции приспособленности в локальных оптимумах, процесс нейроэволюции приводит к уменьшению количества видов. В крайних случаях он создает только один вид, который препятствует инновациям и эволюционному процессу. Чтобы избежать этого, можно повысить видообразование, изменив значение коэффициента совместимости, который используется при расчете коэффициента совместимости генома. Этот коэффициент контролирует вес, который будет присвоен избыточным или непересекающимся частям сравниваемых геномов. Более высокие значения коэффициентов подчеркивают важность топологических различий в сравниваемых геномах и позволяют более разнородным геномам принадлежать к одному и тому же виду.

В следующей главе вы познакомитесь с методом оптимизации поиском новизны, который лучше подходит для решения обманчивых задач, таких как ориентирование в лабиринте.

Глава 6

Метод оптимизации поиском новизны

В этой главе вы узнаете о продвинутом методе оптимизации поиска решения, который можно использовать для создания автономно ориентирующихся агентов. Этот метод называется *поиском новизны* (novelty search, NS). Основная идея данного метода заключается в том, что целевая функция может быть определена исходя из новизны поведения, демонстрируемого агентом, а не расстоянием до цели в пространстве поиска решения.

В этой главе вы также узнаете, как использовать методы оптимизации на основе поиска новизны с алгоритмом нейроэволюции для обучения успешных агентов, ориентирующихся в лабиринте. Проведя эксперименты, представленные в этой главе, вы убедитесь, что в определенных случаях метод поиска новизны превосходит традиционный метод целеориентированной оптимизации поиска. К концу этой главы вы изучите основы метода оптимизации поиском новизны. Вы научитесь определять функцию приспособленности, используя оценку новизны, и применять ее для решения практических задач, связанных с вашей работой или экспериментами.

В этой главе будут рассмотрены следующие темы:

- метод оптимизации поиском новизны;
- основы реализации оценки новизны;
- функция приспособленности на основе оценки новизны;
- эксперимент с простой конфигурацией лабиринта;
- эксперимент со сложной конфигурацией лабиринта.

6.1 Технические требования

Для проведения экспериментов, описанных в этой главе, ваш компьютер должен удовлетворять следующим техническим требованиям:

- Windows 8/10, macOS 10.13 или новее, или современный Linux;
- Anaconda Distribution версия 2019.03 или новее.

Исходный код примеров этой главы можно найти в каталоге `Chapter6` в файловом архиве книги.

6.2 Метод оптимизации поиском новизны

Основная идея, заложенная в основу метода поиска новизны, заключается в том, чтобы вознаграждать *новизну* созданного решения, а не его близость к конечной цели. Эта идея вдохновлена естественной эволюцией. При поиске успешного решения не всегда очевиден точный порядок действий, которые следует предпринять. Естественная эволюция непрерывно производит новые формы с различными фенотипами, пытающимися использовать окружающую среду и приспосабливаться к изменениям. Это привело к взрыву разнообразия форм жизни на Земле и вызвало качественные скачки в эволюции жизни. Тот же процесс позволил формам жизни покинуть море и покорить землю. Необычайный генез эукариот стал источником всех высших форм жизни на планете. Все это примеры поощрения новизны в процессе эволюции. В то же время в естественной эволюции нет четкой задачи или конечной цели.

Как вы узнали из предыдущей главы, обычные целенаправленные функции приспособленности чувствительны к ловушкам локального оптимума. Эта патология оказывает давление на эволюционный процесс, заставляя его сходиться к единому решению, которое часто застревает в тупиках в пространстве поиска, при этом отсутствуют локальные шаги, которые могут улучшить эффективность функции приспособленности. В результате застревания в тупике успешное решение остается неисследованным.

И наоборот, поиск новизны ведет эволюцию к разнообразию. Этот подход помогает процессу нейроэволюции производить успешных решающих агентов, даже для задач с обманчивым ландшафтом функции приспособленности, таких как задача прохождения лабиринта.

Практическим примером такой обманчивой задачи является навигация по неизвестному городу. Если вы посещаете старый город с хаотичной системой дорог, вам нужно использовать иную стратегию, чтобы добраться из точки А в точку В, чем в современных городах с четкой линейной системой улиц. В современных городах достаточно проехать по дорогам, которые более-менее точно указывают в направлении пункта назначения, но навигация в старых городах намного сложнее. Направление к месту назначения часто приводит вас в тупик (обманчивый локальный оптимум). Вы должны использовать более исследовательский подход, тестируя новые и часто нелогичные решения, которые, казалось бы, уводят вас от пункта назначения. Однако, завернув за очередной угол, вы достигнете своей цели. Тем не менее обратите внимание, что с самого начала нам было неочевидно, какие повороты применять, основываясь только на расстоянии до конечного пункта назначения (то есть на целеориентированном показателе приспособленности). Повороты, ведущие к оптимальному решению, часто находятся в нелогичных местах, которые, кажется, уводят вас в сторону, но в конечном итоге помогают достичь цели.

Про оптимизацию поиском новизны мы говорили в главе 1.

6.3 Основы реализации алгоритма поиска новизны

Реализация алгоритма поиска новизны должна включать структуру данных для хранения информации о рассматриваемом новом элементе и структуру для работы со списком новых элементов. В нашей реализации эта функциональность заключена в трех классах Python:

- `NoveltyItem` – структура, которая содержит всю соответствующую информацию о степени новизны особи, которая оценивалась в ходе эволюции;
- `NoveltyArchive` – класс, который поддерживает список соответствующих экземпляров `NoveltyItem`. Он предоставляет методы для оценки новизны отдельных геномов по сравнению с уже накопленными экземплярами `NoveltyItem` и текущей популяцией;
- `ItemsDistance` – вспомогательная структура, которая содержит значение метрики расстояния (новизны) между двумя экземплярами `NoveltyItem`. Она используется в расчетах среднего расстояния до *k*-ближайшего соседа, которое используется в качестве значения оценки новизны в нашем эксперименте.

С подробностями реализации кода можно ознакомиться в файле `novelty_archive.py` в файловом архиве книги.

6.3.1 NoveltyItem

Этот класс является основной структурой, которая содержит информацию о балле новизны каждой особи, получившей оценку в ходе эволюции. Как можно видеть в исходном коде, он имеет несколько полей для хранения соответствующей информации:

```
def __init__(self, generation=-1, genomeId=-1, fitness=-1, novelty=-1):
    self.generation = generation
    self.genomeId = genomeId
    self.fitness = fitness
    self.novelty = novelty
    self.in_archive = False
    self.data = []
```

Поле `generation` содержит идентификатор поколения, в котором был создан этот элемент. По сути, `genomeId` – это идентификатор оцениваемого генома, а `fitness` – это целенаправленная оценка приспособленности оцениваемого генома (близость к выходу из лабиринта). Кроме того, `novelty` – это оценка новизны, присвоение которой оцениваемому геному мы обсудим в следующем разделе, а `data` – это список точек данных, представляющих координаты конкретных положений лабиринта, которые агент решателя посетил во время моделирования. Этот список данных используется для оценки расстояния между текущим и остальными элементами. Вычисленное расстояние после этого можно использовать для оценки показателя новизны, связанной с конкретным обновленным геномом.

6.3.2 NoveltyArchive

Этот класс ведет список соответствующих обновленных элементов и предоставляет методы для оценки новизны отдельных геномов, а также всей популяции геномов в целом. В конструкторе определены следующие поля:

```
def __init__(self, threshold, metric):
    self.novelty_metric = metric
    self.novelty_threshold = threshold
    self.novelty_floor = 0.25
    self.items_added_in_generation = 0
    self.time_out = 0
    self.neighbors = KNNNoveltyScore
    self.generation = 0
    self.novel_items = []
    self.fittest_items = []
```

Обратите внимание, что `novelty_metric` – это ссылка на функцию, которую можно использовать для оценки метрики новизны или расстояния между двумя новинками.

Кроме того, `novelty_threshold` определяет текущее минимальное значение новизны для `NoveltyItem`, при котором новый элемент может быть добавлен в архив. Это значение является динамическим и изменяется во время выполнения, чтобы поддерживать размер архива в определенных пределах; `novelty_floor` – это минимально возможное значение `novelty_threshold`. Поля `items_added_in_generation` и `time_out` используются для планирования динамики изменения значений `novelty_threshold`. Поле `neighbors` – это число *k*-ближайших соседей по умолчанию, которое используется для оценки новизны. `generation` – это нынешнее эволюционное поколение. `novel_items` – это список всех соответствующих экземпляров `NoveltyItem`, собранных до настоящего времени, а `fittest_items` – это список новых элементов, имеющих максимальный целенаправленный показатель приспособленности.

Динамика поля `novelty_threshold` определяется следующим исходным кодом:

```
def _adjust_archive_settings(self):
    if self.items_added_in_generation == 0:
        self.time_out += 1
    else:
        self.time_out = 0
    if self.time_out >= 10:
        self.novelty_threshold *= 0.95
        if self.novelty_threshold < self.novelty_floor:
            self.novelty_threshold = self.novelty_floor
        self.time_out = 0
    if self.items_added_in_generation >= 4:
        self.novelty_threshold *= 1.2
    self.items_added_in_generation = 0
```

Данная функция вызывается в конце каждого эволюционного поколения, чтобы настроить значение поля `novelty_threshold` для следующего поколения.

Как вы знаете, это значение определяет, сколько новых элементов следует добавить в архив в следующем поколении.

Динамическая регулировка этого параметра необходима для поддержания соответствующей сложности поиска новых решений с течением времени при использовании метода поиска новизны. В начале эволюции существуют огромные возможности для поиска новых решений с высокими показателями новизны, поскольку в лабиринте исследовано всего несколько путей. Однако ближе к концу эволюции становится все труднее находить новинки, потому что остается меньше неизведанных путей. Чтобы компенсировать это усложнение, если новый путь не найден в последних 2500 оценках (10 поколений), значение `novelty_threshold` снижается на 5 %. С другой стороны, чтобы снизить скорость добавления нового элемента `NoveltyItem` в архив на ранних этапах эволюции, значение `novelty_threshold` увеличивается на 20 %, если в последнем поколении было добавлено более четырех элементов.

Следующий исходный код показывает, как значение `novelty_threshold` используется для определения, какой элемент `NoveltyItem` заслуживает добавления в архив:

```
def evaluate_individual_novelty(self, genome, genomes, n_items_map,
only_fitness=False):
item = n_items_map[genome.key]
result = 0.0
if only_fitness:
result = self._novelty_avg_knn(item=item, genomes=genomes,
n_items_map=n_items_map)
else:
result = self._novelty_avg_knn(item=item, neighbors=1,
n_items_map=n_items_map)
if result > self.novelty_threshold or \
len(self.novel_items) < ArchiveSeedAmount:
self._add_novelty_item(item)
item.novelty = result
item.generation = self.generation
return result
```

Чтобы оценить новизну предоставленного генома, в данном блоке кода используется функция приспособленности с оценкой новизны, которую мы подробно рассмотрим далее. Если эта функция вызывается в режиме обновления архива (`only_fitness = False`), то полученное значение оценки новизны (`result`) сравнивается с текущим значением поля `novelty_threshold`. На основании результата сравнения принимается решение о добавлении объекта `NoveltyItem` в объект `NoveltyArchive`. Кроме того, для начального заполнения архива экземплярами `NoveltyItem` в начале эволюции, когда архив еще пуст, вводится константа `ArchiveSeedAmount`.

6.4 Функция приспособленности с оценкой новизны

Теперь, когда мы определили основные принципы, лежащие в основе метода поиска новизны, нам нужно найти способ интегрировать его в определение функции приспособленности, которая будет применяться для управления процессом

нейроэволюции. Другими словами, нам нужно определить метрику новизны, способную отразить количество новизны, привнесенное конкретным решающим агентом в ходе эволюционного процесса. Есть несколько характеристик, которые можно использовать как метрики новизны для решающего агента:

- новизна структуры генотипа решателя – *структурная* новизна;
- поведенческие шаги, найденные в пространстве поиска решения, – *поведенческая* новизна.

Наша основная цель в этой главе – обучить успешного агента-решателя лабиринта. Чтобы успешно перемещаться по лабиринту, агент должен уделять одинаковое внимание большинству мест в лабиринте. Такое поведение может быть достигнуто за счет вознаграждения агентов, которые выбирают уникальный путь исследования по сравнению с уже известными путями ранее протестированных агентов. Возвращаясь к ранее упомянутым метрикам новизны, это означает, что нам нужно определить функцию приспособленности, используя метрику, основанную на *поведенческой* новизне.

6.4.1 Оценка новизны

Поведенческое пространство агента, проходящего лабиринт, определяется его траекторией, пролегающей через лабиринт, во время моделирования прохождения лабиринта. Эффективная реализация оценки новизны должна вычислять *разреженность* в любой точке такого поведенческого пространства. Любая область с более плотным кластером посещенных точек пространства поведения является менее новой, что дает меньшее вознаграждение решающему агенту.

Как было сказано в главе 1, наиболее простой мерой разреженности в точке является среднее расстояние от нее до k-ближайших соседей. Разреженные области имеют более высокие значения расстояния, а более плотные области имеют низкие значения расстояния. Следующая формула дает разреженность в точке x поведенческого пространства:

$$\rho(x) = \frac{1}{k}\sum_{i=0}^{k} dist(x, \mu_i).$$

Здесь μ_i – i-й ближайший сосед x для расчета метрики расстояния (новизны) $dist(x, y)$.

Рассчитанная по приведенной выше формуле разреженность в определенной точке поведенческого пространства представляет собой показатель новизны, который может использоваться функцией приспособленности.

Код Python для нахождения показателя новизны представлен в виде следующей функции:

```
def _novelty_avg_knn(self, item, n_items_map, genomes=None,
                     neighbors=None):
    distances = None
    if genomes is not None:
        distances = self._map_novelty_in_population(item=item,
                          genomes=genomes, n_items_map=n_items_map)
    else:
        distances = self._map_novelty(item=item)
```

```
    distances.sort()
    if neighbors is None:
        neighbors = self.neighbors

    density, weight, distance_sum = 0.0, 0.0, 0.0
    length = len(distances)
    if length >= ArchiveSeedAmount:
        length = neighbors
        if len(distances) < length:
            length = len(distances)
        i = 0
        while weight < float(neighbors) and i < length:
            distance_sum += distances[i].distance
            weight += 1.0
            i += 1
        if weight > 0:
            sparsity = distance_sum / weight
    return sparsity
```

Эта функция состоит из нескольких основных частей.

1. Сначала мы проверяем, содержит ли аргумент функции `_novelty_avg_knn` список всех геномов в текущей популяции. В этом случае мы начнем с заполнения списка расстояний между поведенческими характеристиками всех геномов в популяции, включая все объекты `NoveltyItem` из `NoveltyArchive`. В противном случае мы используем предоставленную новинку (`item`), чтобы найти расстояния между ней и всеми объектами `NoveltyItem` из `NoveltyArchive`.

```
distances = None
if genomes is not None:
    distances = self._map_novelty_in_population(item=item,
                    genomes=genomes, n_items_map=n_items_map)
else:
    distances = self._map_novelty(item=item)
```

2. После этого мы сортируем список расстояний в порядке возрастания, чтобы в начале списка стояло наименьшее расстояние, потому что нас интересуют точки, которые являются ближайшими к предоставленному новому элементу в поведенческом пространстве:

```
distances.sort()
```

3. Затем мы инициализируем все промежуточные переменные, необходимые для вычисления оценок *k*-ближайших соседей, и проверяем, превышает ли количество значений расстояний, собранных на предыдущем шаге, постоянное значение `ArchiveSeedAmount`:

```
if neighbors is None:
    neighbors = self.neighbors

density, weight, distance_sum = 0.0, 0.0, 0.0
length = len(distances)
```

4. Теперь мы можем проверить, меньше ли длина списка найденных расстояний, чем количество соседей, которых мы хотим проверить на соответствие (neighbors). Если это так, мы обновляем значение связанной переменной:

```
if length >= ArchiveSeedAmount:
    length = neighbors
    if len(distances) < length:
        length = len(distances)
```

5. После того как всем локальным переменным присвоены правильные значения, мы можем запустить цикл, собирающий сумму всех расстояний и весов для каждой связи:

```
i = 0
while weight < float(neighbors) and i < length:
    distance_sum += distances[i].distance
    weight += 1.0
    i += 1
```

6. Когда предыдущий цикл завершается из-за того, что вычисленное значение веса превышает указанное число соседей, или если мы уже прошлись по всем значениям расстояния в списке distances, мы готовы рассчитать оценку новизны для данного элемента как среднее расстояние к *k*-ближайшим соседям:

```
if weight < 0:
    sparsity = distance_sum / weight
```

Затем функция возвращает вычисленное значение оценки новизны.

Подробности реализации можно найти в файле исходного кода novelty_archive.py в файловом архиве книги.

6.4.2 Метрика новизны

Метрика новизны является мерой того, насколько текущее решение отличается от уже известных. Она используется для вычисления оценки новизны при подсчете расстояния от текущей точки в поведенческом пространстве до ее *k*-ближайших соседей.

В нашем эксперименте метрика новизны, отражающая разницу в поведении двух агентов, определяется *поэлементным расстоянием* между двумя векторами траектории (один вектор на агента). Вектор траектории содержит координаты позиций, которые посетил агент во время симуляции. Следующая формула дает определение метрики:

$$dist(x,\mu)=\frac{1}{n}\sum_{j=0}^{n}\left|x_j-\mu_j\right|.$$

Здесь n – размер вектора траектории, а μ_i и x_j – значения сравниваемых векторов и в позиции j.

В эксперименте по прохождению лабиринта нас больше всего интересует конечная позиция решающего агента. Таким образом, вектор траектории может содержать только окончательные координаты агента после выполнения всех необходимых шагов в симуляторе прохождения лабиринта или при обнаружении выхода из лабиринта.

Код Python для вычисления метрики новизны выглядит следующим образом:

```
def maze_novelty_metric(first_item, second_item):
    diff_accum = 0.0
    size = len(first_item.data)
    for i in range(size):
        diff = abs(first_item.data[i] - second_item.data[i])
        diff_accum += diff
    return diff_accum / float(size)
```

Этот код берет двух новичков-претендентов и находит поэлементное расстояние между двумя векторами траектории, содержащими позиции соответствующих агентов во время симуляции прохождения лабиринта.

6.4.3 Функция приспособленности

Функция приспособленности, используемая в экспериментах данной главы, непосредственно использует показатель новизны, определенный ранее как значение приспособленности генома. В результате процесс нейроэволюции пытается максимизировать новизну произведенных особей, исходя из этой функции приспособленности.

Для различных задач в этом эксперименте мы используем различные факторы приспособленности:

- *оценка новизны* используется, чтобы направлять процесс нейроэволюции (оптимизация поиска решения). Она присваивается каждому геному в качестве значения приспособленности и используется для оценки генома в ходе эволюции;
- *оценка целеориентированной приспособленности* (расстояние до выхода из лабиринта), полученная из симулятора лабиринта, используется для проверки достижения конечной цели (то есть выхода из лабиринта) – это значение записывается как показатель эффективности каждого агента.

Исходный код для вычисления значений приспособленности представлен в двух функциях:

- функция обратного вызова для оценки показателей приспособленности всей популяции (`eval_genomes`);
- функция для оценки отдельных геномов с помощью симуляции прохождения лабиринта (`eval_individual`).

Функция оценки приспособленности популяции

Функция оценки приспособленности – это функция обратного вызова, зарегистрированная в библиотеке NEAT-Python и позволяющая этой библиотеке выполнять оценку геномов популяции в зависимости от определенных условий конкретной задачи, которую необходимо решить. Мы реализуем эту функцию,

чтобы оценить каждый геном в текущей популяции согласно тому, как он решает задачу прохождения лабиринта, и использовать полученный показатель новизны как оценку приспособленности генома.

Библиотека NEAT-Python не позволяет нам отправлять какие-либо указания о завершении задачи из функции обратного вызова, кроме как путем установки конкретного значения приспособленности генома-победителя. Это значение приспособленности должно быть выше порога приспособленности в конфигурации гиперпараметров NEAT-Python. Однако с помощью алгоритма поиска новизны невозможно точно оценить верхнюю границу значения новизны, которая может быть достигнута геномом победителя. Кроме того, геном победителя может иметь значение показателя новизны, которое ниже значений, полученных геномами ранее в процессе эволюции, когда пространство поиска решения не было так тщательно исследовано.

Поэтому, учитывая, что оценка новизны присваивается геномам в качестве их значений приспособленности, нам необходимо найти обходной путь, который позволяет нам использовать стандартные критерии завершения, определенные библиотекой NEAT-Python. Мы достигаем этого, используя заданное оценочное значение новизны, которое является достаточно большим, чтобы не встретиться случайно во время нормального выполнения алгоритма. Это значение определяет критерий завершения, передаваемый через конфигурацию гиперпараметров NEAT-Python. Мы используем `800000` в качестве индикативной меры оценки новизны и ее натуральный логарифм (примерно `13.59`) в качестве подходящего порога приспособленности.

Полный исходный код функции выглядит следующим образом:

```
def eval_genomes(genomes, config):
    n_items_map = {}
    solver_genome = None
    for genome_id, genome in genomes:
        found = eval_individual(genome_id=genome_id,
                                genome=genome,
                                genomes=genomes,
                                n_items_map=n_items_map,
                                config=config)
        if found:
            solver_genome = genome
    trial_sim.archive.end_of_generation()
    # Вычисляем приспособленность каждого генома в популяции
    for genome_id, genome in genomes:
        fitness = trial_sim.archive.evaluate_individual_novelty(
                    genome=genome,
                    genomes=genomes,
                    n_items_map=n_items_map,
                    only_fitness=True)
        if fitness > 1:
            fitness = math.log(fitness)
        else:
            fitness = 0
        genome.fitness = fitness
```

```
    if solver_genome is not None:
        solver_genome.fitness = math.log(800000) # ~=13.59
```

Функция состоит из трех важных частей:

1. Сначала мы создаем словарь для хранения новинок (`n_items_map`) для каждого генома в популяции и циклически перебираем все геномы в популяции, оценивая их эффективность решения лабиринтов:

```
n_items_map = {}
solver_genome = None
for genome_id, genome in genomes:
    found = eval_individual(genome_id=genome_id,
                            genome=genome,
                            genomes=genomes,
                            n_items_map=n_items_map,
                            config=config)
    if found:
        solver_genome = genome
trial_sim.archive.end_of_generation()
```

2. После этого мы перебираем все геномы в популяции еще раз, чтобы назначить оценки приспособленности для геномов, используя оценки новизны. В процессе оценки новизны используются объекты `NoveltyItem`, собранные в `n_items_map` в первом цикле (описанном ранее) во время моделирования прохождения лабиринта:

```
for genome_id, genome in genomes:
    fitness = trial_sim.archive.evaluate_individual_novelty(
                genome=genome,
                genomes=genomes,
                n_items_map=n_items_map,
                only_fitness=True)
    if fitness > 1:
        fitness = math.log(fitness)
    else:
        fitness = 0
    genome.fitness = fitness
```

3. Наконец, если в первом цикле обнаружен геном успешного решателя, мы присваиваем ему значение приспособленности, равное индикативному показателю приспособленности, описанному ранее (`~13.59`):

```
if solver_genome is not None:
    solver_genome.fitness = math.log(800000) # ~13.59
```

Обратите внимание, что мы применяем натуральный логарифм к полученным значениям оценки новизны и к индикативной оценке новизны, чтобы сохранить их числовую близость. В результате мы сможем правильно отображать графики производительности, используя статистику, собранную в ходе эксперимента.

Функция оценки индивидуальной приспособленности

Эта функция является важной частью оценки приспособленности популяции, и она вызывается из функции `eval_genomes`, обсуждавшейся ранее, для оценки эффективности прохождения лабиринта каждым геномом в популяции.

Оценка отдельного генома как агента, проходящего лабиринт, выглядит следующим образом:

```
def eval_individual(genome_id, genome, genomes, n_items_map, config):
    n_item = archive.NoveltyItem(
                        generation=trial_sim.population.generation,
                        genomeId=genome_id)
    n_items_map[genome_id] = n_item
    maze_env = copy.deepcopy(trial_sim.orig_maze_environment)
    control_net = neat.nn.FeedForwardNetwork.create(genome, config)
    goal_fitness = maze.maze_simulation_evaluate(
                                    env=maze_env,
                                    net=control_net,
                                    time_steps=SOLVER_TIME_STEPS,
                                    n_item=n_item,
                                    mcns=MCNS)

    if goal_fitness == -1:
        # Особь не удовлетворяет критерию минимальной приспособленности.
        print("Individ with ID %d marked for extinction, MCNS %f"
              % (genome_id, MCNS))
        return False

    record = agent.AgentRecord(
        generation=trial_sim.population.generation,
        agent_id=genome_id)
    record.fitness = goal_fitness
    record.x = maze_env.agent.location.x
    record.y = maze_env.agent.location.y
    record.hit_exit = maze_env.exit_found
    record.species_id = trial_sim.population.species \
        .get_species_id(genome_id)
    record.species_age = record.generation - \
        trial_sim.population.species.get_species(genome_id).created
    trial_sim.record_store.add_record(record)

    if not maze_env.exit_found:
        record.novelty = trial_sim.archive \
         .evaluate_individual_novelty(genome=genome,
                       genomes=genomes, n_items_map=n_items_map)

    trial_sim.archive.update_fittest_with_genome(genome=genome,
                                       n_items_map=n_items_map)
    return maze_env.exit_found
```

Давайте подробно разберем назначение всех основных частей реализации функции `eval_individual`.

1. Сначала мы создаем объект `NoveltyItem` для хранения информации о степени новизны, связанной с конкретным геномом, и сохраняем его под ключом `genome_id` в словаре `n_items_map`:

```
n_item = archive.NoveltyItem(
                    generation=trial_sim.population.generation,
                    genomeId=genome_id)
n_items_map[genome_id] = n_item
```

2. После этого создаем глубокую копию исходной среды лабиринта, чтобы избежать побочных эффектов во время симуляции, и формируем контрольную нейросеть на основе предоставленного генома:

```
maze_env = copy.deepcopy(trial_sim.orig_maze_environment)
control_net = neat.nn.FeedForwardNetwork.create(genome, config)
```

3. Теперь, используя копию среды лабиринта и созданную контрольную нейросеть, мы запускаем симуляцию прохождения лабиринта для заданного количества шагов:

```
goal_fitness = maze.maze_simulation_evaluate(
                                  env=maze_env,
                                  net=control_net,
                                  time_steps=SOLVER_TIME_STEPS,
                                  n_item=n_item,
                                 mcns=MCNS)
```

4. После завершения симуляции возвращенная оценка целеориентированной приспособленности (оценка близости к выходу из лабиринта) и другие параметры симуляции и генома сохраняются в записи `AgentRecord`, которая затем добавляется в хранилище записей:

```
record = agent.AgentRecord(
    generation=trial_sim.population.generation,
    agent_id=genome_id)
record.fitness = goal_fitness
record.x = maze_env.agent.location.x
record.y = maze_env.agent.location.y
record.hit_exit = maze_env.exit_found
record.species_id = trial_sim.population.species \
    .get_species_id(genome_id)
record.species_age = record.generation - \
   trial_sim.population.species.get_species(genome_id).created
trial_sim.record_store.add_record(record)
```

5. Наконец, мы оцениваем новизну данного генома, если он не является победителем, и при необходимости обновляем список наиболее подходящих геномов в `NoveltyArchive` с помощью `NoveltyItem` текущего генома:

```
if not maze_env.exit_found:
    record.novelty = trial_sim.archive \
     .evaluate_individual_novelty(genome=genome,
          genomes=genomes, n_items_map=n_items_map)

trial_sim.archive.update_fittest_with_genome(genome=genome,
                                        n_items_map=n_items_map)
```

В этом эксперименте оценка приспособленности генома определяется как два отдельных значения, каждое из которых служит своей цели. Показатель целеориентированной приспособленности помогает проверить, было ли найдено решение, и собирает полезную статистику производительности. Оценка приспособленности на основе новизны направляет процесс нейроэволюции в сторону максимального разнообразия поведения решателя, то есть градиент поиска решения направлен на изучение различных вариантов поведения без какой-либо явной цели.

Подробности реализации можно изучить в файле исходного кода `maze_experiment.py` в файловом архиве книги.

6.5 Эксперимент с простой конфигурацией лабиринта

Теперь мы готовы провести эксперименты, используя простую конфигурацию лабиринта, подобную описанной в предыдущей главе. Однако для управления процессом нейроэволюции вместо целеориентированной целевой функции мы используем метод оптимизации поиском новизны. Мы надеемся, что с помощью метода поиска новизны удастся найти успешный решатель с меньшим количеством этапов эволюции.

Схема простого лабиринта изображена на рис. 6.1.

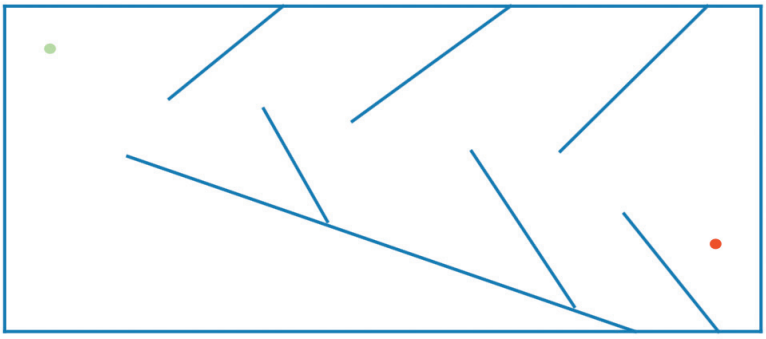


Рис. 6.1. Схема простого лабиринта

Конфигурация лабиринта такая же, как в предыдущей главе. Однако нам необходимо настроить некоторые гиперпараметры NEAT для соответствия спецификациям метода поиска новизны.

6.5.1 Настройка гиперпараметров

Целевая функция, используемая в экспериментах, описанных в этой главе, основана на метрике новизны, которая не имеет четкого верхнего граничного значения. Следовательно, мы не можем точно оценить пороговое значение приспособленности. Чтобы сигнализировать о появлении агента-победителя, мы воспользуемся ориентировочным значением, которое достаточно велико, чтобы его нельзя было случайно встретить при обычном выполнении алгоритма.

В качестве ориентировочного значения оценки новизны мы выбрали `800000`. Однако для улучшения визуального представления графика результатов эксперимента мы используем натуральный логарифм значения новизны. Поэтому пороговое значение приспособленности, используемое в файле конфигурации, становится равным `13.5`, что немного меньше максимально возможной оценки приспособленности (`13.59`), чтобы избежать проблем с округлением чисел с плавающей запятой. Кроме того, мы увеличиваем размер популяции по сравнению со значением, описанным в предыдущей главе (`250`), чтобы сделать пространство поиска решения более обширным, поскольку нам необходимо изучить максимальное количество уникальных мест в лабиринте:

```
[NEAT]
fitness_criterion = max
fitness_threshold = 13.5
pop_size = 500
reset_on_extinction = False
```

Теперь в каждом испытании мы проходим через большее количество поколений, чем в эксперименте в предыдущей главе. Поэтому мы увеличили допустимую продолжительность стагнации, чтобы дольше сохранять разнообразие видов:

```
[DefaultStagnation]
max_stagnation = 100
```

Все остальные гиперпараметры NEAT имеют значения, аналогичные значениям, представленным в предыдущей главе. При необходимости вернитесь к пояснениям относительно выбора конкретных значений гиперпараметров в предыдущей главе.

Полный перечень гиперпараметров, задействованных в эксперименте, содержится в файле `maze_config.ini` в файловом архиве книги.

6.5.2 Настройка рабочей среды

Рабочая среда для проведения эксперимента должна включать все зависимости и может быть создана с помощью следующих команд Anaconda:

```
$ conda create --name maze_ns_neat python=3.5
$ conda activate maze_ns_neat
```

```
$ pip install neat-python==0.92
$ conda install matplotlib
$ conda install graphviz
$ conda install python-graphviz
```

Эти команды создают и активируют виртуальную среду `maze_ns_neat` с Python 3.5. После этого устанавливается библиотека NEAT-Python версии 0.92, а также другие зависимости, используемые нашими утилитами визуализации.

6.5.3 Реализация движка эксперимента

Реализация движка эксперимента, использованная в этой главе, по большей части похожа на реализацию, использованную в предыдущей главе, но имеет существенные отличия, которые мы обсудим в данном разделе.

Цикл испытаний

В этой главе мы представляем обновление реализации эксперимента. Мы реализуем поддержку для запуска нескольких испытаний последовательно, пока не будет найдено решение. Такое обновление значительно упрощает последовательную работу с несколькими экспериментами, особенно с учетом того, что выполнение каждого испытания может занять много времени.

Основной цикл движка эксперимента теперь выглядит следующим образом (см. `__main__` в скрипте `maze_experiment.py`):

```
print("Starting the %s maze experiment (Novelty Search), for %d trials"
      % (args.maze, args.trials))
for t in range(args.trials):
    print("\n\n----- Starting Trial: %d ------" % (t))
    # Создаем архив новинок
    novelty_archive = archive.NoveltyArchive(
                              threshold=args.ns_threshold,
                              metric=maze.maze_novelty_metric)
    trial_out_dir = os.path.join(out_dir, str(t))
    os.makedirs(trial_out_dir, exist_ok=True)
    solution_found = run_experiment(config_file=config_path,
                                    maze_env=maze_env,
                                    novelty_archive=novelty_archive,
                                    trial_out_dir=trial_out_dir,
                                    n_generations=args.generations,
                                    args=args,
                                    save_results=True,
                                    silent=True)
    print("\n------ Trial %d complete, solution found: %s ------\n"
          % (t, solution_found))
```

Цикл запускает испытания в количестве `args.trials`, где значение `args.trials` предоставляется пользователем из командной строки.

Первые строки цикла создают объект `NoveltyArchive`, который является частью алгоритма поиска новизны. Позже, во время конкретного испытания, этот объект будет применяться для хранения всех соответствующих элементов `NoveltyItems`:

```
novelty_archive = archive.NoveltyArchive(
                threshold=args.ns_threshold,
                metric=maze.maze_novelty_metric)
```

Обратите внимание, что `maze.maze_novelty_metric` является ссылкой на функцию, которая используется для оценки новизны каждого агента-решателя.

В исходном коде для этой главы представлена реализация двух функций метрики новизны:

- метрика поэлементного расстояния (`maze.maze_novelty_metric`);
- метрика евклидового расстояния (`maze.maze_novelty_metric_euclidean`).

Однако в наших экспериментах мы используем первую реализацию. Вторая реализация предназначена для запуска дополнительных экспериментов.

Функция движка эксперимента

Данная функция движка имеет много сходств с функцией движка, представленной в предыдущей главе, но в то же время у нее есть уникальные особенности, характерные для алгоритма оптимизации поиском новизны.

Давайте рассмотрим наиболее важные части реализации.

1. Функция начинается с выбора определенного начального значения для генератора случайных чисел на основе текущего системного времени:

```
seed = int(time.time())
random.seed(seed)
```

2. После этого она загружает конфигурацию алгоритма NEAT и создает начальную популяцию геномов:

```
config = neat.Config(neat.DefaultGenome,
                     neat.DefaultReproduction,
                     neat.DefaultSpeciesSet,
                     neat.DefaultStagnation,
                     config_file)
p = neat.Population(config)
```

3. Чтобы сохранить промежуточные результаты после оценки каждого поколения, мы инициализируем глобальную переменную `trial_sim` с объектом `MazeSimulationTrial`. Мы используем глобальную переменную, которая обеспечивает доступ с помощью функции обратного вызова к оценке приспособленности (`eval_genomes(genomes, config)`), передаваемой в среду NEAT-Python:

```
global trial_sim
trial_sim = MazeSimulationTrial(maze_env=maze_env,
                                population=p,
                                archive=novelty_archive)
```

4. Также традиционно мы регистрируем в объекте `Population` несколько репортеров для вывода результатов алгоритма и сбора статистики:

```
p.add_reporter(neat.StdOutReporter(True))
stats = neat.StatisticsReporter()
p.add_reporter(stats)
```

5. Теперь мы готовы запустить алгоритм NEAT для заданного числа поколений и оценить результаты:

```
start_time = time.time()
best_genome = p.run(eval_genomes, n=n_generations)
elapsed_time = time.time() - start_time
# Отображаем лучшие геномы по поколениям.
print('\nBest genome:\n%s' % (best_genome))
solution_found = \
    (best_genome.fitness >= config.fitness_threshold)
if solution_found:
    print("SUCCESS: The stable maze solver controller was found!!!")
else:
    print("FAILURE: Failed to find the stable maze solver controller!!!")
```

6. После этого собранные статистические и архивные записи можно визуализировать и сохранить в файловой системе:

```
node_names = {-1:'RF_R', -2:'RF_FR', -3:'RF_F', -4:'RF_FL',
              -5:'RF_L', -6: 'RF_B', -7:'RAD_F', -8:'RAD_L',
              -9:'RAD_B', -10:'RAD_R', 0:'ANG_VEL', 1:'VEL'}
visualize.draw_net(config, best_genome, view=show_results,
                       node_names=node_names,
                       directory=trial_out_dir, fmt='svg')
if args is None:
    visualize.draw_maze_records(maze_env,
                            trial_sim.record_store.records,
                            view=show_results)
else:
    visualize.draw_maze_records(maze_env,
                       trial_sim.record_store.records,
                       view=show_results, width=args.width,
                       height=args.height,
                       filename=os.path.join(trial_out_dir,
                                             'maze_records.svg'))
visualize.plot_stats(stats, ylog=False,
                     view=show_results,
                     filename=os.path.join(trial_out_dir,
                                           'avg_fitness.svg'))
visualize.plot_species(stats, view=show_results,
                     filename=os.path.join(trial_out_dir,
                                            'speciation.svg'))
# Сохраняем архивные записи NoveltyItems
trial_sim.archive.write_fittest_to_file(
                         path=os.path.join(trial_out_dir,
                                 'ns_items_fittest.txt'))
trial_sim.archive.write_to_file(
                         path=os.path.join(trial_out_dir,
                                    'ns_items_all.txt'))
```

7. Наконец, мы запускаем представленные в этой главе дополнительные процедуры визуализации, отображающие путь агентов через лабиринт. Мы делаем это, запустив симуляцию прохождения лабиринта с контроллером на основе нейросети лучшего решающего агента, найденного в ходе эволюции. Во время этого прогона симуляции все точки пути, посещенные решающим агентом, собираются для последующей визуализации функцией `draw_agent_path`:

```
maze_env = copy.deepcopy(trial_sim.orig_maze_environment)
control_net = neat.nn.FeedForwardNetwork.create(
                                        best_genome, config)
path_points = []
evaluate_fitness = maze.maze_simulation_evaluate(
                                env=maze_env,
                                net=control_net,
                                time_steps=SOLVER_TIME_STEPS,
                                path_points=path_points)
print("Evaluated fitness of best agent: %f"
          % evaluate_fitness)
visualize.draw_agent_path(trial_sim.orig_maze_environment,
                          path_points, best_genome,
                          view=show_results,
                          width=args.width,
                          height=args.height,
                          filename=os.path.join(trial_out_dir,
                                    'best_solver_path.svg'))
```

В конце функция `run_experiment` возвращает логическое значение, указывающее, был найден во время испытания успешный агент-решатель задачи лабиринта или нет.

Изучите реализацию функции `run_experiment(config_file, maze_env, novelty_archive, trial_out_dir, args=None, n_generations=100, save_results=False, silent=False)` в файле `maze_experiment.py` из файлового архива книги.

6.5.4 Простой эксперимент по навигации в лабиринте с поиском новизны

Убедитесь, что вы скопировали все связанные скрипты Python и файлы конфигурации (`maze_config.ini` и `medium_maze.txt`) в локальный каталог из файлового архива книги.

Теперь войдите в рабочий каталог и выполните следующую команду в удобном для вас приложении терминала:

```
python maze_experiment.py -g 500 -t 10 -m medium --width 300 --height 150
```

Не забудьте активировать соответствующую виртуальную среду с помощью команды `conda activate maze_ns_neat`.

Предыдущая команда запускает 10 испытаний эксперимента по навигации в лабиринте с простой конфигурацией, загруженной из файла medium_maze.txt. Алгоритм нейроэволюции оценивает 500 поколений решателей лабиринтов в каждом испытании, используя данные конфигурации NEAT, загруженные из файла maze_config.ini. Параметры width и height задают размеры секции графика (подробнее см. в реализации функции visualize.draw_maze_records).

Спустя 99 поколений эволюции успешный агент-решатель лабиринта был найден в поколении 100. Имеются общие статистические данные о популяции геномов в последнем поколении эволюции. В консольном выводе завершенной программы Python вы увидите следующий текст для последнего поколения эволюции:

```
****** Running generation 100 ******

Maze solved in 391 steps
Population's average fitness: 1.28484 stdev: 0.90091
Best fitness: 13.59237 - size: (2, 8) - species 1 - id 48354

Best individual in generation 100 meets fitness threshold - complexity: (2, 8)
```

Далее отображается конфигурация генома-победителя и общая статистика прогона:

```
Best genome:
Key: 48354
Fitness: 13.592367006650065
Nodes:
  0 DefaultNodeGene(key=0, bias=-2.1711339938349026, response=1.0,
activation=sigmoid, aggregation=sum)
  1 DefaultNodeGene(key=1, bias=6.576480565646596, response=1.0,
activation=sigmoid, aggregation=sum)
Connections:
  DefaultConnectionGene(key=(-10, 1), weight=-0.5207773885939109,
enabled=True)
  DefaultConnectionGene(key=(-9, 0), weight=1.7778928210387814,
enabled=True)
  DefaultConnectionGene(key=(-7, 1), weight=-2.4940590667086524,
enabled=False)
  DefaultConnectionGene(key=(-6, 1), weight=-1.3708732457648565,
enabled=True)
  DefaultConnectionGene(key=(-4, 0), weight=4.482428082179011,
enabled=True)
  DefaultConnectionGene(key=(-4, 1), weight=-1.3103728328721098,
enabled=True)
  DefaultConnectionGene(key=(-3, 0), weight=-0.4583080031587811,
enabled=True)
  DefaultConnectionGene(key=(-3, 1), weight=4.643599450804774,
```

```
enabled=True)
  DefaultConnectionGene(key=(-2, 1), weight=-0.9055329546235956,
enabled=True)
  DefaultConnectionGene(key=(-1, 0), weight=-1.5899992185951817,
enabled=False)
SUCCESS: The stable maze solver controller was found!!!

Record store file: out/maze_ns/medium/0/data.pickle
Random seed: 1567086899
Trial elapsed time: 7452.462 sec
Plot figure width: 6.8, height: 7.0
Maze solved in 391 steps
Evaluated fitness of best agent: 1.000000
Plot figure width: 7.8, height: 4.0
```

Консольный вывод сообщает нам, что геном победителя, кодирующий нейросеть успешного решателя лабиринта, имеет только два гена узла и восемь генов связей. Эти гены соответствуют двум выходным узлам в нейросети контроллера, причем восемь соединений используются для установления связей с входами. Итоговая конфигурация нейросети показана на рис. 6.2.

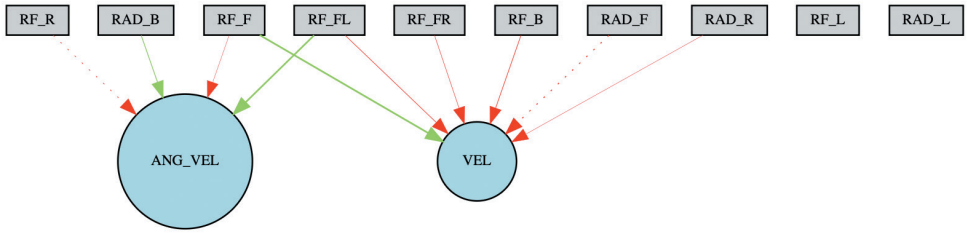


Рис. 6.2. Конфигурация нейросети успешного решателя лабиринта

Данная конфигурация нейросети успешного решателя лучше, чем конфигурация, описанная в предыдущей главе, которая была с использованием *целеориентированного* метода оптимизации. В этом эксперименте оптимальная конфигурация нейросети полностью исключает скрытые узлы, и эволюционный процесс занимает меньше поколений, чтобы найти ее.

Таким образом, мы можем предположить, что метод оптимизации поиском новизны по крайней мере так же эффективен, как и метод, ориентированный на цель. Это наблюдается даже при том, что метод оптимизации поиском новизны вообще не учитывает близость к конечной цели, а всего лишь вознаграждает новое поведение. Процесс нейроэволюции привел к созданию успешного агента-решателя задачи лабиринта, без намеков на близость к конечной цели (выходу из лабиринта), и это просто удивительно.

