ДМК издательство

К. Ма, В. Хегде, Л. Йольан

Трехмерное глубокое обучение на Python

Эта книга дает полное представление о современном трехмерном глубоком обучении и помогает разработчикам в области компьютерного зрения применить свои знания на практике.

Вы научитесь:

- разрабатывать модели трехмерного компьютерного зрения для взаимодействия с окружающей средой;
- обрабатывать 3D-данные с использованием облаков точек, полигональных сеток, применяя файлы форматов PLY и OBJ;
- работать с 3D-геометрией, моделями камеры, системами координат и конвертировать данные из одной в другую;
- с легкостью разбираться в понятиях отрисовки, затенения и т. д.;
- реализовывать дифференцируемую отрисовку во многих моделях трехмерного глубокого обучения;
- применять современные модели трехмерного глубокого обучения, такие как NeRF, SynSin, Mesh R-CNN.

Издание предназначено для специалистов по анализу данных, инженеров машинного и глубокого обучения, которые хотят хорошо разбираться в методах компьютерного зрения с использованием 3D-данных.

Разрабатывайте модели компьютерного зрения с использованием 3D-данных с помощью библиотеки PyTorch3D и других инструментов

Интернет-магазин:
www.dmkpress.com

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

Packt›

ДМК ПРЕСС
издательство
www.дмк.рф

ISBN 978-5-93700-202-0

Ксудонг Ма, Вишах Хегде, Лилит Йольян

Трехмерное глубокое обучение на Python

Разрабатывайте модели компьютерного зрения с использованием 3D-данных с помощью библиотеки PyTorch3D и других инструментов

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2023

УДК 004.04
ББК 32.372
М12

Ма К., Хегде В., Йольян Л.
М12 Трехмерное глубокое обучение на Python / пер. с англ. А. В. Логунова. – М.: ДМК Пресс, 2023. – 226 с.: ил.

ISBN 978-5-93700-202-0

В этом руководстве исследуется современное трехмерное глубокое обучение: приводятся пошаговые объяснения базовых понятий и концепций, а также практические примеры, на основе которых вы сможете создавать собственные модели. Вы научитесь обрабатывать 3D-данные с использованием облаков точек, полигональных сеток; работать с 3D-геометрией, моделями камеры, системами координат; разбираться в понятиях отрисовки, затенения и др.; применять современные продвинутые модели трехмерного глубокого обучения, такие как NeRF, SynSin, Mesh R-CNN.

Издание предназначено для практиков машинного обучения от начального до среднего уровня, исследователей данных, а также инженеров машинного и глубокого обучения, которые хотят изучить и применять методы трехмерного компьютерного зрения.

УДК 004.04
ББК 32.372

First published in the English language under the title '3D Deep Learning with Python' – (9781803247823).

ISBN 978-1-80324-782-3 (англ.)
ISBN 978-5-93700-202-0 (рус.)

Посвящается моей жене и семье,
за их поддержку и вдохновение на всех этапах работы
– Вишах Хегде

Посвящается моей семье и друзьям,
чья любовь и поддержка были моей самой большой мотивацией
– Лилит Йольян

Содержание

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 на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и 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 обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

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 незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторах

Ксудонг Ма – штатный инженер машинного обучения в Grabango Inc. Беркли, штат Калифорния. Работал старшим инженером машинного обучения в Facebook (Meta) Oculus и тесно сотрудничал с коллективом 3D PyTorch в проектах отслеживания лица в 3D. Имеет многолетний опыт работы в области компьютерного зрения, машинного и глубокого обучения и имеет докторскую степень в области электромашиностроения и конструирования вычислительных машин.

Вишах Хегде – исследователь в области машинного обучения и компьютерного зрения. Имеет более 7 лет опыта работы в указанных областях, во время которых стал автором нескольких хорошо процитированных исследовательских работ и опубликованных патентов. Имеет степень магистра Стэнфордского университета по специализации «Прикладная математика и машинное обучение», а также степени бакалавра и магистра по физике университета IIT в Мадрасе. Ранее работал в Schlumberger и Matroid. Является старшим прикладным исследователем в Ambient.ai, где помогал разрабатывать систему обнаружения оружия, которая развернута в нескольких глобальных компаниях из списка Fortune 500. Сейчас он использует свой опыт и страсть к решению деловых задач с целью создания технологического стартапа в Кремниевой долине. Подробнее о нем можно узнать на его сайте.

Хотел бы поблагодарить исследователей в области компьютерного зрения, прорывное исследование которых мне пришлось описывать. Хотел бы поблагодарить рецензентов за их отзыв и замечательный коллектив издательства Packt Publishing за то, что он дал мне возможность проявить творческий подход. Наконец, хочу поблагодарить свою жену и семью за их поддержку и вдохновение в ситуации, когда мне это было нужно больше всего.

Лилит Йольян – исследователь машинного обучения, работающая над докторской диссертацией в университете в YSU. Ее исследования посвящены разработке технологических решений в области компьютерного зрения для умных городов с использованием данных дистанционной съемки. Имеет 5-летний опыт работы в области компьютерного зрения и работала над технически сложным решением по обеспечению безопасности водителя, планируемым к развертыванию многими известными компаниями-производителями автомобилей.

О рецензентах

Эйя Абид – студентка магистратуры в области машиностроения со специализацией «Глубокое обучение и компьютерное зрение». Занимает пост преподавателя ИИ в рамках NVIDIA и квантового машинного обучения в CERN.

Прежде всего хотела бы посвятить эту работу своей семье, друзьям и всем тем, кто помог мне в ходе работы. Особая благодарность Аймену, которому я всегда благодарна.