Также интересно посмотреть на график видообразования во время эволюции (рис. 6.3).

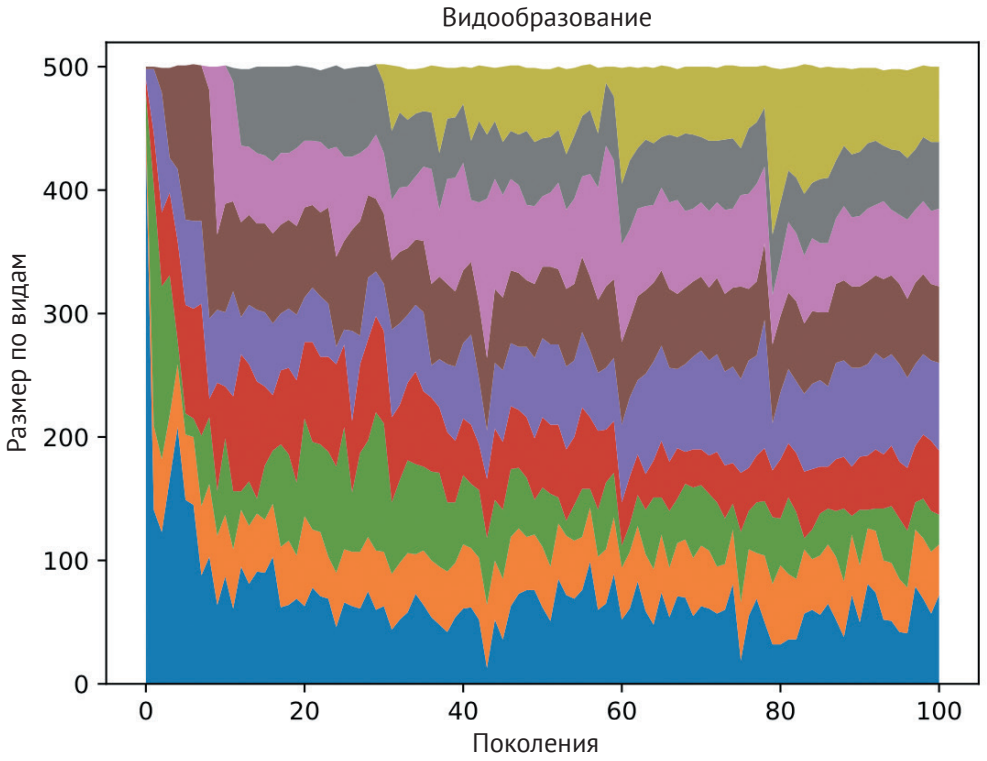


Рис. 6.3. График видообразования

На графике видообразования видно, что общее количество видов в ходе эволюционного процесса не превышает девяти. Кроме того, большинство из них присутствуют с самых первых поколений эволюции, пока не будет найден успешный решатель.

Визуализация записей агента

Мы использовали метод визуализации записей агента, который был представлен в предыдущей главе, и добавили новый метод для визуализации пути агента через лабиринт.

Записи агентов для каждого завершенного испытания автоматически сохраняются в виде SVG-файла `obj_medium_maze_records.svg` в выходном каталоге соответствующего эксперимента.

Визуализация записей агентов для эксперимента, описанного в этой главе, показана на рис. 6.4.

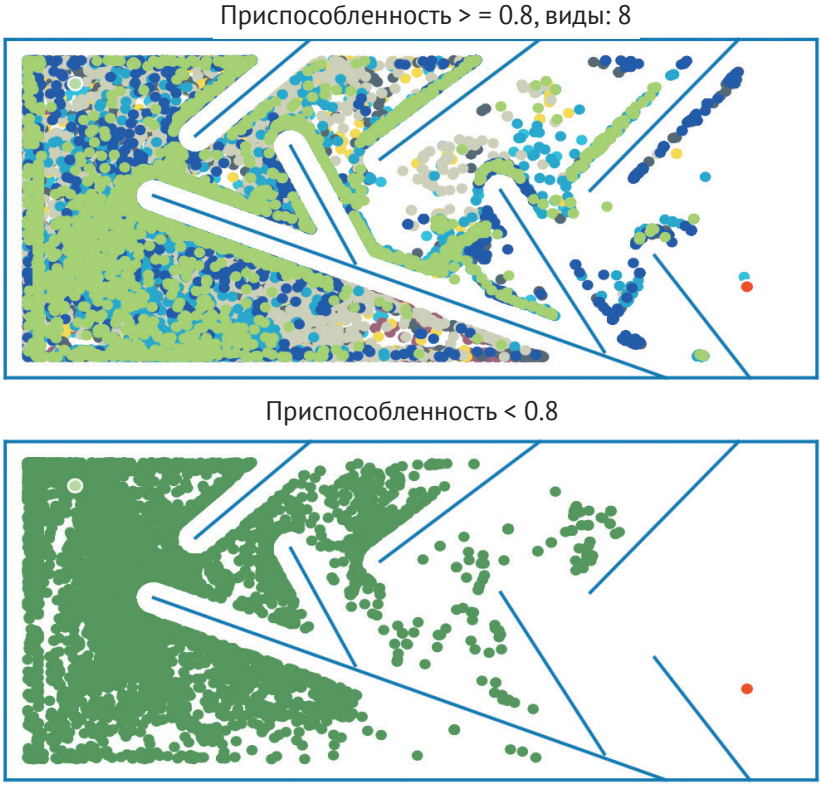


Рис. 6.4. Визуализация записей агента

Верхняя часть графика отображает конечные позиции агентов, принадлежащих к наиболее приспособленным видам, у которых целеориентированное значение показателя приспособленности выше 0,8. Мы смогли найти восемь видов, которые исследовали почти все области лабиринта и наконец смогли найти выход из лабиринта. В то же время даже эволюционные неудачники (нижняя часть графика) продемонстрировали весьма выраженное исследовательское поведение, равномерно заполняя первую половину области лабиринта (сравните это с аналогичным графиком в предыдущей главе).

Кроме того, важно отметить, что восемь из девяти видов, созданных в ходе эволюционного процесса, демонстрируют самые высокие целевые показатели приспособленности; то есть они были почти в состоянии достигнуть выхода из лабиринта (и один из них в конечном счете достиг его). Это достижение резко контрастирует с экспериментом в предыдущей главе, где только половина всех видов (шесть из двенадцати) достигли таких же результатов.

Однако наиболее захватывающая визуализация представляет собой путь успешного агента, который смог найти выход из лабиринта (рис. 6.5).

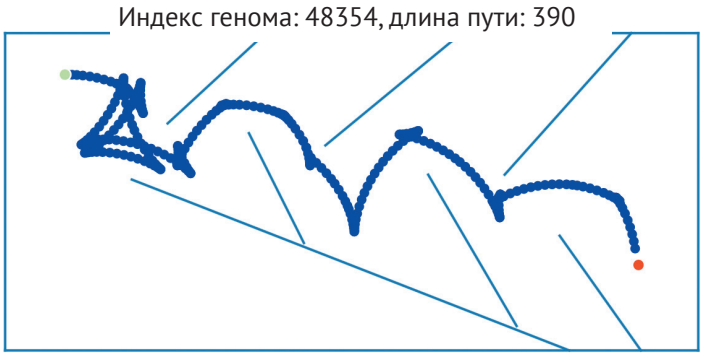


Рис. 6.5. Путь успешного агента, который нашел выход из лабиринта

Эту визуализацию можно найти в выходном каталоге эксперимента в файле `best_solver_path.svg`.

Как вы можете видеть, успешный агент, решающий задачу лабиринта, смог найти почти оптимальный путь через лабиринт, хотя вначале он немного запутался.

Просто удивительно, что такой извилистый путь через лабиринт может быть найден без малейшего намека на местоположение выхода из лабиринта, а только путем вознаграждения за новизну каждого промежуточного решения.

6.5.5 Упражнение 1

1. Установите для параметра размера популяции (`pop_size`) в файле `maze_config.ini` значение `250`. Проверьте, можно ли найти в этом случае решатель лабиринтов.
2. Измените значение параметра, указывающего вероятность добавления нового узла (`node_add_prob`). Был ли процесс нейроэволюции в состоянии найти решение и является ли оно оптимальным с топологической точки зрения?
3. Измените исходную конфигурацию генома, чтобы вначале не было скрытых узлов (`num_hidden`). Как это влияет на производительность алгоритма?
4. Попробуйте использовать другую метрику новизны, которая предоставляется с исходным кодом (`maze.maze_novelty_metric_euclidean`), и посмотрите, что произойдет.
5. Измените параметр командной строки `location_sample_rate` со значения по умолчанию (`4000`), которое позволяет включать только конечную позицию решателя лабиринта в его вектор поведения. Попробуйте значения меньше `400` (количество шагов моделирования лабиринта). Например, если мы установим для этого параметра значение `100`, вектор поведения будет включать в себя координаты максимум четырех точек траектории для каждого решающего агента. Посмотрите, как этот параметр может влиять на производительность алгоритма. Вы можете задать значение этого параметра, выполнив следующую команду:

```
python maze_experiment.py -g 500 -t 10 -r 100 -m medium --width 300 --height 150
```

Предыдущая команда запускает эксперимент простого лабиринта с параметром `location_sample_rate`, равным 100.

6.6 Эксперимент со сложной конфигурацией лабиринта

В следующем эксперименте мы оценим эффективность метода оптимизации поиском новизны в более сложной задаче. В этой задаче мы пытаемся обучить агента, способного найти путь через лабиринт со сложной конфигурацией.

Для этого эксперимента мы используем сложную для прохождения конфигурацию лабиринта, представленную в предыдущей главе. Такой подход позволяет сравнивать результаты, полученные с помощью метода оптимизации поиском новизны, с результатами, полученными с помощью метода оптимизации, основанного на близости к цели, использованного в предыдущей главе. Конфигурация лабиринта повышенной сложности показана на рис. 6.6.

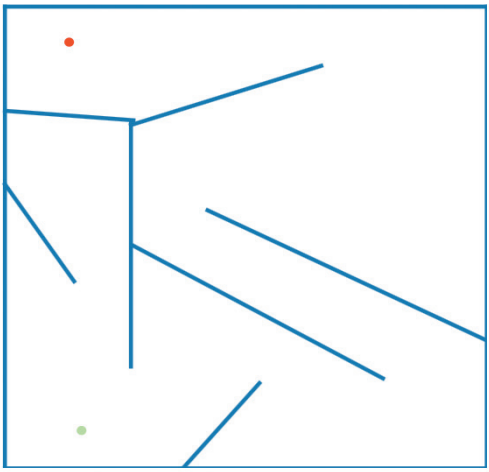


Рис. 6.6. Конфигурация лабиринта повышенной сложности

Эта конфигурация лабиринта идентична описанной в главе 5. При необходимости обратитесь к этой главе за подробным описанием.

6.6.1 Настройка гиперпараметров и рабочей среды

Гиперпараметры для этого эксперимента те же, что мы использовали ранее в данной главе для эксперимента с простым лабиринтом. Мы решили оставить гиперпараметры без изменений, чтобы проверить, насколько хорошо алгоритм обобщается при попытке найти решение задачи в той же области, но с другой конфигурацией.

Рабочая среда для этого эксперимента полностью совместима со средой, уже созданной для эксперимента с простым лабиринтом. Поэтому мы также используем прежнюю среду.

6.6.2 Выполнение эксперимента по прохождению труднодоступного лабиринта

Для запуска этого эксперимента мы можем использовать тот же движок, который разработали для эксперимента с простым лабиринтом, с той лишь разницей, что вначале должны быть указаны другие параметры командной строки.

Вы можете запустить эксперимент со сложным лабиринтом с помощью следующей команды:

```
$ python maze_experiment.py -m hard -g 500 -t 10 --width 200 --height 200
```

Эта команда запускает эксперимент со сложным лабиринтом для серии из `10` испытаний по `500` поколений в каждом. Параметры `width` и `height` определяют размеры рабочей области для визуализации записей, собранных во время эксперимента.

Используя библиотеку NEAT-Python для эксперимента со сложным лабиринтом, мы не смогли найти успешный решатель в течение `10` испытаний, даже с помощью метода оптимизации поиском новизны. Тем не менее результаты, полученные с помощью поиска новизны, являются более многообещающими, чем с помощью метода целеориентированной оптимизации из предыдущей главы. Вы можете увидеть это на рис. 6.7, где показаны конечные позиции агентов во время симуляции прохождения лабиринта.

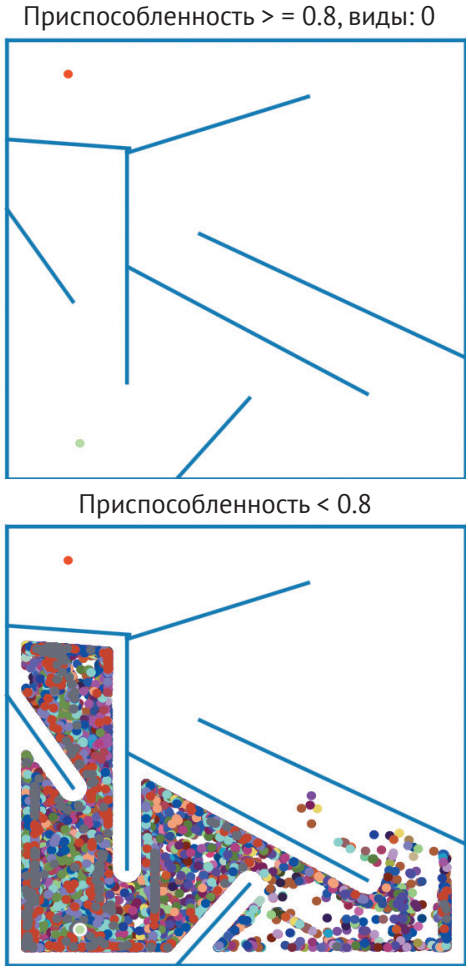


Рис. 6.7. Визуализация конечных позиций агентов-решателей задачи лабиринта

График, который визуализирует конечные позиции всех оцененных агентов, показывает, что в ходе этого эксперимента за счет оптимизации поиском новизны было исследовано больше областей лабиринта, чем при попытке ориентации на цель. Кроме того, вы можете видеть, что некоторые виды почти достигли финиша, оказавшись всего в нескольких шагах от выхода из лабиринта. Путь наиболее успешного агента показан на рис. 6.8.

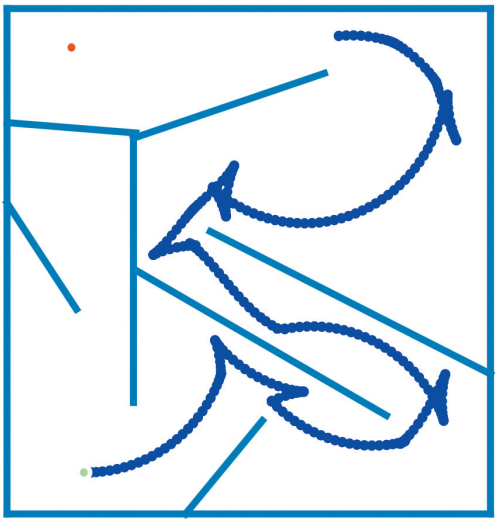


Рис. 6.8. Путь наиболее успешного агента, решающего задачу лабиринта

Путь через лабиринт, пройденный наиболее успешным агентом, демонстрирует, что агент смог обнаружить ключевые отношения между входами датчиков и маневрами, которые необходимо выполнить. Тем не менее ему все еще не хватает точности в применении сигналов управления. Из-за этого недостатка некоторые управляющие действия формируют неэффективную траекторию, напрасно расходующую драгоценное время на прохождение лабиринта.

Наконец, интересно взглянуть на топологию нейросети самого успешного решателя задачи лабиринта (рис. 6.9).

Вы можете видеть, что в принятие решения вовлечены все входы датчиков, в отличие от топологии нейросети, разработанной в предыдущем эксперименте этой главы. Кроме того, топология включает в себя два скрытых узла, что позволяет агенту реализовывать сложную стратегию управления для прохождения более трудного лабиринта.

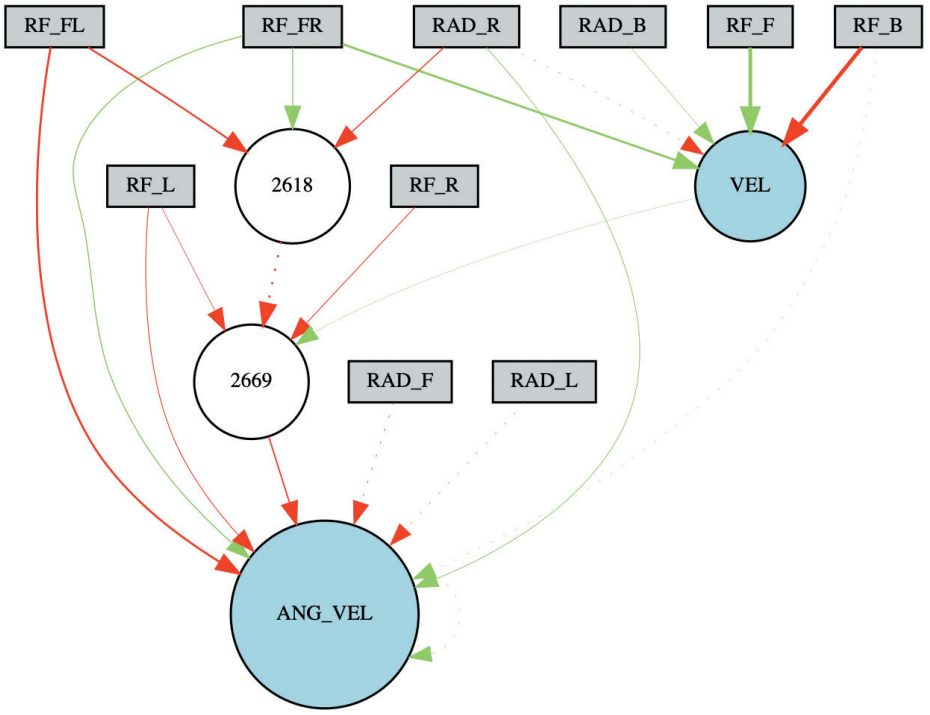


Рис. 6.9. Топология наиболее успешной нейросети

Несмотря на то что в этом эксперименте с использованием библиотеки NEAT-Python нам не удалось до конца обучить решателя задачи лабиринта с помощью метода оптимизации поиска новизны, это скорее проблема неэффективной реализации алгоритма NEAT библиотекой, чем недостаток метода поиска новизны.

Я реализовал алгоритм NEAT на языке программирования GO, который с высокой эффективностью решает задачу прохождения сложного лабиринта. Вы можете получить код на GitHub по адресу **https://github.com/yaricom/goNEAT_ NS**.

6.6.3 Упражнение 2

В файловом архиве исходного кода для этой главы также представлена реализация движка эксперимента на основе библиотеки Python MultiNEAT, которую мы обсуждали в главе 2.

Вы можете попытаться использовать этот код для решения задачи прохождения сложного лабиринта следующим образом.

1. Обновите текущую среду Anaconda, установив библиотеку Python MultiNEAT с помощью следующей команды:

```
$ conda install -c conda-forge multineat
```

2. Запустите движок эксперимента на основе библиотеки MultiNEAT:

```
$ python maze_experiment_multineat.py -m hard -g 500 -t 10 --width 200 --height 200
```

Эти команды устанавливают библиотеку MultiNEAT в текущей среде Anaconda и запускают 10 испытаний (по 500 поколений в каждом) эксперимента со сложным лабиринтом с использованием соответствующего движка эксперимента.

6.7 Заключение

В этой главе вы узнали о методе оптимизации поиском новизны и о том, как его можно использовать для управления процессом нейроэволюции в таких проблемных условиях, как прохождение лабиринта. Мы провели те же эксперименты по прохождению лабиринта, что и в предыдущей главе. После этого сравнили полученные результаты, чтобы определить, имеет ли метод поиска новизны преимущества по сравнению с методом целеориентированной оптимизации, рассмотренным в предыдущей главе.

Вы получили практический опыт написания исходного кода на Python и экспериментировали с настройкой важных гиперпараметров алгоритма NEAT. Также познакомились с новым методом визуализации, позволяющим увидеть путь агента через лабиринт. С помощью этого метода вы можете легко сравнить, как разные агенты пытаются решить проблему навигации в лабиринте, и сделать вывод, является найденный путь через лабиринт оптимальным или нет.

В следующей главе представлены более продвинутые приложения алгоритма NEAT. Мы начнем с задачи зрительного различения и познакомим вас с расширением HyperNEAT алгоритма NEAT. Метод HyperNEAT позволяет работать с крупномасштабными нейросетями, охватывающими тысячи или миллионы параметров. Операции такого масштаба невозможны с классическим алгоритмом NEAT.

Часть III

Передовые методы нейроэволюции

В этой части обсуждаются передовые методы нейроэволюции и способы их использования для решения практических задач. Вы узнаете о продвинутых методах нейроэволюции и найдете идеи для новых проектов.

Часть состоит из следующих глав:

- главы 7 «Зрительное различение с NEAT на основе гиперкуба»;
- главы 8 «Метод ES-HyperNEAT и задача сетчатки»;
- главы 9 «Коэволюция и метод SAFE»;
- главы 10 «Глубокая нейроэволюция».

Глава 7

Зрительное различение с NEAT на основе гиперкуба

В этой главе вы узнаете об основных принципах, заложенных в основу алгоритма NEAT на основе гиперкуба, и задачах, для решения которых он был разработан. Мы рассмотрим проблемы, которые возникают при попытке использовать прямое кодирование генома с крупномасштабными искусственными нейронными сетями, и покажем, как они могут быть решены путем введения схемы косвенного кодирования генома. Вы узнаете, как *сети, производящие составные паттерны* (compositional pattern producing network, CPPN), могут использоваться для хранения информации о кодировании генома с очень высокой степенью сжатия и как CPPN используются алгоритмом HyperNEAT. Наконец, вы познакомитесь с практическими примерами, которые демонстрируют мощь алгоритма HyperNEAT.

В этой главе мы обсудим следующие темы:

- проблема с прямым кодированием масштабных естественных сетей в алгоритме NEAT, и как HyperNEAT может помочь за счет метода косвенного кодирования;
- эволюция CPPN при помощи NEAT для нахождения геометрических закономерностей в гиперкубе, что позволяет нам эффективно кодировать паттерны связей в целевой нейросети;
- применение метода HyperNEAT для обнаружения и распознавания объектов в поле зрения;
- определение целевой функции для эксперимента по зрительному различению;
- обсуждение результатов эксперимента по зрительному различению.

7.1 Технические требования

Для проведения экспериментов, описанных в этой главе, ваш компьютер должен удовлетворять следующим техническим требованиям:

- Windows 8/10, macOS 10.13 или новее, или современный Linux;
- Anaconda Distribution версия 2019.03 или новее.

Исходный код примеров этой главы можно найти в каталоге `Chapter7` в файловом архиве книги.

7.2 Косвенное кодирование нейросетей с CPPN

В предыдущих главах вы узнали о прямом кодировании нейросетей на основе позаимствованной у природы концепции генотипа, который в соотношении 1:1 отображается на фенотип, представляющий топологию нейросети. Это отображение дает возможность использовать расширенные функции алгоритма NEAT, такие как номер инновации, позволяющий отслеживать, когда в ходе эволюции появилась конкретная мутация. Каждый ген в геноме имеет определенное значение числа инноваций, что позволяет быстро и точно скрещивать родительские геномы, чтобы произвести потомство. Несмотря на то что эта возможность предоставляет огромные преимущества, а также снижает вычислительные затраты, необходимые для сопоставления родительских геномов во время рекомбинации, прямое кодирование, используемое для представления топологии фенотипа, имеет существенный недостаток, поскольку ограничивает размер кодируемой нейросети. Чем больше соответствующая нейросеть, тем больше становится геном, который оценивается во время эволюции, и это влечет за собой огромные вычислительные затраты.

Существует много задач, в первую очередь связанных с распознаванием образов в изображениях или других многомерных источниках данных, которые нуждаются в сетях со сложной топологией, включающей большое количество слоев, узлов и связей между ними. Такие конфигурации не могут быть эффективно обработаны классическим алгоритмом NEAT из-за ограничений прямого кодирования.

Для устранения этого недостатка был предложен новый метод кодирования фенотипа нейросети, сохраняющий все преимущества алгоритма NEAT. Мы обсудим его в следующем разделе.

7.2.1 Кодирование CPPN

Предложенная схема кодирования использует метод получения паттернов связей в фенотипе целевой нейросети (которую мы хотим обучить) путем запроса к другой *специализированной* нейронной сети о весах связей между узлами. Эта специализированная нейронная сеть называется CPPN. Ее основная задача состоит в том, чтобы представить паттерны связей фенотипа целевой нейросети как функцию его геометрии. Результирующий паттерн связей представлен в виде четырехмерного гиперкуба. Каждая точка гиперкуба кодирует связь между двумя связанными узлами в фенотипе целевой нейросети и описывается четырьмя числами: координатами исходного узла и координатами целевого узла. В свою очередь, соединительная CPPN принимает в качестве входных данных каждую точку гиперкуба и вычисляет вес связей между каждым узлом в фенотипе целевой нейросети. Кроме того, связь между двумя узлами не экспрессируется, если значение веса связи, возвращаемого CPPN, меньше минимального порогового значения (w_{min}). Таким образом, мы можем определить соединительную CPPN как четырехмерную функцию, возвращающую вес связи в соответствии со следующей формулой:

$$w = CPPN(x_1, y_1, x_2, y_2).$$

Исходный узел фенотипа целевой нейросети находится в (x_1, y_1), а конечный узел в (x_2, y_2).

Другая существенная особенность CPPN состоит в том, что в отличие от обычных нейросетей, которые используют только один тип функции активации для каждого узла (обычно из семейства сигмовидных функций), CPPN могут использовать в качестве активаторов узла несколько геометрических функций. Благодаря этому CPPN могут выражать в создаваемых паттернах связей богатый набор геометрических последовательностей:

- симметрия (функция Гаусса);
- несовершенная симметрия (функция Гаусса в сочетании с асимметричной системой координат);
- повторение (синус);
- повторение с вариациями (синус в сочетании с системой координат, которая не повторяется).

Учитывая упомянутые особенности CPPN, мы можем предположить, что создаваемый ими паттерн связей способен описывать любую топологию фенотипа целевой нейросети. Кроме того, паттерн связей может использоваться для кодирования крупномасштабных топологий путем обнаружения закономерностей в обучающих данных и повторного использования того же набора генов в CPPN для кодирования повторений в фенотипе целевой нейросети.

7.2.2 Нейроэволюция с развитием топологии на основе гиперкуба

Упомянутая выше методика была изобретена Кеннетом О. Стэнли и получила название *нейроэволюции с развитием топологии на основе гиперкуба* (hypercube-based neuroevolution of augmenting topologies, HyperNEAT). Как следует из названия, это расширение алгоритма NEAT, который мы уже использовали в этой книге. Основное различие между старым и новым методами состоит в том, что метод HyperNEAT использует схему косвенного кодирования, основанную на CPPN. В ходе эволюции метод HyperNEAT применяет алгоритм NEAT для развития популяции геномов, кодирующих топологию соединительной CPPN. После этого каждая созданная CPPN может использоваться для формирования паттернов связей в пределах конкретного фенотипа целевой нейросети. Наконец, этот фенотип нейросети можно оценить в пространстве предметной области.

До сих пор мы рассуждали о том, как паттерны связей развиваются с использованием NEAT и CPPN и как они могут применяться к узлам фенотипа целевой нейросети. Однако мы не упомянули, откуда берется начальное геометрическое расположение узлов. Ответственность за объявление узлов и их позиции (расположение) возлагается на человека-архитектора. Архитектор анализирует пространство предметной области и использует наиболее подходящий макет.

Исходную компоновку узлов фенотипа целевой нейросети принято называть *субстратом*. Существует несколько типов конфигурации субстрата (макета), доказавших свою эффективность при решении конкретных задач:

- *двумерная сетка* – регулярная сетка узлов сети в двумерном декартовом пространстве с центром в (0,0);
- *трехмерная сетка* – регулярная сетка узлов сети в трехмерном декартовом пространстве с центром в (0,0,0);
- *сэндвич пространства состояний* – две двумерные плоские сетки с соответствующими исходными и целевыми узлами, в которых один слой может отправлять соединения только в направлении другого;
- *круговая* – регулярная радиальная структура, подходящая для определения закономерностей в радиальной геометрии на основе полярных координат.

Располагая узлы нейросети на субстрате в подходящей компоновке, можно использовать закономерности в геометрии предметной области. Это значительно повышает эффективность кодирования за счет использования соединительной CPPN для создания паттернов связей между узлами субстрата. Далее мы обсудим основы эксперимента по зрительному различению.

Более подробное описание метода HyperNEAT приведено в главе 1.

7.3 Основы эксперимента по зрительному различению

Как мы уже упоминали, основным преимуществом косвенного кодирования, используемого алгоритмом HyperNEAT, является возможность экономично кодировать топологию крупномасштабной нейросети. В этом разделе мы рассмотрим эксперимент, нацеленный на проверку способности метода HyperNEAT обучать крупномасштабные нейросети. Задачи зрительного распознавания образов обычно нуждаются в больших нейросетях из-за высокой размерности входных данных (высота изображения, умноженная на ширину изображения). В этой главе мы рассмотрим разновидность задачи распознавания образов, именуемую задачей зрительного различения.

Задача зрительного различения состоит в том, чтобы отличить крупный объект от маленького объекта в двумерном зрительном пространстве независимо от их расположения в поле зрения и положения относительно друг друга. Задача зрительного различения решается специализированной различающей нейросетью – *зрительным дискриминатором*, – которая развивается на субстрате, сконфигурированном как сэндвич пространства состояний с двумя слоями:

- *зрительное поле* представляет собой двумерный массив датчиков, которые могут находиться в двух состояниях: включено или выключено (черно-белое изображение);
- *целевое поле* – это двумерный массив выходов со значениями активации в диапазоне [0,1].

Схема задачи зрительного различения показана на рис. 7.1.

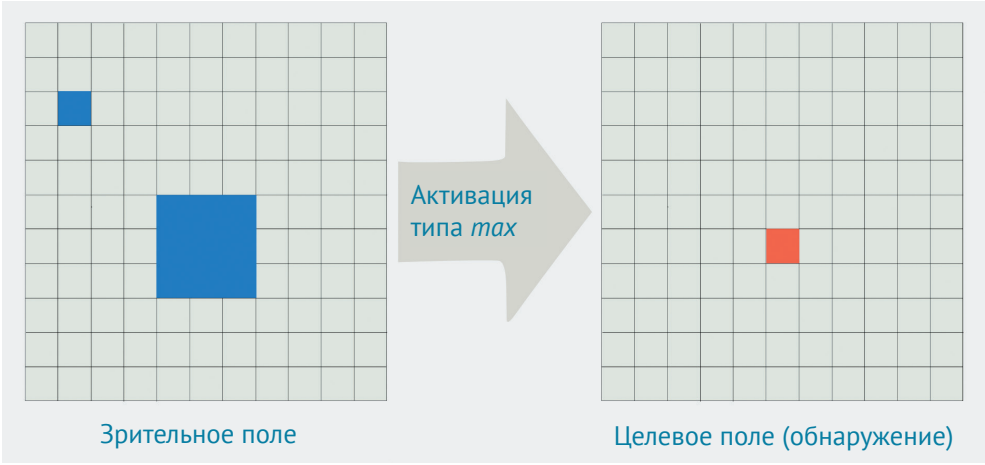


Рис. 7.1. Задача зрительного различения

На рисунке мы видим, что различаемые объекты представлены в виде двух квадратов, разделенных пустым пространством. Большой объект в три раза превосходит маленький по каждому измерению. Алгоритм, который мы пытаемся построить, должен найти центр более крупного объекта. Обнаружение основано на измерении значений активации узлов нейросети в целевом поле. Положение узла с наивысшим значением активации указывает на центр обнаруженного объекта. Наша цель – найти правильный паттерн связей между зрительным полем и целевым полем, который сопоставляет выходной узел с наибольшей активацией в целевом поле с центром большого объекта в зрительном поле. Кроме того, результат работы нейросети не должен зависеть от взаимного расположения объектов.

Алгоритм для задачи зрительного различения должен оценивать большое количество входных данных – значений, представляющих ячейки в зрительном поле. Кроме того, успешный алгоритм должен обнаружить стратегию, способную обрабатывать входные данные из нескольких ячеек одновременно. Такая стратегия должна основываться на общем принципе, позволяющем определять относительные размеры объектов в зрительном поле, которое в нашем эксперименте представлено в виде двумерной сетки. Следовательно, общим геометрическим принципом, который должен быть обнаружен, является локальность.

Мы можем использовать принцип локальности в конфигурации целевой (различающей) нейросети, применяя определенный паттерн в схеме соединений между узлами зрительного поля и целевого поля. В этой схеме соединений отдельные узлы зрительного поля соединены со множеством смежных узлов вывода вокруг определенного местоположения в целевом поле. В результате активация выходного узла зависит от того, как много сигналов поступает в него через соединения с отдельными входными узлами.

Чтобы эффективно использовать упомянутый ранее принцип локальности, представление связей должно учитывать геометрию субстрата различающей нейросети и тот факт, что правильный паттерн связей повторяется по

всей сети. Наилучшим кандидатом для такого представления является CPPN, которая может один раз обнаружить локальный паттерн связей и повторить его по сетке субстрата для сколь угодно большого разрешения.

7.3.1 Определение целевой функции

Основная задача *зрительного дискриминатора* – правильно определить положение более крупного объекта независимо от взаимного расположения обоих объектов. Отсюда мы можем определить целевую функцию для управления процессом нейроэволюции. Целевая функция должна основываться на евклидовом расстоянии между точным положением более крупного объекта в поле зрения и его прогнозируемым положением в целевом поле.

Функция ошибки может быть представлена непосредственно как евклидово расстояние между фактическим и прогнозируемым положениями следующим образом:

$$\mathcal{L} = \sqrt{\sum_{i=1}^{2} \left(G_i - P_i\right)^2},$$

где $\mathcal{L}$ – функция ошибки, G_i – истинные координаты большого объекта, а P_i – координаты, предсказанные нейросетью.

Используя функцию ошибки, определенную ранее, мы можем представить целевую функцию следующим образом:

$$\mathcal{F}_n = 1.0 - \frac{\mathcal{L}}{D_{max}}.$$

Здесь D_{max} – максимально возможное расстояние между двумя точками в пределах целевого пространства поля. Формула целевой функции гарантирует, что рассчитанный показатель приспособленности ($\mathcal{F}_n$) всегда находится в интервале [0,1]. Теперь, когда мы сформировали основы эксперимента по зрительному различению, можно заняться непосредственно подготовкой к эксперименту.

7.4 Подготовка эксперимента по зрительному различению

В нашем эксперименте, во время обучения нейросети зрительного дискриминатора, мы используем фиксированное разрешение зрительного и целевого полей, с размерностью 11×11. Следовательно, соединительная CPPN должна как-то связать между собой 121 вход зрительного поля и 121 выход целевого поля, что в итоге потенциально может дать 14 641 значение веса связи.

На рис. 7.2 показана схема субстрата для нейросети зрительного дискриминатора.

Нейросеть дискриминатора на рис. 7.2 имеет два слоя с узлами, образующими по одной двумерной плоской сетке на слой. Соединительная CPPN «рисует» паттерны связей, проводя связи от узлов одного уровня к узлам другого уровня.

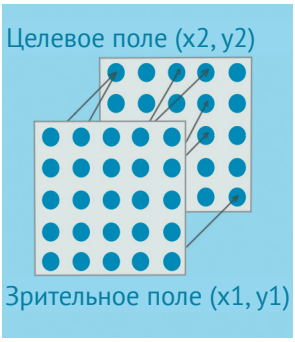


Рис. 7.2. Субстрат пространства состояний для нейросети зрительного дискриминатора

На каждом поколении эволюции каждый индивидуум в популяции (геном, кодирующий CPPN) оценивается на его способность создавать паттерны связей нейросети дискриминатора. Затем зрительный дискриминатор проверяется, чтобы определить, может ли он найти центр большого объекта в поле зрения. Всего для текущей нейросети предусмотрено 75 оценочных испытаний, в которых два объекта расположены в разных местах. В каждом испытании мы помещаем маленький объект в одну из 25 позиций, равномерно распределенных в поле зрения. Центр большого объекта находится в пяти шагах от маленького объекта вправо, вниз или по диагонали. Если большой объект не полностью вписывается в поле зрения, он переходит на другую сторону. Таким образом, учитывая логику размещения объектов относительно друг друга и сетки, мы должны суметь оценить все возможные конфигурации в 75 испытаниях.

Среда эксперимента состоит из двух основных частей, которые мы обсудим в следующих разделах.

7.4.1 Тестовая среда зрительного дискриминатора

Сначала нам нужно определить тестовую среду и предоставить доступ к набору данных, который содержит все возможные конфигурации зрительных полей, как сказано в предыдущем разделе. Набор данных, используемый в этом эксперименте, создается во время инициализации тестовой среды. Мы обсудим создание набора данных позже.

Тестовая среда состоит из двух основных компонентов:

- структура данных для хранения определений зрительного поля;
- менеджер тестовой среды, который хранит набор данных и предоставляет средства для оценки нейросети дискриминатора.

Далее вы познакомитесь с подробным описанием этих компонентов.

Определение зрительного поля

Конфигурация зрительного поля для каждого из упомянутых ранее 75 испытаний сохраняется в классе Python `VisualField`. Он имеет следующий конструктор:

```
def __init__(self, big_pos, small_pos, field_size):
    self.big_pos = big_pos
    self.small_pos = small_pos
    self.field_size = field_size
    self.data = np.zeros((field_size, field_size))
    # Позиция маленького объекта.
    self._set_point(small_pos[0], small_pos[1])

    # Позиция большого объекта.
    offsets = [-1, 0, 1]
    for xo in offsets:
        for yo in offsets:
            self._set_point(big_pos[0] + xo, big_pos[1] + yo)
```

Конструктор `VisualField` принимает в качестве параметров кортеж с координатами (*x*, *y*) большого и маленького объектов, а также размер зрительного поля. Мы рассматриваем квадратное поле, поэтому размер зрительного поля одинаковый вдоль каждой оси. Внутренним представлением зрительного поля является двумерный двоичный массив, где единицы представляют позиции, занятые объектами, а нули – это пустые пространства. Он хранится в поле `self.data`, которое представляет собой массив NumPy с размерностью (2, 2).

Маленький объект имеет размер 1×1, а размерность большого объекта в три раза больше. Следующий фрагмент из исходного кода конструктора создает представление большого объекта в массиве данных:

```
offsets = [-1, 0, 1]
for xo in offsets:
    for yo in offsets:
        self._set_point(big_pos[0] + xo, big_pos[1] + yo)
```

Конструктор класса `VisualField` получает координаты центра большого объекта в виде кортежа (`x`, `y`). Предыдущий фрагмент кода рисует большой объект, начиная с верхнего левого угла (`x-1`, `y-1`) и заканчивая нижним правым углом (`x+1`, `y+1`).

Функция `_set_point(self,x,y)`, упомянутая в предыдущем фрагменте, устанавливает значение `1.0` в определенной позиции в поле `self.data`:

```
def _set_point(self, x, y):
    px, py = x, y
    if px < 0:
        px = self.field_size + px
    elif px >= self.field_size:
        px = px - self.field_size

    if py < 0:
        py = self.field_size + py
    elif py >= self.field_size:
        py = py - self.field_size

    self.data[py, px] = 1 # В Numpy индекс имеет вид [строка, столбец]
```

Функция `_set_point(self, x, y)` выполняет перенос координат, когда значение координат превышает допустимое количество измерений на ось. Например, для оси *x* исходный код для переноса значений координат выглядит следующим образом:

```
if px < 0:
    px = self.field_size + px
elif px >= self.field_size:
    px = px - self.field_size
```

Исходный код для переноса координат вдоль оси *y* аналогичен.

После переноса координат, указанных в качестве параметров функции (при необходимости), мы присваиваем соответствующим позициям в поле `self.data` значение `1.0`.

NumPy выполняет индексацию в виде [`строка, столбец`]. Поэтому мы должны поставить *y* в первую позицию и *x* во вторую позицию индекса.

Рабочая среда зрительного дискриминатора

Рабочая среда зрительного дискриминатора содержит сгенерированный набор данных с описаниями зрительного поля. Кроме того, она предоставляет методы для создания набора данных и оценки нейросети дискриминатора на конкретном наборе данных. Класс Python `VDEnvironment` содержит определения всех упомянутых методов, а также соответствующих структур данных. Далее мы рассмотрим все важные компоненты класса `VDEnvironment`.

- Конструктор класса определяется следующим образом:

```
def __init__(self, small_object_positions, big_object_offset,
             field_size):
    self.s_object_pos = small_object_positions
    self.data_set = []
    self.b_object_offset = big_object_offset
    self.field_size = field_size

    self.max_dist = self._distance((0, 0),
                        (field_size - 1, field_size - 1))

    # Создание тестового набора данных
    self._create_data_set()
```

Первый параметр конструктора `VDEnvironment` представляет собой массив с определениями всех возможных положений малых объектов, определенных как последовательность значений координат для каждой оси. Второй параметр определяет смещение координат центра большого объекта от координат малого объекта. В нашем эксперименте в качестве значения этого параметра мы используем `5`. Наконец, третий параметр – это размер зрительного поля в двух измерениях.

Сохранив все полученные параметры в полях объекта, мы вычисляем максимально возможное расстояние между двумя точками в зрительном поле следующим образом:

```
self.max_dist = self._distance((0, 0),
                    (field_size - 1, field_size - 1))
```

Евклидово расстояние между верхним левым и нижним правым углами зрительного поля сохраняется в поле `self.max_dist`. Это значение будет использовано позже для нормализации расстояний между точками зрительного поля, чтобы сохранить их в интервале `[0,1]`.

- Функция `_create_data_set()` создает все возможные наборы данных с учетом заданных параметров среды. Исходный код этой функции выглядит следующим образом:

```
def _create_data_set(self):
    for x in self.s_object_pos:
        for y in self.s_object_pos:
            # Диагональ
            vf = self._create_visual_field(x, y,
                            x_off=self.b_object_offset,
                            y_off=self.b_object_offset)
            self.data_set.append(vf)
            # Вправо
            vf = self._create_visual_field(x, y,
                            x_off=self.b_object_offset,
                            y_off=0)
            self.data_set.append(vf)
            # Вниз
            vf = self._create_visual_field(x, y,
                            x_off=0,
                            y_off=self.b_object_offset)
            self.data_set.append(vf)
```

Функция выполняет итерацию по позициям маленького объекта вдоль двух осей и пытается создать большой объект по координатам, расположенным справа, снизу или по диагонали от координат маленького объекта.

- Функция `_create_visual_field` создает соответствующую конфигурацию зрительного поля, используя координаты маленького объекта (`sx`, `sy`) и смещение центра большого объекта (`x_off`, `y_off`). Следующий исходный код показывает, как это реализовано:

```
def _create_visual_field(self, sx, sy, x_off, y_off):
    bx = (sx + x_off) % self.field_size # Перенос по координате X
    by = (sy + y_off) % self.field_size # Перенос по координате Y

    # Создаем зрительное поле.
    return VisualField(big_pos=(bx, by), small_pos=(sx, sy),
                       field_size=self.field_size)
```

Если координаты большого объекта, вычисленные предыдущей функцией, находятся вне пространства зрительного поля, мы применяем перенос следующим образом:

```
if bx >= self.field_size:
    bx = bx - self.field_size # Перенос
```

Предыдущий фрагмент показывает перенос по оси *x*. Перенос по оси *y* выполняется аналогично. Наконец, мы создаем объект `VisualField` и возвращаем его для добавления в набор данных.