Рамеш Сехар – генеральный директор и соучредитель компании Dapster.ai, которая разрабатывает доступных и легко развертываемых роботов, выполняющих самые трудные задачи на складах. Рамеш работал в таких компаниях, как Symbol, Motorola и Zebra, и специализируется на разработке продуктов на пересечении компьютерного видения, ИИ и робототехники. Имеет степень бакалавра в области электромашиностроения и магистра в области вычислительных систем. Рамеш основал Dapster.ai в 2020 году. Миссия Dapster состоит в том, чтобы разрабатывать роботов, которые оказывают положительное влияние на людей, выполняя опасные и вредные для здоровья задачи. Видение компании заключается в том, чтобы открывать доступ к более качественным рабочим местам, усиливать цепочки поставок и лучше справляться с вызовами, возникающими в результате изменения климата.

Уткарш Шривастава – профессионал в области ИИ/МО, тренер, ютубер и блогер. Любит решать и разрабатывать алгоритмы МО, обработки естественного языка и компьютерного зрения, чтобы решать сложные задачи. Начал свою карьеру в науке о данных в качестве блогера в своем блоге (datamahadev.com) и на канале YouTube (Datamahadev), после чего перешел на работу старшим тренером по науке о данных в институте в Гуджарате. Кроме того, он обучил и консультировал более 1000 работающих специалистов и студентов по ИИ/МО. Уткарш выполнил более 40 внештатных работ/проектов по тренировке и разработке в области науки о данных и аналитике, ИИ/МО, разработке на Python и SQL. Он родом из Лакхнау и в настоящее время поселился в Бангалоре, Индия, в качестве аналитика в Deloitte USI Consulting.

Хотел бы поблагодарить свою мать, миссис Рупам Шривастава, за ее постоянное руководство и поддержку на протяжении трудных периодов и борьбы. Спасибо также Верховному Пара-Брахману.

Мейсон МакГоу – старший специалист НИОКР в области машиностроения и компьютерного зрения в лаборатории Lowe's Innovation Labs. Страстно увлечен визуализацией и провел более десяти лет, решая задачи компью-

терного зрения в широком спектре промышленных и академических дисциплин, включая геологию, биоинформатику, разработку игр и розничную торговлю. Совсем недавно приступил к разведывательному анализу применения цифровых близнецов и 3D-сканирования применительно к розничным магазинам.

Хотел бы поблагодарить Энди Ликоса, Джозефа Канзано, Александра Аранго, Олега Александра, Эрин Кларк и мою семью за поддержку.

Предисловие

Благодаря этому практическому руководству по трехмерному глубокому обучению разработчики в области трехмерного компьютерного зрения смогут применить свои знания на практике. В данной книге представлен практический подход к реализации вычислительных решений в указанной области и связанных с ней методологий, которые помогут вам быстро начать работу и повысить продуктивность.

Оснащенные пошаговыми объяснениями важных понятий, практическими примерами и вопросами для самопроверки, вы начнете с обследования передовых методов трехмерного глубокого обучения.

Вы познакомитесь с базовой обработкой 3D-данных полигональной сетки и облака точек с помощью библиотеки PyTorch3D, такой как загрузка и сохранение файлов PLY и OBJ, проецирование 3D-точек на координаты камеры с использованием моделей перспективной камеры и ортографической камеры, отрисовка облаков точек и полигональных сеток на изображениях и т. д. Вы также научитесь реализовывать некоторые современные алгоритмы трехмерного глубокого обучения, такие как дифференцируемая отрисовка, NeRF, SynSin и Mesh R-CNN, поскольку благодаря библиотеке PyTorch3D программирование этих моделей глубокого обучения значительно упрощается.

К концу этой книги вы сможете реализовывать свои собственные модели трехмерного глубокого обучения.

Для кого эта книга предназначена

Эта книга предназначена для всех тех, кто начинает свою карьеру в области машинного обучения, а также практиков среднего уровня, исследователей данных, инженеров машинного обучения и инженеров глубокого обучения, которые стремятся хорошо разбираться в методах компьютерного зрения, используя 3D-данные.

О чем эта книга рассказывает

Глава 1 «Введение в обработку 3D-данных» будет посвящена основам 3D-данных, например способам хранения 3D-данных и базовым понятиям полигональной сетки и облаков точек, мировой системы координат и системы

координат поля зрения камеры. В ней также дается объяснение часто используемой системы координат NDC, способов конверсии разных систем координат, перспективной и ортографической камер и моделей камеры, которые следует использовать.

Глава 2 «Введение в трехмерное компьютерное зрение и геометрию» покажет базовые понятия компьютерной графики, такие как отрисовка и затенение. Вы познакомитесь с несколькими фундаментальными понятиями, которые потребуются в последующих главах этой книги, включая 3D-трансформации геометрии, тензоры PyTorch и оптимизацию.

Глава 3 «Подгонка деформируемых сеточных моделей к необработанным облакам точек» представит практический проект применения деформируемой 3D-модели с целью подгонки шумных 3D-наблюдений, используя все знания, которые вы получили в предыдущих главах. Вы познакомитесь с часто используемыми функциями стоимости, причинами важности этих функций и ситуациями, когда эти функции стоимости обычно используются. Наконец, мы обследуем наглядный пример выбора конкретных функций стоимости под конкретную задачу и настройки цикла оптимизации, чтобы получить желаемые результаты.