- Тем не менее наиболее интересная часть описания `VDEnvironment` связана с оценкой нейросети дискриминатора, которая выполняется в функции `evaluate_net(self, net)` следующим образом:

```
def evaluate_net(self, net):
    avg_dist = 0

    # Вычисление предсказанной позиции
    for ds in self.data_set:
        # Вычисляем выходные значения
        outputs, x, y = self.evaluate_net_vf(net, ds)

        # Находим расстояние до большого объекта
        dist = self._distance((x, y), ds.big_pos)
        avg_dist = avg_dist + dist

    avg_dist /= float(len(self.data_set))
    # Нормализация ошибки предсказания позиции
    error = avg_dist / self.max_dist
    # Приспособленность
    fitness = 1.0 - error

    return fitness, avg_dist
```

Эта функция получает нейросеть зрительного дискриминатора в качестве параметра и возвращает вычисленную оценку приспособленности и среднее расстояние между предсказанными и истинными координатами большого объекта, рассчитанное для всех рассмотренных зрительных полей. Среднее расстояние рассчитывается следующим образом:

```
for ds in self.data_set:
    # Вычисляем выходные значения
    _, x, y = self.evaluate_net_vf(net, ds)

    # Находим расстояние до большого объекта
    dist = self._distance((x, y), ds.big_pos)
    avg_dist = avg_dist + dist
avg_dist /= float(len(self.data_set))
```

Предыдущий исходный код перебирает все объекты `VisualField` в наборе данных и использует нейросеть зрительного дискриминатора для предска-

зания координат большого объекта. После этого мы вычисляем расстояние (ошибку предсказания) между истинным и предсказанным положениями большого объекта. Наконец, находим среднее значение ошибки предсказания и нормализуем его следующим образом:

```
# Нормализованная ошибка предсказания
error = avg_dist / self.max_dist
```

В соответствии с предыдущим кодом максимально возможное значение ошибки составляет `1.0`. Значение показателя приспособленности вычисляется как разность между `1.0` и значением ошибки, поскольку приспособленность возрастает по мере уменьшения ошибки:

```
# fitness
fitness = 1.0 - error
```

Функция `evaluate_net` возвращает рассчитанный показатель приспособленности вместе с ненормализованной ошибкой обнаружения.

- Функция `evaluate_net_vf(self, net, vf)` предоставляет средства для оценки нейросети дискриминатора на конкретном объекте `VisualField` и реализована в следующем блоке кода:

```
def evaluate_net_vf(self, net, vf):
    depth = 1 # У нас только 2 слоя

    net.Flush()
    # Подготовка входа
    inputs = vf.get_data()
    net.Input(inputs)
    # Активация
    [net.Activate() for _ in range(depth)]

    # Получаем выходы
    outputs = net.Output()
    # Находим координаты большого объекта
    x, y = self._big_object_coordinates(outputs)

    return outputs, x, y
```

- Предыдущая функция получает нейросеть дискриминатора в качестве первого параметра и объект `VisualField` в качестве второго параметра. После этого она формирует развернутый входной массив из объекта `VisualField` и использует его в качестве входных данных для нейросети дискриминатора:

```
inputs = vf.get_data()
net.Input(inputs)
```

После того как мы определили входы дискриминатора, он должен быть активирован для распространения входных значений по всем сетевым узлам. Наша нейросеть дискриминатора имеет только два слоя, что определяется пространственной конфигурацией субстрата. Следовательно, нам нужно

активировать ее дважды – по одному разу для каждого слоя. После распространения сигнала активации через оба уровня нейросети дискриминатора мы можем определить положение большого объекта в целевом поле через индекс максимального значения в выходном массиве. Используя функцию `_big_object_coordinates(self, outputs)`, мы можем извлечь декартовы координаты (*x*, *y*) большого объекта в целевом поле.

Наконец, функция `evaluate_net_vf` возвращает необработанный выходной массив вместе с извлеченными декартовыми координатами (*x*, *y*) большого объекта в пространстве целевого поля.

- Функция `_big_object_coordinates(self, outputs)` извлекает декартовы координаты большого объекта в пространстве целевого поля из необработанных выходных данных, полученных из нейросети дискриминатора. Исходный код функции выглядит следующим образом:

```
def _big_object_coordinates(self, outputs):
    max_activation = -100.0
    max_index = -1
    for i, out in enumerate(outputs):
        if out > max_activation:
            max_activation = out
            max_index = i

    # Получение координат точки максимальной активации
    x = max_index % self.field_size
    y = int(max_index / self.field_size)

    return (x, y)
```

Сначала функция перебирает выходной массив и находит индекс максимального значения:

```
max_activation = -100.0
max_index = -1
for i, out in enumerate(outputs):
    if out > max_activation:
        max_activation = out
        max_index = I
```

После этого она использует найденный индекс для вычисления декартовых координат с учетом размера целевого поля:

```
x = max_index % self.field_size
y = int(max_index / self.field_size)
```

Наконец, функция возвращает кортеж (*x*, *y*) с декартовыми координатами большого объекта в целевом поле.

Полный исходный код можно найти в файле `vd_environment.py` в файловом архиве книги.

7.4.2 Движок эксперимента

Как мы упоминали ранее, задачу зрительного различения можно решить с помощью метода HyperNEAT. Для этого нам нужно воспользоваться библиотекой, которая обеспечивает реализацию алгоритма HyperNEAT. Библиотека MultiNEAT Python является подходящим кандидатом для нашего эксперимента, поэтому дальше мы будем использовать эту библиотеку.

Далее обсудим наиболее важные компоненты движка эксперимента.

Полный исходный код можно найти в файле `vd_experiment_multineat.py` в файловом архиве книги.

Функция движка эксперимента

Функция `run_experiment` позволяет нам проводить эксперимент, используя доступные гиперпараметры и инициализированную тестовую среду зрительного дискриминатора. Код функции состоит из нескольких важных частей.

Инициализация первой популяции геномов CPPN

В представленном далее блоке кода сначала мы инициализируем начальное число генератора случайных чисел текущим системным временем. После этого мы создаем подходящую конфигурацию субстрата для нейросети дискриминатора, способную работать со зрительным полем заданной размерности. Затем на основе конфигурации субстрата мы создаем геном CPPN:

```
# Начальное значение генератора случайных чисел.
seed = int(time.time())
# Создаем субстрат.
substrate = create_substrate(num_dimensions)
# Создаем геном CPPN и популяцию.
g = NEAT.Genome(0,
                substrate.GetMinCPPNInputs(),
                0,
                substrate.GetMinCPPNOutputs(),
                False,
                NEAT.ActivationFunction.UNSIGNED_SIGMOID,
                NEAT.ActivationFunction.UNSIGNED_SIGMOID,
                0,
                params, 0)
pop = NEAT.Population(g, params, True, 1.0, seed)
pop.RNG.Seed(seed)
```

Геном CPPN, созданный в предыдущем коде, имеет необходимое количество входных и выходных узлов, предоставляемых субстратом. Сначала в качестве функции активации узла он использует беззнаковый сигмоид. Позже, в ходе эволюции тип функции активации каждого узла CPPN будет изменен в соответствии с процедурами алгоритма HyperNEAT. Наконец, на основе инициализированного генома CPPN и гиперпараметров HyperNEAT создается исходная популяция.

Запуск нейроэволюции в течение нужного числа поколений

Сначала мы создаем промежуточные переменные для хранения результатов выполнения и сборщик статистики (`Statistics`). После этого выполняем цикл эволюции для числа поколений, указанного в параметре `n_generations`:

```
start_time = time.time()
best_genome_ser = None
best_ever_goal_fitness = 0
best_id = -1
solution_found = False

stats = Statistics()
for generation in range(n_generations):
```

В рамках цикла эволюции мы получаем список геномов, принадлежащих к популяции в текущем поколении, и оцениваем все геномы из списка на приспособленность к условиям тестовой среды следующим образом:

```
genomes = NEAT.GetGenomeList(pop)
# Получаем геномы.
genome, fitness, distances = eval_genomes(genomes,
                                vd_environment=vd_environment,
                                substrate=substrate,
                                generation=generation)
stats.post_evaluate(max_fitness=fitness, distances=distances)
solution_found = fitness >= FITNESS_THRESHOLD
```

Мы сохраняем в сборщик статистики значения, возвращенные функцией `eval_genomes(genomes, substrate, vd_environment, generation)` для текущего поколения. Кроме того, используем возвращаемую функцией оценку приспособленности, чтобы оценить, было найдено успешное решение или нет. Если показатель приспособленности превышает значение `FITNESS_THRESHOLD`, мы считаем, что было найдено успешное решение.

Далее, если было найдено успешное решение или текущий показатель приспособленности является максимальным из когда-либо достигнутых показателей приспособленности, мы сохраняем геном CPPN и текущий показатель приспособленности:

```
if solution_found or best_ever_goal_fitness < fitness:
    best_genome_ser = pickle.dumps(genome)
    best_ever_goal_fitness = fitness
    best_id = genome.GetID()
```

Кроме того, если будет найдено успешное решение, мы прерываем цикл эволюции и переходим к этапу вывода отчетов, который обсудим позже:

```
if solution_found:
    print('Solution found at generation: %d, best fitness: %f,
    species count: %d' % (generation, fitness, len(pop.Species)))
    break
```

Если удачное решение не было найдено, мы выводим в консоль статистику для текущего поколения и переходим к следующему поколению:

```
# Переходим к следующему поколению
pop.Epoch()
# Выводим статистику в консоль
gen_elapsed_time = time.time() - gen_time
print("Best fitness: %f, genome ID: %d" % (fitness, best_id))
print("Species count: %d" % len(pop.Species))
print("Generation elapsed time: %.3f sec" % (gen_elapsed_time))
print("Best fitness ever: %f, genome ID: %d"
      % (best_ever_goal_fitness, best_id))
```

После завершения основного цикла эволюции в консоль выводятся результаты эксперимента, основанные на статистике, собранной в цикле.

Сохранение результатов эксперимента

Результаты эксперимента собираются и сохраняются в текстовом и графическом представлениях (файлы SVG). Мы начнем с вывода на печать общей статистики производительности алгоритма:

```
print("\nBest ever fitness: %f, genome ID: %d"
      % (best_ever_goal_fitness, best_id))
print("\nTrial elapsed time: %.3f sec" % (elapsed_time))
print("Random seed:", seed)
```

Первые три строки данного кода выводят на консоль лучшую оценку приспособленности, полученную среди всех поколений эволюции. После этого мы выводим на печать продолжительность эксперимента и используемое случайное начальное значение.

Если мы захотим сохранить или показать визуализацию, то будут вызваны соответствующие функции:

```
# Визуализация результатов эксперимента
show_results = not silent
if save_results or show_results:
    net = NEAT.NeuralNetwork()
    best_genome.BuildPhenotype(net)
    visualize.draw_net(net, view=show_results, node_names=None,
                       directory=trial_out_dir, fmt='svg')
```

Этот фрагмент кода рисует сетевой граф CPPN и выводит на печать статистику графа. Далее мы переходим к визуализации вывода нейросети дискриминатора:

```
# Визуализация активаций от наилучшего генома
net = NEAT.NeuralNetwork()
best_genome.BuildHyperNEATPhenotype(net, substrate)
# Выбор случайного зрительного поля
index = random.randint(0, len(vd_environment.data_set) - 1)
vf = vd_environment.data_set[index]
# Отображение активаций
outputs, x, y = vd_environment.evaluate_net_vf(net, vf)
visualize.draw_activations(outputs, found_object=(x, y), vf=vf,
          dimns=num_dimensions, view=show_results,
```

```
            filename=os.path.join(trial_out_dir,
                                  "best_activations.svg"))
```

В предыдущем коде мы создаем нейросеть зрительного дискриминатора, используя лучший геном CPPN, найденный в ходе эволюции. После этого представляем в графическом виде выходные данные активации, полученные путем запуска нейросети дискриминатора в тестовой среде. Мы используем зрительное поле, которое выбирается случайным образом из набора данных эксперимента.

Наконец, отображаем общую статистику, собранную во время эксперимента:

```
# Visualize statistics
visualize.plot_stats(stats, ylog=False, view=show_results,
          filename=os.path.join(trial_out_dir, 'avg_fitness.svg'))
```

Графическое отображение статистики включает в себя лучшие оценки приспособленности и средние расстояния ошибок, собранные за поколения эволюции.

Полный исходный код функций визуализации, упомянутых в этом разделе, можно найти в файле `visualize.py` из файлового архива книги.

Функция конструктора субстрата

Метод HyperNEAT построен вокруг понятия субстрата, определяющего структуру нейросети дискриминатора. Поэтому крайне важно создать правильную конфигурацию субстрата, которая будет задействована во время выполнения эксперимента. Процедуры создания субстрата определены в следующих двух функциях.

- Функция конструктора субстрата `create_substrate` создает объект субстрата:

```
def create_substrate(dim):
    # Строим конфигурации входных и выходных декартовых листов
    inputs = create_sheet_space(-1, 1, dim, -1)
    outputs = create_sheet_space(-1, 1, dim, 0)
    substrate = NEAT.Substrate( inputs, [], # hidden outputs)
    substrate.m_allow_input_output_links = True
    ...
    substrate.m_hidden_nodes_activation = \
                NEAT.ActivationFunction.SIGNED_SIGMOID
    substrate.m_output_nodes_activation = \
                NEAT.ActivationFunction.UNSIGNED_SIGMOID
    substrate.m_with_distance = True
    substrate.m_max_weight_and_bias = 3.0
    return substrate
```

Данная функция сначала создает два разбитых на сетку декартовых листа, которые представляют входные данные (зрительное поле) и выходные данные (целевое поле) в конфигурации субстрата. Как вы помните, для этого

эксперимента мы выбрали конфигурацию слоев типа «сэндвич». Затем экземпляр субстрата инициализируется в соответствии с созданными конфигурациями полей:

```
inputs = create_sheet_space(-1, 1, dim, -1)
outputs = create_sheet_space(-1, 1, dim, 0)
substrate = NEAT.Substrate( inputs, [], # hidden outputs)
```

Обратите внимание, что субстрат не использует никаких скрытых узлов; вместо них мы предоставляем пустой список.

Затем мы настраиваем субстрат, чтобы разрешить только соединения, направленные от входных к выходным узлам, и использовать сигмоидную функцию активации со знаком на выходных узлах. Наконец, мы устанавливаем максимальные значения для смещения и веса соединения.

- Функция `create_sheet_space`, вызываемая конструктором субстрата, определяется следующим образом:

```
def create_sheet_space(start, stop, dim, z):
    lin_sp = np.linspace(start, stop, num=dim)
    space = []
    for x in range(dim):
        for y in range(dim):
            space.append((lin_sp[x], lin_sp[y], z))

    return space
```

Функция `create_sheet_space` получает начальную и конечную координаты сетки в одном измерении вместе с количеством измерений сетки. Также указана координата *z* листа. Используя указанные параметры, предыдущий код создает равномерное линейное пространство с координатами, начинающимися в диапазоне `[start, stop]`, с шагом `dim`:

```
lin_sp = np.linspace(start, stop, num = dim)
```

Затем мы используем это линейное пространство, чтобы заполнить двумерный массив координатами узлов сетки:

```
space = []
for x in range(dim):
for y in range(dim):
space.append((lin_sp[x], lin_sp[y], z))
```

Функция `create_sheet_space` возвращает конфигурацию сетки в виде двумерного массива.

Оценка приспособленности

Оценка приспособленности генома является важной частью любого алгоритма нейроэволюции, включая метод HyperNEAT. Как вы видели, главный цикл эксперимента вызывает функцию `eval_genomes` для оценки приспособленности всех геномов в популяции для каждого поколения. Давайте рассмотрим детали

реализации оценки приспособленности, которая состоит из двух основных функций.

- Функция `eval_genomes` оценивает все геномы в популяции:

```
def eval_genomes(genomes, substrate, vd_environment, generation):
    best_genome = None
    max_fitness = 0
    distances = []
    for genome in genomes:
        fitness, dist = eval_individual(genome, substrate,
                                        vd_environment)
        genome.SetFitness(fitness)
        distances.append(dist)

        if fitness > max_fitness:
            max_fitness = fitness
            best_genome = genome
    return best_genome, max_fitness, distances
```

Функция `eval_genomes` в качестве параметров принимает список геномов, конфигурацию субстрата нейросети дискриминатора, инициализированную тестовую среду и идентификатор текущего поколения. Первые строки функции создают промежуточные переменные для хранения результатов оценки:

```
best_genome = None
max_fitness = 0
distances = []
```

После этого мы перебираем все геномы популяции и собираем соответствующую статистику:

```
for genome in genomes:
    fitness, dist = eval_individual(genome, substrate,
                                    vd_environment)
    genome.SetFitness(fitness)
    distances.append(dist)

    if fitness > max_fitness:
        max_fitness = fitness
        best_genome = genome
```

Наконец, функция `eval_genomes` возвращает собранную статистику в виде кортежа (`best_genome`, `max_fitness`, `distance`).

Функция `eval_individual` позволяет нам оценить приспособленность отдельного генома:

```
def eval_individual(genome, substrate, vd_environment):
    # Создает нейросеть из генома CPPN и субстрата.
    net = NEAT.NeuralNetwork()
    genome.BuildHyperNEATPhenotype(net, substrate)

    fitness, dist = vd_environment.evaluate_net(net)
    return fitness, dist
```

Вначале вышеприведенный исходный код создает фенотип нейросети дискриминатора с использованием генома CPPN, предоставленного в качестве параметра. После этого фенотип нейросети дискриминатора оценивается на приспособленность к тестовой среде.

Функция `eval_individual` возвращает оценку приспособленности и величину ошибки, полученные из тестовой среды во время оценки фенотипа. Теперь, когда мы завершили настройку, давайте приступим к эксперименту по зрительному различению.

7.5 Эксперимент по зрительному различению объектов

Выполнив все необходимые шаги по настройке, мы готовы начать эксперимент.

В эксперименте по зрительному различению мы используем следующую конфигурацию зрительного поля:

Параметр	Значение
Размер видимого поля	11×11
Положение мелких объектов в поле зрения вдоль каждой оси	[1, 3, 5, 7, 9]
Размер маленького объекта	1×1
Размер большого объекта	3×3
Смещение центра большого объекта от маленького объекта	5

Далее нужно выбрать подходящие значения гиперпараметров HyperNEAT, что позволит нам найти успешное решение задачи зрительного различения.

Обратите внимание, что гиперпараметр, который мы опишем далее, определяет, как развивать соединительную CPPN, используя процесс нейроэволюции. Нейросеть зрительного дискриминатора создается путем наложения соединительной CPPN на субстрат.

7.5.1 Выбор гиперпараметра

Библиотека MultiNEAT использует класс `Parameters` для хранения всех необходимых гиперпараметров. Чтобы установить соответствующие значения гиперпараметров, мы определяем функцию `create_hyperparameters` в скрипте Python для запуска эксперимента. Здесь мы опишем основные гиперпараметры, которые оказывают существенное влияние на производительность алгоритма HyperNEAT в этом эксперименте.

1. Функция `create_hyperparameters` начинается с создания объекта `Parameters` для хранения параметров HyperNEAT:

```
params = NEAT.Parameters()
```

2. Мы решили начать с популяции геномов среднего размера, чтобы ускорить вычисления. В то же время мы хотим сохранить достаточное количество организмов в популяции для эволюционного разнообразия. Численность популяции определяется следующим образом:

```
params.PopulationSize = 150
```

3. Мы заинтересованы в создании компактных геномов CPPN, которые имеют как можно меньше узлов для повышения эффективности косвенного кодирования. Поэтому устанавливаем очень маленькую вероятность добавления нового узла во время эволюции, а также оставляем довольно низкую вероятность создания новой связи:

```
params.MutateAddLinkProb = 0.1
params.MutateAddNeuronProb = 0.03
```

4. Метод HyperNEAT создает геномы CPPN с различными типами функций активации в скрытых и выходных узлах. Следовательно, мы должны определить вероятность мутации, которая меняет тип активации узла. Кроме того, в этом эксперименте мы заинтересованы в использовании только четырех типов функции активации: гауссовой со знаком, сигмоиды со знаком, синусоиды со знаком и линейной функции. Мы устанавливаем вероятности выбора любого из упомянутых четырех типов активации равными `1.0`, что фактически уравнивает вероятность выбора каждого типа:

```
params.MutateNeuronActivationTypeProb = 0.3
params.ActivationFunction_SignedGauss_Prob = 1.0
params.ActivationFunction_SignedSigmoid_Prob = 1.0
params.ActivationFunction_SignedSine_Prob = 1.0
params.ActivationFunction_Linear_Prob = 1.0
```

5. Наконец, мы определяем количество видов в популяции, которое должно оставаться в интервале `[5,10]`, и устанавливаем допустимую длительность стагнации вида в течение 100 поколений. Эта конфигурация поддерживает умеренное видовое разнообразие, но сохраняет виды достаточно долго, чтобы позволить им развиваться и создавать полезные конфигурации генома CPPN:

```
params.SpeciesMaxStagnation = 100
params.MinSpecies = 5
params.MaxSpecies = 10
```

Представленные здесь гиперпараметры продемонстрировали высокую эффективность получения успешных геномов CPPN в ходе эволюции.

7.5.2 Настройка рабочей среды

В этом эксперименте мы используем библиотеку MultiNEAT, которая обеспечивает реализацию алгоритма HyperNEAT. Нам остается лишь создать соответствующую среду Python, которая включает в себя библиотеку MultiNEAT и все необходимые зависимости. Это можно сделать с помощью Anaconda, выполнив следующие команды в командной строке:

```
$ conda create --name vd_multineat python=3.5
$ conda activate vd_multineat
$ conda install -c conda-forge multineat
$ conda install matplotlib
```

```
$ conda install -c anaconda seaborn
$ conda install graphviz
$ conda install python-graphviz
```

Эти команды создают и активируют виртуальную среду vd_multineat на основе Python 3.5. После этого они устанавливают последнюю версию библиотеки MultiNEAT вместе с зависимостями, которые используются нашим кодом для визуализации результатов.

7.5.3 Запуск эксперимента по зрительному различению

Чтобы запустить эксперимент, вам нужно перейти в локальный каталог, содержащий скрипт vd_experiment_multineat.py, и выполнить следующую команду:

```
$ python vd_experiment_multineat.py
```

Не забудьте активировать соответствующую виртуальную среду с помощью команды $ conda activ vd_multineat.

Спустя определенное количество поколений будет найдено успешное решение, и вы увидите в окне терминала строки наподобие таких:

```
****** Generation: 16 ******

Best fitness: 0.995286, genome ID: 2410
Species count: 11
Generation elapsed time: 3.328 sec
Best fitness ever: 0.995286, genome ID: 2410

****** Generation: 17 ******
Solution found at generation: 17, best fitness: 1.000000, species count: 11

Best ever fitness: 1.000000, genome ID: 2565

Trial elapsed time: 57.753 sec
Random seed: 1568629572

CPPN nodes: 10, connections: 16

Running test evaluation against random visual field: 41
Substrate nodes: 242, connections: 14641
found (5, 1)
target (5, 1)
```

В данном консольном выводе говорится, что решение было найдено в 17-м поколении. Идентификатор успешного генома CPPN – 2565, и этот геном имеет 10 узлов и 16 связей между ними. Кроме того, вы можете увидеть результаты оценки нейросети дискриминатора, полученного с помощью лучшего генома CPPN в отношении случайно выбранного зрительного поля.

В нашем случае найденные декартовы координаты большого объекта в целевом поле и фактические координаты в зрительном поле совпадают (5, 1), что свидетельствует о способности найденного решения различать объекты с высокой точностью.

Далее интересно взглянуть на визуальное представление уровней активации выходов нейросети зрительного дискриминатора, полученных во время тестового прогона (рис. 7.3).

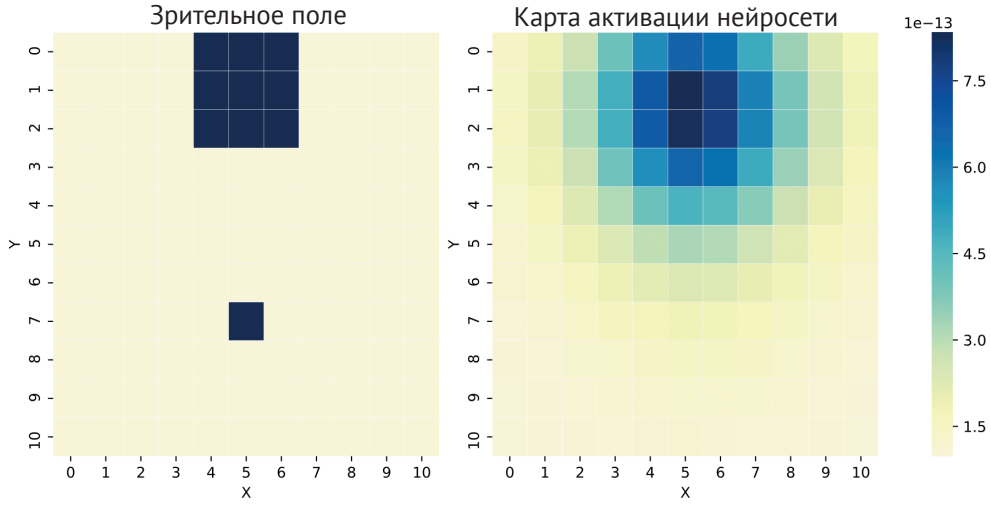


Рис. 7.3. Активации целевого поля зрительного дискриминатора

Правая часть рисунка отображает значения активации целевого поля (выходного слоя) нейросети дискриминатора, которые мы получили во время оценки на случайном зрительном поле. В левой части графика вы можете увидеть фактическую конфигурацию зрительного поля. Как видите, максимальное значение активации целевого поля (самая темная ячейка) находится точно в той же позиции, что и центр большого объекта с координатами в зрительном поле (`5,1`).

Мы также видим, что значениям активации нейросети присущ чрезвычайно низкий уровень: минимальная активация составляет ~ 1×10^{-13}, а максимальная – только ~ 9×10^{-13}. Разработанная человеком нейросеть, вероятно, была бы нормализована так, чтобы выходной сигнал лежал в интервале `[0,1]`, имея минимум, близкий к нулю, и максимум, близкий к единице. Однако нам достаточно, чтобы активация просто имела максимум в нужном месте, и нейросеть вправе выбрать любую схему активации выходов, которую большинство людей сочли бы необычной.

Следующий график позволяет изучить ход процесса эволюции в течение нескольких поколений и сделать вывод о том, насколько хорошо полученные соединительные CPPN справляются с задачей построения успешной нейросети зрительного дискриминатора (рис. 7.4).

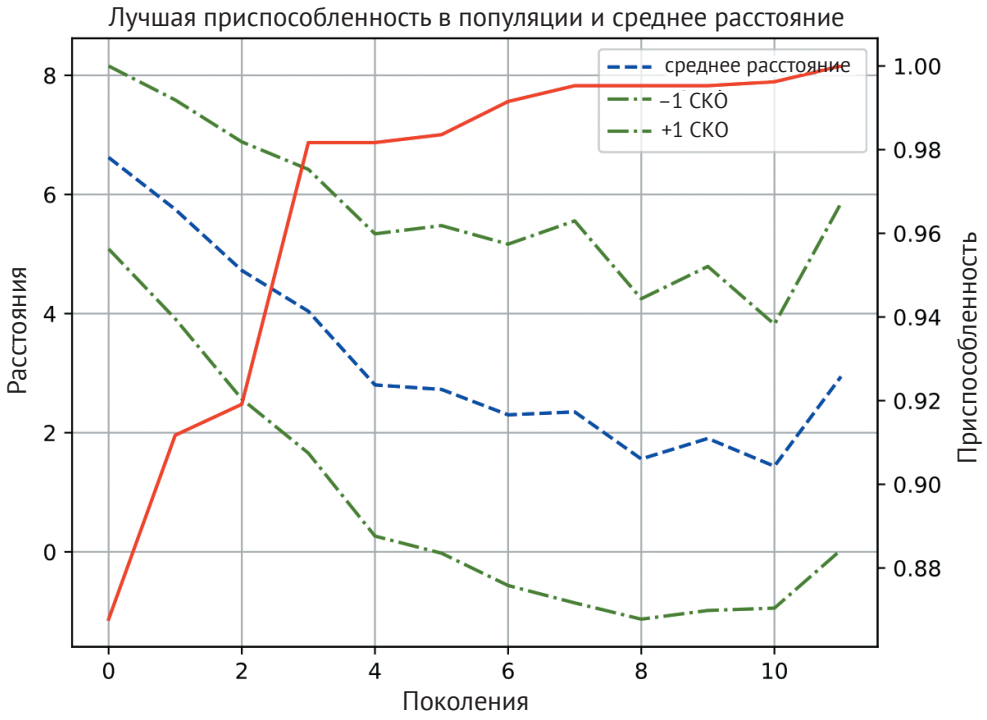


Рис. 7.4. Наилучшие оценки приспособленности и средние ошибки для нейросети зрительного дискриминатора

На рис. 7.4 показано изменение оценок приспособленности (восходящая линия) и средних ошибок (нисходящая линия) для каждого поколения эволюции. Вы можете видеть, что показатели приспособленности почти достигли максимального значения уже в третьем поколении эволюции, но потребовалось еще семь поколений, чтобы проработать конфигурации генома CPPN и найти победителя. Кроме того, вы можете видеть, что среднее расстояние между предсказанным и истинным положениями большого объекта постепенно уменьшается в процессе эволюции.

Однако самая захватывающая часть данного эксперимента представлена на графе фенотипа CPPN (рис. 7.5).

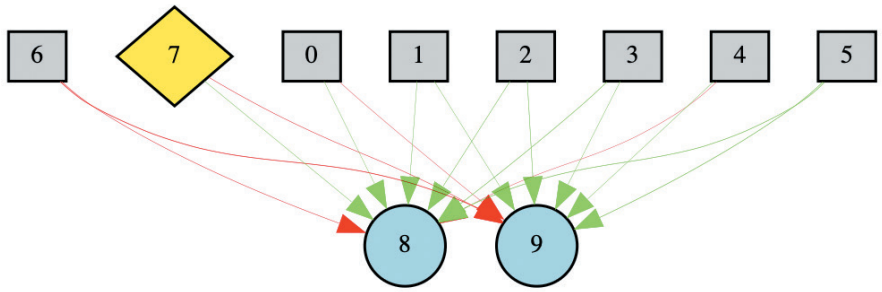


Рис. 7.5. Граф фенотипа CPPN лучшего генома

Граф демонстрирует топологию сети фенотипа CPPN, который использовался для построения связей в нейросети успешного зрительного дискриминатора. На графе фенотипа CPPN входные узлы помечены квадратами, выходные узлы – закрашенные кружки, а узел смещения – ромб.

Два выходных узла CPPN имеют следующее значение:

- первый узел (8) предоставляет вес связи;
- второй узел (9) определяет, экспрессирована ли связь.

Назначение входных узлов CPPN определено следующим образом:

- первые два узла (0 и 1) задают координаты точки (x, y) во входном слое субстрата;
- следующие два узла (2 и 3) задают координаты точки (x, y) в скрытом слое субстрата (не использовались в нашем эксперименте);
- следующие два узла (4 и 5) задают координаты точки (x, y) в выходном слое субстрата;
- последний узел (6) задает евклидово расстояние от точки на входном слое до начала координат.

Также вы можете видеть, что фенотип CPPN не содержит скрытых узлов. Нейроэволюционному процессу удалось найти подходящие типы функций активации для выходных узлов CPPN в задаче зрительного различения. Это решение позволяет соединительной CPPN «выращивать» правильные паттерны связей на субстрате нейросети дискриминатора.

Подсчитав количество узлов и соединений между ними, представленных на графе рис. 7.5, вы можете почувствовать мощь метода косвенного кодирования, предложенного алгоритмом HyperNEAT. Имея всего 16 соединений между 10 узлами, фенотип CPPN смог закодировать паттерн связей, который способен покрыть субстрат зрительного поля разрешением 11×11, потенциально имеющего 14 641 соединение между узлами зрительного и целевого полей. Таким образом, мы достигли степени сжатия информации около 0,11 %, что весьма впечатляет.

Такая высокая степень сжатия стала возможной благодаря обнаружению геометрических закономерностей в связях субстрата соединительной CPPN. Используя обнаруженные закономерности, CPPN может обойтись только несколькими паттернами (мотивами локальной связности) для всего пространства связей субстрата. После этого CPPN может неоднократно применять эти локальные паттерны в разных позициях субстрата, чтобы нарисовать полную схему связей между слоями субстрата – в нашем случае чтобы нарисовать связи между входным слоем (зрительное поле) и выходным слоем (целевое поле).

7.6 Упражнения

1. Попытайтесь уменьшить значение параметра `params.PopulationSize` и посмотрите, что произойдет. Как это повлияло на производительность алгоритма?
2. Попробуйте установить нулевые вероятности для значений следующих гиперпараметров: `params.ActivationFunction_SignedGauss_Prob`, `params.ActivationFunction_SignedSigmoid_Prob` и `params.ActivationFunction_Signed-`

`Sine_Prob`. Было ли найдено успешное решение с этими изменениями? Как это повлияло на конфигурацию связей субстрата?

3. Распечатайте геном-победитель, попробуйте придумать визуализацию, а затем посмотрите, насколько ваше представление о геноме совпадает с визуализированной CPPN.

7.7 Заключение

В этой главе мы узнали о методе косвенного кодирования топологии нейросети с использованием CPPN. Узнали о расширении HyperNEAT алгоритма NEAT, использующем соединительную CPPN для выявления паттернов связей в субстрате фенотипа нейросети зрительного дискриминатора. Также мы продемонстрировали, как схема косвенного кодирования позволяет алгоритму HyperNEAT работать с топологиями крупномасштабных нейросетей, что часто встречается в задачах распознавания образов и зрительного различения.

Благодаря полученным теоретическим знаниям у вас появилась возможность улучшить свои навыки кодирования, реализовав решение задачи зрительного различения объектов с использованием Python и библиотеки MultiNEAT. Кроме того, вы узнали о новом методе визуализации, изображающем уровни активации узлов в выходном слое нейросети зрительного дискриминатора, и научились использовать эту визуализацию для проверки решения.

В следующей главе мы обсудим, как можно улучшить метод HyperNEAT, введя автоматическое создание соответствующей конфигурации субстрата. Мы рассмотрим расширение Evolvable Substrate HyperNEAT (ES-HyperNEAT) алгоритма NEAT и посмотрим, как его можно применять для решения практических задач, требующих наличия модульной топологии нейросети решающего устройства.

Глава 8

Метод ES-HyperNEAT и задача сетчатки

В этой главе вы познакомитесь с расширением ES-HyperNEAT метода HyperNEAT, о котором мы говорили в предыдущей главе. Как вы уже знаете, метод HyperNEAT позволяет кодировать топологии крупномасштабных искусственных нейронных сетей, что важно для работы в областях, где входные данные имеют большое количество измерений, таких как компьютерное зрение. Однако, несмотря на всю свою мощь, метод HyperNEAT имеет существенный недостаток – конфигурация субстрата должна быть заранее разработана архитектором-человеком. Метод ES-HyperNEAT решает эту проблему путем введения концепции *развиваемого субстрата*, который позволяет автоматически создавать необходимую конфигурацию субстрата в процессе эволюции.

После ознакомления с основами метода ES-HyperNEAT у вас будет возможность применить эти знания для решения *задачи модульной сетчатки*. В ходе обсуждения этой задачи вы узнаете, как выбрать подходящую начальную конфигурацию субстрата, которая поможет эволюционному процессу выявить модульные структуры. Кроме того, мы обсудим исходный код решателя задачи модульной сетчатки вместе с тестовой средой, которую можно использовать для оценки приспособленности нейросети детектора.

Изучив эту главу, вы получите практический опыт применения метода ES-HyperNEAT с использованием библиотеки MultiNEAT.

В этой главе мы рассмотрим следующие темы:

- сравнение ручного и эволюционного формирования топографии узлов;
- извлечение информации из квадродерева и основы ES-HyperNEAT;
- эксперимент с двусторонней модульной сетчаткой;
- обсуждение результатов эксперимента.

8.1 Технические требования

Для проведения экспериментов, описанных в этой главе, ваш компьютер должен удовлетворять следующим техническим требованиям:

- Windows 8/10, macOS 10.13 или новее, или современный Linux;
- Anaconda Distribution версия 2019.03 или новее.

Исходный код примеров этой главы можно найти в каталоге `Chapter8` в файловом архиве книги.

8.2 Ручное и эволюционное формирование топографии узлов

Метод HyperNEAT, который мы обсуждали в главе 7, позволяет нам использовать нейроэволюцию для решения различных задач, требующих наличия крупномасштабных нейросетевых структур. Этот класс задач охватывает множество прикладных областей, включая визуальное распознавание образов. Главной отличительной чертой всех этих задач является высокая размерность входных/выходных данных.

В предыдущей главе вы узнали, как определить конфигурацию субстрата дискриминатора для решения задачи зрительного различения. Вы также узнали, что крайне важно использовать правильную конфигурацию субстрата, которая соответствует геометрическим особенностям пространства поиска конкретной задачи. Используя метод HyperNEAT, вы как архитектор должны заранее определить конфигурацию субстрата, используя только ваше понимание пространственной геометрии задачи. Однако не всегда возможно заранее узнать обо всех геометрических закономерностях, скрытых за конкретным пространством поиска.

Если вы проектируете субстрат вручную, то создаете непреднамеренное ограничение для паттерна весов, налагаемого поверх субстрата соединительной сетью (CPPN). Размещая узлы в определенных местах на субстрате, вы мешаете способности CPPN обнаруживать геометрические закономерности мира природы. CPPN вынуждена создавать паттерн связей, который идеально соответствует структуре придуманного вами субстрата, где связи возможны только между узлами этой структуры. Это ограничение приводит к ненужным ошибкам аппроксимации, которые портят результаты, когда вы используете усовершенствованную CPPN для создания топологии нейросети решателя (фенотип).

Однако почему мы должны начинать с ограничений, которые налагает ручная настройка субстрата? Не лучше ли позволить CPPN самостоятельно детализировать схемы связей между узлами субстрата, которые автоматически размещаются в правильных местах? Похоже, что развиваемые паттерны связей в субстрате предоставляют ценные неявные подсказки, помогающие оценить размещение узлов для следующей эпохи эволюции. Поход, основанный на эволюции конфигурации субстрата во время обучения CPPN, получил название *развиваемого субстрата* (evolvable substrate, ES).

Неявные данные, позволяющие нам предсказать положение следующего узла, представляют собой некоторый объем информации, закодированный паттерном связей в конкретной области субстрата. Области с равномерным распределением весов связей содержат небольшое количество информации, что требует наличия лишь нескольких узлов субстрата в этих областях. В то же время области субстрата с большими градиентами весов связей являются информационно насыщенными и могут извлечь выгоду из дополнительных узлов, размещенных в этих областях. Размещая дополнительные узлы в таких областях субстрата, вы помогаете CPPN формировать гораздо более детальную кодировку сущностей окружающего мира. Иными словами, размещение узлов и паттерн связей могут зависеть от распределения весов связей, в то время как CPPN находит веса связей в ходе эволюции.

HyperNEAT представляет каждую связь между двумя узлами субстрата в виде точки в четырехмерном гиперкубе. Алгоритм HyperNEAT с развиваемым субстратом расширяет HyperNEAT, автоматически размещая меньше гиперточек в областях гиперкуба с меньшим градиентом весов связей. Следовательно, ES-HyperNEAT использует плотность информации в качестве основного критерия при формировании топологии субстрата в процессе эволюции. В следующем разделе мы обсудим особенности алгоритма ES-HyperNEAT.

8.3 Извлечение информации из квадродерева и основы ESHyperNEAT

Для эффективного расчета плотности информации в паттернах связей субстрата мы должны использовать соответствующую структуру данных. Нам необходимо применять структуру данных, которая позволяет выполнять эффективный поиск в двухмерном пространстве субстрата на разных уровнях детализации. В информатике существует структура данных, которая идеально соответствует нашим потребностям. Эта структура называется квадродеревом.

Квадродерево – это структура данных, которая позволяет организовать эффективный поиск по двумерному пространству разбиением любой интересующей области на четыре подрегиона. Соответственно, каждый из этих подрегионов становится листом дерева, а корневой узел представляет начальную область.

ES-HyperNEAT использует структуру данных квадродерева для итеративного поиска новых соединений и узлов в субстрате, начиная с входных и выходных узлов, предварительно определенных специалистом по данным. Использование квадродерева для поиска новых соединений и узлов намного эффективнее в вычислительном отношении, чем поиск в четырехмерном пространстве гиперкуба.

Схема извлечения информации с использованием квадродерева показана на рис. 8.1.

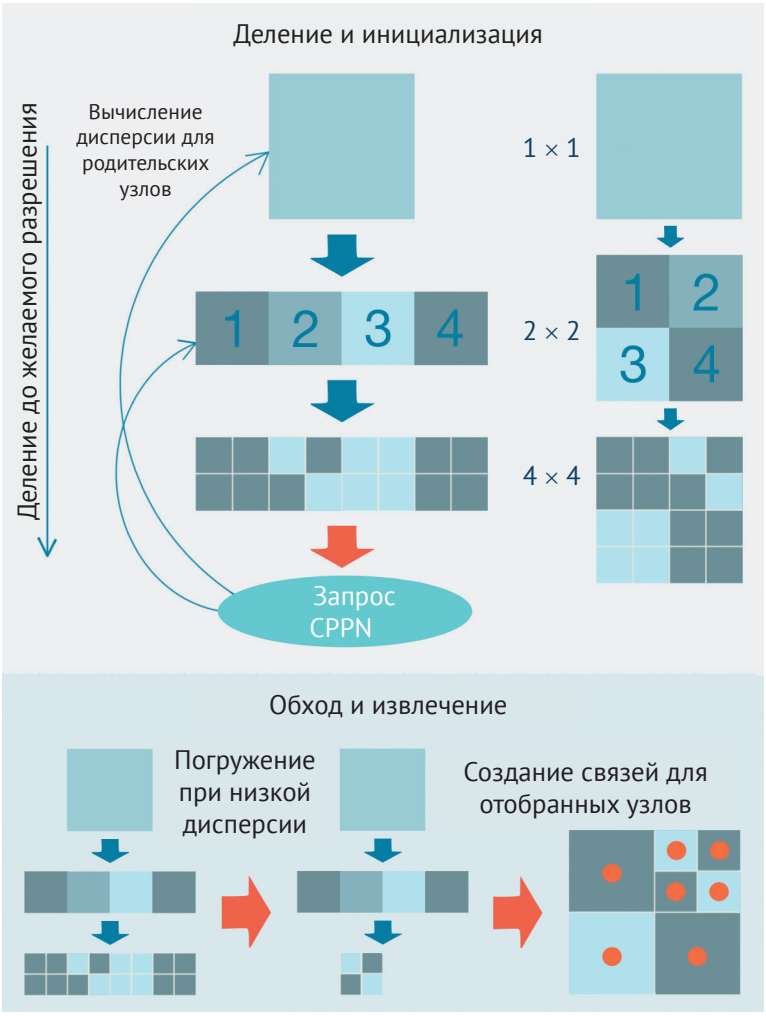


Рис. 8.1. Схема извлечения информации с использованием квадродерева

Метод извлечения информации из квадродерева состоит из двух основных частей.

1. В верхней части рис. 8.1 представлен этап деления и инициализации. На этом этапе квадродерево создается путем рекурсивного деления начальной области субстрата, которая охватывает от (-1,-1) до (1,1). Деление прекращается, когда достигается желаемая глубина квадродерева. Теперь у нас есть несколько подрегионов, которые составляют субстрат, определяя начальное разрешение субстрата (r). Затем для каждого узла квадродерева с центром в (x_i,y_i) мы обращаемся к CPPN, чтобы найти вес соединения (w) между этим узлом и конкретным входным или выходным нейроном в координатах (a, b). После того как мы вычислили веса связей для k конечных узлов в поддереве квадродерева p, мы готовы вычислить информационную дисперсию узла в квадродереве следующим образом:

$$\sigma^2 = \sum_{i=1}^{k} \left(\overline{w} - w_i \right)^2 .$$

Здесь $\overline{w}$ – средний вес соединения среди k конечных узлов и w_i – вес соединения с каждым конечным узлом.

Мы можем использовать это оценочное значение дисперсии в качестве эвристического показателя плотности информации в конкретном подрегионе субстрата. Чем выше это значение, тем выше плотность информации. Дисперсию можно использовать для управления плотностью информации в конкретном подрегионе субстрата путем введения *порога деления*. Если дисперсия превышает порог деления, то этап деления повторяется до тех пор, пока не будет достигнута требуемая плотность информации.

На этом этапе мы создаем ориентировочную структуру, которая позволяет CPPN решать, где устанавливать связи в пределах данного субстрата. На следующем этапе обработки все необходимые связи размещаются с использованием созданной структуры квадродерева.

2. В нижней части рис. 8.1 представлен этап обрезки и извлечения. На этом этапе мы используем развернутую структуру квадродерева предыдущего этапа, чтобы найти регионы с высокой дисперсией и убедиться, что между узлами этих регионов экспрессировано больше связей. Мы обходим квадродерево в глубину и останавливаем обход в узле, у которого значение дисперсии меньше заданного *порога дисперсии* (σ_t^2), или когда текущий узел не имеет дочерних элементов (то есть имеет нулевую дисперсию). Для каждого узла квадродерева, найденного поиском по глубине, мы экспрессируем (делаем действующей) связь между текущим узлом (x, y) и каждым родительским узлом, который уже определен. Родительский узел может быть определен архитектором (узлы ввода/вывода) или найден в предыдущих запусках метода извлечения информации, то есть из скрытых узлов, уже созданных методом ES-HyperNEAT. Когда этот этап завершается, конфигурация субстрата будет иметь больше узлов в областях с высокой плотностью информации и меньше узлов в областях, кодирующих небольшое количество информации.

В следующем разделе мы обсудим, как использовать алгоритм ES-HyperNEAT, который мы только что описали, при решении задачи модульной сетчатки.

Для получения более подробной информации об алгоритме ES-HyperNEAT см. главу 1.

8.4 Основы задачи модульной сетчатки

Иерархические модульные структуры являются неотъемлемой частью сложных биологических организмов и играют незаменимую роль в их эволюции. Модульность повышает способность к развитию, позволяя выполнять рекомбинацию различных модулей в процессе эволюции. Развитая

иерархия модульных компонентов ускоряет процесс эволюции, позволяя выполнять операции над совокупностью сложных структур, а не над базовыми генами. При таком подходе нейроэволюционному процессу не нужно тратить времени на повторное развитие необходимой функциональности. Вместо этого готовые к использованию модульные компоненты могут использоваться в качестве строительных блоков для создания очень сложных нейронных сетей.

В данной главе мы будем искать решение задачи сетчатки при помощи ES-HyperNEAT. *Задача сетчатки* заключается в одновременном опознавании подходящих рисунков с разрешением 2×2 левой и правой частями искусственной сетчатки с разрешением 4×2. Инициализируя среду эксперимента, мы создаем рисунки, предназначенные только для правой стороны, только для левой стороны или для обеих сторон одновременно, а затем случайным образом «показываем» их нейросети зрительного детектора. Нейросеть должна решить, являются ли рисунки, проецируемые на левую и правую стороны сетчатки, подходящими для соответствующей стороны сетчатки (левой или правой).

В задаче сетчатки левый и правый компоненты четко разделены на различные функциональные единицы. В то же время одни компоненты могут присутствовать на каждой стороне сетчатки, тогда как другие являются уникальными для определенной стороны сетчатки. Таким образом, чтобы создать успешную нейросеть детектора, процесс нейроэволюции должен найти модульные структуры отдельно для левой и правой стороны.

Задача сетчатки схематически показана на рис. 8.2.

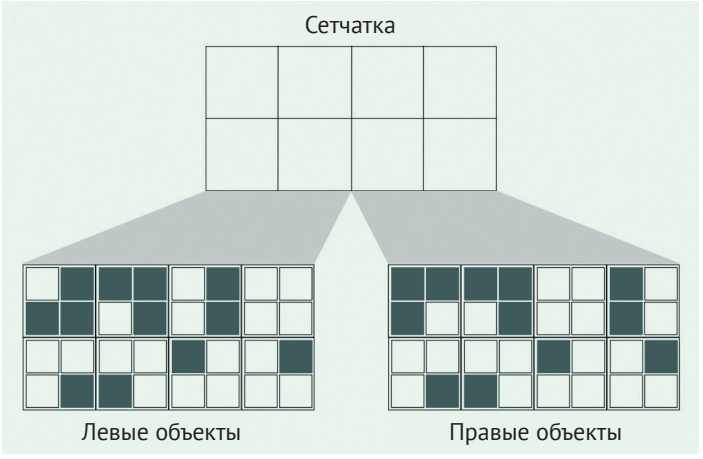


Рис. 8.2. Схематическое представление задачи сетчатки

На этой схеме искусственная сетчатка представлена в виде 2D-сетки с разрешением 4×2 пикселей. Значения двумерного массива, представляющие рисунки, спроецированные на сетчатку, составляют входы нейросети детектора. Закрашенным пикселям в массиве соответствует значение `1.0`, а пустым пикселям соответствует значение `0.0`. С данным разрешением можно нарисо-

вать 16 различных рисунков 2×2 для левой и правой частей сетчатки. Мы назначим восемь подходящих образцов для левой стороны сетчатки и восемь подходящих образцов для правой стороны сетчатки. Некоторые из упомянутых рисунков являются подходящими для обеих сторон сетчатки.

Схема принятия решения нейросетью детектора в пространстве решений задачи сетчатки выглядит, как показано на рис. 8.3.

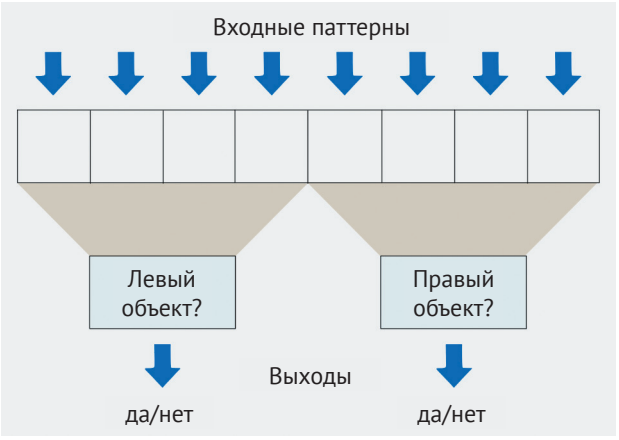


Рис. 8.3. Схема принятия решения нейросетью детектора

Нейросеть детектора имеет восемь входов для ввода входных данных с обеих сторон сетчатки и два выходных узла. Каждый из выходных узлов производит значение, которое можно использовать для классификации рисунка на каждой стороне сетчатки. Первый выходной узел присвоен левой, а второй выходной узел – правой стороне сетчатки соответственно. Значение активации выходного узла, которое больше или равно `0.5`, классифицирует рисунок для соответствующей стороны сетчатки как подходящий. Если значение активации меньше `0.5`, рисунок считается неподходящим. Чтобы еще больше упростить детектирование, мы применяем округление к значениям выходных узлов до уровня «да/нет», как показано на рисунке. Таким образом, каждый выходной узел нейросети детектора служит двоичным классификатором для соответствующей стороны сетчатки и выдает значение `0.0` или `1.0`, чтобы пометить входной рисунок соответственно как неподходящий или подходящий.

8.4.1 Определение целевой функции

Задача нейросети детектора состоит в том, чтобы правильно классифицировать входы с левой и правой сторон сетчатки как подходящие или нет, создав вектор двоичных выходов со значениями `0.0` или `1.0`. Выходной вектор имеет длину 2, которая равна числу выходных узлов.

Мы можем определить ошибку классификации как евклидово расстояние между вектором с контрольными данными и вектором с выходными значениями нейросети, как сделано в следующей формуле:

$$e^2 = \sum_{i=1}^{2} (a_i - b_i)^2.$$

Здесь e^2 – квадрат ошибки классификации для одного испытания, a – вектор выходных данных нейросети детектора и b – вектор с контрольными данными.

На каждом поколении эволюции мы оцениваем каждую нейросеть детектора (фенотип) по всем 256 возможным комбинациям рисунков сетчатки 4×2, которые получаются путем комбинирования 16 различных рисунков 2×2 для каждой стороны сетчатки. Таким образом, чтобы получить окончательное значение ошибки классификации для конкретной нейросети детектора, мы вычисляем сумму 256 значений ошибок, полученных для каждой конфигурации рисунков сетчатки, как указано в следующей формуле:

$$\mathcal{E} = \sum_{i=1}^{256} e_i^2.$$

Здесь $\mathcal{E}$ является суммой всех ошибок, полученных в ходе 256 испытаний, и e_i^2 является квадратом ошибки классификации для конкретного испытания.

Функция приспособленности может быть определена как обратная сумма ошибок, полученных из всех 256 испытаний на всех возможных рисунках сетчатки, как показано в следующей формуле:

$$\mathcal{F} = \frac{1000}{1 + \mathcal{E}}.$$

Мы добавляем 1 к сумме ошибок ($\mathcal{E}$) в знаменателе, чтобы избежать деления на 0 в тех случаях, когда все испытания не дают ошибок. Таким образом, в соответствии с формулой функции приспособленности максимальное значение показателя приспособленности, которое мы позже будем использовать в качестве порогового значения приспособленности, в нашем эксперименте составляет 1000.

8.5 Подготовка эксперимента с модульной сетчаткой

В этом разделе мы обсудим детали эксперимента, направленного на поиск успешного решения задачи модульной сетчатки. В нашем эксперименте мы используем эту задачу в качестве эталона для проверки способности метода ES-HyperNEAT обнаруживать модульные топологии в фенотипе нейросети.

8.5.1 Начальная конфигурация субстрата

Как было сказано ранее в этой главе, сетчатка имеет размеры 4×2, с двумя областями 2×2, одна с левой стороны и одна с правой стороны. Особенности геометрии сетчатки должны быть представлены в геометрии исходной конфигурации субстрата. В нашем эксперименте мы используем трехмерный субстрат, показанный на рис. 8.4.

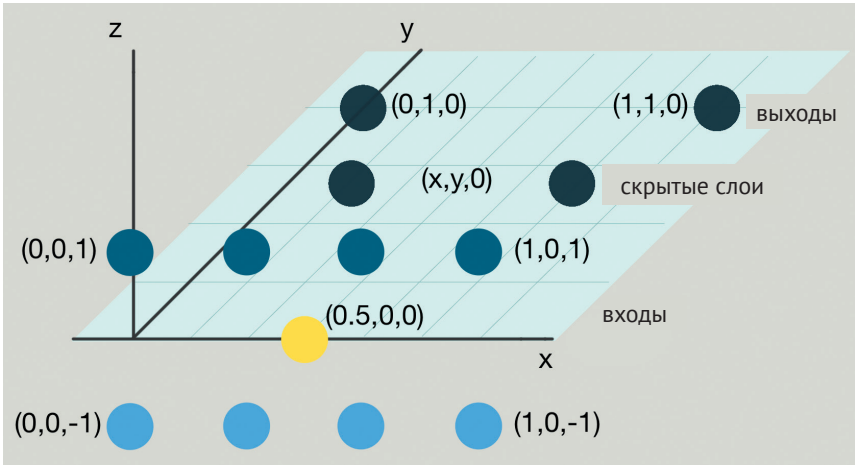


Рис. 8.4. Начальная конфигурация субстрата

Как видно по этому рисунку, входные узлы расположены в плоскости XZ, которая ортогональна плоскости XY. Они представлены в двух группах, с четырьмя узлами для описания левой и правой сторон сетчатки. Два выходных узла и узлы смещения расположены в плоскости XY, которая делит плоскость Z пополам между группами входных узлов. Эволюция субстрата создает новые скрытые узлы в той же плоскости XY, где расположены выходные узлы. Развитая соединительная CPPN рисует паттерны связей между всеми узлами внутри субстрата. Наша конечная цель состоит в том, чтобы развить CPPN и конфигурацию субстрата, которая дает подходящий модульный граф нейросети детектора. Этот граф должен включать два модуля, каждый из которых представляет соответствующую конфигурацию для двоичного классификатора, рассмотренного нами ранее. Далее мы перейдем к тестовой среде для задачи модульной сетчатки.

8.5.2 Тестовая среда для задачи модульной сетчатки

Мы начнем с создания тестовой среды, которая будет использоваться для оценки результатов процесса нейроэволюции, направленного на создание успешной нейросети детектора. Тестовая среда должна создать набор рисунков, включающий все возможные сочетания расположения пикселей на сетчатке. Кроме того, она должна предоставлять функции для оценки нейросети по каждому рисунку в наборе данных. Таким образом, тестовую среду можно разделить на две основные части:

- структура данных для хранения рисунков, относящихся к левой, правой или обеим сторонам сетчатки;
- тестовая среда, хранящая набор данных и предоставляющая функции для оценки нейросети.

Далее мы подробно рассмотрим каждую часть.

Определение визуального объекта

Каждая из разрешенных конфигураций пикселей в определенной части пространства сетчатки может быть представлена как отдельный визуальный объект. Класс Python, инкапсулирующий соответствующую функциональность, называется `VisualObject` и определяется в файле `retina_experiment.py`. Он имеет следующий конструктор:

```
def __init__(self, configuration, side, size=2):
    self.size = size
    self.side = side
    self.configuration = configuration
    self.data = np.zeros((size, size))
    # Чтение конфигурации
    lines = self.configuration.splitlines()
    for r, line in enumerate(lines):
        chars = line.split(" ")
        for c, ch in enumerate(chars):
            if ch == 'o':
                # Пиксель активен
                self.data[r, c] = 1.0
            else:
                # Пиксель не активен
                self.data[r, c] = 0.0
```

Конструктор получает конфигурацию конкретного визуального объекта в виде строки вместе с надлежащим местоположением этого объекта в пространстве сетчатки (справа, слева или обе стороны). После этого он назначает полученные параметры внутренним полям и создает двумерный массив данных, содержащий состояния пикселей в визуальном объекте.

Состояния пикселей получают путем анализа строки конфигурации визуального объекта следующим образом:

```
# Чтение конфигурации
lines = self.configuration.splitlines()
for r, line in enumerate(lines):
    chars = line.split(" ")
    for c, ch in enumerate(chars):
        if ch == 'o':
            # Пиксель активен
            self.data[r, c] = 1.0
        else:
            # П
            self.data[r, c] = 0.0
```

Строка конфигурации визуального объекта содержит четыре символа, исключая разрыв строки, которые определяют состояние соответствующего пикселя в визуальном объекте. Если в определенной позиции строки конфигурации стоит символ `o`, тогда пиксель в соответствующей позиции визуального объекта устанавливается в состояние ON (активен), и для этой позиции в массиве данных сохраняется значение `1.0`.

Определение среды сетчатки

Среда сетчатки создает и хранит набор данных, состоящий из всех возможных визуальных объектов, и предоставляет функции для оценки приспособленности нейросети. Ее реализация также состоит из нескольких основных частей.

Функция создания набора данных со всеми возможными визуальными объектами

В этой функции мы создаем визуальные объекты для набора данных следующим образом:

```
def create_data_set(self):
# Объекты для левой стороны сетчатки
self.visual_objects.append(VisualObject(". .\n. .",
                                        side=Side.BOTH))
self.visual_objects.append(VisualObject(". .\n. o",
                                        side=Side.BOTH))
self.visual_objects.append(VisualObject(". o\n. o",
                                        side=Side.LEFT))
self.visual_objects.append(VisualObject(". o\n. .",
                                        side=Side.BOTH))
self.visual_objects.append(VisualObject(". o\no o",
                                        side=Side.LEFT))
self.visual_objects.append(VisualObject(". .\no .",
                                        side=Side.BOTH))
self.visual_objects.append(VisualObject("o o\n. o",
                                        side=Side.LEFT))
self.visual_objects.append(VisualObject("o .\n. .",
                                       side=Side.BOTH))
```

Предыдущий код создает визуальные объекты для левой стороны сетчатки. Визуальные объекты для правой стороны создаются аналогичным образом:

```
# Объекты для правой стороны сетчатки
self.visual_objects.append(VisualObject(". .\n. .",
                                        side=Side.BOTH))
self.visual_objects.append(VisualObject("o .\n. .",
                                        side=Side.BOTH))
self.visual_objects.append(VisualObject("o .\no .",
                                        side=Side.RIGHT))
self.visual_objects.append(VisualObject(". .\no .",
                                        side=Side.BOTH))
self.visual_objects.append(VisualObject("o o\no .",
                                        side=Side.RIGHT))
self.visual_objects.append(VisualObject(". o\n. .",
                                        side=Side.BOTH))
self.visual_objects.append(VisualObject("o .\no o",
                                        side=Side.RIGHT))
self.visual_objects.append(VisualObject(". .\n. o",
                                       side=Side.BOTH))
```

Созданные визуальные объекты представляют собой набор данных для оценки приспособленности нейросети зрительного детектора, развившегося в процессе нейроэволюции на основе исходного субстрата.

Функция для оценки нейросети детектора по двум визуальным объектам

Эта функция оценивает качество нейросети зрительного детектора по отношению к двум заданным визуальным объектам – по одному на каждую сторону пространства сетчатки. Полный исходный код функции `def _evaluate (self, net, left, right, deep, debug = False)` вы можете найти в файле `retina_environment.py`.

Исходный код функции состоит из следующих основных частей.

1. Сначала мы подготавливаем входы для нейросети детектора в том порядке, в котором они определены для конфигурации субстрата:

```
inputs = left.get_data() + right.get_data()
inputs.append(0.5) # Смещение

net.Input(inputs)
```

Массив входных данных начинается с данных левой части и продолжается данными правой части. После этого в конец массива входных данных добавляется значение смещения, и полный массив передается в качестве входных данных в нейросеть детектора.

2. После заданного количества активаций нейросети выходные значения извлекаются и округляются:

```
outputs = net.Output()
outputs[0] = 1.0 if outputs[0] >= 0.5 else 0.0
outputs[1] = 1.0 if outputs[1] >= 0.5 else 0.0
```

3. Затем нам нужно вычислить квадратичную ошибку обнаружения, которая представляет собой евклидово расстояние между вектором выходных сигналов и вектором истинных значений. Сначала мы создаем вектор истинных значений:

```
left_target = 1.0 if left.side == Side.LEFT or \
                     left.side == Side.BOTH else 0.0
right_target = 1.0 if right.side == Side.RIGHT or \
                      right.side == Side.BOTH else 0.0
targets = [left_target, right_target]
```

Соответствующее значение истинности устанавливается равным `1.0`, если визуальный объект предназначен для данной стороны сетчатки или обеих сторон. В противном случае значение истинности устанавливается равным `0.0`, чтобы указать на неправильное назначение визуального объекта.

4. Наконец, квадратичная ошибка обнаружения рассчитывается следующим образом:

```
error = (outputs[0]-targets[0]) * (outputs[0]-targets[0]) + \
        (outputs[1]-targets[1]) * (outputs[1]-targets[1])
```

Функция возвращает ошибку обнаружения и выходы нейросети зрительного детектора. В следующем разделе мы обсудим реализацию движка эксперимента.

Для получения полной информации о реализации обратитесь к файлу `retina_environment.py` в файловом архиве книги.

8.5.3 Движок эксперимента

Чтобы решить задачу модульной сетчатки, нам нужно использовать библиотеку Python, которая предоставляет реализацию алгоритма ES-HyperNEAT. Если вы читали предыдущую главу, то уже знакомы с библиотекой MultiNEAT, в которой также есть реализация алгоритма ES-HyperNEAT. Мы можем использовать эту библиотеку для создания движка нового эксперимента с сетчаткой.

Давайте обсудим основные компоненты реализации движка.

Для получения полной информации о реализации обратитесь к файлу `retina_experiment.py` в файловом архиве книги.

Функция движка эксперимента

Функция `run_experiment` запускает эксперимент, используя предварительно настроенные гиперпараметры и инициализированную среду тестирования, чтобы оценить найденные нейросети детектора применительно к разным рисункам на сетчатке. Реализация функции имеет следующие важные части:

1. Сначала идет инициализация популяции исходных геномов CPPN:

```
seed = 1569777981
# Создаем субстрат
substrate = create_substrate()
# Создаем геном CPPN и популяцию
g = NEAT.Genome(0,
          substrate.GetMinCPPNInputs(),
          2, # скрытые узлы
          substrate.GetMinCPPNOutputs(),
          False,
          NEAT.ActivationFunction.TANH,
          NEAT.ActivationFunction.SIGNED_GAUSS, # скрытый
          1, # начальное количество скрытых слоев
          params,
          1) # один скрытый слой
pop = NEAT.Population(g, params, True, 1.0, seed)
pop.RNG.Seed(seed)
```

Сначала этот код присваивает такое начальное значение генератора случайных чисел, которое оказалось полезным для генерации успешных решений путем последовательного запуска многих экспериментальных испытаний. После этого мы создаем конфигурацию субстрата, подходящую для эксперимента с сетчаткой, с учетом геометрии пространства сетчатки.

Затем мы создаем исходный геном CPPN, используя конфигурацию субстрата, которую уже имеем. Геном CPPN должен иметь несколько входных и выходных узлов, совместимых с конфигурацией субстрата. Кроме того, мы инициа-

лизируем исходный геном CPPN двумя скрытыми узлами с гауссовой функцией активации, чтобы ускорить процесс нейроэволюции в правильном направлении. Скрытые узлы с гауссовой активацией начинают нейроэволюционный поиск с уклоном в сторону создания определенных топологий нейросети детектора. Этими скрытыми узлами мы вводим в паттерны связей субстрата принцип симметрии, чего и ожидаем достичь в топологии успешного детектора. Для решения задачи сетчатки мы должны найти симметричную конфигурацию нейросети, включающую два симметричных модуля классификатора.

2. Затем мы готовим промежуточные переменные для хранения результатов выполнения эксперимента и сборщик статистики. После этого запускаем цикл эволюции для заданного количества поколений:

```
start_time = time.time()
best_genome_ser = None
best_ever_goal_fitness = 0
best_id = -1
solution_found = False

stats = Statistics()
...
```

3. Внутри цикла эволюции мы получаем список геномов, принадлежащих текущей популяции, и оцениваем его в тестовой среде следующим образом:

```
# Получаем список текущих геномов
genomes = NEAT.GetGenomeList(pop)

# Оценка геномов
genome, fitness, errors = eval_genomes(genomes,
                rt_environment=rt_environment,
                substrate=substrate,
                params=params)
stats.post_evaluate(max_fitness=fitness, errors=errors)
solution_found = fitness >= FITNESS_THRESHOLD
```

Функция `eval_genomes` возвращает кортеж, который имеет следующие компоненты: наиболее подходящий геном, наивысший показатель приспособленности среди всех оцененных геномов и список ошибок обнаружения для каждого оцененного генома. Мы сохраняем соответствующие параметры в сборщике статистики и оцениваем полученную оценку соответствия по критерию завершения поиска, который определяется как константа `FITNESS_THRESHOLD` со значением `1000.0`. Эволюционный поиск успешно завершается, если лучший показатель приспособленности в популяции превышает или равен значению `FITNESS_THRESHOLD`.

4. Если было найдено успешное решение или текущий лучший показатель приспособленности популяции выше, чем максимальный показатель приспособленности, который был достигнут ранее, мы сохраняем лучший геном CPPN и текущий показатель приспособленности следующим образом:

```
if solution_found or best_ever_goal_fitness < fitness:
    # Сохраняем, чтобы заморозить состояние генома.
    best_genome_ser = pickle.dumps(genome)
    best_ever_goal_fitness = fitness
    best_id = genome.GetID()
```

5. После этого, если значение переменной `solution_found` было установлено в `True`, мы завершаем цикл эволюции:

```
if solution_found:
    print('Solution found at generation: %d, best fitness:
    %f, species count: %d' % (generation, fitness, len(pop.Species)))
    break
```

6. Если эволюция не привела к успешному решению, мы выводим в консоль статистику для текущего поколения и переходим к следующему поколению:

```
# Переход к следующему поколению.
pop.Epoch()

# Печать статистики.
gen_elapsed_time = time.time() - gen_time
print("Best fitness: %f, genome ID: %d" %
       (fitness, best_id))
print("Species count: %d" % len(pop.Species))
print("Generation elapsed time: %.3f sec" %
       (gen_elapsed_time))
print("Best fitness ever: %f, genome ID: %d" %
       (best_ever_goal_fitness, best_id))
```

Остальная часть кода движка выводит результаты эксперимента в различных форматах.

7. Мы выводим результаты эксперимента в текстовом и визуальном форматах, используя статистику, собранную в цикле эволюции. Кроме того, файлы визуализации сохраняются в локальной файловой системе в векторном формате SVG:

```
print("\nBest ever fitness: %f, genome ID: %d" %
         (best_ever_goal_fitness, best_id))
print("\nTrial elapsed time: %.3f sec" % (elapsed_time))
print("Random seed:", seed)
```

Эти строки кода выводят в консоль общую статистику о выполнении эксперимента, такую как наивысшая достигнутая оценка приспособленности, продолжительность эксперимента и начальное значение генератора случайных чисел.

8. Следующая часть кода посвящена наиболее информативной части эксперимента – визуализации результатов, – и вам следует уделить ей особое внимание. Мы начнем с визуализации сети CPPN, которая создана из лучшего генома, найденного в ходе эволюции:

```
if save_results or show_results:
# Рисуем граф CPPN.
net = NEAT.NeuralNetwork()
best_genome.BuildPhenotype(net)
visualize.draw_net(net, view=False, node_names=None,
                   filename="cppn_graph.svg",
                   directory=trial_out_dir, fmt='svg')
print("\nCPPN nodes: %d, connections: %d" %
(len(net.neurons), len(net.connections)))
```

9. После этого мы визуализируем топологию нейросети детектора, которая создана с использованием лучшего генома CPPN и субстрата сетчатки:

```
net = NEAT.NeuralNetwork()
best_genome.BuildESHyperNEATPhenotype(net, substrate,
                                      params)
visualize.draw_net(net, view=False, node_names=None,
                   filename="substrate_graph.svg",
                   directory=trial_out_dir, fmt='svg')
print("\nSubstrate nodes: %d, connections: %d" %
         (len(net.neurons),
       len(net.connections)))
inputs = net.NumInputs()
outputs = net.NumOutputs()
hidden = len(net.neurons) - net.NumInputs() - \
         net.NumOutputs()
print("\n\tinputs: %d, outputs: %d, hidden: %d" %
         (inputs, outputs, hidden))
```

10. Кроме того, мы выводим на печать результаты оценки нейросети детектора, созданного предыдущим кодом, по двум случайно выбранным визуальным объектам и полному набору визуальных данных:

```
# Тест на двух случайно выбранных визуальных объектах.
l_index = random.randint(0, 15)
r_index = random.randint(0, 15)
left = rt_environment.visual_objects[l_index]
right = rt_environment.visual_objects[r_index]
err, outputs = rt_environment._evaluate(net, left,
                                        right, 3)
print("Test evaluation error: %f" % err)
print("Left flag: %f, pattern: %s" % (outputs[0], left))
print("Right flag: %f, pattern: %s" % (outputs[1], right))

# Тест на полном наборе визуальных объектов.
fitness, avg_error, total_count, false_detections = \
             rt_environment.evaluate_net(net, debug=True)
print("Test evaluation against full data set [%d], fitness:
%f, average error: %f, false detections: %f" % (total_count,
fitness, avg_error, false_detections))
```

Наконец, мы отображаем статистические данные, собранные в ходе эксперимента:

```
# Визуализация статистики
visualize.plot_stats(stats, ylog=False, view=show_results,
                     filename=os.path.join(trial_out_dir,
                     'avg_fitness.svg'))
```

Все упомянутые здесь графики визуализации могут быть найдены после выполнения эксперимента в каталоге `trial_out_dir` локальной файловой системы. Далее вы узнаете, как реализована функция построителя субстрата.

Функция построителя субстрата

Метод ES-HyperNEAT запускает процесс нейроэволюции, включающий в себя эволюцию геномов CPPN наряду с эволюцией конфигурации субстрата. Однако, хотя субстрат способен успешно эволюционировать, очень выгодно начинать с подходящей *начальной конфигурации субстрата*. Эта конфигурация должна соответствовать геометрии проблемного пространства.

Для эксперимента с сетчаткой соответствующая конфигурация субстрата создается следующим образом.

1. Сначала создадим конфигурацию входного слоя субстрата. Как вы знаете из раздела 8.5.1, восемь узлов входного слоя расположены в плоскости XZ, которая ортогональна плоскости XY. Кроме того, чтобы отразить геометрию пространства сетчатки, узлы левого объекта должны быть соответственно размещены на левой стороне плоскости, а узлы правого объекта – на правой стороне плоскости. Узел смещения должен быть расположен в центре плоскости входных узлов. В коде эксперимента входной слой создается следующим образом:

```
# Входной слой.
x_space = np.linspace(-1.0, 1.0, num=4)
inputs = [
    # Левая сторона.
    (x_space[0], 0.0, 1.0), (x_space[1], 0.0, 1.0),
    (x_space[0], 0.0, -1.0), (x_space[1], 0.0, -1.0),
    # the right side
    (x_space[2], 0.0, 1.0), (x_space[3], 0.0, 1.0),
    (x_space[2], 0.0, -1.0), (x_space[3], 0.0, -1.0),
    (0,0,0) # the bias
    ]
```

Два выходных узла расположены в плоскости XY, которая ортогональна плоскости ввода. Эта конфигурация допускает естественную эволюцию субстрата путем размещения найденных скрытых узлов в плоскости XY.

2. Выходной слой создается следующим образом:

```
# Выходной слой
outputs = [(-1.0, 1.0, 0.0), (1.0, 1.0, 0.0)]
```

3. Далее мы определяем общие параметры конфигурации субстрата:

```
# Разрешаем следующие связи: входной -> скрытый, скрытый -> выходной
# и скрытый -> скрытый.
substrate.m_allow_input_hidden_links = True
substrate.m_allow_hidden_output_links = True
substrate.m_allow_hidden_hidden_links = True

substrate.m_allow_input_output_links = False
substrate.m_allow_output_hidden_links = False
substrate.m_allow_output_output_links = False
substrate.m_allow_looped_hidden_links = False
substrate.m_allow_looped_output_links = False

substrate.m_hidden_nodes_activation = \
       NEAT.ActivationFunction.SIGNED_SIGMOID
substrate.m_output_nodes_activation = \
       NEAT.ActivationFunction.UNSIGNED_SIGMOID

# Отправляем длину связи в CPPN как параметр
substrate.m_with_distance = True
substrate.m_max_weight_and_bias = 8.0
```

Мы разрешаем субстрату иметь связи между входными и скрытыми узлами, между скрытыми узлами и от скрытых к выходным узлам. Мы указываем, что скрытые узлы должны использовать функцию активации «знаковая сигмоида», в то время как выходные узлы должны использовать функцию активации «беззнаковая сигмоида». Мы выбираем для выходных узлов беззнаковую сигмоиду, чтобы выходные значения нейросети детектора находились в интервале [0,1].

В следующем разделе мы обсудим реализацию функций для оценки приспособленности решений.

Оценка фитнеса

Процесс нейроэволюции требует средств для оценки приспособленности популяции генома в каждом поколении эволюции. Оценка приспособленности в нашем эксперименте состоит из двух частей, которые мы обсуждаем здесь.

Функция eval_genomes

Эта функция оценивает приспособленность всей популяции. У нее есть следующее определение:

```
def eval_genomes(genomes, substrate, rt_environment, params):
    best_genome = None
    max_fitness = 0
    errors = []
    for genome in genomes:
        fitness, error, total_count, false_detetctions = eval_individual(
                                genome, substrate, rt_environment, params)
        genome.SetFitness(fitness)
        errors.append(error)
```

```
        if fitness > max_fitness:
            max_fitness = fitness
            best_genome = genome
    return best_genome, max_fitness, errors
```

Функция `eval_genomes` в качестве параметров принимает список геномов CPPN из текущей популяции, конфигурацию субстрата, инициализированную тестовую среду и гиперпараметры ES-HyperNEAT.

В начале кода мы создаем промежуточный объект для сбора результатов оценки каждого конкретного генома:

```
best_genome = None
max_fitness = 0
errors = []
```

После этого мы запускаем цикл, который перебирает все геномы и оценивает каждый геном в соответствии с условиями текущей тестовой среды:

```
for genome in genomes:
    fitness, error, total_count, false_detetctions = eval_individual(
                          genome, substrate, rt_environment, params)
    genome.SetFitness(fitness)
    errors.append(error)

    if fitness > max_fitness:
        max_fitness = fitness
        best_genome = genome
```

Наконец, функция возвращает результаты оценки в виде кортежа, который включает лучший геном, наивысшую оценку приспособленности и список всех ошибок классификации для каждого оцениваемого генома.

Функция eval_individual

Эта функция оценивает приспособленность каждого отдельного генома и имеет следующее определение:

```
def eval_individual(genome, substrate, rt_environment, params):
    # Создаем нейросеть на основе генома CPPN и субстрата
    net = NEAT.NeuralNetwork()
    genome.BuildESHyperNEATPhenotype(net, substrate, params)

    fitness, dist, total_count, false_detetctions = \
       rt_environment.evaluate_net(net, max_fitness=MAX_FITNESS)
    return fitness, dist, total_count, false_detetctions
```

Она получает в качестве параметров оцениваемый геном CPPN, конфигурацию субстрата, среду тестирования и гиперпараметры ES-HyperNEAT. Используя предоставленные параметры, мы создаем конфигурацию нейронной сети детектора и оцениваем ее в текущей тестовой среде. Затем функция возвращает результат оценки.

8.6 Эксперимент с модульной сетчаткой

Теперь мы готовы начать эксперименты в тестовой среде, которая имитирует проблемное пространство модульной сетчатки. В следующих разделах вы узнаете, как выбрать подходящие гиперпараметры, настроить среду и запустить эксперимент. После этого мы обсудим результаты эксперимента.

8.6.1 Настройка гиперпараметров

Гиперпараметры определены как класс Python, и библиотека MultiNEAT ссылается на него для получения необходимых параметров конфигурации. В исходном коде эксперимента мы определяем специализированную функцию `create_hyperparameters`, которая инкапсулирует логику инициализации гиперпараметров. Далее мы опишем наиболее важные гиперпараметры и причины выбора конкретных значений.

1. Мы используем популяцию генома CPPN среднего размера. Это сделано для ускорения эволюции за счет доступа к изначально большому пространству вариантов для поиска решения. Численность популяции определяется следующим образом:

```
params.PopulationSize = 300
```

2. Затем мы определяем количество видов, которые будут сохраняться в ходе эволюции в интервале `[5,15]`, и устанавливаем продолжительность стагнации видов до `100` поколений. Эта конфигурация позволяет нам иметь здоровое разнообразие видов и поддерживать их в течение достаточно долгого времени, чтобы успеть найти решение:

```
params.SpeciesMaxStagnation = 100
params.MinSpecies = 5
params.MaxSpecies = 15
```

3. Мы заинтересованы в создании сверхкомпактной конфигурации геномов CPPN, поэтому задаем очень маленькие значения вероятности появления новых узлов и связей в геномах:

```
params.MutateAddLinkProb = 0.03
params.MutateAddNeuronProb = 0.03
```

4. Метод ES-HyperNEAT является расширением метода HyperNEAT, который в ходе эволюции может изменять тип функций активации в скрытом и выходном узлах. В этом эксперименте мы задаем равную вероятность выбора следующих типов активации:

```
params.ActivationFunction_SignedGauss_Prob = 1.0
params.ActivationFunction_SignedStep_Prob = 1.0
params.ActivationFunction_Linear_Prob = 1.0
params.ActivationFunction_SignedSine_Prob = 1.0
params.ActivationFunction_SignedSigmoid_Prob = 1.0
```

5. Наконец, мы определяем специфические гиперпараметры ES-HyperNEAT, которые управляют развитием субстрата. Следующие гиперпараметры управляют динамикой создания узлов и связей внутри субстрата в процессе эволюции:

```
params.DivisionThreshold = 0.5
params.VarianceThreshold = 0.03
```

Параметр `params.DivisionThreshold` определяет, сколько новых узлов и соединений вводится в субстрат при каждом поколении эволюции. Параметр `params.VarianceThreshold` определяет, сколько узлов и соединений может оставаться в субстрате после фазы обрезки и извлечения. Вы можете вернуться к разделу 8.3 за более подробной информацией об этих пороговых значениях.

8.6.2 Настройка рабочей среды

В этом эксперименте мы используем библиотеку MultiNEAT, которая обеспечивает реализацию алгоритма ES-HyperNEAT. Нам нужно создать соответствующую среду Python, которая включает в себя библиотеку MultiNEAT и все необходимые зависимости. Это можно сделать с помощью Anaconda, выполнив следующие команды в командной строке:

```
$ conda create --name rt_multineat python=3.5
$ conda activate vd_multineat
$ conda install -c conda-forge multineat
$ conda install matplotlib
$ conda install -c anaconda seaborn
$ conda install graphviz
$ conda install python-graphviz
```

Эти команды создают и активируют виртуальную среду `rt_multineat` на основе Python 3.5. Затем они устанавливают последнюю версию библиотеки MultiNEAT вместе с зависимостями, которые нужны вашему коду для визуализации результатов.

8.6.3 Запуск эксперимента с модульной сетчаткой

На этом этапе у вас уже есть сценарий запуска эксперимента, полностью определенный в файле `retina_experiment.py`. Вы можете начать эксперимент, клонировав соответствующий репозиторий Git и запустив скрипт с помощью следующих команд:

```
$ git clone
https://github.com/PacktPublishing/Hands-on-Neuroevolution-with-Python.git
$ cd Hands-on-Neuroevolution-with-Python/Chapter8
$ python retina_experiment.py -t 1 -g 1000
```

Не забудьте активировать соответствующее виртуальное окружение при помощи команды `conda activate rt_multineat`.

Предыдущая команда запускает один прогон эксперимента для 1000 поколений эволюции. Спустя некоторое количество поколений должно быть найдено успешное решение, которое сопровождается приблизительно таким выводом в консоль:

```
****** Generation: 949 ******

Solution found at generation: 949, best fitness: 1000.000000, species
count: 6

Best ever fitness: 1000.000000, genome ID: 284698

Trial elapsed time: 1332.576 sec
Random seed: 1569777981

CPPN nodes: 21, connections: 22

Substrate nodes: 15, connections: 28
```

Как следует из этого примера вывода, в данном случае успешное решение было найдено в поколении 949. Оно было создано геномом CPPN, включающим 21 узел и 22 связи между ними. В то же время субстрат, определяющий топологию нейросети детектора, имеет 15 узлов и 28 связей между ними. Успешное решение было получено с использованием случайного начального значения 1569777981. Использование других случайных начальных значений может не привести к успешному решению или потребует еще многих поколений эволюции.

Далее интересно взглянуть на график средней приспособленности и погрешности за время эволюции (рис. 8.5).

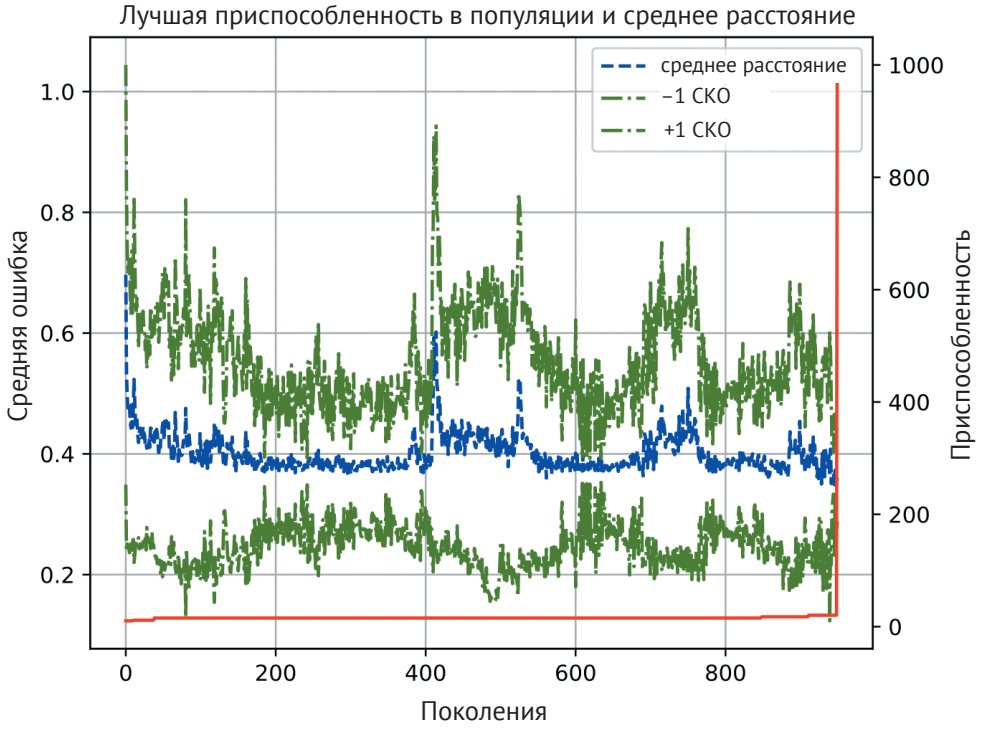


Рис. 8.5. Средняя приспособленность и ошибка по поколениям

Можно сделать вывод, что в течение большинства поколений эволюции оценка приспособленности была очень мала (около 20), но неожиданно нашелся успешный геном CPPN, который совершил резкий эволюционный скачок всего за одно поколение.

Конфигурация успешного генома CPPN показана на рис. 8.6.

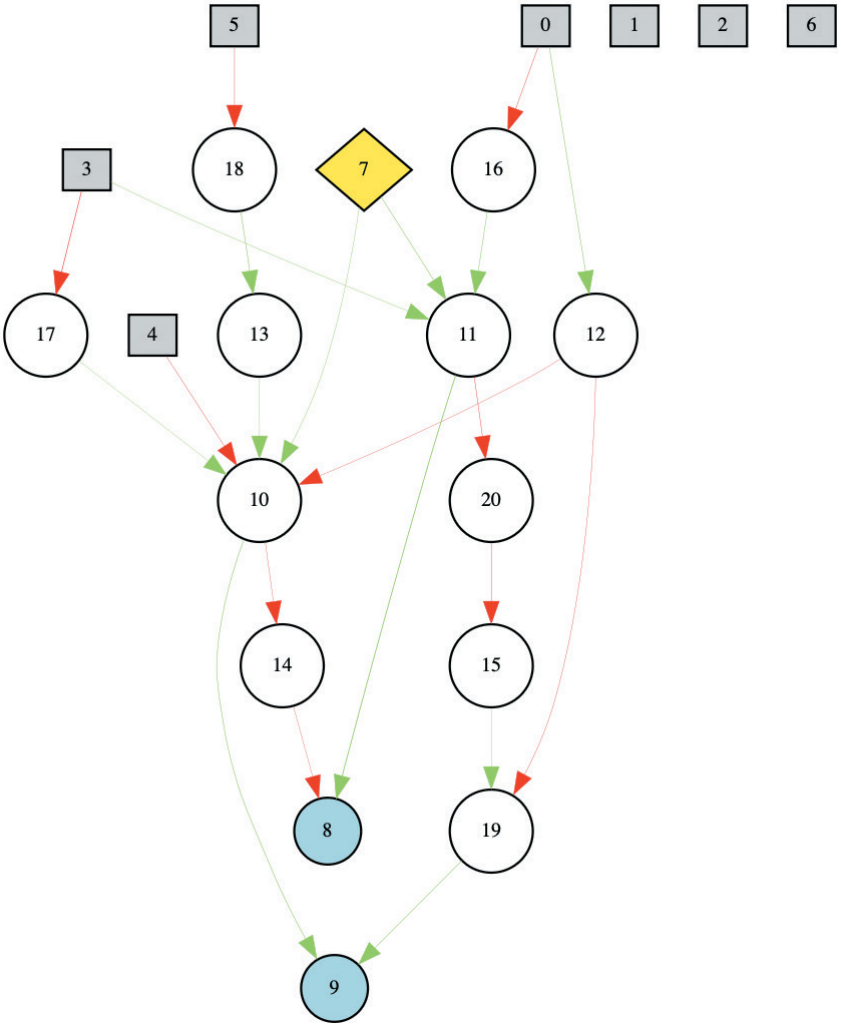


Рис. 8.6. Граф фенотипа CPPN успешного генома

Схема на рис. 8.6 чрезвычайно любопытна, потому что, как вы можете видеть, успешный геном CPPN использует не все доступные входные данные (серые квадраты) для предсказания выходных данных. Более того, еще более удивительным является использование только координаты *x* входного (0) и координаты *y* скрытого (3) узлов субстрата при принятии решения об активации связи между этими узлами субстрата. В то же время обе координаты *x* и *y* выходных узлов субстрата участвуют в процессе принятия решений (4 и 5).

Если вы посмотрите на исходную конфигурацию субстрата, которую мы представили ранее (рис. 8.4), то увидите, что упомянутые нами особенности полностью основаны на топологии субстрата. Мы разместили входные узлы в плоскости XZ. Таким образом, координата *y* для них не имеет значения. В то же время скрытые узлы расположены внутри плоскости XY, причем координата *y* определяет расстояние от плоскости входов. Наконец, выходные узлы также расположены в плоскости XY. Их координата *x* определяет сторону сетчатки, к которой относится каждый выходной узел. Следовательно, для выходных узлов естественно учитывать обе координаты *x* и *y*.

На графе фенотипа CPPN входные узлы помечены квадратами, выходные узлы – закрашенные кружки, узел смещения – ромб, а скрытые узлы – пустые кружки.

Два выходных узла на диаграмме CPPN имеют следующее назначение:

- первый узел (8) обеспечивает вес связи;
- второй узел (9) определяет, активирована связь или нет.