Глава 4 «Обнаружение и отслеживание позы объекта с помощью дифференцируемой отрисовки» расскажет о базовых концепциях дифференцируемой отрисовки. Она поможет разобраться в основных понятиях и выяснить, в каких ситуациях следует эти методы применять для решения своих собственных задач.

Глава 5 «Понятие дифференцируемой объемометрической отрисовки» представит практический проект с использованием дифференцируемой отрисовки для оценивания позиций камеры по одному изображению и известной трехмерной сеточной модели. Вы научитесь применять библиотеку PyTorch3D на практике, чтобы настраивать камеры, отрисовщики и затенители. Вы также получите практический опыт использования разных функций стоимости, чтобы получать результаты оптимизации.

Глава 6 «Обследование нейронных полей яркости излучения (NeRF)» предоставит практический проект с использованием дифференцируемого отрисовщика для оценивания трехмерных сеточных моделей по нескольким изображениям и текстурным моделям.

Глава 7 «Обследование контролируемых нейронных полей признаков» охватывает очень важный алгоритм синтеза ракурсов под названием Nerf. Вы познакомитесь с тем, что это вообще такое, как его использовать и где он проявляет свою ценность.

Глава 8 «Моделирование человеческого тела в 3D» посвящена обследованию подгонки 3D-тела человека с использованием алгоритма SMPL.

Глава 9 «Сквозной синтез ракурсов с помощью модели SynSin» посвящена передовой модели глубокого обучения, применяемой для синтеза других ракурсов изображения.

Глава 10 «Модель Mesh R-CNN» познакомит еще с одним передовым методом предсказания трехмерных воксельных моделей по одному входному изображению под названием Mesh R-CNN.

Что нужно, чтобы извлечь максимум пользы из этой книги

Описанное в книге программное/аппаратное обеспечение	Требования к операционной системе
Python 3.6+	Windows, MacOS или Linux

Если вы используете цифровую версию этой книги, то советуем набирать исходный код самостоятельно либо обращаться к исходному коду в репозитории книги на GitHub (ссылка на репозиторий доступна в следующем разделе). Это поможет избежать любых потенциальных ошибок, связанных с копированием и вставкой исходного кода.

Для справки, пожалуйста, ознакомьтесь с перечисленными ниже статьями.

Глава 6: https://arxiv.org/abs/2003.08934, https://github.com/yenchenlin/nerf-pytorch.

Глава 7: https://m-niemeyer.github.io/project-pages/giraffe/index.html, https://arxiv.org/abs/2011.12100.

Глава 8: https://smpl.is.tue.mpg.de/, https://smplify.is.tue.mpg.de/, https://smpl-x.is.tue.mpg.de/.

Глава 9: https://arxiv.org/pdf/1912.08804.pdf.

Глава 10: https://arxiv.org/abs/1703.06870, https://arxiv.org/abs/1906.02739.

Используемые обозначения

В этой книге используется ряд текстовых обозначений.

`Исходный код в тексте`: указывает слова исходного кода в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, имена путей, фиктивные URL-адреса, вводимые пользователем данные и дескрипторы Twitter. Например, «Далее нужно обновить файл `./options/options.py`».

Блок исходного кода задается, как показано ниже:

```
elif opt.dataset == 'kitti':
    opt.min_z = 1.0
    opt.max_z = 50.0
    opt.train_data_path = (
        './DATA/dataset_kitti/'
    )
    from data.kitti import KITTIDataLoader
    return KITTIDataLoader
```

Когда мы хотим привлечь ваше внимание к определенной части блока исходного кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
wget https://dl.fbaipublicfiles.com/synsin/checkpoints/realestate/synsin.pth
```

Любые данные на входе или на выходе из команды командой оболочки записываются, как показано ниже:

```
bash ./download_models.sh
```

Жирный шрифт: выделяет новый термин, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговых окнах пишутся в тексте следующим образом: «Модуль детализации (**g**) получает входные данные от нейронного отрисовщика облака точек и затем выводит окончательное реконструированное изображение».

Подсказки и важные замечания выглядят так.

Часть I

ОСНОВЫ ОБРАБОТКИ 3D-ДАННЫХ

Первая часть книги посвящена определению самых базовых понятий обработки данных и изображений, поскольку указанные понятия лягут в основу последующего изложения. Данная часть делает книгу самодостаточной, вследствие чего читателям не придется читать какие-либо другие книги, чтобы приступить к изучению библиотеки PyTorch3D.

Эта часть содержит следующие главы:

- глава 1 «Введение в обработку 3D-данных»;
- глава 2 «Введение в трехмерное компьютерное зрение и геометрию».

Глава 1

Введение в обработку 3D-данных

В этой главе мы обсудим несколько базовых понятий, весьма существенных для трехмерного глубокого обучения, которые будут часто использоваться в последующих главах. Вы начнете со знакомства с наиболее часто используемыми форматами 3D-данных, а также многими способами манипулирования ими и конвертации их в разные форматы. Мы начнем с настройки среды разработки и установкой всех необходимых программных пакетов, включая Anaconda, Python, PyTorch и PyTorch3D. Затем мы поговорим о наиболее часто используемых способах представления 3D-данных – например, облаках точек, полигональных сетках и вокселах. Затем мы перейдем к форматам файлов 3D-данных, таким как файлы PLY и OBJ. Затем обсудим системы 3D-координат. Наконец, мы обсудим модели камеры, которые в основном связаны со способом отображения 3D-данных в 2D-изображения[1].

После прочтения этой главы вы сможете легко отлаживать алгоритмы трехмерного глубокого обучения, проводя инспекцию файлов выходных данных. Благодаря четкому пониманию систем координат и моделей камеры вы будете готовы опираться на эти знания и узнать о более продвинутых темах трехмерного глубокого обучения.

В данной главе будут охвачены следующие ниже главные темы:

- настройка среды разработки и установка дистрибутива Anaconda, библиотек PyTorch и PyTorch3D,
- представление 3D-данных,
- форматы 3D-данных – файлы PLY и OBJ,
- системы 3D-координат и конверсия между ними,
- модели камеры – перспективная и ортографическая камеры.

[1] Син. соотнесение 3D-данных с 2D-изображениями. – *Прим. перев.*

Технические требования

Для выполнения примеров исходного кода этой книги в идеале понадобится компьютер с графическим процессором. Тем не менее для выполнения фрагментов исходного кода вполне будет достаточно только центрального процессора(ов).

Рекомендуемая компьютерная конфигурация включает следующее:

- GPU, такой как серия GTX или серия RTX с не менее 8 Гб памяти,
- Python 3,
- библиотеки PyTorch и PyTorch3D.

Фрагменты исходного кода к этой главе находятся по адресу https://github.com/packtpublishing/3d-deep-learning-with-python.

Настройка среды разработки

Сначала давайте создадим среду разработки для всех прилагаемых к этой книге примеров программирования. Для всех примеров исходного кода Python в этой книге рекомендуется использовать машину Linux.

1. Сначала мы настроим широко используемый дистрибутив Python под названием Anaconda, который идет в комплекте с мощной реализацией CPYTHON. Одним из преимуществ использования дистрибутива Anaconda является его система управления пакетами, позволяющая пользователям легко создавать виртуальные среды. Индивидуальная редакция дистрибутива является бесплатной для одиночных практиков, студентов и исследователей. В целях установки дистрибутива мы рекомендуем посетить его веб-сайт anaconda.com, на котором можно получить подробные инструкции. Самым простым способом установки дистрибутива Anaconda, как правило, является выполнение скрипта, который нужно скачать с веб-сайта дистрибутива. После настройки дистрибутива выполните следующую ниже команду, чтобы создать виртуальную среду Python 3.7:

   ```
   $ conda create -n python3d python=3.7
   ```

   Эта команда создаст виртуальную среду Python версии 3.7. Для того чтобы использовать эту виртуальную среду, ее нужно сначала активировать.
2. Активируйте только что созданную виртуальную среду следующей ниже командой:

   ```
   $ source activate python3d
   ```

3. Установите библиотеку PyTorch. Подробные инструкции по установке PyTorch находятся на ее веб-странице по адресу www.pytorch.org/get-

started/locally/. Например, на своем рабочем столе Ubuntu с CUDA 11.1 я установлю PyTorch 1.9.1 следующим образом:

```
$ conda install pytorch torchvision torchaudio
cudatoolkit-11.1 -c pytorch -c nvidia
```

4. Установите библиотеку PyTorch3D. Это библиотека Python с открытым исходным кодом для трехмерного компьютерного зрения, недавно выпущенная исследовательской группой Facebook AI Research. Библиотека PyTorch3D предоставляет много функций-утилит, позволяющих с легкостью манипулировать 3D-данными. Будучи спроектированной специально для глубокого обучения, она позволяет обрабатывать почти все 3D-данные мини-пакетами, такими как камеры, облака точек и полигональные сетки. Еще одной ключевой особенностью библиотеки PyTorch3D является реализация очень важной техники трехмерного глубокого обучения, именуемой *дифференцируемой отрисовкой*[1]. Тем не менее самым большим преимуществом данной библиотеки трехмерного глубокого обучения является ее тесная связь с PyTorch.
   Библиотеке PyTorch3D могут понадобиться некоторые зависимости, и подробные инструкции по установке этих зависимостей находятся на домашней странице PyTorch3D на Github по адресу github.com/facebookresearch/pytorch3d. После установки всех зависимостей, если следовать инструкциям веб-сайта, установка PyTorch3D легко выполняется следующей ниже командой:

```
$ conda install pytorch3d -c pytorch3d
```

Теперь, когда мы создали среду разработки, давайте продолжим и займемся изучением представления данных.

Представление 3D-данных

В этом разделе вы познакомитесь с наиболее часто используемыми представлениями 3D-данных. Выбор представления данных является особенно важным конструктивным решением для многих систем трехмерного глубокого обучения. Например, облака точек не имеют решетчатых структур, поэтому свертки обычно невозможно использовать для них напрямую. Представления в виде вокселов имеют решетчатые структуры, однако они, как правило, потребляют большой объем компьютерной памяти. Мы обсудим плюсы и минусы этих 3D-представлений подробнее в этом разделе. Пред-

[1] Дифференцируемая отрисовка (differentiable rendering) – это относительно новая и захватывающая область исследований в компьютерном зрении, преодолевающая разрыв между 2D и 3D, позволяющая связывать пикселы 2D-изображения с 3D-свойствами сцены. – *Прим. перев.*

ставлениями 3D-данных, получившими наиболее широкое применение на практике, обычно являются облака точек, полигональные сетки и вокселы.

Представление в виде облака точек

Облако 3D-точек – это очень простое представление 3D-объектов, в котором каждое облако точек – это просто множество 3D-точек, и каждая 3D-точка представлена одним трехмерным кортежем (*x*, *y* и *z*). Сырые мерные данные многих камер глубины обычно представляют собой трехмерные облака точек.

С точки зрения глубокого обучения облака 3D-точек являются одним из неупорядоченных и нерегулярных типов данных. В отличие от регулярных изображений, в которых по каждому отдельному пикселу можно определить соседствующие ему пикселы, в облаке точек нет четких и регулярных определений соседних точек по каждой точке – т. е. применить свертки к облакам точек обычно невозможно. И поэтому для обработки облаков точек необходимо использовать специальные типы моделей глубокого обучения, такие как PointNet: https://arxiv.org/abs/1612.00593.

Еще одной проблемой облаков точек в качестве тренировочных данных для трехмерного глубокого обучения является проблема разнородности данных – т. е. по каждому тренировочному набору данных разные облака точек могут содержать разное число 3D-точек. Один из подходов к решению проблемы разнородности данных заключается в вынужденном назначении всем облакам точек одинакового числа точек. Однако это не всегда возможно – например, число возвращаемых камерами глубины точек может отличаться от кадра к кадру.

При тренировке моделей глубокого обучения разнородные данные могут создавать некоторые трудности для мини-пакетного градиентного спуска. В большинстве систем глубокого обучения подразумевается, что каждый мини-пакет содержит тренировочные примеры одинакового размера и мерности. Предпочитаются именно такие однородные данные, поскольку их можно обрабатывать на современном оборудовании для параллельной обработки наиболее эффективным образом, таком как графические процессоры. Эффективная обработка разнородных мини-пакетов требует дополнительной работы. К счастью, PyTorch3D предоставляет целый ряд способов эффективной обработки разнородных мини-пакетов, которые очень важны для трехмерного глубокого обучения.

Представление в виде полигональной сетки

Полигональные сетки, или меши, – это еще одно широко используемое представление 3D-данных. Как и точки в облаках точек, каждая полигональная сетка содержит множество 3D-точек, именуемых вершинами. Кроме того, каждая полигональная сетка содержит множество многоугольников, именуемых гранями, которые определены на вершинах.

В большинстве основанных на данных приложений полигональные сетки являются результатом постобработки сырых мерных данных камер глубины. Нередко они создаются вручную в процессе конструирования 3D-ресурсов. По сравнению с облаками точек полигональные сетки содержат дополнительную геометрическую информацию, кодируют топологию и имеют информацию о нормалях к поверхности. Эта дополнительная информация становится особенно полезной в тренировке обучающихся моделей. Например, графовые сверточные нейронные сети обычно трактуют полигональные сетки как графы и определяют сверточные операции, используя информацию о соседстве вершин.

Подобно облакам точек, полигональные сетки также имеют схожие проблемы разнородности данных. И снова PyTorch3D предоставляет эффективные способы оперирования разнородными мини-пакетами данных полигональной сетки, что делает трехмерное глубокое обучение весьма эффективным.

Представление в виде воксела

Еще одним важным представлением 3D-данных является представление в виде воксела. Воксел – это аналог пиксела в трехмерном компьютерном зрении. Пиксел определяется путем деления прямоугольника в 2D на меньшие прямоугольники, и каждый малый прямоугольник – это один пиксел. По аналогии с этим воксел определяется путем деления трехмерного куба на кубы меньшего размера, и каждый такой куб называется одним вокселом. Соответствующие процессы показаны на следующем ниже рисунке:

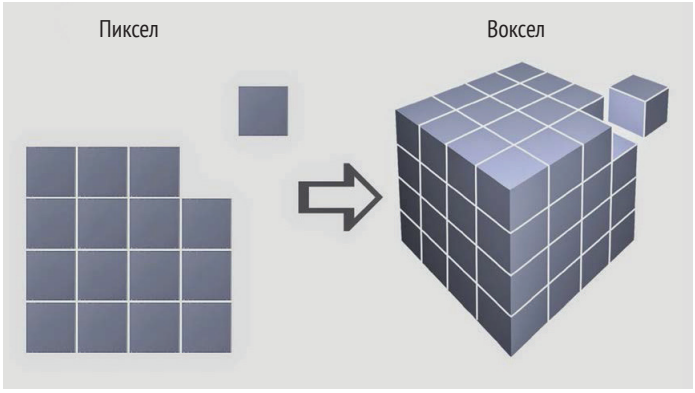


Рис. 1.1 ❖ Представление в виде воксела является трехмерным аналогом двумерного представления в виде пиксела, где кубическое пространство делится на малообъемные элементы

В представлениях в виде воксела для представления 3D-поверхностей обычно используются **функции усеченных расстояний со знаком** (**TSDF**)[1].

[1] От англ. *Truncated Signed Distance Function. – Прим. перев.*

Функция расстояния со знаком (**SDF**)[1] может быть определена на каждом вокселе в качестве расстояния (со знаком) между центром воксела до ближайшей точки на поверхности. Положительный знак в SDF указывает на то, что центр воксела находится вне объекта. Единственное различие между TSDF и SDF заключается в том, что значения TSDF усекаются, в результате чего значения TSDF всегда варьируются в интервале от –1 до +1.

В отличие от представлений в виде облаков точек и полигональных сеток представление в виде вокселов упорядочено и является регулярным. Это свойство похоже на пикселы в изображениях и позволяет использовать сверточные фильтры в моделях глубокого обучения. Одним из потенциальных недостатков представления в виде вокселов является то, что для них обычно требуется больше компьютерной памяти, но указанный недостаток можно уменьшить за счет таких методов, как хеширование. Тем не менее представление в виде вокселов является важным представлением 3D-данных.

Существуют представления 3D-данных, отличные от упомянутых выше. Например, в многоракурсных представлениях используется несколько изображений, взятых с разных точек зрения, чтобы представлять трехмерную сцену. В представлениях RGB-D используется дополнительный канал глубины, чтобы представлять 3D-сцену. Однако в этой книге мы не будем погружаться в эти 3D-представления слишком глубоко. Теперь, когда вы познакомились с основами представлений 3D-данных, самое время заняться несколькими форматами файлов, часто используемыми для облаков точек и полигональных сеток.

Формат файла 3D-данных – файлы PLY

Формат файла PLY[2] был разработан в середине 1990-х годов группой исследователей из Стэнфордского университета. С тех пор он превратился в один из наиболее широко используемых форматов файлов 3D-данных. Формат файла PLY имеет как ASCII-версию, так и двоичную версию. Двоичная версия предпочтительнее в тех случаях, когда необходимо минимизировать размеры файлов и обеспечить эффективность обработки. ASCII-версию легко отлаживать. Здесь мы обсудим базовый формат PLY-файлов и технику использования как пакета Open3D, так и библиотеки PyTorch3D для загрузки и визуализации 3D-данных из PLY-файлов.

В этом разделе мы собираемся обсудить два наиболее часто используемых формата файлов данных, чтобы представлять облака точек и полигональные сетки, формат PLY-файла и формат OBJ-файла. Мы обсудим сами форматы и способы загрузки и сохранения этих форматов файлов с помощью библиотеки PyTorch3D. Библиотека PyTorch3D предоставляет отличные функции-утилиты, поэтому с помощью этих утилит загрузка из этих форматов и сохранение в них проста и эффективна.

---

[1] От англ. *Signed Distance Function. – Прим. перев.*

[2] От англ. *Polygon File Format. – Прим. перев.*

Пример PLY-файла `cube.ply` показан в следующем ниже фрагменте исходного кода (данный файл находится в папке `ply_io` главы книги в репозитории на GitHub):