Входные узлы CPPN определены следующим образом:

- первые два узла (0 и 1) задают координаты точки (*x*, *z*) во входном слое субстрата;
- следующие два узла (2 и 3) задают координаты точки (*x*, *y*) в скрытом слое субстрата;
- следующие два узла (4 и 5) задают координаты точки (*x*, *y*) в выходном слое субстрата;
- последний узел (6) устанавливает евклидово расстояние до точки во входном слое от начала координат.

Тем не менее наиболее захватывающая часть результатов эксперимента показана на следующей схеме (рис. 8.7). Это конфигурация успешной нейросети зрительного детектора.

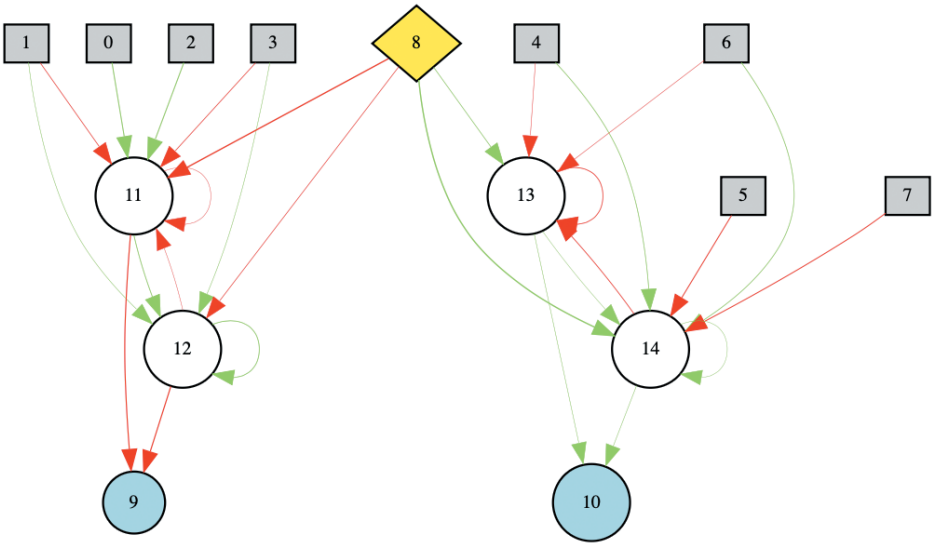


Рис. 8.7. Конфигурация успешной нейросети зрительного детектора

Как и на предыдущей схеме, мы помечаем входные узлы квадратами, выходные узлы – закрашенными кружками, узел смещения – ромбом, а скрытые узлы – пустыми кружками.

Как видите, у нас есть две четко разделенные модульные структуры с левой и правой сторон графика. Каждый модуль подключен к соответствующим входам с левой (узлы 0, 1, 2 и 3) и правой (узлы 4, 5, 6 и 7) сторон сетчатки. Оба модуля имеют одинаковое количество скрытых узлов, которые связаны с соответствующими выходными узлами: узел 9 для левой стороны и узел 10 для правой стороны сетчатки. Также вы можете видеть, что паттерны связей в левом и правом модулях похожи. Скрытый узел 11 слева имеет связность, аналогичную узлу 14 справа, и то же самое можно сказать об узлах 12 и 13.

Просто удивительно, как стохастический эволюционный процесс смог обнаружить такое простое и элегантное решение. Результаты этого эксперимента полностью подтвердили нашу гипотезу о том, что задача сетчатки может быть решена путем поиска модульных топологий нейросети зрительного детектора.

Более подробную информацию о задаче модульной сетчатки можно найти в оригинальной статье по адресу **http://eplex.cs.ucf.edu/papers/risi_ alife12.pdf**.

8.7 Упражнения

1. Попробуйте запустить эксперимент с различными значениями генератора случайных чисел, которые можно изменить в строке 101 скрипта `retina_experiment.py`. Посмотрите, получится ли найти успешные решения с другими значениями.
2. Попытайтесь увеличить начальный размер популяции до 1000, изменив значение параметра `params.PopulationSize`. Как это повлияло на производительность алгоритма?
3. Попробуйте изменить количество типов функций активации, использованных в ходе эволюции, устанавливая вероятность выбора определенной функции равной 0. Особенно интересно посмотреть, что произойдет, если вы исключите из выбора типы активации `ActivationFunction_SignedGauss_Prob` и `ActivationFunction_SignedStep_Prob`.

8.8 Заключение

В этой главе вы узнали о методе нейроэволюции, который позволяет конфигурации субстрата развиваться в процессе поиска решения задачи. Такой подход освобождает разработчика от необходимости проработки подходящей конфигурации субстрата до мельчайших деталей и позволяет обойтись только основными очертаниями. Алгоритм автоматически найдет оставшиеся подробности конфигурации субстрата в процессе эволюции.

Кроме того, вы узнали о модульных структурах нейросети, которые можно использовать для решения различных задач, включая задачу модульной сетчатки. Модульные топологии нейросетей – это очень мощная концепция,

которая позволяет многократно использовать успешный модуль фенотипа нейросети для построения сложной иерархической топологии. Кроме того, у вас была возможность отточить свои навыки работы с языком программирования Python, реализовав решение задачи с использованием библиотеки MultiNEAT.

В следующей главе мы обсудим увлекательную концепцию коэволюции и то, как ее можно использовать для одновременного развития решателя и целевой функции, которая используется для оптимизации. Мы обсудим метод эволюции решения и приспособленности, и вы узнаете, как применить его в модифицированном эксперименте по прохождению лабиринта.

Глава 9

Коэволюция и метод SAFE

В этой главе мы представим концепцию коэволюции и объясняем, как ее можно использовать для совместной эволюции решателя и целевой функции, которая оптимизирует эволюцию решателя. Затем обсудим метод *коэволюции решателя и приспособленности* (solution and fitness evolution, SAFE) и дадим краткий обзор различных стратегий коэволюции. Вы узнаете, как объединить коэволюцию с нейроэволюционными методами, а также получите практический опыт реализации модифицированного эксперимента по прохождению лабиринта.

В данной главе мы рассмотрим следующие темы:

- принцип коэволюции и общие стратегии коэволюции;
- основы метода SAFE;
- модифицированный эксперимент по прохождению лабиринта;
- обсуждение результатов эксперимента.

9.1 Технические требования

Для проведения экспериментов, описанных в данной главе, ваш компьютер должен удовлетворять следующим техническим требованиям:

- Windows 8/10, macOS 10.13 или новее, или современный Linux;
- Anaconda Distribution версия 2019.03 или новее.

Исходный код примеров этой главы можно найти в каталоге `Chapter9` в файловом архиве книги.

9.2 Общие стратегии коэволюции

Естественная эволюция биологических систем не может рассматриваться отдельно от концепции коэволюции. *Коэволюция* является одной из основных движущих сил эволюции, от которой зависит текущее состояние биосферы и разнообразие организмов.

Мы можем определить коэволюцию как взаимовыгодную стратегию одновременной эволюции множества генеалогий разных организмов. Эволюция одного вида невозможна без других видов. В ходе эволюции совместно эволюционирующие виды взаимодействуют друг с другом, и эти межвидовые

отношения формируют их эволюционную стратегию. Существует три основных типа коэволюции:

- *мутуализм*, когда два или более вида мирно сосуществуют и взаимно выигрывают друг от друга;
- *конкурентная коэволюция*:
  - *хищничество*, когда один организм убивает другой и потребляет его ресурсы;
  - *паразитизм*, когда один организм использует ресурсы другого, но не убивает его;
- *комменсализм*, когда представители одного вида получают выгоды, не причиняя вреда и не принося выгоды другим видам.

Исследователи изучили каждый тип коэволюции и нашли у них не только достоинства, но и недостатки при использовании в качестве основного принципа нейроэволюции. Однако недавно группа исследователей достигла многообещающих результатов при использовании комменсализма. Они создали алгоритм SAFE, который мы обсудим в этой главе.

Подробнее об алгоритме SAFE можно прочитать в статье **https://doi.org/10.1007/978-3-030-16670-0_10**.

Теперь, когда вы познакомились с общими типами коэволюции, давайте перейдем к изучению метода SAFE.

9.3 Метод SAFE

Как следует из названия, метод коэволюции решателя и приспособленности основан на совместной эволюции решения и функции приспособленности, которая направляет оптимизацию поиска решения. Метод SAFE построен вокруг стратегии *комменсалистической коэволюции* двух популяций:

- популяция потенциальных решений, которые развиваются, чтобы решить непосредственно поставленную задачу;
- популяция кандидатов на целевые функции, которые эволюционируют, чтобы направлять эволюцию популяции решений.

В этой книге мы уже обсудили несколько стратегий оптимизации, которые можно использовать для управления развитием потенциальных кандидатов на решение. Эти стратегии основаны на объективной оптимизации приспособленности и оптимизации поиском новизны. Первая стратегия оптимизации идеальна в ситуациях, когда у нас простой ландшафт функции приспособленности и мы можем сосредоточить оптимизацию на близости к конечной цели. В этом случае мы можем использовать объективную метрику, которая оценивает в каждом поколении эволюции, насколько близко наше текущее решение к месту назначения.

Стратегия оптимизации поиском новизны устроена иначе. В этой стратегии нас не интересует близость к конечной цели, но вместо этого нас больше всего интересует путь, который выбирают потенциальные решения. Основная идея метода поиска новизны – постепенно исследовать все новые сту-

пеньки, которые в конечном итоге ведут к месту назначения. Эта стратегия оптимизации идеальна для ситуаций, когда у нас сложный ландшафт функции приспособленности со многими обманчивыми тупиками и локальными оптимумами.

Основная идея метода SAFE состоит в том, чтобы извлечь выгоду из обоих методов оптимизации поиска, упомянутых здесь. Далее мы обсудим модифицированный эксперимент с лабиринтом, который для руководства процессом нейроэволюции использует оба метода поисковой оптимизации.

9.4 Модифицированный эксперимент с лабиринтом

В этой книге мы уже обсуждали, как применять методы поисковой оптимизации на основе близости к цели или поиска новизны к задаче решения лабиринта. В данной главе мы представляем модифицированный эксперимент по прохождению лабиринта, в котором пытаемся объединить оба метода оптимизации поиска, используя алгоритм SAFE.

Мы представляем коэволюцию двух популяций: агентов-решателей лабиринта и кандидатов на целевые функции. Следуя методу SAFE, мы используем в нашем эксперименте комменсалистическую стратегию коэволюции. Давайте сначала обсудим устройство агента, который будет проходить лабиринт.

9.4.1 Агент-решатель задачи лабиринта

Агент-решатель задачи лабиринта оснащен набором датчиков, позволяющих ему воспринимать среду лабиринта и знать направление к выходу из лабиринта на каждом этапе. Расположение датчиков показано на рис. 9.1.

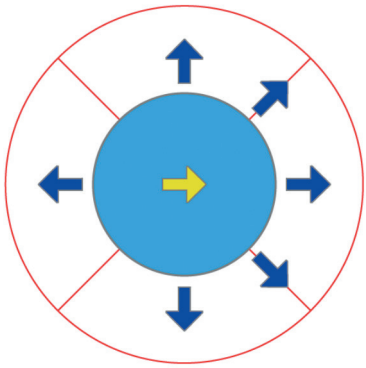


Рис. 9.1. Расположение датчиков на агенте-решателе

На этом рисунке темные стрелки обозначают дальномерные датчики, позволяющие агенту обнаруживать препятствия и находить расстояние до препятствия в заданном направлении. Четыре сектора, нарисованных вокруг тела робота, представляют собой радиолокаторы с секторным обзором, которые определяют направление к выходу из лабиринта на каждом шаге симуляции. Светлая стрелка внутри тела робота определяет направление, в котором он движется.

Также у робота есть два привода: один для изменения его угловой скорости (вращение) и другой для изменения его линейной скорости (передвижение).

Мы используем ту же конфигурацию робота, что и в главе 5. Рекомендую вернуться к этой главе и перечитать раздел 5.3.1. Далее мы рассмотрим среду лабиринта.

9.4.2 Среда лабиринта

Лабиринт определяется как пространство, окруженное снаружи сплошными стенами. Внутри лабиринта несколько внутренних перегородок создают множество тупиков с локальными оптимумами, что делает оптимизацию по близости к цели не очень эффективной. Кроме того, сосредоточенные на близости к цели агенты могут застревать в тупиках, полностью останавливая процесс эволюции. Тупики локальных оптимумов показаны на рис. 9.2.

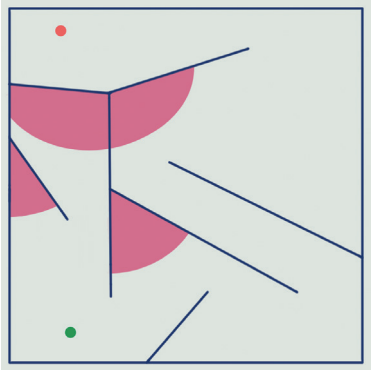


Рис. 9.2. Расположение локальных оптимумов внутри лабиринта

На рис. 9.2 начальная позиция агента-решателя отмечена закрашенным кружком в нижнем левом углу, а выход из лабиринта отмечен закрашенным кружком в верхнем левом углу. Обманчивые области локальных оптимумов показаны в виде закрашенных секторов относительно начальной позиции агента.

Среда лабиринта определена в файле конфигурации, а для имитации прохождения агента через лабиринт применяется симулятор. Мы обсуждали реализацию симулятора лабиринта в разделе 5.3.2, и сейчас вы можете перечитать этот раздел, чтобы вспомнить подробности.

В данной главе мы обсудим модификации, которые были внесены в исходный эксперимент для реализации стратегии SAFE. Самое важное различие заключается в том, как определяется функция приспособленности, и мы обсудим это в следующем разделе.

Вы найдете полный исходный код симулятора лабиринта в файле `maze_environment.py`.

9.4.3 Определение функции приспособленности

Метод SAFE – это коэволюция кандидатов в успешные решатели и кандидатов в целевые функции, то есть мы имеем две популяции эволюционирующих видов. Таким образом, нам нужно определить две функции приспособленности: одну для кандидатов на решение (решатели лабиринтов) и другую для кандидатов на целевые функции. В этом разделе мы обсудим оба варианта.

Функция приспособленности для решателя задачи лабиринта

В каждом поколении эволюции каждый индивидуум (решатель лабиринта) оценивается всеми кандидатами на целевые функции. Мы используем максимальную оценку приспособленности, полученную во время оценивания решателя каждым кандидатом на целевую функцию, в качестве оценки приспособленности решения.

Функция приспособленности решателя лабиринта представляет собой совокупность двух метрик – расстояния от выхода из лабиринта (оценка близости к цели) и новизны конечной позиции решателя (оценка новизны). Эти оценки арифметически объединяются с использованием пары коэффициентов, полученных в качестве выходных данных от конкретного индивидуума в популяции кандидатов на целевую функцию.

Следующая формула дает комбинацию этих показателей в качестве оценки приспособленности:

$$\mathcal{O}_i(\mathcal{S}_i) = a \times \frac{1}{D_i} + b \times NS_i.$$

Здесь $\mathcal{O}_i(\mathcal{S}_i)$ – это значения приспособленности, полученные путем оценки кандидата на решение $\mathcal{S}_i$, относительно целевой функции $\mathcal{O}_i$. Пара коэффициентов $[a, b]$ – это выход конкретного кандидата на целевую функцию. Эта пара определяет, в какой мере расстояние до выхода из лабиринта (D_i) и поведенческая новизна (NS_i) решения влияют на конечную оценку приспособленности решателя лабиринта в конце траектории.

Расстояние до выхода из лабиринта (D_i) определяется как евклидово расстояние между последними координатами решателя лабиринта в его траектории и координатами выхода из лабиринта. Расстояние вычисляется по следующей формуле:

$$D_i = \sqrt{(x_s - x_m)^2 + (y_s - y_m)^2},$$

x_s и y_s – последние координаты решателя, а x_m и y_m – координаты выхода из лабиринта.

Оценка новизны NS_i каждого решателя лабиринта определяется его окончательным положением в лабиринте (точка x). Она рассчитывается как среднее расстояние от этой точки до k-ближайших соседних точек, которые являются окончательными позициями других решателей лабиринтов.

Следующая формула дает значение оценки новизны в точке x поведенческого пространства:

$$NS_i = \frac{1}{k}\sum_{i=0}^{k} dist(x, \mu_i),$$

где μ_i – это i-й ближайший сосед x, а $dist(x, \mu_i)$ – расстояние между x и μ_i.

Расстояние между двумя точками является метрикой новизны, измеряющей, насколько текущее решение (x) отличается от другого (μ_i), созданного различными решателями лабиринтов. Показатель новизны рассчитывается как евклидово расстояние между двумя точками:

$$dist(x, \mu) = \sqrt{\sum_{j=1}^{2}\left(x_j - \mu_j\right)^2}.$$

Здесь μ_j и x_j – значения в позиции j координатных векторов, содержащих координаты точек μ и x соответственно.

Далее мы обсудим, как определить функцию приспособленности для оптимизации кандидатов на роль целевой функции.

Функция приспособленности для кандидатов на целевую функцию

Метод SAFE основан на комменсалистическом коэволюционном подходе, который означает, что одна из совместно эволюционирующих популяций не получает ни пользы, ни вреда в ходе эволюции. В нашем эксперименте комменсалистическая популяция представляет собой совокупность кандидатов на целевые функции. Для этой совокупности нам нужно определить такую функцию приспособленности, которая не зависит от качества работы решателей лабиринтов.

Подходящим вариантом является функция приспособленности, которая в качестве оценки приспособленности использует показатель новизны. Формула для расчета оценки новизны каждого кандидата на роль целевой функции такая же, как и для решателей лабиринтов. Единственное отличие состоит в том, что в случае кандидатов на роль целевой функции мы рассчитываем оценку новизны, используя векторы с выходными значениями каждого индивидуума. После этого применяем значение показателя новизны в качестве показателя приспособленности особи.

Этот метод оценки новизны является частью модифицированного метода поиска новизны, который мы обсудим в следующем разделе.

9.5 Модифицированный поиск новизны

Вы познакомились с методом поиска новизны в главе 6. В текущем эксперименте мы используем слегка модифицированный вариант метода, который обсудим далее.

Модификации метода поиска новизны, которые мы представим в этом эксперименте, относятся к новому способу сохранения архива баллов новизны (новинок). Балл новизны отражает местоположение агента-решателя в конце траектории, которое комбинируется с оценкой новизны.

В более традиционной версии метода размер архива новинок является динамическим, что позволяет добавлять новый элемент, если его оценка но-

визны превышает определенный порог (порог новизны). Кроме того, порог новизны может быть скорректирован во время выполнения с учетом того, как быстро обнаруживаются новые точки новизны в ходе эволюции. Эти настройки позволяют нам в некоторой степени контролировать максимальный размер архива. Однако нам нужно начать с начального порогового значения новизны, и этот выбор не очевиден.

Модифицированный метод поиска новизны подразумевает архив фиксированного размера, чтобы решить вопрос выбора правильного порогового значения новизны. Новые элементы добавляются в архив до тех пор, пока он не заполнится. После этого новый элемент добавляется в архив путем вытеснения текущего элемента с минимальной оценкой только в том случае, если его оценка новизны превышает текущий минимальный балл архива. Таким образом, мы можем поддерживать фиксированный размер архива новинок и хранить в нем только самые ценные новинки, обнаруженные в ходе эволюции.

Исходный код модифицированного поиска новизны представлен в файле `novelty_archive.py` в файловом архиве книги.

Далее давайте обсудим наиболее интересные части реализации.

9.5.1 Функция _add_novelty_item

Эта функция позволяет добавлять новые элементы в архив при сохранении его размера. Она имеет следующую реализацию:

```
if len(self.novel_items) >= MAXNoveltyArchiveSize:
    # Проверяем, превышает ли новизна этой точки check
    # минимальное значение в архиве (минимальную новизну).
    if item > self.novel_items[-1]:
        # Заменяем значение
        self.novel_items[-1] = item
else:
    # Просто добавляем новое значение.
    self.novel_items.append(item)

# Сортируем элементы массива по убыванию значений
self.novel_items.sort(reverse=True)
```

Код сначала проверяет, не был ли еще превышен размер архива новинок, и в этом случае напрямую добавляет к нему новый элемент. В противном случае новый элемент вытесняет последний элемент в архиве, то есть элемент с наименьшим баллом новизны. Мы можем быть уверены, что последний элемент в архиве имеет наименьшую оценку новизны, потому что после добавления нового элемента в архив мы сортируем элементы в порядке убывания баллов.

9.5.2 Функция evaluate_novelty_score

Эта функция предоставляет механизм для вычисления баллов новизны по всем элементам, уже собранным в архиве новинок, и всем новинкам, обнару-

женным в текущей популяции. Мы рассчитываем балл новизны как среднее расстояние до $k = 15$ ближайших соседей, выполнив следующие шаги.

1. Собираем расстояния от рассматриваемой точки до всех элементов, уже хранящихся в архиве новинок:

```
distances = []
for n in self.novel_items:
    if n.genomeId != item.genomeId:
        distances.append(self.novelty_metric(n, item))
    else:
        print("Novelty Item is already in archive: %d" %
              n.genomeId)
```

2. После этого складываем расстояния от рассматриваемой точки до всех особей в текущей популяции:

```
for p_item in n_items_list:
    if p_item.genomeId != item.genomeId:
        distances.append(self.novelty_metric(p_item, item))
```

3. Наконец, вычисляем среднее значение k-ближайших соседей:

```
distances = sorted(distances)
item.novelty = sum(distances[:KNN])/KNN
```

Мы сортируем список по расстояниям в порядке возрастания, чтобы гарантировать, что ближайшие элементы находятся первыми в списке. После этого мы вычисляем сумму первых $k = 15$ элементов в списке и делим ее на количество суммированных значений. Таким образом, мы получаем значение среднего расстояния до k-ближайших соседей.

Модифицированный метод оптимизации поиском новизны лежит в основе оценки приспособленности как для популяции решателей лабиринта, так и для популяции кандидатов на роль целевой функции. Мы широко используем его в реализации эксперимента, о которой поговорим в следующем разделе.

9.6 Движок модифицированного эксперимента с лабиринтом

Реализация движка эксперимента основана на библиотеке MultiNEAT, которую мы использовали в нескольких экспериментах в этой книге. Развитие каждой совместно эволюционирующей популяции происходит в соответствии с базовым алгоритмом NEAT, который обсуждался в главах 3–5.

Тем не менее в этом разделе мы покажем, как использовать алгоритм NEAT для поддержания коэволюции двух независимых популяций видов: решателей лабиринтов и кандидатов в целевые функции.

Далее мы обсудим основные части модифицированного движка эксперимента.

Для детального изучения исходного кода обратитесь к файлу `maze_experiment_safe.py` из файлового архива книги.

9.6.1 Создание совместно эволюционирующих популяций

В этом эксперименте нам необходимо создать две совместно развивающиеся популяции видов с различными исходными конфигурациями генотипа, чтобы удовлетворить фенотипические требования продуцируемых видов.

Фенотип решателя лабиринтов имеет 11 входных узлов для приема сигналов от датчиков и два выходных узла для выдачи управляющих сигналов. В то же время фенотип кандидата на целевую функцию имеет один входной узел, принимающий фиксированное значение (0,5), которое преобразуется в два выходных значения, которые используются в качестве коэффициентов функции приспособленности решателя лабиринта.

Начнем с обсуждения того, как создать популяцию кандидатов на целевую функцию.

Создание популяции кандидатов на целевую функцию

Генотип, кодирующий фенотип кандидатов на целевую функцию, должен создавать конфигурации фенотипа, которые имеют по крайней мере один входной узел и два выходных узла, как было сказано ранее. Создание популяции в функции `create_objective_fun` реализовано следующим образом:

```
params = create_objective_fun_params()
# Геном имеет один вход (0.5) и два выхода (a и b).
genome = NEAT.Genome(0, 1, 1, 2, False,
    NEAT.ActivationFunction.TANH, # hidden layer activation
    NEAT.ActivationFunction.UNSIGNED_SIGMOID, # output layer activation
    1, params, 0)
pop = NEAT.Population(genome, params, True, 1.0, seed)
pop.RNG.Seed(seed)

obj_archive = archive.NoveltyArchive(
                          metric=maze.maze_novelty_metric_euclidean)
obj_fun = ObjectiveFun(archive=obj_archive,
                         genome=genome, population=pop)
```

В этом коде мы создаем генотип NEAT с одним входным узлом, двумя выходными узлами и одним скрытым узлом. Скрытый узел предварительно внесен в исходный геном, чтобы ускорить эволюцию с заранее определенной нелинейностью. В качестве функции активации скрытого слоя с поддержкой отрицательных выходных значений выбран гиперболический тангенс. Эта особенность важна для нашей задачи. Отрицательное значение одного из коэффициентов, созданных кандидатом на целевую функцию, может указывать на то, что конкретный компонент функции приспособленности решателя лабиринта оказывает отрицательное влияние, и таким образом посылает сигнал, что эволюция должна пробовать другие пути.

В конце мы создаем объект `ObjectiveFun` для хранения растущей популяции кандидатов на целевую функцию.

Далее обсудим, как создается популяция решателей лабиринтов.

Создание популяции решателей лабиринтов

Агент-решатель лабиринта должен получать входные данные от 11 датчиков и генерировать два управляющих сигнала, которые влияют на угловую и линейную скорости робота. Следовательно, геном, кодирующий фенотип решателя лабиринта, должен определять конфигурации, которые включают 11 входных узлов и два выходных узла. Чтобы узнать, как создается начальная популяция геномов для решателя, изучите код функции `create_robot`:

```
params = create_robot_params()
# Геном имеет 11 входов и два выхода
genome = NEAT.Genome(0, 11, 0, 2, False,
                     NEAT.ActivationFunction.UNSIGNED_SIGMOID,
                     NEAT.ActivationFunction.UNSIGNED_SIGMOID,
                     0, params, 0)
pop = NEAT.Population(genome, params, True, 1.0, seed)
pop.RNG.Seed(seed)

robot_archive = archive.NoveltyArchive(metric=maze.maze_novelty_metric)
robot = Robot(maze_env=maze_env, archive=robot_archive, genome=genome,
              population=pop)
```

В коде мы получаем соответствующие гиперпараметры NEAT из функции `create_robot_params`. После этого используем их для создания исходного NEAT-генотипа с соответствующим количеством входных и выходных узлов. Наконец, мы создаем объект `Robot`, который инкапсулирует все данные, относящиеся к популяции решателей лабиринтов, а также среду симулятора лабиринта.

Теперь, когда мы создали две совместно эволюционирующие популяции, нам необходимо реализовать оценку приспособленности для особей из обеих популяций. В следующем разделе обсудим реализацию оценки приспособленности.

9.6.2 Оценка приспособленности совместно развивающихся популяций

Определив две совместно развивающиеся популяции, нам необходимо создать функции для оценки показателей приспособленности особей в каждой популяции. Как мы уже упоминали, показатели приспособленности особей в популяции решателей лабиринта зависят от выходных значений, производимых популяцией кандидатов на целевую функцию. В то же время оценка приспособленности каждого кандидата на целевую функцию полностью определяется оценкой новизны этой особи.

Таким образом, у нас есть два разных подхода к оценке показателей приспособленности, и нам нужно реализовать две разные функции. Далее мы обсудим обе реализации.

Оценка приспособленности кандидатов на целевую функцию

Оценка приспособленности каждого индивидуума в популяции кандидатов на целевые функции определяется его баллом новизны, который рассчитывается, как мы обсуждали ранее. Реализация оценки приспособленности состоит из двух функций – `evaluate_obj_functions` и `evaluate_individ_obj_function`.

Давайте рассмотрим реализации обеих функций.

Реализация функции evaluate_obj_functions

Эта функция принимает объект `ObjectiveFun`, который содержит совокупность кандидатов на целевые функции, и использует его для оценки приспособленности каждого индивидуума в популяции, выполнив следующие действия.

1. Сначала мы перебираем все геномы в популяции и собираем баллы новизны для каждого генома:

```
obj_func_genomes = NEAT.GetGenomeList(obj_function.population)
for genome in obj_func_genomes:
    n_item = evaluate_individ_obj_function(genome=genome,
                                           generation=generation)
    n_items_list.append(n_item)
    obj_func_coeffs.append(n_item.data)
```

В коде баллы новизны, полученные с помощью функции `evaluate_individ_obj_function`, добавляются к списку баллов новизны в популяции. Также мы добавляем данные балла новизны в список пар коэффициентов. Позже список пар коэффициентов будет использоваться для вычисления показателей приспособленности индивидуальных решателей лабиринтов.

2. Затем мы перебираем список геномов популяции и получаем оценку новизны каждого генома, используя баллы новизны, собранные на предыдущем шаге:

```
max_fitness = 0
for i, genome in enumerate(obj_func_genomes):
    fitness = obj_function.archive.evaluate_novelty_score(
           item=n_items_list[i],n_items_list=n_items_list)
    genome.SetFitness(fitness)
    max_fitness = max(max_fitness, fitness)
```

Оценка новизны, вычисленная с использованием баллов новизны, уже включена в архив новинок и в список новинок, созданный для текущей популяции. После этого мы устанавливаем найденный показатель новизны как показатель приспособленности соответствующего генома. Кроме того, находим максимальное значение показателя приспособленности и возвращаем его вместе со списком пар коэффициентов.

Реализация функции evaluate_individ_obj_function

Эта функция принимает NEAT-геном кандидата в целевую функцию и возвращает результаты расчета балла новизны:

```
n_item = archive.NoveltyItem(generation=generation, genomeId=genome_id)
# Запуск симуляции
multi_net = NEAT.NeuralNetwork()
genome.BuildPhenotype(multi_net)
depth = 2
try:
    genome.CalculateDepth()
    depth = genome.GetDepth()
except:
    pass
```

```
obj_net = ANN(multi_net, depth=depth)

# Задаем вход и получаем выход ([a, b])
output = obj_net.activate([0.5])

# Сохраняем коэффициенты
n_item.data.append(output[0])
n_item.data.append(output[1])
```

Мы начнем с создания объекта `NoveltyItem` для хранения данных балла новизны для данного генома. После этого строим фенотип нейросети и активируем его значением `0.5`. Наконец, используем выходные данные нейросети, чтобы создать очередную точку новизны.

В следующем разделе мы обсудим оценку приспособленности особей в популяции решателей лабиринтов.

Оценка приспособленности агентов, решающих лабиринт

Мы находим оценку приспособленности каждой особи в популяции лабиринтов как сочетание двух компонентов: оценки новизны и расстояния до выхода из лабиринта в конце траектории. Влияние каждого компонента определяется парой коэффициентов, возвращаемой особями из популяции кандидатов на целевые функции.

Код для расчета приспособленности состоит из трех функций, которые мы обсудим далее.

Реализация функции evaluate_solutions

Функция `evaluate_solutions` в качестве входного параметра получает объект `Robot`, который инкапсулирует популяцию агентов-решателей и симулятор среды лабиринта. Кроме того, она получает список пар коэффициентов, сгенерированных во время оценки популяции кандидатов на целевые функции.

Мы используем входные параметры функции для оценки каждого генома в популяции и вычисления его приспособленности. Функция состоит из следующих основных частей.

1. Сначала мы запускаем симуляцию прохождения лабиринта каждой особью в популяции и находим расстояние до выхода из лабиринта в конце траектории:

```
robot_genomes = NEAT.GetGenomeList(robot.population)
for genome in robot_genomes:
    found, distance, n_item = evaluate_individual_solution(
        genome=genome, generation=generation, robot=robot)
    # Сохраняем полученное значение.
    distances.append(distance)
    n_items_list.append(n_item)
```

2. Затем перебираем все геномы в популяции и вычисляем оценку новизны каждой особи. Кроме того, мы берем полученное ранее соответствующее расстояние до выхода из лабиринта и объединяем его с рассчитанной оценкой новизны, чтобы получить окончательную оценку приспособленности генома:

```
for i, n_item in enumerate(n_items_list):
    novelty = robot.archive.evaluate_novelty_score(item=n_item,
                                    n_items_list=n_items_list)
    # Проверка правильности.
    assert robot_genomes[i].GetID() == n_item.genomeId

    # Вычисление приспособленности.
    fitness, coeffs = evaluate_solution_fitness(distances[i],
                                   novelty, obj_func_coeffs)
    robot_genomes[i].SetFitness(fitness)
```

В первой половине кода мы используем функцию `robot.archive.evaluate_novelty_score`, чтобы получить оценку новизны каждой особи в популяции. Вторая половина вызывает функцию `evaluate_solution_fitness`, чтобы получить оценку приспособленности каждой особи, используя оценку новизны и расстояние до выхода из лабиринта.

3. Наконец, мы собираем статистику об эффективности лучшего генома решателя в популяции:

```
if not solution_found:
    # Находим лучший геном в популяции
    if max_fitness < fitness:
        max_fitness = fitness
        best_robot_genome = robot_genomes[i]
        best_coeffs = coeffs
        best_distance = distances[i]
        best_novelty = novelty
elif best_robot_genome.GetID() == n_item.genomeId:
    # Сохраняем приспособленность решения-победителя
    max_fitness = fitness
    best_coeffs = coeffs
    best_distance = distances[i]
    best_novelty = novelty
```

Завершая работу, функция возвращает всю статистику, собранную во время оценки популяции.

Далее мы обсудим, как с помощью симулятора среды лабиринта оценивается отдельный геном решателя.

Реализация функции `evaluate_individual_solution`

Данная функция оценивает производительность отдельного генома решателя в симуляторе среды лабиринта. Это происходит следующим образом.

1. Сначала мы создаем фенотип нейросети решателя и используем эту нейросеть в качестве контроллера для управления движением робота через лабиринт:

```
n_item = archive.NoveltyItem(generation=generation,
                             genomeId=genome_id)
# Запуск симуляции
maze_env = copy.deepcopy(robot.orig_maze_environment)
multi_net = NEAT.NeuralNetwork()
```

```
genome.BuildPhenotype(multi_net)
depth = 8
try:
    genome.CalculateDepth()
    depth = genome.GetDepth()
except:
    pass
control_net = ANN(multi_net, depth=depth)
distance = maze.maze_simulation_evaluate(
    env=maze_env, net=control_net,
    time_steps=SOLVER_TIME_STEPS, n_item=n_item)
```

В этой части кода мы создаем объект `NoveltyItem` для хранения *точки новизны*, которая определяется конечной позицией робота в лабиринте. После этого создаем фенотип нейросети и запускаем симулятор лабиринта, используя нейросеть в качестве контроллера робота на заданном количестве временных шагов (400). После завершения симуляции получаем расстояние между конечной позицией решателя и выходом из лабиринта.

2. Затем мы сохраняем статистику моделирования в объекте `AgentRecord`, который проанализируем в конце эксперимента:

```
record = agent.AgenRecord(generation=generation,
                          agent_id=genome_id)
record.distance = distance
record.x = maze_env.agent.location.x
record.y = maze_env.agent.location.y
record.hit_exit = maze_env.exit_found
record.species_id = robot.get_species_id(genome)
robot.record_store.add_record(record)
```

Функция возвращает кортеж со следующими значениями: флаг, указывающий, найдено ли решение, расстояние до выхода из лабиринта в конце траектории робота и объект `NoveltyItem`, инкапсулирующий информацию об обнаруженной точке новизны.

В следующем разделе мы обсудим реализацию функции приспособленности решателя лабиринта.

Реализация функции evaluate_solution_fitness

Это реализация функции приспособленности решателя лабиринта, которую мы обсуждали ранее. Функция получает расстояние до выхода из лабиринта, оценку новизны и список пар коэффициентов, сгенерированных текущим поколением кандидатов на целевую функцию. Затем она использует полученные входные параметры для расчета оценки приспособленности:

```
normalized_novelty = novelty
if novelty >= 1.00:
    normalized_novelty = math.log(novelty)
norm_distance = math.log(distance)

max_fitness = 0
best_coeffs = [-1, -1]
```

```
for coeff in obj_func_coeffs:
    fitness = coeff[0] / norm_distance + coeff[1] * normalized_novelty
    if fitness > max_fitness:
        max_fitness = fitness
        best_coeffs[0] = coeff[0]
        best_coeffs[1] = coeff[1]
```

Сначала мы должны нормализовать значения расстояния и оценки новизны, используя натуральный логарифм. Нормализация приводит значения расстояния и оценки новизны к одному масштабу. Это важно, потому что пара коэффициентов всегда находится в интервале [0,1]. Следовательно, если значения расстояния и оценки новизны имеют разный масштаб, пара коэффициентов не сможет правильно влиять на значимость каждого значения при расчете оценки приспособленности.

Код перебирает список пар коэффициентов и для каждой пары коэффициентов вычисляет оценку приспособленности, комбинируя значения расстояния и оценки новизны.

В качестве окончательного показателя приспособленности решателя берется максимальное значение среди всех найденных показателей приспособленности. Затем функция возвращает это значение и соответствующую пару коэффициентов.

9.6.3 Выполнение эксперимента модифицированного лабиринта

Теперь, когда мы подготовили все необходимые процедуры для создания совместно развивающихся популяций и оценки приспособленности особей в этих популяциях, мы готовы приступить к реализации цикла эксперимента.

Полный исходный код функции `run_experiment` можно найти в файле `maze_experiment_safe.py` в файловом архиве книги.

Полный рабочий цикл эксперимента состоит из следующих шагов.

1. Начинаем с создания популяций совместно эволюционирующих видов:

```
robot = create_robot(maze_env, seed=seed)
obj_func = create_objective_fun(seed)
```

2. Запускаем цикл эволюции и оцениваем обе популяции:

```
for generation in range(n_generations):
    # Оцениваем популяцию кандидатов на функцию приспособленности.
    obj_func_coeffs, max_obj_func_fitness = \
                evaluate_obj_functions(obj_func, generation)
    # Оцениваем популяцию решателей.
    robot_genome, solution_found, robot_fitness, distances, \
    obj_coeffs, best_distance, best_novelty = \
      evaluate_solutions(robot=robot,
      obj_func_coeffs=obj_func_coeffs, generation=generation)
```

3. Получив оценки популяций, сохраняем результаты в виде статистики текущего поколения эволюции:

```
stats.post_evaluate(max_fitness=robot_fitness,
                    errors=distances)
# Сохраняем лучший геном
best_fitness = robot.population.GetBestFitnessEver()
if solution_found or best_fitness < robot_fitness:
    best_robot_genome_ser = pickle.dumps(robot_genome)
    best_robot_id = robot_genome.GetID()
    best_obj_func_coeffs = obj_coeffs
    best_solution_novelty = best_novelty
```

4. Если в текущем поколении решение не нашлось, то в конце цикла эволюции мы просим обе популяции перейти к следующему поколению:

```
if solution_found:
    print('Solution found at generation: %d, best fitness:
          %f, species count: %d' % (generation, robot_fitness,
          len(pop.Species)))
    break
# Переход к следующему поколению.
robot.population.Epoch()
obj_func.population.Epoch()
```

5. После того как цикл эволюции совершил итерации в течение заданного числа поколений, мы визуализируем накопленные записи о прохождении лабиринта:

```
if args is None:
    visualize.draw_maze_records(maze_env,
                robot.record_store.records,
                view=show_results)
else:
    visualize.draw_maze_records(maze_env,
              robot.record_store.records,
              view=show_results, width=args.width,
              height=args.height,
              filename=os.path.join(trial_out_dir,
                            'maze_records.svg'))
```

Упомянутые здесь записи прохождения лабиринта содержат статистику оценок каждого генома решателя в симуляторе лабиринта, собранную во время эволюции в виде объектов `AgentRecord`. В визуализации мы отображаем конечную позицию каждого рассмотренного решателя в лабиринте.

6. Далее мы моделируем прохождение лабиринта с помощью управляющей нейросети, которая была создана с использованием лучшего генома, найденного в ходе эволюции. Траектория движения по лабиринту во время моделирования может быть визуализирована следующим образом:

```
multi_net = NEAT.NeuralNetwork()
best_robot_genome.BuildPhenotype(multi_net)
control_net = ANN(multi_net, depth=depth)
```

```
path_points = []
distance = maze.maze_simulation_evaluate(
                            env=maze_env,
                            net=control_net,
                            time_steps=SOLVER_TIME_STEPS,
                            path_points=path_points)
print("Best solution distance to maze exit: %.2f, novelty:
       %.2f" % (distance, best_solution_novelty))
visualize.draw_agent_path(robot.orig_maze_environment,
                  path_points, best_robot_genome,
                  view=show_results, width=args.width,
                  height=args.height,
                  filename=os.path.join(trial_out_dir,
                             'best_solver_path.svg'))
```

Сначала код создает фенотип нейросети из лучшего решающего генома. Затем он запускает симулятор лабиринта, используя созданный фенотип в качестве контроллера робота. Потом создается рисунок, отображающий точки траектории робота.

7. Наконец, мы отображаем график со средними показателями приспособленности для каждого поколения:

```
visualize.plot_stats(stats, ylog=False, view=show_results,
   filename=os.path.join(trial_out_dir,'avg_fitness.svg'))
```

Все упомянутые здесь визуализации также сохраняются в локальной файловой системе в виде файлов SVG и могут быть использованы позже для анализа результатов.

В следующем разделе мы обсудим выполнение модифицированного эксперимента с лабиринтом и результаты эксперимента.

9.7 Эксперимент с модифицированным лабиринтом

Мы почти готовы запустить процесс коэволюции в модифицированном эксперименте с лабиринтом. Однако сначала мы должны настроить гиперпараметры для каждой эволюционирующей популяции.

9.7.1 Гиперпараметры для популяции агентов-решателей

Для этого эксперимента мы решили использовать библиотеку MultiNEAT, которая использует класс Python `Parameters` для ведения списка всех поддерживаемых гиперпараметров. Инициализация гиперпараметров для популяции агентов-решателей происходит в функции `create_robot_params`. Далее мы обсудим основные гиперпараметры и причины выбора конкретных значений.

1. Мы решили с самого начала иметь популяцию среднего размера, обеспечивающую достаточно обширное разнообразие особей:

```
params.PopulationSize = 250
```

2. Мы заинтересованы в создании компактной топологии генома во время эволюции и ограничении числа видов в популяции. Поэтому определяем маленькие вероятности добавления новых узлов и соединений в ходе эволюции:

```
params.MutateAddNeuronProb = 0.03
params.MutateAddLinkProb = 0.05
```

3. Оценка новизны служит вознаграждением за нахождение уникальных позиций в лабиринте. Одним из способов заработать вознаграждение является усиление численной динамики в рамках фенотипа. Поэтому мы увеличили интервал значений весов связей:

```
params.MaxWeight = 30.0
params.MinWeight = -30.0
```

4. Чтобы поддержать эволюционный процесс, мы решили ввести элитарность, определив долю геномов, которые будут переданы следующему поколению:

```
params.Elitism = 0.1
```

Значение элитарности определяет, что примерно одна десятая часть популяции будет отнесена к следующему поколению.

9.7.2 Гиперпараметры популяции кандидатов на целевую функцию

Инициализация гиперпараметров для эволюции популяции кандидатов на целевую функцию происходит в функции `create_objective_fun_params`. Наиболее важными являются следующие гиперпараметры.

1. Мы решили начать с небольшой популяции, чтобы уменьшить вычислительные затраты. Кроме того, не ожидается, что генотипы кандидатов на целевую функцию будут очень сложными. Следовательно, небольшой популяции должно быть достаточно:

```
params.PopulationSize = 100
```

2. Как и в случае с решателями, мы заинтересованы в создании компактных геномов. Поэтому задаем очень маленькую вероятность добавления новых узлов и соединений:

```
params.MutateAddNeuronProb = 0.03
params.MutateAddLinkProb = 0.05
```

Мы не ожидаем получить сложную топологию геномов в популяции кандидатов на целевую функцию. Поэтому для большинства гиперпараметров оставлены значения по умолчанию.

9.7.3 Настройка рабочей среды

В этом эксперименте мы используем библиотеку MultiNEAT. Перед запуском эксперимента нам нужно создать соответствующую среду Python, которая включает эту библиотеку и другие зависимости. Вы можете настроить среду Python с помощью Anaconda, выполнив следующие команды:

```
$ conda create --name maze_co python=3.5
$ conda activate maze_co
$ conda install -c conda-forge multineat
$ conda install matplotlib
```

```
$ conda install graphviz
$ conda install python-graphviz
```

Эти команды создают виртуальную среду `maze_co` с Python 3.5 и устанавливают в нее все необходимые зависимости.

9.7.4 Проведение модифицированного эксперимента с лабиринтом

Теперь мы готовы запустить эксперимент в созданной виртуальной среде. Вы можете начать эксперимент, клонировав соответствующий репозиторий Git и запустив скрипт с помощью следующих команд:

```
$ git clone
https://github.com/PacktPublishing/Hands-on-Neuroevolution-with-Python.git
$ cd Hands-on-Neuroevolution-with-Python/Chapter9
$ python maze_experiment_safe.py -t 1 -g 150 -m medium
```

Не забудьте активировать рабочую среду при помощи команды `conda activate maze_co`.

Предыдущая команда запускает одно испытание эксперимента для 150 поколений эволюции, используя конфигурацию лабиринта средней сложности. Примерно через 100 поколений эволюции, в процессе нейроэволюции, будет найдено успешное решение, и вы сможете увидеть следующий вывод в консоли:

```
****** Generation: 105 ******

Maze solved in 338 steps

Solution found at generation: 105, best fitness: 3.549289, species count: 7

================================
Record store file: out/maze_medium_safe/5/data.pickle
Random seed: 1571021768
Best solution fitness: 3.901621, genome ID: 26458
Best objective func coefficients: [0.7935419704765059, 0.9882050653334634]
----------------------------
Maze solved in 338 steps
Best solution distance to maze exit: 3.56, novelty: 19.29
------------------------
Trial elapsed time: 4275.705 sec
================================
```

В данном примере успешный решатель лабиринтов был найден в поколении 105 и смог пройти лабиринт за 338 шагов из отведенных 400. Кроме того, интересно отметить, что пара коэффициентов, созданная лучшим кандидатом на целевую функцию, придает немного большее значение компоненту оценки новизны в функции приспособленности решателя.

Интересно взглянуть на график лучших показателей приспособленности по поколениям (рис. 9.3).

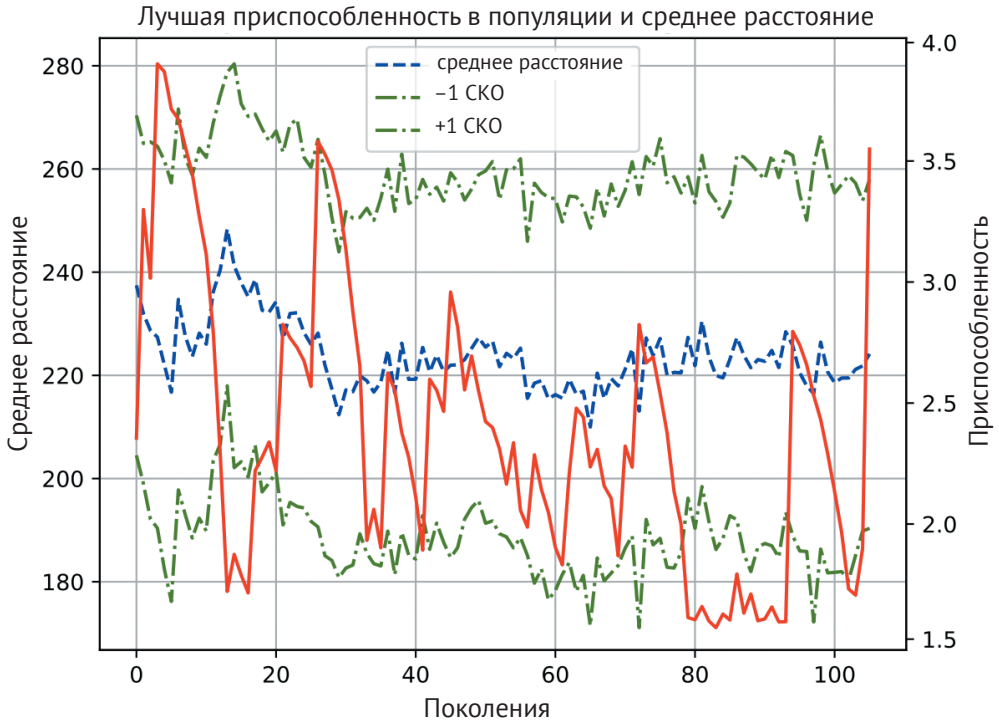


Рис. 9.3. Показатели лучшей приспособленности по поколениям

На этом графике хорошо видно, что лучший показатель приспособленности имеет максимум в ранних поколениях эволюции. Это связано с высокими значениями оценки новизны, которые легче получить в начале эволюции, потому что есть много областей лабиринта, которые не были исследованы. Еще один важный момент, на который следует обратить внимание, – это то, что среднее расстояние до выхода из лабиринта остается почти на одном уровне для большинства поколений эволюции. Отсюда мы можем предположить, что правильное решение было найдено не постепенными улучшениями, а скорее за счет качественного скачка генома победителя. Этот вывод также подтверждается следующей визуализацией, на которой показаны записи прохождения лабиринта по видам (рис. 9.4).

Визуализация состоит из двух частей: верхняя часть для видов с целеориентированной оценкой приспособленности (на основе расстояния от выхода из лабиринта) более 0,8 и нижняя часть для других видов. Вы можете видеть, что только один вид произвел геном потомка, который смог достичь окрестности выхода из лабиринта. Кроме того, вы можете видеть, что геномы, принадлежащие этому виду, демонстрируют выраженное исследовательское поведение, исследуя больше областей лабиринта, чем все другие виды вместе взятые.

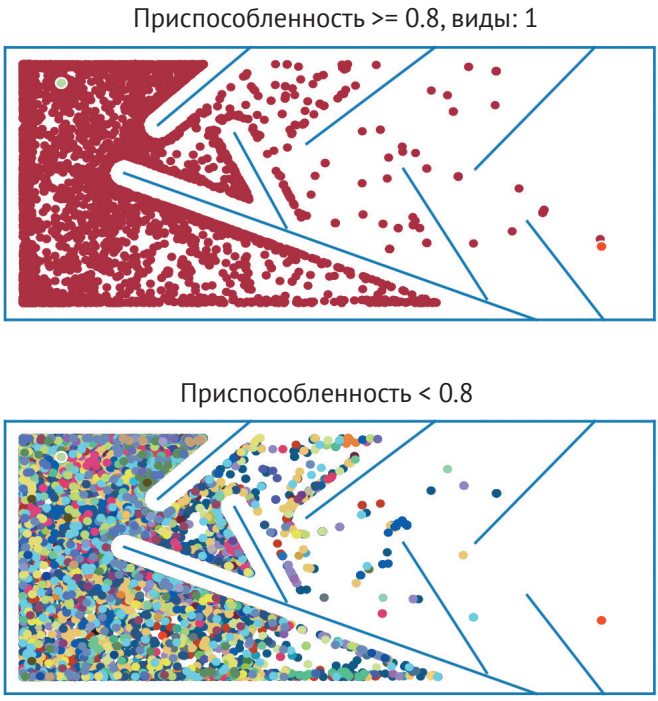


Рис. 9.4. Записи прохождения лабиринтов решателями с указанием финальных позиций

Наконец, мы можем рассмотреть визуализацию пути успешного решателя через лабиринт (рис. 9.5).

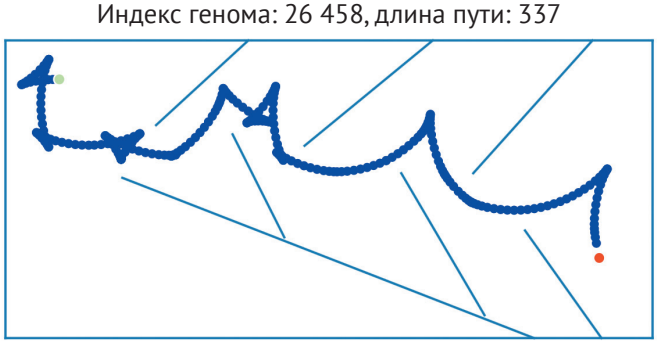


Рис. 9.5. Путь успешного решателя через лабиринт

Траектория успешного решателя является почти оптимальной для данной конфигурации лабиринта.

Этот эксперимент также демонстрирует важность начальных условий в поиске успешного решения. Начальные условия определяются начальным значением генератора случайных чисел, которое мы выбираем перед запуском эксперимента.

9.8 Упражнения

1. Мы включили сложную конфигурацию лабиринта в исходный код эксперимента в файле `hard_maze.txt` в файловом архиве книги. Вы можете попытаться пройти сложный лабиринт с помощью следующей команды: `python maze_experiment_safe.py -g 120 -t 5 -m hard --width 200 -- height 200`.
2. Мы нашли успешное решение, используя `1571021768` в качестве начального значения генератора случайных чисел. Попробуйте найти другое случайное начальное значение, дающее успешное решение. Через сколько поколений удалось найти успешное решение в вашем случае?

9.9 Заключение

В этой главе мы обсудили коэволюцию двух популяций. Вы узнали, как может быть реализована комменсалистическая коэволюция, создающая популяцию успешных решателей задачи лабиринта. Мы познакомили вас с захватывающим подходом к разработке функции приспособленности агента-решателя, которая комбинирует оценку близости к цели и оценку новизны при помощи коэффициентов, возвращаемых популяцией кандидатов на целевую функцию. Кроме того, вы узнали о модифицированном методе поиска новизны и о том, как он отличается от оригинального метода, о котором мы говорили в главе 6.

Используя знания, полученные в этой главе, вы сможете применять комменсалистический коэволюционный подход к своей работе или исследовательским задачам, которые не имеют четкого определения функции приспособленности.

В следующей главе вы узнаете о методе глубокой нейроэволюции и о том, как использовать его для развития агентов, способных играть в классические игры Atari.

Глава 10

Глубокая нейроэволюция

В этой главе вы узнаете о методе глубокой нейроэволюции, который можно использовать для обучения *глубоких нейронных сетей* (deep neural networks, DNN). Глубокие нейросети обычно обучаются с использованием методов *обратного распространения*, основанных на снижении градиента ошибки, который вычисляется относительно весов связей между нейронными узлами. Хотя градиентное обучение является мощным методом, который положил начало нынешней эпохе глубокого машинного обучения, у него есть свои недостатки, такие как длительное время обучения и огромные требования к вычислительной мощности.

В этой главе мы покажем, как методы глубокой нейроэволюции можно использовать для обучения с подкреплением и как они значительно превосходят традиционные методы обучения нейросетей, основанные на градиентах. К концу этой главы у вас сформируется ясное понимание методов глубокой нейроэволюции, и вы получите практический опыт их применения. Вы узнаете, как научить нейросеть играть в классические игры Atari, используя метод глубокой нейроэволюции. Также узнаете, как использовать *визуальный контроль нейроэволюции* (visual inspector for neuroevolution, VINE) для проверки результатов экспериментов.

В этой главе мы рассмотрим следующие темы:

- глубокая нейроэволюция для глубокого обучения с подкреплением;
- обучение агента игре Atari Frostbite с использованием глубокой нейроэволюции;
- эксперимент с игрой Atari Frostbite;
- запуск эксперимента Atari Frostbite;
- проверка результатов при помощи VINE.

10.1 Технические требования

Для проведения экспериментов, описанных в этой главе, ваш компьютер должен удовлетворять следующим техническим требованиям:

- современный ПК с графическим ускорителем Nvidia GeForce GTX 1080Ti или лучше;
- MS Windows 10, Ubuntu Linux 16.04 или macOS 10.14 с дискретным графическим процессором;
- Anaconda Distribution версии 2019.03 или новее.

Исходный код примеров этой главы можно найти в каталоге `Chapter10` в файловом архиве книги.

10.2 Глубокая нейроэволюция для глубокого обучения с подкреплением

В этой книге уже говорилось о том, как метод нейроэволюции можно применять для решения простых задач обучения с подкреплением, таких как балансировка одинарного и двойного обратного маятника на тележке в главе 4. Однако хотя эксперимент по балансировке маятника выглядит захватывающе, он слишком прост и работает с крошечными искусственными нейронными сетями. В этой главе мы обсудим, как применить нейроэволюцию к задачам *обучения с подкреплением* (reinforcement learning, RL), которые требуют огромных нейросетей для аппроксимации *функции истинности*[1] (value function) алгоритма RL.

Алгоритм RL учится методом проб и ошибок. Почти все варианты алгоритмов RL пытаются оптимизировать функцию истинности, отображающую текущее состояние системы на соответствующее действие, которое будет выполнено в следующем шаге времени. Наиболее широко используемая классическая версия алгоритма RL использует метод *Q-обучения*. В основе этого метода лежит *матрица состояний* (*Q-матрица*), связанная с правилами поведения, которым должен следовать алгоритм после завершения обучения. Обучение состоит в обновлении ячеек Q-матрицы путем итеративного выполнения некоторых действий в определенных состояниях и последующего получения сигналов вознаграждения. Следующая формула описывает процесс обновления конкретной ячейки в Q-матрице:

$$Q^{new}(s_t, a_t) = (1-\alpha)Q(s_t, a_t) + \alpha\left(r_t + \gamma \max_a Q(s_{t+1}, a)\right).$$

Здесь r_t – вознаграждение, получаемое при переходе системы от состояния s_t к состоянию s_{t+1}, a_t – действие, предпринимаемое во время t, ведущее к изменению состояния, α – скорость обучения и γ – фактор дисконтирования, от которого зависит весомость будущих вознаграждений. Скорость обучения определяет, в какой степени новая информация переопределяет существующую информацию в конкретной ячейке Q-матрицы. Если мы установим нулевую скорость обучения, то не получим новые знания, а если установим ее на 1, то не сохраним ранее полученные знания. Таким образом, скорость обучения определяет, насколько быстро система может получать новые знания, сохраняя при этом полезные, уже проверенные данные.

Простая версия алгоритма Q-обучения перебирает все возможные комбинации действий и состояний и обновляет ячейки Q-матрицы, как мы уже обсуждали. Этот подход довольно хорошо работает для простых задач с небольшим количеством пар действие–состояние, но быстро терпит неудачу с увеличением числа таких пар, то есть с увеличением размерности простран-

[1] Иногда переводится как функция ценности или общее подкрепление (на всем сроке жизни агента). – *Прим. перев.*

ства действия–состояния. Большинство реальных задач имеют большую размерность пространства действия–состояния, непосильную для классической версии Q-обучения.

Для решения проблемы высокой размерности был предложен метод аппроксимации Q-функцией. В этом методе правила поведения определяются не таблицей состояний, которую мы упоминали ранее, а аппроксимируются функцией. Одним из способов достижения этой аппроксимации является использование нейросети в качестве универсального аппроксиматора. Благодаря использованию нейросетей, особенно глубоких, для аппроксимации Q-матрицы удается использовать алгоритмы RL для очень сложных задач, даже на непрерывном пространстве состояний. С этой целью был разработан метод *глубоких Q-сетей* (deep Q-network, DQN), который использует глубокие нейросети для аппроксимации Q-значений. Алгоритм RL, основанный на аппроксимации функции истинности, получил название *глубокого обучения с подкреплением* (deep RL, глубокое RL).

С глубоким RL можно извлечь правила поведения непосредственно из пикселей видеопотока. Это позволяет нам использовать видеопоток, например, для обучения агентов видеоиграм. Тем не менее метод DQN можно считать разновидностью метода градиентного обучения. Он использует обратное распространение ошибки для обучения аппроксиматора функции Q-значений. Будучи мощным методом, он имеет высокую вычислительную сложность и нуждается в использовании графических процессоров для умножения матриц во время вычислений, связанных с градиентным спуском.

Одним из методов, которые можно использовать для уменьшения вычислительных затрат, являются *генетические алгоритмы* (genetic algorithms, GA), такие как нейроэволюция. Нейроэволюция позволяет нам обучать нейросеть для аппроксимации Q-матрицы без каких-либо градиентных вычислений. В недавних исследованиях было показано, что методы генетических алгоритмов без градиента показывают отличную производительность, когда речь идет о сложных задачах с глубоким RL, и что они могут даже превзойти традиционные аналоги. В следующем разделе вы узнаете, как метод глубокой нейроэволюции можно использовать для обучения успешных агентов прохождению одной из классических игр Atari, просто считывая пиксельное изображение игрового поля.

10.3 Обучение агента игре Atari Frostbite с использованием глубокой нейроэволюции

Недавно классические игры Atari были инкапсулированы в так называемую *обучающую среду Atari* (Atari learning environment, ALE), которая служит эталоном для тестирования различных реализаций алгоритмов RL. Алгоритмы, которые подвергаются проверке на ALE, должны уметь считывать состояние игры в виде пикселей игрового экрана и развивать сложную логику управления, позволяющую агенту выигрывать игру. Таким образом, задача алгоритма состоит в том, чтобы развить понимание игровой ситуации с точки зрения игрового персонажа и его противников. Кроме того, алгоритм должен правильно понимать сигнал вознаграждения, поступающий с игрового экрана в виде окончательного счета в конце одиночного прогона игры.

10.3.1 Игра Atari Frostbite

Frostbite[1] – это классическая игра Atari, в которой вы управляете игровым персонажем, возводящим и́глу – домик из ледяных блоков. Экран игры показан на рис. 10.1.

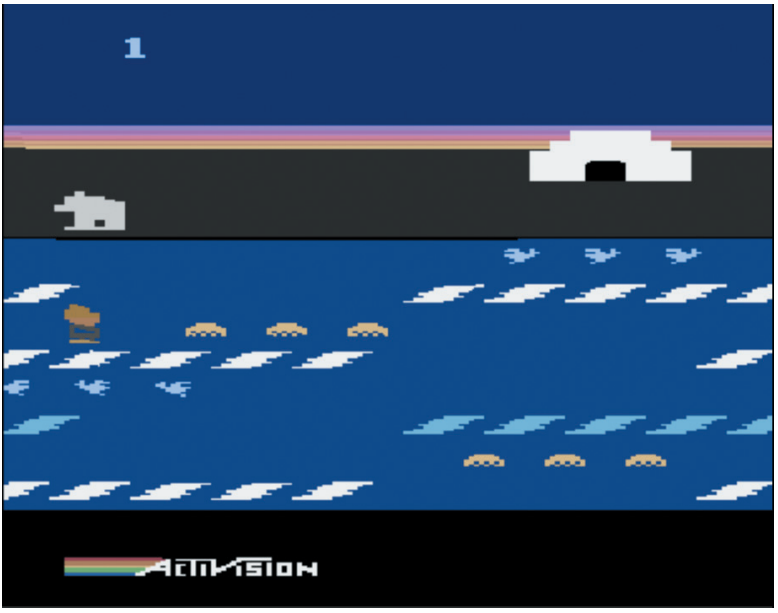


Рис. 10.1. Экран игры Atari Frostbite

Нижняя часть экрана – вода с плавающими в ней ледяными блоками, расположенными в четыре ряда. Персонаж прыгает из одного ряда в другой, пытаясь избежать различных противников. Если игровой персонаж попадает на белый ледяной блок, этот блок достается персонажу и используется для строительства и́глу на берегу в правом верхнем углу экрана. После этого белый ледяной блок меняет свой цвет и больше не может использоваться.

Чтобы построить иглу, персонаж должен собрать 15 ледяных блоков в течение 45 секунд. В противном случае игра заканчивается, потому что персонаж погибает от переохлаждения. Когда иглу построено, персонаж должен оказаться внутри него, чтобы завершить текущий уровень. Чем быстрее персонаж завершает уровень, тем больше бонусных очков начисляется игроку.

Далее мы обсудим, как состояние экрана игры отображается на входные данные, задействованные в нейроэволюции.

10.3.2 Отображение игрового экрана на действия

Чтобы научиться играть в игры Atari, глубокая нейросеть должна иметь возможность напрямую отображать пиксели на экране в систему управления игрой. Это означает, что наш алгоритм должен прочитать игровой экран и ре-

[1] Обморожение, гибель от холода. – *Прим. перев.*

шить, какое игровое действие предпринять в данный момент, чтобы получить максимально возможный игровой счет.

Эта задача может быть разделена на две логические подзадачи:

- задача анализа изображения, которая кодирует состояние текущей игровой ситуации на экране, включая положение игрового персонажа, препятствия и противников;
- задача обучения с подкреплением, которая сводится к обучению нейросети, аппроксимирующей Q-матрицу для построения правильного соответствия между конкретным состоянием игры и действиями игрока.

В задачах, связанных с анализом визуальных образов или других евклидовых данных большой размерности, обычно используются *сверточные нейронные сети* (convolutional neural network, CNN). Мощь CNN основана на их способности значительно сократить количество параметров обучения по сравнению с другими типами нейросетей в задачах зрительного распознавания. Иерархия CNN обычно имеет несколько последовательных сверточных слоев в сочетании с нелинейными полностью связанными слоями и заканчивается полностью связанным слоем, за которым следует слой потерь. Последние полностью связанные слои и слой потерь образуют архитектуру построения логического вывода нейронной сети. В случае глубокого RL эти слои аппроксимируют Q-значения. Далее мы рассмотрим детали реализации сверточного слоя.

Сверточные слои

Изучая организацию зрительной коры высших форм жизни (в том числе людей), исследователи нашли источник вдохновения для разработки CNN. Каждый нейрон зрительной коры головного мозга реагирует на сигналы, поступающие из ограниченной области поля зрения – рецептивного поля нейрона. Рецептивные поля различных нейронов частично перекрываются, что позволяет им покрывать все поле зрения, как показано на рис. 10.2.

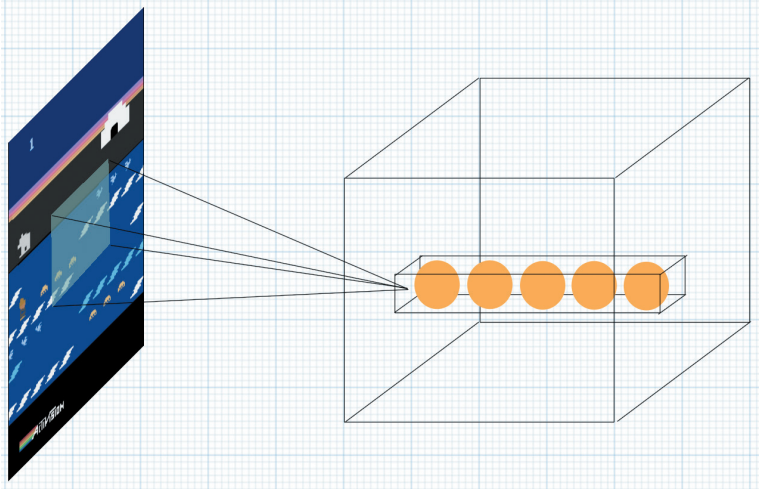


Рис. 10.2. Схема связей между рецептивным полем (слева) и нейронами в сверточном слое (справа)

Сверточный слой состоит из столбца нейронов, где каждый нейрон в одном столбце связан с одним и тем же рецептивным полем. Этот столбец представляет собой набор фильтров[1] (ядер, kernel). Каждый фильтр определяется размером рецептивного поля и количеством каналов. Количество каналов определяет *глубину* столбца нейронов, в то время как размер рецептивного поля определяет количество столбцов в сверточном слое. Когда рецептивное поле перемещается над полем зрения, на каждом шаге активируется новый столбец нейронов.

Как мы упоминали ранее, каждый сверточный слой обычно объединяется с полностью связанным слоем с нелинейной активацией, таким как *выпрямленный линейный блок* (rectified linear unit, ReLU). Функция активации ReLU позволяет отфильтровывать отрицательные значения в соответствии со следующей формулой:

$$f(x) = x^{+} = max(0, x).$$

Здесь x – вход нейрона.

В архитектуре нейросети несколько сверточных уровней соединены с несколькими полностью связанными уровнями, которые осуществляют построение логического вывода. Далее мы обсудим архитектуру CNN, которая используется в нашем эксперименте.

Архитектура CNN для обучения игрового агента Atari

В нашем эксперименте мы будем использовать архитектуру CNN, состоящую из трех сверточных слоев с 32, 64 и 64 каналами, за которыми следует полностью связанный слой с 512 узлами и выходной слой с количеством узлов, соответствующих количеству игровых действий. Сверточные слои имеют размеры ядра 8×8, 4×4 и 3×3 и перемещаются с шагом 4, 2 и 1 соответственно. За всеми сверточными и полностью связными слоями следует нелинейный блок ReLU.

Исходный код для создания описанной модели сетевого графа с использованием инфраструктуры TensorFlow выглядит следующим образом:

```
class LargeModel(Model):
    def _make_net(self, x, num_actions):
        x = self.nonlin(self.conv(x, name='conv1', num_outputs=32,
                                  kernel_size=8, stride=4, std=1.0))
        x = self.nonlin(self.conv(x, name='conv2', num_outputs=64,
                                  kernel_size=4, stride=2, std=1.0))
        x = self.nonlin(self.conv(x, name='conv3', num_outputs=64,
                                  kernel_size=3, stride=1, std=1.0))
        x = self.flattenallbut0(x)
        x = self.nonlin(self.dense(x, 512, 'fc'))

        return self.dense(x, num_actions, 'out', std=0.1)
```

[1] Иногда эти фильтры называют *ядрами* или кернелами, от англ. kernel. – *Прим. перев.*

В результате построения этой архитектуры CNN содержит около 4 млн обучаемых параметров. Далее мы обсудим, как происходит процесс глубокого обучения с подкреплением в нашем эксперименте.

Полный исходный код представлен в файле `dqn.py` в файловом архиве книги.

10.3.3 Обучение игрового агента

В нашем эксперименте для обучения с подкреплением использован метод нейроэволюции. Этот метод основан на простом генетическом алгоритме, который развивает популяцию особей. Генотип каждой особи кодирует вектор обучаемых параметров нейросети игрового агента. Под обучаемыми параметрами мы подразумеваем веса связей между узлами сети. В каждом поколении каждый генотип оценивается в тестовой среде на игре Frostbite и возвращает определенный показатель приспособленности. Мы оцениваем каждого агента (геном) на 20 000 фреймов игры. В течение периода оценки игровой персонаж может играть несколько раз, и итоговая оценка игры Atari – это оценка приспособленности, которая служит сигналом вознаграждения с точки зрения RL.

Далее мы обсудим схему кодирования генома, которая позволяет нам закодировать более 4 млн обучаемых параметров нейросети игрового агента.

Схема кодирования генома

Глубокая нейросеть, которую мы используем в качестве контроллера игрового агента, имеет около 4 млн обучаемых параметров. Каждый обучаемый параметр – это вес связи между двумя узлами нейронной сети. Традиционно обучение нейронных сетей заключается в поиске подходящих значений всех весов связей, что позволяет нейронной сети аппроксимировать функцию, которая описывает специфику моделируемого процесса.

Традиционный способ оценки этих обучаемых параметров состоит в использовании некоторой формы обратного распространения ошибки на основе градиентного спуска, которая является вычислительно затратной. С другой стороны, алгоритм нейроэволюции позволяет нам обучать нейросеть, используя заимствованный у природы генетический алгоритм. Алгоритм нейроэволюции ищет правильную конфигурацию нейросети, применяя к обучаемым параметрам ряд мутаций и рекомбинаций. Однако для использования генетического алгоритма мы должны разработать соответствующую схему кодирования фенотипа нейросети. После этого популяция особей (геномы, кодирующие фенотип нейросети) может создаваться и развиваться с использованием простого генетического алгоритма, который мы обсудим позже.

Как мы упоминали ранее, схема кодирования должна создавать компактные геномы, способные кодировать значения более 4 млн весов связей между узлами глубокой нейросети, управляющей игровым агентом. Мы нуждаемся в компактных геномах, чтобы уменьшить вычислительные затраты, связанные с оценкой генетического алгоритма. Далее обсудим схему кодирования генома, которая может быть использована для построения нейросети с большим фенотипом.

Схема кодирования генома

Исследователи из лаборатории Uber AI предложили схему кодирования, которая использует для кодирования фенотипа нейросети начальное число генератора псевдослучайных чисел. В этой схеме геном представлен в виде списка начальных значений, который применяется последовательно для генерации значений всех весов связей (обучаемых параметров), экспрессированных между узлами нейросети.

Другими словами, первое начальное значение в списке представляет начальное значение инициализации системы правил, которое совместно используется генеалогией потомков одного родителя. Все последующие начальные значения представляют специфические мутации, которые приобретаются потомством в процессе эволюции. Каждое начальное случайное значение применяют последовательно для получения вектора параметров нейросети определенного фенотипа. Следующая формула описывает получение вектора параметров фенотипа для конкретной особи (n):

$$\theta^n = \theta^0 + \sigma\sum_{i=1}^{n-1}\varepsilon(\tau_i).$$

Здесь τ – кодировка θ^n и состоит из списка начальных значений мутации; $\varepsilon(\tau_i) \sim \mathcal{N}(0, I)$ представляет собой детерминированный генератор гауссовых псевдослучайных чисел с начальным числом τ_i, которое создает вектор длиной $|\theta|$ θ^0; – вектор начальных параметров, который создается во время инициализации следующим образом: $\theta^0 = \phi(\tau_0)$, где ϕ – детерминированная функция инициализации; σ – степень мутации, которая определяет силу влияния всех последующих векторов параметров на вектор начальных параметров θ^0.

В текущей реализации $\varepsilon(\tau_i)$ – это предварительно вычисленная таблица с 250 млн случайных векторов, проиндексированных с использованием 28-разрядных начальных чисел. Это сделано для ускорения обработки во время выполнения, поскольку поиск по индексу выполняется быстрее, чем генерация новых случайных чисел. Далее мы обсудим, как реализовать схему кодирования в исходном коде Python.

Реализация схемы кодирования генома

Следующий исходный код реализует получение параметров нейросети, как определено формулой из предыдущего раздела (см. функцию `compute_weights_from_seeds`):

```
idx = seeds[0]
theta = noise.get(idx, self.num_params).copy() * self.scale_by

for mutation in seeds[1:]:
    idx, power = mutation
    theta = self.compute_mutation(noise, theta, idx, power)
return theta
```

Функция `compute_mutation` выполняет один шаг вычисления мутировавших параметров нейросети следующим образом:

```
def compute_mutation(self, noise, parent_theta, idx, mutation_power):
return parent_theta + mutation_power * noise.get(idx, self.num_params)
```

Этот код берет вектор обучаемых параметров родителя и добавляет к нему случайный вектор, который создается детерминированным псевдослучайным генератором с использованием определенного начального индекса. Параметр степени мутации масштабирует сгенерированный случайный вектор перед его добавлением к вектору параметров родителя.

Для получения дополнительной информации о реализации обратитесь к файлу `base.py` в файловом архиве книги.

Далее мы обсудим особенности простого генетического алгоритма, который используется для обучения игрового агента навыкам игры в Atari Frostbite.

Простой генетический алгоритм

Простой генетический алгоритм, который используется в нашем эксперименте, развивает популяцию *N* особей в течение нескольких поколений эволюции. Как мы упоминали ранее, каждый отдельный геном кодирует вектор обучаемых параметров нейросети. Кроме того, в каждом поколении мы выбираем *T* лучших особей, которые станут родителями следующего поколения.

Процесс воспроизводства следующего поколения реализуется следующим образом. Для *N* – 1 повторов мы делаем вот что.

1. Родитель выбирается равномерным случайным образом и удаляется из списка выбора.
2. К выбранному родителю применяется мутация путем прибавления аддитивного гауссова шума к вектору параметров, который закодирован в особи.
3. Затем мы добавляем новый организм в список особей для следующего поколения.

После этого лучшая особь из текущего поколения копируется в неизменном состоянии в следующее поколение (элитарность). Чтобы гарантировать, что была выбрана лучшая особь, мы оцениваем каждого из 10 лучших игроков в текущем поколении на 30 дополнительных сеансах игры. Особь с самым высоким средним показателем приспособленности затем выбирается в качестве элиты для копирования в следующее поколение.

Мутация родительской особи реализована следующим образом:

```
def mutate(self, parent, rs, noise, mutation_power):
    parent_theta, parent_seeds = parent
    idx = noise.sample_index(rs, self.num_params)
    seeds = parent_seeds + ((idx, mutation_power), )
    theta = self.compute_mutation(noise, parent_theta, idx,
                                  mutation_power)
    return theta, seeds
```

Эта функция получает фенотип и генотип родительской особи, источник случайных чисел, а также таблицу предварительно подготовленных шумов

(250 млн векторов) и значение степени мутации. Источник случайных чисел выдает случайное начальное число (`idx`), которое используется в качестве индекса, чтобы мы могли выбрать соответствующий вектор параметров из таблицы шумов. После этого создаем геном потомства, комбинируя список родительских начальных случайных чисел с новым начальным случайным числом. Наконец, создаем фенотип потомства, комбинируя фенотип родителя с гауссовым шумом, который был извлечен из общей таблицы шума, используя случайный начальный индекс, который мы получили ранее (`idx`). В следующем разделе рассмотрим эксперимент по обучению агента навыкам игры в Frostbite.

10.4 Обучение агента навыкам игры в Frostbite

Теперь, когда мы обсудили теорию реализации игрового агента, мы готовы приступить к практике. Наша реализация основана на исходном коде, предоставленном Uber AI Lab на GitHub по адресу **https://github.com/uber-research/deep-neuroevolution**. Исходный код в этом репозитории содержит реализацию двух методов обучения глубокой нейросети: методы на основе ЦП для многоядерных систем (до 720 ядер) и методы на основе графического процессора. Мы заинтересованы в реализации на основе графического процессора, потому что большинство пользователей не имеют доступа к таким продвинутым технологиям, как компьютер с 720 процессорными ядрами. В то же время не составит особого труда обзавестись графическим процессором Nvidia.

Далее мы обсудим детали программной реализации.

10.4.1 Учебная среда Atari

Во время обучения агентов нам нужно смоделировать реальный игровой процесс в системе Atari. Это можно сделать с помощью среды эмуляции игр ALE, которая имитирует систему Atari, запускающую образы игр из ПЗУ. ALE предоставляет интерфейс, позволяющий нам захватывать кадры игрового экрана и управлять игрой, эмулируя игровой контроллер. Здесь мы будем использовать модификацию ALE, которая доступна по адресу **https://github.com/yari-com/atari-py**.

Наша реализация использует инфраструктуру TensorFlow для реализации моделей искусственных нейронных сетей и выполнения их на графическом процессоре. Следовательно, мы должны выстроить мост между ALE и TensorFlow. Это делается путем добавления *пользовательской операции* TensorFlow с использованием языка программирования C++ для повышения эффективности. Соответствующий интерфейс Python также доступен в виде класса Python `AtariEnv` , который можно найти в файле `tf_atari.py` в файловом архиве книги.

Класс `AtariEnv` включает функции, позволяющие выполнить один шаг игры, сбросить настройки игры и получить текущее состояние игры (наблюдение). Далее мы обсудим каждую функцию.

Функция шага игры

Функция шага игры выполняет один шаг игры, используя переданные ей действия:

```
def step(self, action, indices=None, name=None):
    if indices is None:
        indices = np.arange(self.batch_size)
    with tf.variable_scope(name, default_name='AtariStep'):
        rew, done = gym_tensorflow_module.environment_step(
                            self.instances, indices, action)
        return rew, done
```

Эта функция применяет игровое действие, полученное от нейросети контроллера, к текущей игровой среде. Обратите внимание, что эта функция может выполнять один шаг игры одновременно в нескольких игровых экземплярах. Параметр `self.batch_size` или длина входного тензора `indices` определяет количество экземпляров игры, которыми мы располагаем. Функция возвращает два тензора: один тензор с вознаграждениями (счет игры) и другой с флагами, указывающими, завершена ли текущая сессия игры после этого шага (пройдена или провалена). Оба тензора имеют длину, равную `self.batch_size`, или длину входного тензора `indices`.

Далее мы обсудим, как выполняется наблюдение за игрой.

Функция наблюдения за игрой

Эта функция получает текущее состояние игры из среды Atari в качестве буфера экрана и реализована следующим образом:

```
def observation(self, indices=None, name=None):
    if indices is None:
        indices = np.arange(self.batch_size)
    with tf.variable_scope(name, default_name='AtariObservation'):
        with tf.device('/cpu:0'):
            obs = gym_tensorflow_module.environment_observation(
                              self.instances, indices, T=tf.uint8)
        obs = tf.gather(tf.constant(self.color_pallete),
                                             tf.cast(obs,tf.int32))
        obs = tf.reduce_max(obs, axis=1)
        obs = tf.image.resize_bilinear(obs, self.warp_size,
                                               align_corners=True)
        obs.set_shape((None,) + self.warp_size + (1,))
        return obs
```

Данная функция получает скриншот из среды Atari и оборачивает его в тензор, совместимый со средой TensorFlow. Функция наблюдения за игрой тоже позволяет нам получать состояние из нескольких игр, количество которых определяется либо параметром `self.batch_size`, либо длиной входного параметра `indices`. Функция возвращает скриншоты из нескольких игр, обернутые в тензор.

Нам также необходимо иметь функцию для сброса среды Atari в случайное начальное состояние.

Функция сброса среды Atari

Чтобы обучить игровых агентов, нам нужно реализовать функцию, которая запускает среду Atari из определенного случайного состояния. Крайне важно реализо-

вать стохастическую функцию сброса среды, чтобы наш агент мог играть в игру из любого начального состояния. Функция реализована следующим образом:

```
def reset(self, indices=None, max_frames=None, name=None):
    if indices is None:
        indices = np.arange(self.batch_size)
    with tf.variable_scope(name, default_name='AtariReset'):
        noops = tf.random_uniform(tf.shape(indices), minval=1,
                                  maxval=31, dtype=tf.int32)
        if max_frames is None:
            max_frames = tf.ones_like(indices, dtype=tf.int32) * \
                                       (100000 * self.frameskip)
        import collections
        if not isinstance(max_frames, collections.Sequence):
            max_frames = tf.ones_like(indices, dtype=tf.int32) * \
                                    max_frames
        return gym_tensorflow_module.environment_reset(self.instances,
                        indices, noops=noops, max_frames=max_frames)
```

Эта функция использует входной параметр `indices` для одновременного сброса нескольких экземпляров игр Atari в случайные начальные состояния. Функция также определяет максимальное количество фреймов для каждого экземпляра игры.

Далее мы обсудим, как выполняется расчет RL на ядрах графического процессора.

10.4.2 Расчет RL на ядрах GPU

В нашем эксперименте мы реализуем расчет RL с использованием инфраструктуры TensorFlow на устройствах с графическим процессором. Это означает, что все вычисления, связанные с распространением входных сигналов через контроллер нейросети, выполняются на графическом процессоре. Такой прием позволяет нам эффективно рассчитывать более 4 млн параметров обучения – весов связей между управляющими узлами нейросети – для каждого шага игры. Кроме того, мы можем одновременно моделировать несколько сеансов игры, каждый из которых управляется отдельной нейросетью контроллера.

Одновременный обсчет нескольких нейросетей игровых контроллеров реализуется двумя классами Python: `RLEvalutionWorker` и `ConcurrentWorkers`. Далее мы обсудим каждый класс.

Полный исходный код представлен в файле `concurrent_worker.py` в файловом архиве книги.

Класс RLEvalutionWorker

Этот класс содержит конфигурацию и сетевой граф нейросети контроллера. Он предоставляет нам методы, позволяющие создавать сетевой граф нейросети контроллера, запускать цикл расчетов для созданного сетевого графа и помещать новые задачи в цикл обсчета. Далее мы обсудим, как из сетевой модели получается сетевой граф.

Создание графа сети

Сетевой граф TensorFlow создается функцией `make_net`, которая в качестве входных параметров получает конструктор нейросети модели, идентификатор устройства GPU и размер пакета. Сетевой граф создается следующим образом.

1. Начнем с создания модели нейросети контроллера и среды выполнения игры:

```
self.model = model_constructor()
…
with tf.variable_scope(None, default_name='model'):
    with tf.device('/cpu:0'):
        self.env = self.make_env_f(self.batch_size)
```

2. Далее мы создадим заполнители, чтобы могли получать значения во время вычисления сетевого графа. Также создадим оператор для сброса игры перед началом нового игрового эпизода:

```
self.placeholder_indices = tf.placeholder(tf.int32,
                                           shape=(None, ))
self.placeholder_max_frames = tf.placeholder(
                                  tf.int32, shape=(None, ))
self.reset_op = self.env.reset(
                    indices=self.placeholder_indices,
                   max_frames=self.placeholder_max_frames)
```

3. После этого, используя контекст доступного устройства графического процессора, мы создадим двух операторов, которые будут получать наблюдения за состоянием игры и вычислять последующие действия игры:

```
with tf.device(device):
    self.obs_op = self.env.observation(
                    indices=self.placeholder_indices)
    obs = tf.expand_dims(self.obs_op, axis=1)
    self.action_op = self.model.make_net(obs,
                    self.env.action_space,
                    indices=self.placeholder_indices,
                    batch_size=self.batch_size,
                   ref_batch=ref_batch)
```

4. Оператор `action` возвращает массив значений вероятности действия, которые необходимо отфильтровать, если пространство действия дискретно:

```
if self.env.discrete_action:
    self.action_op = tf.argmax(
                self.action_op[:tf.shape(
                self.placeholder_indices)[0]],
               axis=-1, output_type=tf.int32)
```

Код проверяет, требует ли текущая игровая среда дискретных действий, и оборачивает оператор `action`, используя встроенный оператор `tf.argmax` платформы TensorFlow. Оператор `tf.argmax` возвращает индекс действия

с наибольшим значением, который может служить указанием на необходимость выполнения конкретного игрового действия.

Игровая среда Atari представляет собой дискретную среду, что означает, что на каждый временной шаг принимается только одно действие.

5. Наконец, мы создаем оператор для выполнения одного игрового шага:

```
with tf.device(device):
    self.rew_op, self.done_op = \
                self.env.step(self.action_op,
                indices=self.placeholder_indices)
```

Этот код создает оператор, который возвращает операции для получения вознаграждений `self.rew_op` и статус завершения игры `self.done_op` после выполнения одного шага игры.

Далее мы обсудим, как реализован цикл расчетов.

Цикл оценочного испытания графа

Это цикл, который мы используем для оценочного испытания ранее созданного сетевого графа на нескольких играх параллельно – количество игр, которые можно оценить одновременно, определяется параметром `batch_size`.

Цикл оценки определен в функции `_loop` и реализован следующим образом.

1. Мы начинаем с создания массивов для хранения значений оценки игры в нескольких эпизодах:

```
running = np.zeros((self.batch_size,), dtype=np.bool)
cumrews = np.zeros((self.batch_size, ), dtype=np.float32)
cumlen = np.zeros((self.batch_size, ), dtype=np.int32)
```

2. Затем запускаем цикл и устанавливаем соответствующие индексы массива `running`, который мы только что создали, равными `True`:

```
while True:
    # Ничего не загружено
    if not any(running):
        idx = self.queue.get()
        if idx is None:
           break
        running[idx] = True
    while not self.queue.empty():
       idx = self.queue.get()
       if idx is None:
             break
       running[idx] = True
```

3. Используя массив `indices`, мы готовы выполнить один оператор шага игры и собрать результаты:

```
indices = np.nonzero(running)[0]
rews, is_done, _ = self.sess.run(
          [self.rew_op, self.done_op, self.incr_counter],
          {self.placeholder_indices: indices})
```

```
cumrews[running] += rews
cumlen[running] += 1
```

4. Наконец, нам нужно проверить, не завершилась ли какая-либо из оцененных игр выигрышем или достижением максимального количества фреймов. Для всех выполненных задач мы выполняем ряд операций, а именно:

```
if any(is_done):
    for idx in indices[is_done]:
        self.sample_callback[idx](self, idx,
              (self.model.seeds[idx],cumrews[idx],
                                        cumlen[idx]))
    cumrews[indices[is_done]] = 0.
    cumlen[indices[is_done]] = 0.
    running[indices[is_done]] = False
```

Предыдущий код использует индексы всех выполненных задач и вызывает соответствующие зарегистрированные обратные вызовы перед сбросом переменных коллектора по определенным индексам.

Теперь мы готовы обсудить, как добавить и запустить новое задание с помощью *воркера* (worker).

Система запуска асинхронных задач

Эта функция регистрирует конкретную задачу для выполнения воркером в контексте графического процессора. В качестве входных данных она принимает идентификатор задачи, владельца объекта задачи и обратный вызов, который должен быть выполнен при завершении задачи. Эта функция определена под именем `run_async` и реализована следующим образом.

1. Сначала она извлекает соответствующие данные из объекта задачи и загружает их в текущий сеанс TensorFlow:

```
theta, extras, max_frames=task
self.model.load(self.sess, task_id, theta, extras)
if max_frames is None:
    max_frames = self.env.env_default_timestep_cutoff
```

Здесь `theta` представляет собой массив со всеми весами связей в нейросети контроллера, `extras` содержит список начальных случайных чисел соответствующего генома, а `max_frames` – это предельное количество фреймов игры.

2. Затем мы запускаем сеанс TensorFlow с `self.reset_op`, который сбрасывает определенную игровую среду с указанным индексом:

```
self.sess.run(self.reset_op, {self.placeholder_indices:[task_id],
              self.placeholder_max_frames:[max_frames]})
self.sample_callback[task_id] = callback
self.queue.put(task_id)
```

Код запускает `self.reset_op` в сеансе TensorFlow. Кроме того, мы регистрируем текущий идентификатор задачи с оператором `reset` и предельным количеством фреймов игры для данной задачи. Идентификатор задачи используется в цикле оценки, чтобы связать результаты оценки сетевого графа

с определенным геномом в популяции. Далее мы обсудим, как поддерживаются параллельные асинхронные воркеры.

Класс ConcurrentWorkers

Класс `ConcurrentWorkers` содержит конфигурацию среды параллельного выполнения, которая включает в себя несколько оценивающих воркеров (экземпляры `RLEvalutionWorker`) и вспомогательные подпрограммы для поддержки множественного выполнения параллельных задач.

Создание оценивающих воркеров

Одной из основных обязанностей класса `ConcurrentWorkers` является создание и управление экземплярами `RLEvalutionWorker`. Это делается в конструкторе класса следующим образом:

```
self.workers = [RLEvalutionWorker(make_env_f, *args,
     ref_batch=ref_batch,
     **dict(kwargs, device=gpus[i])) for i in range(len(gpus))]
self.model = self.workers[0].model
self.steps_counter = sum([w.steps_counter for w in self.workers])
self.async_hub = AsyncTaskHub()
self.hub = WorkerHub(self.workers, self.async_hub.input_queue,
                     self.async_hub)
```

Здесь мы создаем определенное количество экземпляров `RLEvalutionWorker`, которые соотносятся с количеством графических процессоров, доступных в системе. Затем инициализируем выбранную графовую модель нейросети и создаем вспомогательные подпрограммы для управления параллельным выполнением асинхронных задач. Далее мы обсудим, как планируется выполнение рабочих заданий.

Выполнение рабочих заданий и результаты мониторинга

Чтобы использовать механизм оценки RL, который мы описали ранее, нам нужен метод планирования рабочих заданий для оценки и мониторинга результатов. Он реализован в функции `monitor_eval`, которая получает список геномов в популяции и оценивает их успешность в игровой среде Atari. Эта функция состоит из двух важных частей.

1. Сначала мы перебираем все геномы в списке и создаем асинхронное рабочее задание, чтобы каждый геном получил свою оценку в игровой среде Atari:

```
tasks = []
for t in it:
    tasks.append(self.eval_async(*t, max_frames=max_frames,
                                 error_callback=error_callback))
    if time.time() - tstart > logging_interval:
        cur_timesteps = self.sess.run(self.steps_counter)
        tlogger.info('Num timesteps:', cur_timesteps,
         'per second:',
         (cur_timesteps-last_timesteps)//(time.time()-tstart),
         'num episodes finished: {}/{}'.format(
```

```
            sum([1 if t.ready() else 0 for t in tasks]),
            len(tasks)))
        tstart = time.time()
        last_timesteps = cur_timesteps
```

Этот код планирует асинхронную оценку каждого генома из списка и сохраняет ссылку на каждое асинхронное задание для последующего использования. Также мы периодически выводим результаты процесса оценки уже запланированных заданий. Теперь давайте посмотрим, как отслеживать результаты оценки.

2. Следующий блок кода ожидает завершения асинхронных заданий:

```
while not all([t.ready() for t in tasks]):
    if time.time() - tstart > logging_interval:
        cur_timesteps = self.sess.run(self.steps_counter)
        tlogger.info('Num timesteps:', cur_timesteps, 'per
   second:', (cur_timesteps-last_timesteps)//(time.time()-tstart),
   'num episodes:', sum([1 if t.ready() else 0 for t in tasks]))
        tstart = time.time()
        last_timesteps = cur_timesteps
    time.sleep(0.1)
```

Здесь перебираем все ссылки на запланированные асинхронные задания и ожидаем их завершения. Также мы периодически выводим результаты оценивания.

3. Наконец, после выполнения всех заданий мы собираем оценки:

```
tlogger.info(
   'Done evaluating {} episodes in {:.2f} seconds'.format(
                       len(tasks), time.time()-tstart_all))
return [t.get() for t in tasks]
```

Код перебирает все ссылки на запланированные асинхронные задания и создает список результатов оценивания. Далее мы обсудим реализацию движка эксперимента.

10.4.3 Движок эксперимента

Движок эксперимента считывает конфигурацию эксперимента, определенную в файле JSON, и запускает процесс нейроэволюции в течение указанного количества игровых шагов. В нашем эксперименте оценивание генома прекращается после достижения 1,5 млрд шагов по времени Frostbite. Далее мы обсудим детали конфигурации эксперимента.

Файл конфигурации эксперимента

Файл конфигурации для нашего эксперимента имеет следующее содержание:

```
{
    "game": "frostbite",
    "model": "LargeModel",
    "num_validation_episodes": 30,
    "num_test_episodes": 200,
    "population_size": 1000,
```

```
    "episode_cutoff_mode": 5000,
    "timesteps": 1.5e9,
    "validation_threshold": 10,
    "mutation_power": 0.002,
    "selection_threshold": 20
}
```

Назначение параметров конфигурации:

- `game` – это название игры, под которым она зарегистрирована в ALE. Полный список поддерживаемых игр доступен в файле `tf_atari.py` в файловом архиве книги;
- `model` – название модели сетевого графа, которую следует использовать для построения нейросети контроллера. Модели определены в файле `dqn.py` в файловом архиве книги;
- `num_validation_episodes` – определяет, сколько игровых эпизодов используется для оценки лучших особей в популяции. После этого шага мы можем выбрать истинную элиту популяции;
- `num_test_episodes` – устанавливает количество игровых эпизодов, которые можно использовать для проверки производительности выбранной элиты популяции;
- `population_size` – определяет количество геномов в популяции;
- `episode_cutoff_mode` – определяет, по какому условию останавливается оценочная игра для определенного генома. Текущий эпизод игры может остановиться либо после выполнения определенного количества временных шагов, либо с помощью сигнала остановки по умолчанию соответствующей игровой среды;
- `timesteps` – устанавливает общее количество временных шагов игры, которые должны быть выполнены в процессе нейроэволюции;
- `validation_threshold` – задает количество лучших особей, выбранных из каждого поколения для дополнительной проверки. Элита популяции выбирается из этих особей;
- `mutation_power` – определяет, в какой степени последующие мутации, применяемые к особи, влияют на обучаемые параметры (веса связей);
- `selection_threshold` – определяет, сколько родительских особей может производить потомство в следующем поколении.

Теперь мы готовы обсудить подробности реализации движка эксперимента.

Конфигурация эксперимента хранится в файле `ga_atari_config.json` в файловом архиве книги.

Реализация движка эксперимента

Движок эксперимента создает среду параллельной оценки геномов и запускает эволюционный цикл для множества особей. Давайте обсудим основные детали реализации.

1. Мы начинаем с настройки среды оценки, загружая модель нейросети контроллера и создавая параллельные воркеры:

```
Model = neuroevolution.models.__dict__[config['model']]
all_tstart = time.time()
def make_env(b):
    return gym_tensorflow.make(game=config["game"],
                               batch_size=b)
worker = ConcurrentWorkers(make_env, Model, batch_size=64)
```

2. Затем создаем таблицу со случайными шумоподобными значениями, которая будет использоваться в качестве случайных начальных чисел, и определяем функцию для создания потомства в следующем поколении:

```
noise = SharedNoiseTable()
rs = np.random.RandomState()
def make_offspring():
    if len(cached_parents) == 0:
        return worker.model.randomize(rs, noise)
    else:
       assert len(cached_parents) == config['selection_threshold']
       parent = cached_parents[rs.randint(len(cached_parents))]
       theta, seeds = worker.model.mutate( parent, rs, noise,
                      mutation_power=state.sample(
                      state.mutation_power))
       return theta, seeds
```

3. После этого начинается основной цикл эволюции. Мы используем ранее определенную функцию, чтобы создать популяцию потомства для текущего поколения:

```
tasks = [make_offspring() for _ in range(
                           config['population_size'])]
for seeds, episode_reward, episode_length in \
    worker.monitor_eval(tasks, max_frames=state.tslimit * 4):
    results.append(Offspring(seeds,
                  [episode_reward], [episode_length]))

state.num_frames += sess.run(worker.steps_counter) - \
                            frames_computed_so_far
```

Здесь мы создаем рабочие задания для каждого потомства в популяции и ставим каждое задание в очередь на оценку в игровой среде.

4. Завершив оценку каждой особи в популяции, мы начинаем оценивать лучших особей, чтобы выбрать элиту:

```
state.population = sorted(results,
              key=lambda x:x.fitness, reverse=True)
…
validation_population = state.\
               population[:config['validation_threshold']]
if state.elite is not None:
    validation_population = [state.elite] + \
                                validation_population[:-1]
validation_tasks = [
```

```
    (worker.model.compute_weights_from_seeds(noise,
    validation_population[x].seeds, cache=cached_parents),
    validation_population[x].seeds) for x in range(
                        config['validation_threshold'])]
_,population_validation, population_validation_len =\
    zip(*worker.monitor_eval_repeated(validation_tasks,
    max_frames=state.tslimit * 4,
    num_episodes=config['num_validation_episodes']))
```

5. Используя результаты оценки 10 лучших особей, мы выбираем элиту популяции и проводим финальные тесты, чтобы оценить ее эффективность:

```
population_elite_idx = np.argmax(population_validation)
state.elite = validation_population[population_elite_idx]
elite_theta = worker.model.compute_weights_from_seeds(
          noise, state.elite.seeds, cache=cached_parents)
_,population_elite_evals,population_elite_evals_timesteps=\
              worker.monitor_eval_repeated(
              [(elite_theta, state.elite.seeds)],
              max_frames=None,
              num_episodes=config['num_test_episodes'])[0]
```

Элитные особи будут скопированы в следующее поколение, как есть.

6. Наконец, мы выбираем лучшие особи из текущей популяции, которые станут родителями следующего поколения:

```
if config['selection_threshold'] > 0:
    tlogger.info("Caching parents")
    new_parents = []
    if state.elite in \
        state.population[:config['selection_threshold']]:
        new_parents.extend([
             (worker.model.compute_weights_from_seeds(
             noise, o.seeds, cache=cached_parents), o.seeds)
for o in state.population[:config['selection_threshold']]])
    else:
        new_parents.append(
            (worker.model.compute_weights_from_seeds(
            noise, state.elite.seeds, cache=cached_parents),
            state.elite.seeds))
        new_parents.extend([
            (worker.model.compute_weights_from_seeds(
            noise, o.seeds, cache=cached_parents), o.seeds)
for o in state.population[:config['selection_threshold']-1]])
```

Предыдущий код собирает лучшие особи популяции, которые достойны стать родителями следующего поколения. Кроме того, он добавляет текущую элиту в список родителей, если ее еще нет.

Теперь мы готовы обсудить, как запустить эксперимент.

10.5 Запуск эксперимента с игрой Atari Frostbite

Итак, мы обсудили все нюансы исходного кода, и пришло время запустить эксперимент. Тем не менее первое, что нам нужно сделать, – это создать рабочую среду, о которой мы поговорим позже.

10.5.1 Настройка рабочей среды

Обучение агента навыкам прохождения игр Atari подразумевает, что в процессе эксперимента необходимо обучить большую нейросеть контроллера. Мы уже говорили, что нейросеть имеет более 4 млн обучаемых параметров и требует много вычислительных ресурсов для оценки. К счастью, современные графические ускорители позволяют выполнять тяжелые параллельные вычисления. Эта возможность пригодится для нашего эксперимента, потому что в процессе эволюции нам нужно неоднократно оценивать прохождение игры каждой особью. Без графического ускорителя эти расчеты заняли бы много времени или потребовали огромного количества процессорных ядер (около 720).

Итак, давайте приступим к пошаговой подготовке рабочей среды.

1. Для работы требуется наличие в системе видеоускорителя Nvidia (например, GeForce 1080Ti); также нужно установить соответствующую версию Nvidia CUDA SDK. Более подробную информацию о CUDA SDK и его установке можно найти по адресу **https://developer.nvidia.com/cuda-toolkit**.
2. Далее необходимо установить инструмент сборки CMake, как описано на **https://cmake.org**.
3. Теперь при помощи Anaconda нужно создать новую среду Python и установить все зависимости, которые используются в реализации эксперимента:

```
$ conda create -n deep_ne python=3.5
$ conda activate deep_ne
$ conda install -c anaconda tensorflow-gpu
$ pip install gym
$ pip install Pillow
```

Эти команды создают и активируют новую среду Python 3.5, а затем устанавливают TensorFlow, OpenAI Gym и библиотеку изображений Python в качестве зависимостей.

4. После этого вам необходимо клонировать репозиторий с исходным кодом эксперимента:

```
$ git clone
https://github.com/PacktPublishing/Hands-on-Neuroevolution-with-Pyt
hon.git
$ cd Hands-on-Neuroevolution-with-Python/Chapter10
```

После выполнения этих команд в вашем текущем рабочем каталоге появится исходный код эксперимента.

5. Теперь вам нужно построить среду ALE и интегрировать ее в свой эксперимент. Клонируйте репозиторий ALE в соответствующий каталог и соберите среду с помощью следующих команд:

```
$ cd cd gym_tensorflow/atari/
$ git clone https://github.com/yaricom/atari-py.git
$ cd ./atari-py && make
```

Теперь у вас есть рабочая среда ALE, интегрированная с TensorFlow. Мы можем использовать ее для оценки нейросетей контроллера, которые создаются из популяции геномов, претендующих на победу в игре Atari (в нашем эксперименте это игра Frostbite).

6. После завершения сборки ALE вам нужно организовать связь между OpenAI Gym и TensorFlow, которая является специфической для реализации данного эксперимента:

```
$ cd ../..gym_tensorflow && make
```

Теперь у вас есть полностью настроенная рабочая среда, и вы готовы приступить к экспериментам. Далее мы обсудим, как запустить эксперимент.

10.5.2 Запуск эксперимента

Итак, мы настроили рабочую среду и готовы начать эксперимент. Для запуска эксперимента достаточно войти в каталог `Chapter10` и выполнить следующую команду:

```
$ python ga.py -c configurations/ga_atari_config.json -o out
```

Эта команда запускает эксперимент с использованием файла конфигурации, указанного в качестве первого параметра. Результаты эксперимента будут сохранены в каталоге `out`.

После завершения эксперимента консольный вывод должен выглядеть примерно так:

```
...
| PopulationEpRewMax | 3.47e+03 |
| PopulationEpRewMean | 839 |
| PopulationEpCount | 1e+03 |
| PopulationTimesteps | 9.29e+05 |
| NumSelectedIndividuals | 20 |
| TruncatedPopulationRewMean | 3.24e+03 |
| TruncatedPopulationValidationRewMean | 2.36e+03 |
| TruncatedPopulationEliteValidationRew | 3.1e+03 |
| TruncatedPopulationEliteIndex | 0 |
...
| TruncatedPopulationEliteTestRewMean | 3.06e+03 |
...
Current elite: (47236580, (101514609, 0.002), (147577692, 0.002),
(67106649, 0.002), (202520553, 0.002), (230555280, 0.002), (38614601,
0.002), (133511446, 0.002), (27624159, 0.002), (233455358, 0.002),
(73372122, 0.002), (32459655, 0.002), (181449271, 0.002), (205743718,
0.002), (114244841, 0.002), (129962094, 0.002), (24016384, 0.002),
(77767788, 0.002), (90094370, 0.002), (14090622, 0.002), (171607709,
0.002), (147408008, 0.002), (150151615, 0.002), (224734414, 0.002),
(138721819, 0.002), (154735910, 0.002), (172264633, 0.002))
```

В выводе представлен набор статистических данных после определенного поколения эволюции. Вы можете увидеть следующие результаты:

- максимальный балл вознаграждения, который достигнут во время оценки популяции, составляет 3470 (`PopulationEpRewMax`);
- максимальный балл, полученный лучшими особями в дополнительных 30 эпизодах проверки, составляет 3240 (`TruncatedPopulationRewMean`);
- средний балл оценки лучших особей составляет 2360 (`TruncatedPopulationValidationRewMean`);
- средний балл элитных особей, полученный во время дополнительных 200 прогонов теста, составляет 3060 (`TruncatedPopulationEliteTestRewMean`).

Достигнутые оценки вознаграждений довольно высоки по сравнению с другими методами обучения, если сравнить их с результатами, которые были опубликованы на **https://arxiv.org/abs/1712.06567v3**.

Кроме того, в конце выходных данных вы можете увидеть геномное представление элиты популяции. Мы можем воспользоваться элитным геномом для визуализации прохождения игры Frostbite нейросетью, созданной по этому геному. Далее мы обсудим, как выполнить такую визуализацию.

10.5.3 Визуализация прохождения игры Frostbite

Теперь, когда у нас есть обученный игровой агент, было бы интересно посмотреть, как найденное решение играет в игру Frostbite в среде Atari. Чтобы запустить симуляцию, вам нужно скопировать текущее представление элитного генома из выходных данных и вставить его в поле `seeds` файла `display.py`. После этого можно запустить симуляцию с помощью следующей команды:

```
$ python display.py
```

Эта команда использует предоставленный элитный геном для создания нейросети, управляющей действиями игрового агента Frostbite. Откроется окно игры, где вы сможете воочию наблюдать, как работает нейросеть контроллера. Игра будет продолжаться до тех пор, пока у игрового персонажа не закончатся «жизни». На рис. 10.3 показано несколько скриншотов игровых экранов во время выполнения `display.py` в среде Ubuntu 16.04.

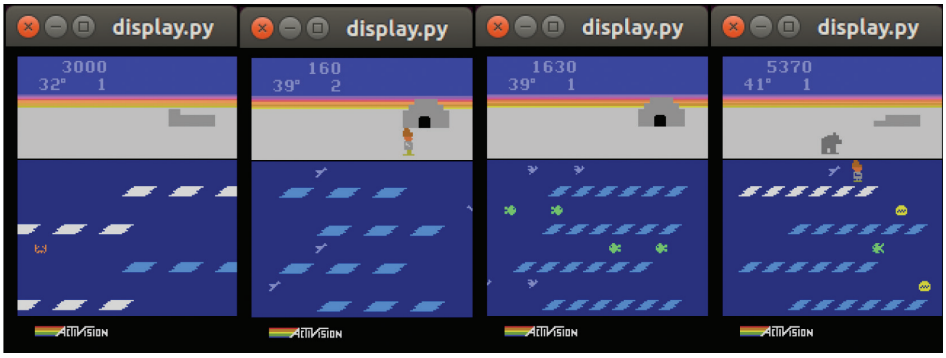


Рис. 10.3. Скриншоты игры Frostbite, полученные во время игрового сеанса с элитным геномом

Удивительно наблюдать, насколько плавный игровой процесс демонстрирует нейросеть, обученная исключительно на основе наблюдений за игровым экраном.

Далее мы обсудим дополнительный метод визуализации, который позволяет нам анализировать результаты.

10.6 Визуальный инспектор нейроэволюции

В процессе нейроэволюции мы развиваем популяцию особей. Каждая особь оценивается в тестовой среде (такой как игра Atari), и для каждой особи на каждом поколении эволюции записываются баллы вознаграждений. Чтобы исследовать общую динамику процесса нейроэволюции, нам нужен инструмент, способный визуализировать облако результатов, достигнутых каждой особью в каждом поколении эволюции. Кроме того, интересно наблюдать за изменениями в оценке приспособленности элитной особи, чтобы понять прогресс процесса эволюции.

Для решения этих задач исследователи из Uber AI разработали инструмент VINE, который мы и обсудим далее.

10.6.1 Настройка рабочей среды

Чтобы воспользоваться инструментом VINE, установите дополнительные библиотеки в виртуальную среду Python с помощью следующих команд:

```
$ pip install click
$ conda install matplotlib
$ pip install colour
$ conda install pandas
```

Эти команды устанавливают все необходимые зависимости в виртуальную среду Python, которую мы создали для нашего эксперимента. Далее мы обсудим, как использовать инструмент VINE для визуализации результатов.

Не забудьте предварительно активировать соответствующую виртуальную среду с помощью следующей команды: `conda activ deep_ne`.

10.6.2 Использование VINE для визуализации эксперимента

После установки зависимостей в виртуальной среде Python мы готовы использовать инструмент VINE. Сначала вам нужно клонировать его из репозитория Git с помощью следующих команд:

```
$ git clone https://github.com/uber-research/deep-neuroevolution.git
$ cd visual_inspector
```

Мы клонировали репозиторий `deep-neuroevolution` в текущий каталог и перешли в папку `visual_inspector`, где находится исходный код инструмента VINE.

Давайте рассмотрим пример того, как можно использовать VINE для визуализации результатов эксперимента по нейроэволюции, для чего воспользуемся результатами эксперимента с человеческой походкой в симуля-

торе Mujoco, предоставленными Uber AI Lab. Более подробную информацию об эксперименте с симулятором Mujoco можно найти по адресу **https://eng.uber.com/deep-neuroevolution/**.

Теперь мы можем запустить визуализацию результатов эксперимента с симулятором Mujoco, которые находятся в папке `sample_data`, с помощью следующей команды:

```
$ python -m main_mujoco 90 99 sample_data/mujoco/final_xy_bc/
```

Эта команда использует те же данные, которые были предоставлены Uber AI Lab из их эксперимента по обучению человекоподобным перемещениям, и выводит на экран визуализацию (рис. 10.4).

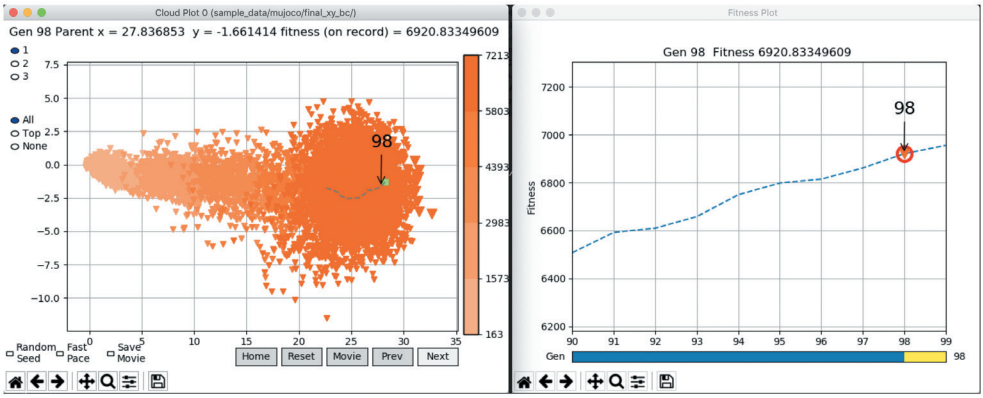


Рис. 10.4. Инструмент VINE для визуализации результатов обучения персонажа человеческой походке

В левой части рис. 10.4 вы можете увидеть облако результатов для каждой особи в популяции, начиная с 90-го поколения и заканчивая 99-м поколением. В правой части графика вы можете увидеть оценку приспособленности элиты популяции по поколениям. На графике справа видно, что эволюционный процесс демонстрирует положительную динамику от поколения к поколению, по мере того как оценка элиты возрастает.

Каждая точка на левом графике демонстрирует точки поведенческой характеристики для каждой особи в популяции. Поведенческая характеристика для задачи обучения человеческой походке – это конечная позиция гуманоидного персонажа в конце траектории. Чем дальше он ушел от исходных координат (0,0), тем выше оценка приспособленности особи. С развитием эволюции облако результатов удаляется от исходных координат. Это движение облака результатов также является сигналом о положительной динамике обучения, потому что каждая особь может сохранять равновесие на ногах в течение более длительного периода.

Для получения более подробной информации об эксперименте Mujoco Humanoid locomotion, пожалуйста, обратитесь к статье на **https://eng.uber.com/deep-neuroevolution/**.

10.7 Упражнения

1. Попробуйте увеличить параметр `pop_size` в настройках эксперимента и посмотрите, что произойдет.
2. Попробуйте вывести результаты эксперимента, которые можно визуализировать с помощью VINE. Для этого вы можете использовать вспомогательные функции `master_extract_parent_ga` и `master_extract_cloud_ga` в скрипте `ga.py`.

10.8 Заключение

В этой главе мы обсудили, как использовать нейроэволюцию для обучения больших нейросетей с более чем 4 млн обучаемых параметров. Вы узнали, как применить данный метод обучения для создания успешных игровых агентов, способных играть в классические игры Atari, изучая правила игры исключительно из наблюдения за игровыми экранами. При подготовке к проведению эксперимента с игрой Atari вы узнали о сверточных нейронных сетях и о том, как их можно использовать для отображения многомерных входных данных, таких как изображения на игровом экране, в соответствующие игровые действия. Теперь у вас есть четкое понимание того, как можно использовать сверточные нейросети для аппроксимации Q-матрицы в методе глубокого обучения с подкреплением, основанном на алгоритме глубокой нейроэволюции.

Обладая знаниями, приобретенными в этой главе, вы сможете применять методы глубокой нейроэволюции для работы с входными данными большого размера, например полученными с камер или других источников изображений.

В следующей главе мы кратко обобщим то, о чем говорилось в этой книге, и дадим несколько советов о том, как вы можете продолжить свое самообразование.

Часть IV

Обсуждение результатов и заключительные замечания

В заключительной части книги представлено краткое изложение того, что вы узнали в предыдущих главах, и рассказано о ресурсах, которые вы можете использовать, чтобы узнать больше об алгоритмах на основе нейроэволюции.

Эта часть состоит из следующих глав:

- главы 11 «Лучшие методы, советы и подсказки»;
- главы 12 «Заключительные замечания».

Глава 11

Лучшие методы, советы и подсказки

В этой главе вы получите несколько советов по практическому использованию передовых методов и узнаете о некоторых приемах разработки и анализа алгоритмов нейроэволюции. К концу главы вы узнаете, как начать работу с прикладной задачей, как настроить гиперпараметры алгоритма нейроэволюции, как использовать расширенные инструменты визуализации и какие метрики можно применять для анализа производительности алгоритма. Кроме того, вы узнаете о лучших приемах кодирования на языке Python, которые помогут вам в реализации ваших проектов.

В данной главе мы рассмотрим следующие темы:

- первичный анализ задачи;
- выбор оптимального метода поисковой оптимизации;
- использование современных инструментов визуализации;
- назначение и правильная настройка гиперпараметров;
- выбор нужных показателей производительности алгоритма;
- программирование на языке Python, советы и рекомендации.

11.1 Первичный анализ задачи

Рецепт вашего успеха – начинать с правильного анализа проблемного пространства. Нейроэволюция прощает многие ошибки программиста. С точки зрения нейроэволюции подобные ошибки являются частью окружающей среды, и процесс эволюции способен адаптироваться к ним. Однако существует определенная категория ошибок, которые могут помешать процессу эволюции найти успешное решение: числовая стабильность процесса эволюции. Большинство функций активации предназначены для работы в интервале входных значений от нуля до единицы. В результате слишком большие или отрицательные значения не оказывают заметного влияния на процесс эволюции.

Таким образом, возможно, вам придется предварительно обработать входные данные, чтобы избежать проблемы числовой стабильности. Не пропускайте этапы анализа и предварительной обработки данных.

Далее мы обсудим предварительную обработку входных данных.

11.1.1 Предварительная обработка данных

Всегда исследуйте возможный диапазон входных данных и проверяйте наличие выбросов. Если вы обнаружите, что масштаб одного входного параметра отличается от другого на порядок, вам необходимо предварительно обработать выборки входных данных. В противном случае *признаки* (features)[1], представленные во входных данных с большей величиной, окажут настолько значительное влияние на процесс обучения, что в конечном итоге перекроют вклад других признаков. Однако небольшие сигналы, генерируемые входными данными с малой величиной, часто имеют решающее значение для поиска успешного решения. Слабые входные сигналы могут характеризовать тонкие, но ценные нюансы базового процесса обучения.

Стандартизация данных

Большинство алгоритмов машинного обучения значительно выигрывают от входных данных, которые имеют нормальное распределение; то есть среднее значение и единичная дисперсия равны нулю. Такой подход называется *стандартизацией* данных. В общем виде стандартизация входных данных для получения нулевого среднего значения и единичной дисперсии выражается следующей формулой:

$$z = \frac{x - u}{s}.$$

Здесь z – это стандартизованная входная оценка, x – выборка входных данных, u – среднее значение обучающих выборок и s – стандартное отклонение обучающих выборок.

Вы можете использовать библиотеку Python Scikit-learn, чтобы применить стандартизацию к вашим образцам входных данных. Следующий исходный код демонстрирует практический пример стандартизации:

```
>>> from sklearn.preprocessing import StandardScaler
>>> data = [[0, 0], [0, 0], [1, 1], [1, 1]]
>>> scaler = StandardScaler()
>>> print(scaler.fit(data))
StandardScaler(copy=True, with_mean=True, with_std=True)
>>> print(scaler.mean_)
[0.5 0.5]
>>> print(scaler.transform(data))
[[-1. -1.]
```

[1] Здесь автор внезапно упоминает понятие *признаков* (features), никак его не объясняя. В машинном обучении под признаками понимают такие данные из общего массива больших данных (big data), которые имеют значение для обучения модели в предметной области. Например, в предсказании погоды значащими признаками будут влажность, атмосферное давление, температура воздуха и т. д. В общем случае признаки имеют разную значимость (вес) для модели. Составление перечня признаков и присвоение им индивидуального веса называется *извлечением*, или *конструированием*, *признаков* (feature engineering) и обычно выполняется разработчиком вручную на этапе анализа предметной области, хотя, как упоминалось в этой книге, нейроэволюция иногда способна самостоятельно оценить важность признаков. – *Прим. перев.*

```
[-1. -1.]
[ 1. 1.]
[ 1. 1.]]
```

В этом коде мы сначала создаем выборки входных данных. После этого для центрирования и масштабирования входных выборок используется класс `StandardScaler`. Результаты преобразования данных показаны в последних строках листинга.

Другой метод предварительной обработки данных – это масштабирование входных данных для приведения к определенному интервалу.

Приведение входных данных к заданному интервалу

Масштабирование входных данных, то есть приведение их к заданному интервалу значений, является еще одним методом предварительной обработки данных. Этот метод является альтернативой стандартизации. В результате масштабирования получаются выборки данных, которые лежат строго в интервале между минимальным и максимальным значениями. Часто этот метод используется для масштабирования входных данных в интервале от нуля до единицы. Для масштабирования данных до нужного интервала вы можете использовать `MinMaxScaler` из библиотеки Python Scikit-learn, как показано в следующем примере:

```
>>> import sklearn.preprocessing
>>> X_train = np.array([[ 1., -1., 2.],
... [ 2., 0., 0.],
... [ 0., 1., -1.]])
...
>>> min_max_scaler = preprocessing.MinMaxScaler()
>>> X_train_minmax = min_max_scaler.fit_transform(X_train)
>>> X_train_minmax
array([[0.5 , 0. , 1. ],
       [1. , 0.5 , 0.33333333],
       [0. , 1. , 0. ]])
```

Пример начинается с создания выборки данных и преобразования с помощью класса `MinMaxScaler`. В последних строках листинга вы можете видеть масштабированные данные.

Иногда вам нужно иметь образцы данных с одинаковыми единицами измерения. Этот метод предварительной обработки называется *нормализацией*. Мы обсудим его в следующем разделе.

Нормализация данных

Зачастую ваши входные данные имеют разные единицы измерения. Например, в эксперименте по балансировке обратного маятника положение тележки измерялось в метрах, линейная скорость выражалась в метрах в секунду, а угловая скорость – в радианах в секунду. Полезно нормализовать входные данные, чтобы упростить сравнение признаков, представленных во входных данных.

Процесс *нормализации* эффективно исключает единицы измерения из выборок входных данных. После этого все выборки будут находиться в интервале от нуля до единицы.

В статистике существуют разные типы нормализации. Мы уже упоминали два метода: стандартизацию и масштабирование. Кроме того, Scikit-learn предоставляет специальный преобразователь для нормализации данных, который масштабирует отдельные выборки до так называемой единичной нормы. Следующий код демонстрирует, как его использовать:

```
>>> import sklearn.preprocessing
>>> X = [[ 1., -1., 2.],
... [ 2., 0., 0.],
... [ 0., 1., -1.]]
>>> X_normalized = preprocessing.normalize(X, norm='l2')
>>> X_normalized
array([[ 0.40..., -0.40..., 0.81...],
       [ 1. ..., 0. ..., 0. ...],
       [ 0. ..., 0.70..., -0.70...]])
```

Код создает образец тестовых данных и применяет к нему нормализацию с использованием нормы `l2` и выводит результаты.

Библиотека Scikit-learn обеспечивает реализацию многих других методов предварительной обработки данных. Вам было бы полезно познакомиться с ними. Вы можете найти отличный учебник по адресу **https://scikit-learn.org/stable/modules/preprocessing.html**.

11.1.2 Исследование проблемной области

В этой книге некоторые из экспериментов, которые мы обсуждали, были связаны с реальными процессами в физическом мире. Чтобы найти успешные решения для таких процессов, вам необходимо хорошо понимать основные физические законы и принципы. Например, задача балансировки обратного маятника на тележке требует, чтобы мы определили полный набор уравнений движения и написали точный симулятор задачи.

Также для большинства задач в области робототехники вам необходимо написать симулятор, который использует корректную физическую модель и уравнения базового аппарата. Вам необходимо полностью понять физику процесса, чтобы правильно реализовать симулятор. И даже если вы используете готовый симулятор, понимание физических принципов, реализованных в нем, чрезвычайно полезно для вас, потому что понимание динамики реального процесса позволяет вам настроить гиперпараметры алгоритма обучения соответствующим образом.

11.1.3 Написание хороших симуляторов

При работе над конкретной задачей крайне важно написать качественный симулятор, который правильно реализует особенности моделируемого процесса. Если вы используете такой симулятор, то сможете запускать длинные эпизоды обучения, что невозможно при использовании прямого ввода с физических устройств.

Хороший симулятор должен позволять вам контролировать продолжительность одного временно́го шага моделируемого процесса. Во время про-

гона нейроэволюции вам необходимо оценить каждую особь в популяции по отношению к данному симулятору. Таким образом, во время обучения имеет смысл сделать длительность одного временного шага как можно меньше, чтобы увеличить скорость выполнения. С другой стороны, когда решение найдено и вам необходимо проверить его вручную, было бы полезно иметь возможность запускать симулятор с нормальной скоростью выполнения.

Кроме того, вы можете использовать для своих проектов готовые тщательно проработанные симуляторы, что может сэкономить вам много времени. Ознакомьтесь с хорошо известными пакетами симуляторов с открытым исходным кодом. Они часто предоставляют расширенные физические модели, а также набор готовых строительных блоков для ваших виртуальных роботов и сред. Вы можете начать поиск по адресу **https://github.com/cyberbotics/webots**.

Далее мы обсудим, как выбрать правильный метод оптимизации поиска для вашего эксперимента.

11.2 Выбор оптимального метода поисковой оптимизации

В этой книге мы представили вам два основных метода поиска оптимального решения: поиск по близости к цели (целеориентированный) и поиск новизны. Первый способ более прост в реализации и проще для понимания. Однако поиск новизны удобен в тех случаях, когда функция приспособленности имеет обманчивый ландшафт со многими локальными ловушками оптимальности.

В следующем разделе мы кратко обсудим оба метода, чтобы напомнить вам о деталях и помочь выбрать, какой из них использовать в вашей ситуации. Начнем с целеориентированного поиска.

11.2.1 Целеориентированный поиск оптимального решения

Целеориентированный поиск оптимального решения основан на измерении близости решения к конечной цели. Чтобы вычислить среднее расстояние до цели, он часто использует такую метрику, как среднеквадратичная ошибка. Далее мы обсудим особенности метрики среднеквадратичной ошибки.

Среднеквадратичная ошибка

Среднеквадратичная ошибка – это среднеквадратичная разница между полученными результатами и фактическими значениями, которая вычисляется по следующей формуле:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \overline{y}_i\right)^2.$$

Здесь y_i является предсказанным значением, а $\overline{y}_i$ является фактическим значением.

Мы использовали изменение среднеквадратичной ошибки, чтобы определить целевую функцию для эксперимента XOR. Далее обсудим целеориентированные метрики для задач, связанных с позиционированием в евклидовом пространстве.

Евклидово расстояние

Евклидово расстояние является удобной метрикой для задач, связанных с навигацией в евклидовом пространстве. В евклидовом пространстве мы определяем цель задачи как точку с определенными координатами.

Используя евклидово расстояние, легко рассчитать расстояние между положением навигационного агента и целевой точкой, которую он пытается достичь. Следующая формула вычисляет евклидово расстояние между двумя векторами:

$$\mathcal{D} = \sqrt{\sum_{i=1}^{2} \left(a_i - b_i\right)^2}.$$

Здесь $\mathcal{D}$ – евклидово расстояние между вектором положения агента a_i и вектором конечной цели b_i. Мы использовали эту метрику для определения целевой функции агента, проходящего через лабиринт, в главе 5.

Однако главная проблема автономного прохождения лабиринта заключается в обманчивом ландшафте функции приспособленности, что делает целеориентированную оптимизацию неэффективной. Далее мы обсудим метод оптимизации поиском новизны, который способен устранить эту неэффективность.

11.2.2 Оптимизация поиском новизны

Как мы уже упоминали, прохождение лабиринта является обманчивой задачей, которая требует использования особой функции приспособленности. В главе 5 мы представили вам конкретную конфигурацию лабиринта, в которой существуют области с выраженными локальными оптимумами целеориентированной функции приспособленности. В результате процесс обучения может оказаться в ловушке внутри этих областей и не сможет прийти к успешному решению. Для решения проблем с обманчивыми ландшафтами функций приспособленности был разработан метод оптимизации поиском новизны.

Поиск новизны вознаграждает за новизну решения, а не за его близость к конечной цели. Кроме того, метрика новизны, которая используется для расчета оценки приспособленности каждого решения, полностью игнорирует близость решения к конечной цели. Существует два популярных подхода для расчета оценки новизны:

- исходя из различий в архитектуре решений;
- на основе уникальных вариаций поведения решений в общем поведенческом пространстве.

Первый подход рассчитывает разницу между кодировкой текущего решения и всех предыдущих решений. Второй подход сравнивает результат, полученный текущим решением в поведенческом пространстве, с результатами, полученными другими решениями.

Для вычисления приспособленности найденных решений мы использовали оценки новизны, основанные на уникальности проявленного поведения. Траектория движения через лабиринт полностью определяет поведенческое пространство агента и может использоваться для расчета оценки новизны. В данном случае оценка новизны – это евклидово расстояние между векторами траекторий текущего решения и всех остальных решений.

Теперь, когда мы обсудили важность выбора подходящего метода оптимизации поиска, мы можем перейти к обсуждению еще одного важного аспекта успешного эксперимента. Вы должны иметь хорошую визуализацию результатов эксперимента, чтобы получить представление о его результатах. Далее мы обсудим визуализацию результатов.

11.3 Качественная визуализация

Почти всегда правильная визуализация входных данных и результатов имеет решающее значение для успеха вашего эксперимента. Благодаря правильной визуализации вы получите наглядное представление о том, что пошло не так и что нужно исправить.

Всегда старайтесь визуализировать процесс работы симулятора. Такая визуализация может сэкономить вам часы отладки, когда вы получите неожиданный результат. Обычно при надлежащей визуализации вы можете сразу увидеть, что что-то пошло не так – например, агент-решатель лабиринта застрял в углу.

При использовании алгоритмов нейроэволюции вам также необходимо визуализировать ход выполнения генетического алгоритма для каждого поколения. Вы должны визуализировать видообразование от поколения к поколению, чтобы увидеть, не застыл ли эволюционный процесс в одном состоянии. Застоявшаяся эволюция не в состоянии создать достаточно видов для поддержания здорового разнообразия среди решателей задачи. С другой стороны, чрезмерное разнообразие видов препятствует эволюции, уменьшая шансы спаривания между различными организмами.

Еще одна важная визуализация позволяет нам увидеть топологию фенотипа искусственной нейронной сети. Полезно визуально проверить топологию полученного решения, чтобы проверить, соответствует ли она вашим ожиданиям. Например, когда мы обсуждали проблему модульной сетчатки в главе 8, было полезно убедиться, что модульные структуры эволюционировали в рациональные топологии успешных решений.

Вам необходимо ознакомиться со стандартными научными библиотеками построения графиков на языке Python, чтобы создавать адекватную визуализацию для результатов ваших экспериментов. Следует развивать практические навыки работы с такими библиотеками визуализации, как Matplotlib (**https://matplotlib.org**) и Seaborn (**https://seaborn.pydata.org**).

Далее мы обсудим важность настройки гиперпараметров перед запуском процесса нейроэволюции.

11.4 Настройка гиперпараметров

При правильной настройке гиперпараметров вы можете значительно повысить скорость обучения и эффективность процесса нейроэволюции. Вот несколько практических советов:

- сделайте короткие прогоны с различными начальными значениями генератора случайных чисел и обратите внимание, как изменяется качество работы алгоритма. После этого выберите начальное значение, которое дает наилучший результат, и используйте его для длительных прогонов;

- вы можете увеличить количество видов в популяции, уменьшив порог совместимости и немного увеличив значение весового коэффициента пересечения/несовпадения генов;
- если процесс нейроэволюции споткнулся при попытке найти решение, попытайтесь уменьшить значение порога выживания NEAT. Этот коэффициент определяет долю лучших организмов в популяции, которые получили возможность размножаться. Уменьшая коэффициент, вы повышаете качество особей, которым разрешено воспроизводить потомство в зависимости от их приспособленности;
- увеличив максимальный возраст стагнации, вы можете гарантировать, что виды проживут достаточно долго, чтобы иметь возможность получить полезные мутации на более поздних стадиях эволюции. Иногда такая операция помогает подтолкнуть остановившийся процесс нейроэволюции. Тем не менее вы всегда должны начинать с небольших значений допустимой стагнации (15–20 поколений), чтобы инициировать быстрое обновление видов, и значительно увеличивать этот параметр только в случае неудачи всех других настроек;
- после настройки гиперпараметров выполните прогон эволюции на нескольких десятках поколений, чтобы увидеть динамику изменения приспособленности. Обратите особое внимание на количество видов – в популяции должно быть не менее одного вида. Слишком много видов – это тоже плохой знак. Хорошо, если видовое разнообразие варьируется в интервале от 5 до 20 видов;
- используйте визуализацию, чтобы быстро получить наглядное представление о результатах эксперимента. Никогда не упускайте возможность визуализировать топологию нейросетей обнаруженных решений. Эти визуализации могут дать вам бесценное понимание того, как настроить процесс нейроэволюции;
- не тратьте свое время на длинные эволюционные прогоны. Если в ходе эксперимента не удается найти успешное решение через 1000 поколений, есть большая вероятность, что что-то не так с вашим кодом или библиотекой, которую вы используете. Для большинства простых проблем удачное решение может быть найдено всего через 100 поколений;
- численность популяции является критическим параметром эволюционного процесса. С большой популяцией вы с самого начала получаете большое разнообразие особей, что ускоряет процесс эволюции. Тем не менее большая популяция увеличивает вычислительную нагрузку. Поэтому всегда приходится искать компромисс между численностью популяции и вычислительными затратами. Как правило, если вам трудно найти другие подходящие гиперпараметры, попробуйте увеличить размер популяции и посмотреть, поможет ли это. Но будьте готовы дольше ждать, пока завершится процесс нейроэволюции;
- всегда распечатывайте отладочную информацию и сохраняйте промежуточные точки, что позволит вам перезапустить эксперимент с любого этапа вычислений. Всегда больно, когда вы находите решение после двух дней вычислений, но из-за нелепой ошибки в коде происходит

сбой вашей программы при попытке вывести сообщение с поздравлением. Вам нужно вывести, по крайней мере, начальное значение генератора случайных чисел в начале каждого испытания. Это поможет вам точно воссоздать все поколения эволюции в случае неудачи.

Не стоит недооценивать важность настройки гиперпараметров. Даже принимая во внимание, что процесс нейроэволюции может справиться со многими ошибками программирования, выбор правильных гиперпараметров может значительно повысить производительность процесса. В результате вы сможете найти успешное решение за сотни поколений, а не за тысячи или более.

Чтобы сравнить успешность различных решений, вам нужно использовать соответствующие метрики качества, которые мы обсудим далее.

11.5 Метрики качества модели

После того как найдено успешное решение, важно сравнить его с другими решениями, чтобы оценить, насколько оно хорошее. Есть много важных статистических показателей, относительно которых сравнивают разные модели.

Ознакомьтесь с такими понятиями, как оценка точности, оценка отзыва, оценка F1, ROC AUC и достоверность. Понимание этих метрик поможет вам сравнить результаты, полученные различными моделями в различных задачах классификации. Далее мы даем краткий обзор этих метрик.

11.5.1 Точность

Точность (precision) отвечает на вопрос о том, какая доля положительных ответов соответствует истине. Оценка точности может быть рассчитана следующим образом:

$$precision = \frac{TP}{TP + FP}.$$

TP (true positive) – это истинно положительные ответы, а *FP* (false positive) – ложноположительные ответы.

11.5.2 Отклик

Отклик (recall) отвечает на вопрос о том, какая доля от фактически положительных результатов была определена правильно. Оценка отклика вычисляется по следующей формуле:

$$recall = \frac{TP}{TP + FN}.$$

TP – это истинно положительные результаты, а *FN* (false negative) – ложноотрицательные.

11.5.3 Оценка F1

Оценка F1 – это средневзвешенное значение между оценками точности и отклика. Наилучшее значение в баллах F1 – один, а худшее – ноль. Оценка F1 по-

зволяет измерить точность классификации, характерную для определенного класса, и определяется так:

$$F_1 = 2\frac{precision \times recall}{precision + recall}.$$

Здесь *precision* – оценка точности, а *recall* – оценка отклика относительно определенного положительного класса.

В следующем разделе мы рассмотрим кривую *рабочей характеристики приемника* (receiver operating characteristic, ROC) и *площадь под кривой* (area under curve, AUC).

11.5.4 ROC AUC

Мы строим кривую ROC, нанося на график точки в координатах истинно положительных и ложноположительных показателей при разных пороговых значениях. Кривая показывает точность классифицирующей модели при различных пороговых значениях.

Коэффициент истинно положительных результатов (true positive rate, TPR) является синонимом отклика, о котором мы говорили ранее. Он вычисляется по формуле:

$$TPR = \frac{TP}{TP + FN}.$$

Коэффициент ложноположительных результатов (false positive rate, FPR) рассчитывается следующим образом:

$$FPR = \frac{FP}{FP + TN}.$$

Здесь *TN* – это истинно отрицательные результаты.

AUC позволяет нам оценить *различающую способность* (discrimination power) классифицирующей модели, то есть способность модели правильно присваивать более высокую оценку случайным положительным точкам по сравнению со случайными отрицательными точками.

На рис. 11.1 показан пример кривой ROC. Чем больше AUC (площадь под кривой), тем точнее модель классификатора. Пунктирная линия показывает наихудшую точность классификатора. Как правило, чем ближе кривая ROC к верхнему левому углу, тем выше качество классифицирующей модели.

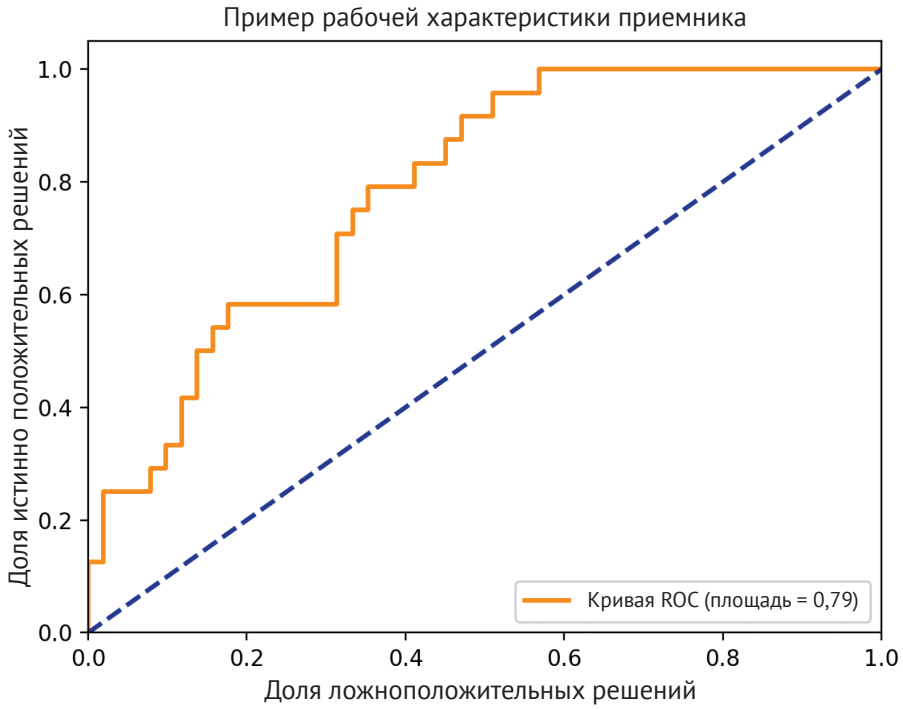


Рис. 11.1. Пример кривой ROC

11.5.5 Достоверность

Достоверность (accuracy) – это показатель, сообщающий, сколько правильных предсказаний смогла сделать наша модель. Достоверность определяется по следующей формуле:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}.$$

Здесь *FP* – ложноположительные срабатывания, а *FN* – ложноотрицательные.

Более подробную информацию можно найти по адресу **https://scikitlearn.org/stable/auto_examples/model_selection/plot_precision_recall.html**.

Далее вы найдете советы по программированию на языке Python.

11.6 Python, кодирование, советы и рекомендации

Решив работать с Python, важно изучить лучшие практики использования языка. В этом разделе я дам вам несколько советов и рекомендаций для самостоятельного обучения.

11.6.1 Советы и рекомендации

Следующие советы и рекомендации помогут вам освоить программирование на языке Python:

- научитесь пользоваться популярными библиотеками машинного обучения, такими как NumPy (**https://numpy.org**), pandas (**https://pandas.pydata.org**) и Scikitlearn (**https://scikit-learn.org/stable**). Освоение этих библиотек откроет перед вами огромные возможности в обработке и анализе данных. Навык работы с этими библиотеками поможет вам избежать многих ошибок и позволит легко отлаживать программы, исходя из результатов экспериментов;
- изучите объектно-ориентированную парадигму программирования. Это позволит вам писать чистый и понятный исходный код, который легко разобрать. Вы можете начать с учебника **https://www.datacamp.com/community/tutorials/python-oop-tutorial**;
- не сваливайте весь код в одну огромную функцию. Разбейте ваш код на более мелкие блоки многократного использования, реализованные в виде функций или классов, которые можно повторно использовать в нескольких проектах и легко отлаживать;
- всегда выводите на печать промежуточные и отладочные данные, чтобы понимать, что происходит в вашей программе. Обладая достаточным количеством отладочных данных, намного проще понять, что не так с выполнением программы;
- пишите комментарии, относящиеся к функциям, классам и сложным местам в вашем исходном коде. Хорошие комментарии значительно облегчают понимание кода. Написание комментариев по ходу реализации алгоритма также поможет прояснить ваши собственные мысли;
- при написании комментариев к функции опишите все входные и выходные параметры и их значения по умолчанию, если таковые имеются;
- если вы решили продолжить работу с Python, потратьте некоторое время на изучение стандартных библиотек. Python – это зрелый язык программирования со множеством служебных функций, встроенных в его стандартные библиотеки. У него также есть много функций, реализующих продвинутые манипуляции с данными, которые могут пригодиться в задачах машинного обучения. Более подробную информацию о стандартных библиотеках Python можно найти по адресу **https://docs.python.org/3/library/index.html**;
- следуйте стандартным соглашениям исходного кода Python при присвоении имен переменным и классам. Следование стандартным соглашениям об именах делает ваш код более читабельным и понятным для всех, кто имеет опыт работы с Python. Вы можете найти более подробную информацию по адресу **https://docs.pythonguide.org/writing/style/** и **https://www.python.org/dev/peps/pep-0008/**;
- ознакомьтесь с современными системами контроля версий, такими как Git. *Система контроля версий* (version control system, VCS) – это мощный инструмент, который может сэкономить вам часы и даже дни попыток восстановить утраченную работу, вызванную поломкой жестко-

го диска. Вы можете узнать о Git по адресу **https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf** и **https://www.atlassian.com/git/tutorials**;

- используйте онлайн-хранилища кода, такие как GitHub (**https://github.com**) и Bitbucket (**https://bitbucket.org**), где вы можете поделиться своим исходным кодом и изучить исходный код других исследователей данных.

Другим важным условием написания хороших программ является правильная настройка рабочей среды и использование подходящих инструментов программирования.

11.6.2 Рабочая среда и инструменты программирования

Для правильной настройки рабочей среды всегда полезно использовать один из развитых менеджеров пакетов Python, например Anaconda Distribution. В качестве дополнительного преимущества вы получите множество бесплатных пакетов для научного и машинного обучения, которые готовы к установке одной командой. Кроме того, Anaconda Distribution управляет всеми косвенными зависимостями и помогает поддерживать все ваши пакеты в актуальном состоянии. Вы можете найти Anaconda Distribution по адресу **https://www.anaconda.com/distribution/**.

Всегда создавайте новую виртуальную среду Python для каждого из ваших экспериментов. После этого, если что-то пойдет не так с зависимостями, вы сможете очистить все одной командой и начать с нуля. Новая среда Python может быть создана с помощью Anaconda Distribution следующим образом:

```
$ conda create --name <name>
$ conda activate <name>
```

При создании новой среды всегда указывайте точную версию Python, которую вы планируете использовать. Упоминание точной версии поможет вам избежать многих неприятностей, вызванных несовместимостью. Версия Python для новой среды может быть определена следующим образом:

```
$ conda create --name <name> python=3.5
```

Если вам нужно использовать новую зависимость в вашем проекте, сначала убедитесь, что в Anaconda Cloud существует соответствующий установочный пакет. Используя библиотеки из Anaconda Cloud, вы можете избежать проблем с установкой косвенных зависимостей. Кроме того, некоторые платформы, такие как TensorFlow, требуют установки дополнительных системных драйверов и заголовочных файлов. Эта задача может оказаться очень громоздкой и потребовать дополнительных навыков.

Используйте хороший редактор кода, который поддерживает автозавершение ввода, просмотр документации и поддержку виртуальных сред Python. Удачный вариант для начала – бесплатный редактор Visual Studio Code, предоставляемый Microsoft. Вы можете найти его на **https://code.visualstudio.com**.

Познакомьтесь с современными операционными системами Linux, такими как Ubuntu. Большинство библиотек машинного обучения намного проще использовать с Linux. Это особенно верно для библиотек, которые

используют графические ускорители. Более подробную информацию об Ubuntu и установке этой операционной системы можно найти по адресу **https://ubuntu.com**.

11.7 Заключение

В этой главе вы получили практические советы, которые, как я надеюсь, облегчат вашу жизнь. Вы узнали о стандартных методах предварительной обработки данных и о популярных статистических показателях, которые можно использовать для оценки производительности созданных вами моделей. Наконец, вы узнали, как улучшить свои навыки программирования и где искать дополнительную информацию по темам Python и машинного обучения.

В следующей главе приведены заключительные замечания, основанные на том, что вы узнали в книге, и соображения о том, где вы можете применить полученные знания в будущем.

Глава 12

Заключительные замечания

Настало время кратко обобщить все, что вы узнали в этой книге, и предоставить дополнительную информацию, чтобы вы могли продолжить свое самообразование. Эта глава поможет вам объединить темы, которые мы рассмотрели в формате отдельных глав, а затем подскажет направление дальнейшего развития, поделившись некоторыми подробностями об Uber AI Labs, alife.org и открытой эволюции в Reddit. Вы также найдете здесь краткий обзор каталога программного обеспечения NEAT и статьи про алгоритм NEAT.

В этой главе мы рассмотрим следующие темы:

- что вы узнали в этой книге;
- в каком направлении развиваться дальше.

12.1 Что вы узнали в этой книге

Я надеюсь, что благодаря экспериментам и упражнениям вы получили четкое представление о нейроэволюционном методе обучения искусственных нейронных сетей. Мы использовали нейроэволюцию, чтобы найти решения различных задач, от классических проблем информатики до создания агентов, способных играть в игры Atari. Мы также рассмотрели задачи, связанные с компьютерным зрением и зрительным различением.

12.1.1 Обзор методов нейроэволюции

В первой главе вы узнали об основных понятиях генетических алгоритмов, таких как генетические операторы и схемы кодирования генома.

Мы обсудили два основных генетических оператора, которые позволяют нам поддерживать эволюционный процесс:

- *оператор мутации* – реализует случайные мутации потомства, что вносит генетическое разнообразие в популяцию;
- *оператор кроссовера* – генерирует потомство путем отбора генов от каждого родителя.

После этого мы продолжили дискуссию о важности выбора правильной схемы кодирования генома. Мы рассмотрели два основных формата кодирования: *прямое* и к*освенное* кодирования генома. Первый формат вводит взаимно-однозначное соотношение между геномом и закодированным фенотипом нейросети. Обычно прямое кодирование применяется для кодирования

небольших нейросетей, которые имеют ограниченное количество подключенных узлов. Более продвинутая схема косвенного кодирования позволяет нам кодировать развивающуюся топологию крупномасштабных нейросетей, часто с миллионами связей. Косвенное кодирование дает возможность повторно использовать повторяющиеся паттерны кодирования, тем самым значительно уменьшая размер генома.

Ознакомившись с основными схемами кодирования генома, мы приступили к обсуждению метода нейроэволюции, который использует различные схемы кодирования. Мы начали с введения в алгоритм NEAT, который использует схему прямого кодирования генома и дополняет ее концепцией числа инноваций. Число инноваций, связанное с каждым геном генотипа, позволяет точно отслеживать, когда была введена конкретная мутация. Эта функция делает операции кроссовера между двумя родителями простыми и легкими в реализации. Метод NEAT подчеркивает важность запуска эволюции с очень простого генома, который постепенно становится более сложным. Таким образом, эволюционный процесс имеет прекрасную возможность найти оптимальное решение.

Кроме того, была введена концепция *видообразования*, которая сохраняет полезные мутации, выделяя их у определенных видов (ниш). Видам в пределах одной ниши разрешено скрещиваться только друг с другом. Видообразование является великой движущей силой естественной эволюции, и было показано, что оно также оказывает большое влияние на нейроэволюцию.

Обсудив основной алгоритм NEAT, мы приступили к изучению его расширений, устраняющих ограничения исходного алгоритма. Один из существенных недостатков алгоритма NEAT вызван использованием схемы прямого кодирования генома. Эта схема, хотя и проста для визуализации и реализации, кодирует только топологии небольших нейросетей. С увеличением размера фенотипа нейросети размер генома увеличивается в линейной пропорции. Это увеличение размера генома в конечном итоге затрудняет реализацию алгоритма. Для устранения этого недостатка был представлен ряд расширений, основанных на схемах косвенного кодирования генома, таких как HyperNEAT и ES-HyperNEAT.

Метод HyperNEAT использует расширенный формат для представления связей между узлами фенотипа нейросети в виде четырехмерных точек в гиперкубе. Размерность выбранного гиперкуба основана на том факте, что связи между двумя узлами в нейросети могут быть закодированы координатами конечных точек соединения в среде, называемой субстратом. Топология субстрата обусловливает структуру, которая образует связи между узлами фенотипа нейросети. Вес связи, которая устанавливается между двумя конкретными узлами на субстрате, оценивается вспомогательной нейронной сетью, известной как *сеть, производящая составные паттерны* (compositional pattern producing network, CPPN). CPPN получает координаты *гиперточки* (координаты конечных точек связи) в качестве входных данных и возвращает вес связи. Кроме того, она вычисляет значение флага, который указывает, должно соединение быть экспрессировано или нет. Экспериментатор заранее определяет конфигурацию субстрата. Она определяется геометрическими свойствами решаемой задачи. В то же время топология CPPN развивается в процессе нейроэволюции с использованием алгоритма NEAT. Таким образом, мы получа-

ем лучшее из обоих миров. Мощь алгоритма NEAT позволяет нам развивать оптимальные конфигурации CPPN. В то же время CPPN поддерживает схему непрямого кодирования и позволяет представлять масштабные нейросети.

Метод ES-HyperNEAT является продолжением развития оригинальных методов NEAT и HyperNEAT и добавляет возможность развивать субстрат по мере эволюции CPPN. Развитие субстрата основано на понятии *плотности информации*, что позволяет более плотно размещать узлы в областях с большей изменчивостью информации. Этот подход позволяет процессу нейроэволюции находить конфигурации субстрата, которые точно следуют геометрическим закономерностям, скрытым в решаемой задаче.

Мы завершили первую главу обсуждением увлекательного метода поисковой оптимизации, известного как *поиск новизны* (novelty search, NS). В основе этого метода управления эволюционным поиском лежит метрика новизны найденных решений. Традиционно поисковая оптимизация основана на целеориентированных критериях пригодности, которые измеряют, насколько мы близки к цели. Но есть целый ряд реальных задач, которые имеют обманчивые ландшафты функций приспособленности с глубокими ловушками локальных оптимумов. Целеориентированный поиск имеет большой шанс застрять в одной из этих ловушек и не найти окончательного решения. В то же время метод поисковой оптимизации, поощряющий новизну найденного решения, позволяет нам избежать этих ловушек, полностью игнорируя близость к конечной цели. Было показано, что метод поиска новизны эффективен в задачах автономного прохождения лабиринта с ловушками оптимумов и заметно превосходит целеориентированные методы.

В следующей главе говорилось о том, как правильно настроить рабочую среду и какие библиотеки Python можно использовать для экспериментов с нейроэволюцией.

12.1.2 Библиотеки Python и настройка среды разработки

Во второй главе мы начали с обсуждения практических аспектов нейроэволюции. Вы узнали про достоинства и недостатки популярных библиотек Python, которые предоставляют реализации алгоритма NEAT и его расширений.

Наряду с обсуждением каждой библиотеки Python вы рассмотрели небольшие фрагменты кода, дающие вам представление о том, как использовать каждую конкретную библиотеку в предстоящих экспериментах.

После этого мы приступили к обсуждению правильной настройки рабочей среды. В рабочей среде должны быть установлены необходимые зависимости, позволяющие использовать упомянутые ранее библиотеки Python. Установка может быть выполнена несколькими способами. Мы рассмотрели два наиболее распространенных из них – стандартный установщик пакетов для утилиты Python (PIP) и Anaconda Distribution. Другим важным аспектом подготовки рабочей среды является создание *изолированных виртуальных сред* Python для каждого конкретного эксперимента. Виртуальные среды допускают наличие различных конфигураций зависимостей для разных экспериментов и используемых в них библиотек NEAT.

Размещение зависимостей в виртуальной среде также позволяет легко управлять всеми установленными зависимостями в целом. Среда может быть

быстро удалена с вашего ПК вместе со всеми установленными файлами, освобождая тем самым место на диске. Вы также можете повторно использовать определенную виртуальную среду для различных экспериментов, которые зависят от той же библиотеки реализации NEAT.

Вы познакомились со всеми необходимыми инструментами, чтобы начать эксперименты с нейроэволюцией. В следующей главе мы приступили к обсуждению эксперимента с решателем задачи XOR на основе базового алгоритма NEAT.

12.1.3 Использование NEAT для оптимизации решения задачи XOR

Это была первая глава, в которой мы начали экспериментировать с алгоритмом NEAT. Мы сделали это, реализовав решатель одной из классических задач информатики. Мы начали с создания решателя для задачи XOR. Решатель задачи XOR – это компьютерный эксперимент в области обучения с подкреплением. Проблема XOR не может быть линейно разделена и, следовательно, нуждается в решателе, предлагающем нелинейный путь выполнения. Однако мы можем ввести нелинейность, внедряя скрытые слои в структуру нейросети.

Вы убедились, что алгоритм NEAT идеально справляется с этой задачей благодаря присущей ему способности развивать нейросеть из очень простой или сложной топологии путем постепенной оптимизации. В эксперименте XOR мы начали с простой топологии нейросети, которая состояла из двух входных узлов и одного выходного узла. В ходе эксперимента была найдена подходящая топология нейросети решателя, и алгоритм ввел дополнительный скрытый узел, представляющий нелинейность, как мы и ожидали.

Кроме того, вы узнали, как определить подходящую функцию приспособленности для управления эволюционным поиском и реализовать ее в скрипте Python. Мы уделили большое внимание описанию гиперпараметров, которые точно настраивают характеристики библиотеки NEAT для эксперимента XOR.

В этой главе вы приобрели навыки, необходимые для реализации базовых решателей, и приготовились перейти к более сложным экспериментам.

12.1.4 Балансировка тележки с обратным маятником

В четвертой главе вы продолжили эксперименты, связанные с классическими задачами информатики в области обучения с подкреплением. Глава началась с обсуждения того, как при помощи алгоритма NEAT оптимизировать нейросеть контроллера балансировки тележки с обратным маятником. Вы начали с балансировки одиночного маятника и рассмотрели все необходимые уравнения движения, которые позволяют численно аппроксимировать физическое устройство реального мира.

Вы узнали, как выполняется балансировка тележки при помощи двухпозиционного регулятора. *Двухпозиционный регулятор* – это уникальная система управления, которая прикладывает к объекту управляющие воздействия всегда с одинаковой силой, но в одном из двух противоположных направлений. Для управления двухпозиционным регулятором нейросеть должна постоянно считывать и анализировать состояние тележки с обратным ма-

ятником и генерировать соответствующие управляющие сигналы. Входные сигналы системы определяются горизонтальным положением тележки на дорожке, ее линейной скоростью, текущим углом наклона маятника и угловой скоростью маятника. Выход системы представляет собой двоичный сигнал, указывающий направление воздействия, прикладываемого к тележке.

Процесс нейроэволюции использует симулятор тележки с маятником для реализации поиска решения методом проб и ошибок в стиле обучения с подкреплением. Он поддерживает популяцию геномов, которые эволюционируют из поколения в поколение до тех пор, пока не будет найден успешный решатель. В ходе своей эволюции каждый организм в популяции проверяется на приспособленность путем моделирования тележки с обратным маятником. В конце симуляции организм получает сигнал вознаграждения в виде количества временных шагов, в течение которых он мог сохранять баланс маятника в пределах дорожки. Полученный сигнал вознаграждения определяет приспособленность организма и решает его судьбу в процессе нейроэволюции.

Затем вы узнали, как можно определить целевую функцию, используя упомянутый сигнал вознаграждения, и реализовать эту функцию на языке Python.

Закончив первый эксперимент по балансировке маятника, мы перешли к модифицированной версии этого эксперимента. Модифицированная версия состояла из двух маятников разной длины, соединенных с подвижной тележкой. Во втором случае необходимо было поддерживать баланс двух маятников одновременно. Этот эксперимент имел более сложную физику и нуждался в поиске гораздо более совершенного контроллера.

Оба эксперимента, представленных в этой главе, подчеркнули важность поддержания сбалансированной популяции решателей с умеренным числом видов. Слишком большое видовое разнообразие популяции может препятствовать процессу нейроэволюции, уменьшая вероятность спаривания организмов, принадлежащих к разным видам.

Кроме того, принимая во внимание, что размер популяции фиксирован, чем больше видов в популяции, тем меньше численность каждого вида. Малочисленные виды снижают вероятность возникновения полезных мутаций. С другой стороны, наличие отдельных видов позволяет нам поддерживать полезные мутации в каждой нише видообразования и использовать каждую мутацию в последующих поколениях. Таким образом, слишком низкое видовое разнообразие также вредит эволюции. В конце эксперимента по балансировке тележки с маятником вы приобрели некоторые практические навыки, связанные с поддержанием разумного баланса числа видов путем настройки соответствующих гиперпараметров алгоритма NEAT (таких как порог совместимости видов).

Другая существенная особенность процесса нейроэволюции, которая была подчеркнута в эксперименте по балансировке, связана с выбором правильного начального числа для генератора псевдослучайных чисел, который управляет эволюционным процессом. Реализация метода нейроэволюции построена вокруг генератора псевдослучайных чисел, который определяет вероятность мутаций генома и скорость кроссовера. В генераторе псевдослучайных чисел последовательность выходных значений определяется

только начальным значением, которое подается в генератор вначале. Используя одно и то же начальное значение, можно создавать одинаковые последовательности псевдослучайных чисел.

В результате эксперимента с поиском нейросети контроллера для балансировки тележки мы обнаружили, что вероятность нахождения успешного решения сильно зависит от начального значения генератора случайных чисел.

Освоение экспериментов по балансировке маятников позволило вам подготовиться к решению более сложных проблем, связанных с автономной навигацией, которые обсуждались в следующей главе.

12.1.5 Автономное прохождение лабиринта

В пятой главе вы продолжили эксперименты с нейроэволюцией и попытались создать решатель, который может найти выход из лабиринта. Прохождение лабиринта является увлекательной задачей, поскольку оно позволяет изучить новый метод поисковой оптимизации – поиск новизны. В главах 5 и 6 мы рассмотрели серию экспериментов по прохождению лабиринта, используя целеориентированную оптимизацию поиска и метод оптимизации поиском новизны.

В этой главе вы ознакомились с симулятором робота, оснащенного массивом датчиков, которые обнаруживают препятствия и отслеживают положение робота в лабиринте. Также мы обсудили, как реализовать целевую функцию для управления эволюционным процессом. Упомянутая реализация целевой функции рассчитывается как евклидово расстояние между конечным положением робота и выходом из лабиринта.

Используя симулятор лабиринта и выбранную целевую функцию, вы провели два эксперимента с простой и сложной конфигурациями лабиринта. Результаты экспериментов дают представление о влиянии обманчивого ландшафта функции приспособленности на эффективность эволюционного процесса. В областях локального оптимума нейроэволюция склонна производить меньше видов, что ограничивает ее способность исследовать новые решения. В крайних случаях это приводит к вырождению эволюционного процесса, и может случиться так, что во всей популяции останется только один вид.

В то же время вы узнали, как избежать подобных неприятностей, настроив гиперпараметры NEAT, такие как коэффициент совместимости видов. Этот параметр контролирует степень влияния топологических различий в сравниваемых геномах на их совместимость для последующего скрещивания. В результате мы смогли увеличить видообразование и повысить разнообразие популяции. Это изменение оказало положительное влияние на поиск успешного решения, и мы смогли найти его для простой конфигурации лабиринта. Тем не менее сложная конфигурация лабиринта с более экстремальными локальными оптимумами не поддалась нашим попыткам найти удачный решатель с помощью целеориентированной функции приспособленности.

Таким образом, вы пришли к необходимости узнать о методе оптимизации поиском новизны, который был разработан для преодоления ограничений целеориентированного поиска.

12.1.6 Метод оптимизации поиском новизны

Во всех экспериментах, предшествующих шестой главе, мы определяли целевую функцию на основе метрики близости к конечной цели. Тем не менее задача прохождения лабиринта в общем случае не может быть решена с помощью измерения текущего расстояния до цели. Определенные конфигурации лабиринта могут создавать сильные локальные оптимумы, в которых застревает процесс поиска, ориентированный на близость к цели.

Поэтому мы воспользовались практическим опытом, полученным при создании решателя задачи лабиринта в предыдущей главе, и вступили на путь создания более совершенного решателя. Для управления эволюционным процессом новый решатель использовал метод оптимизации поиском новизны. Однако перед этим нам пришлось определить соответствующую метрику новизны для оценки степени новизны каждого решения в каждом поколении. Полученный с помощью этой метрики балл новизны служил показателем приспособленности, который присваивался геномам в популяции решателей. Таким образом, нам удалось интегрировать метрику новизны в стандартный процесс нейроэволюции.

Метрика новизны должна показывать, насколько новое решение отличается от решений, найденных в прошлых поколениях, и всех других решений текущего поколения. Существует два способа измерения новизны решения:

- *генотипическая новизна* является показателем внутренней новизны и показывает, как генотип текущего решения отличается от генотипов всех других найденных решений;
- *поведенческая новизна* показывает, как поведение текущего решения в проблемном пространстве отличается от поведения всех остальных решений.

Для решения задачи лабиринта хорошим выбором является использование оценки поведенческой новизны, потому что, в конце концов, мы заинтересованы в достижении выхода из лабиринта, чему может способствовать определенный тип поведения. Кроме того, оценку новизны поведения гораздо легче рассчитать, чем оценку новизны генотипа.

Траектория движения конкретного решателя через лабиринт определяет его поведенческое пространство. Следовательно, мы можем оценить степень новизны, сравнивая векторы траекторий решателей.

Численно оценка новизны может быть рассчитана путем вычисления евклидова расстояния между векторами траектории. Чтобы еще больше упростить эту задачу, мы можем использовать для вычисления балла новизны только координаты последней точки траектории решателя.

Определив метрику новизны, вы узнали, как реализовать ее в исходном коде на языке Python и интегрировать в симулятор лабиринта, созданный вами в главе 5. После этого повторили эксперименты из предыдущей главы и сравнили результаты.

Эксперимент с решателем простого лабиринта продемонстрировал улучшение топологии полученной нейросети. Топология стала оптимальной и менее сложной.

К сожалению, эксперимент со сложной конфигурацией лабиринта снова не привел к успешному решению, как это уже было в главе 5. Похоже, сбой

вызван неэффективностью конкретной реализации алгоритма NEAT, использованной в эксперименте. Я реализовал алгоритм NEAT на языке Go, чтобы он легко находил решение задачи сложного лабиринта, используя оптимизацию поиском новизны. Вы можете найти исходный код на GitHub по адресу **https://github.com/yaricom/goNEAT_NS**.

В главе 6 вы узнали, что метод оптимизации поиском новизны позволяет найти решение, даже если функция приспособленности имеет обманчивый ландшафт со многими локальными ловушками оптимальности, разбросанными внутри. Вы узнали, что шаги, из которых состоит путь к решению, не всегда очевидны. Иногда нужно сделать шаг назад, чтобы найти правильный путь. Это основная идея метода поиска новизны. Он пытается найти решение, полностью игнорируя близость к конечной цели и поощряя новизну каждого промежуточного решения, которое встречается на пути.

В этой главе вы познакомились со стандартным алгоритмом NEAT и подготовились к экспериментам с его более продвинутыми расширениями.

12.1.7 Зрительное различение с NEAT на основе гиперкуба

Седьмая глава была первой из четырех глав, в которых мы обсуждали передовые методы нейроэволюции. В этой главе вы узнали о схеме косвенного кодирования генома, в которой для кодирования топологий нейросетей с большим фенотипом используется CPPN. Схема кодирования CPPN, представленная расширением NEAT, называется HyperNEAT. Это расширение построено вокруг концепции субстрата связности, который представляет топологию нейросети. В то же время связи между узлами субстрата представлены точками внутри четырехмерного гиперкуба. В методе HyperNEAT топология CPPN эволюционирует в соответствии с алгоритмом NEAT. Мы уже обсуждали особенности HyperNEAT, поэтому пропустили остальные детали HyperNEAT для краткости.

В этой главе вы встретили интересную задачу зрительного различения, которая подчеркивает способность алгоритма HyperNEAT различать шаблоны в поле зрения. Вы узнали, что метод HyperNEAT может найти успешный дискриминатор зрительных образов благодаря присущей ему способности повторно использовать успешные шаблоны связей, найденные в субстрате, который кодирует фенотип нейросети решателя. Это стало возможным благодаря CPPN, которая способна находить правильную стратегию передачи сигналов от входных узлов (восприятие изображения) к выходным узлам (представление результатов).

Вы узнали, как выбрать правильную геометрию субстрата, чтобы эффективно использовать возможности CPPN для поиска геометрических закономерностей. После этого у вас была возможность применить полученные знания на практике, реализовав зрительный дискриминатор, обученный с использованием алгоритма HyperNEAT.

Кроме того, завершив эксперимент со зрительным дискриминатором, вы смогли проверить эффективность схемы косвенного кодирования. Эффективность кодирования вычислялась путем сравнения топологии, найденной CPPN, с максимально возможным количеством соединений в различающей нейросети. Результаты эксперимента со зрительным дискриминатором оказались довольно впечатляющими. Благодаря кодированию схемы связей среди

14 641 возможной связи субстрата нам удалось достичь степени сжатия информации 0,11 %, причем у нас осталось только 16 связей между 10 узлами CPPN.

Задачи компьютерного зрения предъявляют высокие требования к архитектуре нейросети из-за высокой размерности входного сигнала. Поэтому в главе 8 мы приступили к изучению другого класса задач зрительного распознавания.

12.1.8 Метод ES-HyperNEAT и задача сетчатки

В восьмой главе вы узнали, как выбрать конфигурацию субстрата, которая лучше всего подходит для конкретной проблемной области. Однако выбор конфигурации не всегда очевиден. Неправильно выбранная конфигурация может существенно повлиять на производительность процесса обучения. В результате процесс нейроэволюции может не найти успешное решение. Кроме того, конкретные особенности конфигурации субстрата обнаруживаются только во время процесса обучения и не могут быть известны заранее.

Проблема с поиском подходящей конфигурации субстрата была решена с помощью метода ES-HyperNEAT. В этой главе вы узнали, что процесс нейроэволюции может автоматически дорабатывать конфигурацию субстрата в ходе эволюции CPPN-связей. Вы познакомились с концепцией структуры данных в виде квадродерева, которая позволяет совершать эффективный обход топологии субстрата и обнаруживать области с высокой плотностью информации. Оказывается, полезно автоматически размещать новые узлы в этих областях для создания более точных паттернов связей, которые описывают скрытые закономерности, существующие в реальном мире.

Ознакомившись с деталями работы алгоритма ES-HyperNEAT, вы узнали, как применять его для решения задачи визуального распознавания, известной как проблема сетчатки. В этой задаче процесс нейроэволюции должен найти решателя, который способен распознавать подходящие изображения одновременно в двух отдельных полях зрения. Иными словами, нейросеть зрительного детектора должна решить, подходят ли изображения, представленные в правом и левом полях зрения, для каждого поля. Решение этой задачи может быть найдено путем введения модульной архитектуры в топологию нейросети детектора. В такой конфигурации каждый модуль нейросети отвечает за распознавание образов только на соответствующей стороне сетчатки.

В этой главе мы реализовали успешное решение задачи сетчатки, используя метод ES-HyperNEAT, и смогли визуально доказать, что полученная топология нейросети детектора содержит модульные структуры. Кроме того, из результатов эксперимента вы узнали, что найденная структура нейросети детектора имеет почти оптимальную сложность. Этот эксперимент еще раз продемонстрировал потенциал нейроэволюции в поиске эффективных решений методом постепенного усложнения.

Во всех экспериментах, включая описанный в этой главе, использовалась особая форма функции приспособленности, которая была заранее определена до начала экспериментов. Тем не менее было бы интересно изучить, как изменяется производительность алгоритма нейроэволюции, если функция приспособленности развивается вместе с решением, которое она пытается оптимизировать.

12.1.9 Коэволюция и метод SAFE

В девятой главе вы узнали, что стратегия *коэволюции* широко распространена в природе и может быть перенесена в область нейроэволюции. Мы перечислили наиболее распространенные стратегии коэволюции, которые можно найти в природе: мутуализм, конкуренция (хищничество или паразитизм) и комменсализм. В нашем эксперименте мы исследовали комменсалистический тип эволюции, который можно определить следующим образом: представители одного вида получают выгоды, не причиняя вреда и не принося пользы другим сосуществующим видам.

Зная о стратегиях эволюции в мире природы, вам было проще понять концепции, лежащие в основе метода SAFE. Название метода SAFE (solution and fitness evolution, коэволюции решателя и приспособленности) говорит о том, что у нас есть две совместно развивающиеся популяции: совокупность потенциальных решений и совокупность кандидатов на роль функции приспособленности. В каждом поколении эволюции мы оцениваем каждое потенциальное решение при помощи всех кандидатов на роль функции приспособленности и выбираем лучший показатель приспособленности, который рассматривается как пригодность решения для кодирования генома. В то же время мы развиваем комменсалистическую совокупность кандидатов на роль функции приспособленности, используя метод поиска новизны. Поиск новизны использует геномную новизну каждого генома в популяции в качестве метрики для оценки индивидуальной приспособленности.

В этой главе вы узнали, как реализовать модифицированный эксперимент по прохождению лабиринтов на основе метода SAFE для проверки эффективности стратегии коэволюции. Кроме того, вы узнали, как определить целевую функцию, чтобы направлять эволюцию популяции потенциальных решений. Эта целевая функция включает в себя две метрики приспособленности: первая – это расстояние до выхода из лабиринта, а вторая – поведенческая новизна найденного решения. Эти метрики объединяются с использованием коэффициентов, найденных популяцией кандидатов на функцию приспособленности.

Как и во всех предыдущих главах, вы продолжали совершенствовать свои навыки работы с Python, реализуя метод SAFE с использованием библиотеки MultiNEAT. В следующей главе мы продолжили изучение еще более продвинутых методов, что позволило использовать нейроэволюцию для обучения игровых агентов Atari.

12.1.10 Глубокая нейроэволюция

В десятой главе вы познакомились с концепцией глубокой нейроэволюции, которую можно использовать для обучения глубоких искусственных нейронных сетей (DNN). Вы узнали, как глубокая нейроэволюция может быть использована для обучения игровых агентов Atari с использованием алгоритма глубокого подкрепления.

Мы начали с обсуждения основных концепций обучения с подкреплением, уделив особое внимание популярному алгоритму Q-обучения, который является одной из классических реализаций обучения с подкреплением. После этого вы узнали, как DNN можно использовать для аппроксимации функции

Q-значений для сложных задач, которые не могут быть аппроксимированы простой матрицей действий с Q-значениями. Далее мы обсудили, как можно найти обучаемые параметры DNN при помощи нейроэволюции. Вы узнали, что нейроэволюция развивает DNN для аппроксимации Q-значений. В результате мы можем обучить DNN, не используя трудоемкий способ обратного распространения ошибки, который является обычным в традиционных методах обучения DNN.

Узнав о глубоком обучении с подкреплением, вы перешли к применению полученных знаний на практике, разработав агента-решателя для прохождения игр Atari. Чтобы агент научился играть в игру Atari, ему нужно прочитать пиксели игрового экрана и определить текущее состояние игры. После этого, используя извлеченное игровое состояние, агент должен выбрать соответствующее действие, которое будет выполнено в игровой среде. Конечная цель агента – максимизировать окончательное вознаграждение в конце определенного игрового эпизода. Таким образом, у нас получается классическое обучение методом проб и ошибок, в котором и заключается суть обучения с подкреплением.

Как мы уже упоминали, игровой агент должен анализировать пиксели игрового экрана. Лучший способ сделать это – использовать *сверточную нейронную сеть* (convolutional neural network, CNN) для обработки входных данных, полученных с игрового экрана. В этой главе мы обсудили основы архитектуры CNN и ее интеграцию с игровым агентом. Вы узнали, как реализовать CNN в Python, используя популярный фреймворк TensorFlow.

Кроме того, вы узнали об уникальной схеме кодирования генома, которая была разработана специально для задач, связанных с глубокой нейроэволюцией. Эта схема позволяет нам кодировать фенотип нейросети с миллионами обучаемых параметров. Предложенная схема использует начальные числа генератора псевдослучайных чисел для кодирования весов связей нейросети. В этой схеме кодирования геном представлен в виде списка случайных начальных чисел. Каждое начальное число используется последовательно для генерации всех весов связей при помощи генератора псевдослучайных чисел.

Разобравшись с реализацией кодирования генома, мы перешли к эксперименту, целью которого было создание агента, способного играть в игру Atari Frostbite. Кроме того, вы узнали, как использовать современный графический процессор для ускорения вычислений в процессе обучения. В конце этой главы вы познакомились с продвинутым инструментом визуализации (VINE), который позволяет изучать результаты экспериментов по нейроэволюции.

Этой главой мы закончили наше краткое знакомство с наиболее популярными методами нейроэволюции, существовавшими на момент написания книги. Тем не менее есть еще много вещей, которые заслуживают вашего внимания в быстро растущей области прикладного искусственного интеллекта и методов нейроэволюции.

12.2 Куда двигаться дальше

Я надеюсь, что ваше путешествие по методам нейроэволюции, которые были представлены в этой книге, было приятным и полезным. Я сделал все возможное, чтобы представить вам самые последние достижения в области нейроэво-

люции. Однако эта область прикладной информатики стремительно развивается, и почти каждый месяц исследователи объявляют о новых достижениях.

В университетах, а также в корпорациях по всему миру существует множество лабораторий, работающих над применением методов нейроэволюции для решения задач, которые выходят за рамки традиционных алгоритмов глубокого обучения.

Я надеюсь также, что вам понравились методы нейроэволюции, которые мы обсуждали, и вы готовы применить их в своей работе и экспериментах. Тем не менее вам необходимо продолжить самообразование, чтобы идти в ногу с последними достижениями в этой области. Дальше я расскажу про несколько мест, где вы можете продолжить свое образование.

12.2.1 Uber AI Labs

В основе исследовательского центра Uber AI Labs лежит стартап Geometric Intelligence, основанный совместно с Кеннетом О. Стэнли – одним из выдающихся пионеров в области нейроэволюции. Он является автором алгоритма NEAT, который мы часто использовали в этой книге. Вы можете следить за работами Uber AI Labs на сайте **https://eng.uber.com/category/articles/ai/**.

12.2.2 alife.org

Международное общество искусственной жизни (international society for artificial life, ISAL) – это солидное сообщество исследователей и энтузиастов со всего мира, которые вовлечены в научно-исследовательскую деятельность, связанную с искусственной жизнью. Генетические алгоритмы и нейроэволюция также входят в сферу интересов этого общества. ISAL издает журнал Artificial Life и спонсирует различные конференции. Вы можете узнать больше о деятельности ISAL на сайте **http://alife.org**.

12.2.3 Открытая эволюция в Reddit

Концепция открытой эволюции напрямую связана с генетическими алгоритмами в целом и нейроэволюцией в частности. *Открытая эволюция* предполагает наличие эволюционного процесса, который не связан какой-либо конкретной целью. Этот подход вдохновлен естественной эволюцией биологических организмов, которые произвели нас, людей. Существует специальный сабреддит[1], где все желающие обсуждают исследования по открытой эволюции. Вы можете найти его по адресу **https://www.reddit.com/r/oee/**.

12.2.4 Каталог программного обеспечения NEAT

Университет Центральной Флориды ведет список программных библиотек, которые реализуют алгоритм NEAT и его расширения. Модератором программы является Кеннет О. Стэнли, автор алгоритма NEAT. Моя реализация алгоритмов NEAT и поиска новизны на языке Go также присутствует в этом каталоге. Вы можете найти каталог по адресу **http://eplex.cs.ucf.edu/neat_software/**.

---

[1] Раздел социально-новостного сайта reddit.com, посвященный отдельной теме. – *Прим. перев.*

12.2.5 arXiv.org

arXiv.org – это известный сервис, публикующий препринты статей по многим отраслям науки. Как правило, это отличный источник новейшей информации в области компьютерных наук. Вы можете искать в нем документы, связанные с нейроэволюцией, используя следующий поисковый запрос: **http://search.arxiv.org:8081/?query=neuroevolution&in=grp_cs**.

12.2.6 Оригинальная публикация про алгоритм NEAT

Оригинальная диссертация, написанная Кеннетом О. Стэнли с описанием алгоритма NEAT, очень полезна для чтения и рекомендуется всем, кто интересуется нейроэволюцией. Она доступна по адресу **http://nn.cs.utexas.edu/downloads/papers/stanley.phd04.pdf**.

12.3 Заключение

В этой главе кратко изложено то, что вы узнали в данной книге. Вы также узнали о местах, где можно искать дальнейшие идеи и продолжать самообразование.

Я счастлив жить в эпоху, когда будущее становится реальностью настолько быстро, что мы совершенно не замечаем огромных изменений, которые происходят в нашей жизни. Человечество стремительно движется по пути к освоению чудес редактирования генов и синтетической биологии. Мы продолжаем проникать в тайны человеческого мозга, что открывает путь к окончательному пониманию сути сознания. Сложнейшие эксперименты в области космологии приближают нас к самым первым мгновениям существования Вселенной.

Мы создали усовершенствованный математический аппарат, который позволяет описывать такие загадочные объекты, как нейтрино, которое на своем пути может стать электроном, а затем снова нейтрино. Как говорил Артур Кларк, наши технологические достижения неотличимы от магии.

Жизнь – это ощущение красоты сущего. Держите свой ум острым и всегда будьте любопытным. Мы стоим на пороге великих событий, когда исследования искусственного сознания породят эволюцию новых форм жизни. И кто знает – может быть, это предстоит сделать именно вам.

Спасибо, мой дорогой читатель, за потраченное время и усилия. Я с нетерпением жду встречи с тем, что вы создадите, используя знания, полученные из этой книги.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Ярослав Омельяненко

Эволюционные нейросети на языке Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Яценков В. С.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 25,19. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

Отпечатано в ПАО «Т8 Издательские Технологии»
109316, Москва, Волгоградский проспект, д. 42, корпус 5.