```
ply
format ascii 1.0
comment создан для книги Глубокое обучение в 3-D на Python
element vertex 8
property float32 x
property float32 y
property float32 z
element face 12
property list uint8 int32 vertex_indices
end_header
-1 -1 -1
1 -1 -1
1 1 -1
-1 1 -1
-1 -1 1
1 -1 1
1 1 1
-1 1 1
3 0 1 2
3 5 4 7
3 6 2 1
3 3 7 4
3 7 3 2
3 5 1 0
3 0 2 3
3 5 7 6
3 6 1 5
3 3 4 0
3 7 2 6
3 5 0 4
```

Как видно из приведенного выше примера, каждый PLY-файл содержит секцию заголовка и секцию данных. Первая строка каждого PLY-файла в кодировке ASCII всегда содержит ключевое слово `ply`, указывая на то, что это PLY-файл. Вторая строка `format ascii 1.0` показывает, что файл имеет кодировку ASCII с номером версии. Любые строки, начинающиеся с ключевого слова `comment`, будут рассматриваться как строка комментариев, и, следовательно, все, что следует за комментарием, будет игнорироваться при загрузке PLY-файла компьютером. Строка `element vertex 8` означает, что первым типом данных в PLY-файле является вершина, и всего имеется восемь вершин. Выражение `property float32 x` означает, что в каждой вершине есть свойство `x` с типом `float32`. Аналогичным образом каждая вершина также имеет свойства `y` и `z`. Здесь каждая вершина – это одна 3D-точка. Строка `element face 12` означает, что второй тип данных в указанном PLY-файле имеет тип `face`, и всего имеется 12 граней. Выражение `property list uint8 int32 vertex_indices` показывает, что каждая грань будет списком индексов вершин. Секция заголовка PLY-файла всегда заканчивается строкой `end_header`.

Первая часть секции данных PLY-файла состоит из восьми строк, каждая строка которой является записью одной вершины. Три числа в каждой строке представляют три свойства `x`, `y` и `z` вершины. Например, три числа `-1`, `-1`, `-1` указывают на то, что координата `x` вершины равна `-1`, координата `y` вершины равна `-1` и координата `z` вершины равна `-1`.

Вторая часть секции данных PLY-файла состоит из 12 строк, каждая строка которой является записью одной грани. Первое число в последовательности чисел указывает число имеющихся у грани вершин, а последующие числа являются индексами вершин. Индексы вершин определяются по порядку, в котором вершины объявлены в PLY-файле.

Для открытия приведенного выше файла можно использовать как пакет Open3D, так и библиотеку PyTorch3D. Пакет Python Open3D очень удобен для визуализации 3D-данных, а библиотека PyTorch3D удобна для применения этих данных в моделях глубокого обучения. Ниже приведен фрагмент исходного кода файла Python `ply_example1.py` в папке `ply_io` главы книги в репозитории на GitHub для визуализации полигональной сетки в PLY-файле `cube.ply` и загрузки вершин и полигональной сетки в виде тензоров PyTorch:

```
import open3d
from pytorch3d.io import load_ply

mesh_file = 'cube.ply'
print('Визуализация полигональной сетки с помощью Open3D')
mesh = open3d.io.read_triangle_mesh(mesh_file)
open3d.visualization.draw_geometries([mesh],
       mesh_show_wireframe = True,
       mesh_show_back_face = True)

print('Загрузка того же файла с помощью PyTorch3D')
vertices, faces = load_ply(mesh_file)
print('Тип vertices = ', type(vertices))
print('Тип faces = ', type(faces))
print('vertices = ', vertices)
print('faces = ', faces)
```

В приведенном выше фрагменте исходного кода Python PLY-файл полигональной сетки `cube.ply` сначала открывается с помощью функции `read_triangle_mesh`[1] пакета Open3D, и все 3D-данные читаются в переменную `mesh`. Затем полигональную сетку можно визуализировать, используя функцию `draw_geometries`[2] данного пакета. При выполнении этой функции пакет Open3D выведет на экран окно для интерактивной визуализации полигональной сетки, в котором ее можно интерактивно поворачивать, увеличивать и уменьшать ее масштаб, используя мышь. PLY-файл `cube.ply`, как можно догадаться, определяет полигональную сетку куба с восемью вершинами и шестью сторонами, каждая сторона которого покрыта двумя гранями.

---

[1] Прочитать треугольную сетку. – *Прим. перев.*

[2] Начертить геометрические объекты. – *Прим. перев.*

Для загрузки той же сетки также можно использовать библиотеку PyTorch3D. Однако на этот раз мы собираемся получить несколько тензоров PyTorch – например, один тензор для вершин и один тензор для граней. Эти тензоры можно напрямую водить в любую модель глубокого обучения PyTorch. В данном примере функция `load_ply` возвращает кортеж с вершинами и гранями, обе из которых обычно находятся в формате тензоров PyTorch. При выполнении исходного кода файла Python `ply_example1.py` возвращенные вершины должны быть тензором PyTorch с очертанием[1] `[8, 3]`, т. е. восемь вершин, и в каждой вершине по три координаты. Аналогичным образом возвращенные грани должны быть тензором PyTorch с очертанием `[12, 3]`, т. е. 12 граней, и у каждой грани по 3 индекса вершин.

В следующем ниже фрагменте исходного кода демонстрируется еще один пример PLY-файла, `parallel_plane_mono.ply`, который тоже можно скачать из репозитория книги на GitHub. Единственное различие между полигональной сеткой в этом примере и полигональной сеткой в PLY-файле `cube.ply` состоит в том, что теперь вместо шести сторон куба у нас только четыре грани, которые образуют две параллельные плоскости:

```
ply
format ascii 1.0
comment создан для книги Глубокое обучение в 3-D на Python
element vertex 8
property float32 x
property float32 y
property float32 z
element face 4
property list uint8 int32 vertex_indices
end_header
-1 -1 -1
1 -1 -1
1 1 -1
-1 1 -1
-1 -1 1
1 -1 1
1 1 1
-1 1 1
3 0 1 2
3 0 2 3
3 5 4 7
3 5 7 6
```

Полигональную сетку можно интерактивно визуализировать с помощью следующего ниже фрагмента исходного кода файла Python `ply_example2.py`.

1. Сначала импортируем все необходимые библиотеки Python:

```
import open3d
from pytorch3d.io import load_ply
```

[1] Также форма (англ. *shape*). – *Прим. перев.*

2. Затем, используя пакет Open3D, загружаем полигональную сетку:

```
mesh_file = 'parallel_plane_mono.ply'
print('Визуализация полигональной сетки с помощью Open3D')
mesh = open3d.io.read_triangle_mesh(mesh_file)
```

3. Далее применяем его метод `draw_geometries`, чтобы открыть окно для интерактивной визуализации полигональной сетки:

```
open3d.visualization.draw_geometries([mesh],
                                     mesh_show_wireframe=True,
                                     mesh_show_back_face=True)
```

4. Затем используем библиотеку PyTorch3D, чтобы открыть ту же полигональную сетку:

```
print('Загрузка того же файла с помощью PyTorch3D')
vertices, faces = load_ply(mesh_file)
```

5. И в конце распечатываем информацию о загруженных вершинах и гранях. На самом деле это просто обычные тензоры PyTorch3D:

```
print('Тип vertices = ', type(vertices),
      ', тип faces = ', type(faces))
print('vertices = ', vertices)
print('faces = ', faces)
```

Для каждой вершины также можно задать свойства, отличные от координат *x*, *y* и *z*. Например, можно задать цвета каждой вершины. Ниже приведен пример `parallel_plane_color.ply`:

```
ply
format ascii 1.0
comment создан для книги Глубокое обучение в 3-D на Python
element vertex 8
property float32 x
property float32 y
property float32 z
property uchar red
property uchar green
property uchar blue
element face 4
property list uint8 int32 vertex_indices
end_header
-1 -1 -1 255 0 0
1 -1 -1 255 0 0
1 1 -1 255 0 0
-1 1 -1 255 0 0
-1 -1 1 0 0 255
1 -1 1 0 0 255
1 1 1 0 0 255
-1 1 1 0 0 255
```

```
3 0 1 2
3 0 2 3
3 5 4 7
3 5 7 6
```

Обратите внимание, что в приведенном выше примере, наряду с *x*, *y* и *z*, мы также задаем несколько дополнительных свойств по каждой вершине, т. е. свойства `red`, `green` и `blue`, все это в типе данных `uchar`. Теперь каждая запись одной вершины представляет собой одну строку из шести чисел. Первые три числа – координаты *x*, *y* и *z*. Следующие три – значения RGB.

Полигональную сетку можно визуализировать с использованием файла Python `ply_example3.py` следующим образом:

```
import open3d
from pytorch3d.io import load_ply

mesh_file = 'parallel_plane_color.ply'
print('Визуализация полигональной сетки с помощью Open3D')

mesh = open3d.io.read_triangle_mesh(mesh_file)
open3d.visualization.draw_geometries([mesh],
                                     mesh_show_wireframe=True,
                                     mesh_show_back_face=True)
print('Загрузка того же файла с помощью PyTorch3D')
vertices, faces = load_ply(mesh_file)
print('Тип vertices = ', type(vertices),
      ', тип faces = ', type(faces))
print('vertices = ', vertices)
print('faces = ', faces)
```

В PLY-файле `cow.ply` мы также предоставляем реальный пример полигональной 3D-сетки. Читатели могут визуализировать указанную сетку с использованием файла Python `ply_example4.py`.

До сего момента мы говорили о базовых элементах формата файла 3D-данных PLY, таких как вершины и грани. Далее мы обсудим формат файла 3D-данных OBJ.

Формат файла 3D-данных – файлы OBJ

В этом разделе мы обсудим еще один широко используемый формат файла 3D-данных, формат файла OBJ. Формат файла OBJ был впервые разработан компанией Wavefront Technologies Inc. Как и формат файла PLY, формат OBJ также имеет как ASCII-версию, так и двоичную версию. Двоичная версия является проприетарной и незарегистрированной. И поэтому в данном разделе мы обсудим ASCII-версию.

Как и в предыдущем разделе, мы собираемся обследовать этот формат файла, посмотрев на примеры. Первый пример, `cube.obj`, приведен ниже. Как можно догадаться, в OBJ-файле определяется полигональная сетка куба.

```
mtllib ./cube.mtl
o cube
# Список вершин
v -0.5 -0.5 0.5
v -0.5 -0.5 -0.5
v -0.5 0.5 -0.5
v -0.5 0.5 0.5
v 0.5 -0.5 0.5
v 0.5 -0.5 -0.5
v 0.5 0.5 -0.5
v 0.5 0.5 0.5

# Список точек/линий/граней
usemtl Door

f 1 2 3
f 6 5 8
f 7 3 2
f 4 8 5
f 8 4 3
f 6 2 1
f 1 3 4
f 6 8 7
f 7 2 6
f 4 5 1
f 8 3 7
f 6 1 5
```

Первая строка, `mtlib ./cube.mtl`, объявляет сопровождающий файл библиотеки шаблонов материалов (**MTL**)[1].

MTL-файл описывает свойства затенения поверхности, которые будут объяснены в следующем далее фрагменте исходного кода.

В строке `o cube` начальная буква `o` указывает на то, что в строке задается объект, имя которого – `cube`. Любая строка, начинающаяся с `#`, является строкой комментариев, т. е. остальная часть строки будет компьютером проигнорирована. В каждой строке, начинающейся с `v`, `v` указывает на то, что в данной строке задается вершина. Например, строка `v -0.5 -0.5 0.5` задает вершину с *x*-координатой `0.5`, *y*-координатой `0.5` и *z*-координатой `0.5`. В каждой строке, начинающейся с `f`, `f` указывает на то, что в данной строке содержится определение одной грани. Например, строка `f 1 2 3` задает грань, причем три ее вершины являются вершинами с индексами 1, 2 и 3.

Строка `usemtl Door` объявляет, что объявленные после этой строки поверхности должны быть затенены с помощью определенного в MTL-файле материального свойства под именем `Door`.

Сопровождающий MTL-файл `cube.mtl` приведен ниже. Указанный файл задает материальное свойство, именуемое `Door`:

```
newmtl Door
Ka  0.8 0.6 0.4
```

[1] От англ. *Material Template Library. – Прим. перев.*

```
Kd  0.8 0.6 0.4
Ks  0.9 0.9 0.9
d   1.0
Ns  0.0
illum 2
```

Мы не будем подробно обсуждать эти материальные свойства, кроме `map_Kd`. Если вам интересно, то можете обратиться к стандартному учебнику по компьютерной графике, такому как «Компьютерная графика: принципы и практика»[1]. Мы перечислим некоторые грубые описания этих свойств следующим образом, просто ради полноты:

- `Ka`: задает окружающий цвет,
- `Kd`: задает рассеянный цвет,
- `Ks`: задает бликовый цвет,
- `Ns`: определяет фокус бликовых моментов,
- `Ni`: определяет оптическую плотность (он же индекс рефракции),
- `d`: задает коэффициент растворения,
- `illum`: задает модель освещения,
- `map_Kd`: задает файл цветной текстуры, который будет применен к диффузной отражательной способности материала.

OBJ-файл `cube.obj` можно открыть как с помощью пакета Open3D, так и с помощью библиотеки PyTorch3D. Следующий ниже фрагмент исходного кода, содержащийся в файле Python `obj_example1.py`, можно скачать из репозитория книги на GitHub:

```
import open3d
from pytorch3d.io import load_obj

mesh_file = 'cube.obj'
print('Визуализация полигональной сетки с помощью Open3D')
mesh = open3d.io.read_triangle_mesh(mesh_file)
open3d.visualization.draw_geometries([mesh],
                                     mesh_show_wireframe=True,
                                     mesh_show_back_face=True)

print('Загрузка того же файла с помощью PyTorch3D')
vertices, faces, aux = load_obj(mesh_file)
print('Тип vertices = ', type(vertices))
print('Type faces = ', type(faces))
print('Тип aux = ', type(aux))
print('vertices = ', vertices)
print('faces = ', faces)
print('aux = ', aux)
```

В приведенном выше фрагменте исходного кода заданная полигональная сетка куба интерактивно визуализируется с помощью функции Open3D `draw_geometries`. Указанная сетка будет показана в окне, и вы сможете ее поворачивать, увеличивать и уменьшать ее масштаб, используя мышь. Полигональную

[1] От англ. *Computer Graphics: Principles and Practice. – Прим. перев.*

сетку также можно загрузить с помощью функции PyTorch3D `load_obj`. Функция `load_obj` вернет переменные `vertices`[1], `faces`[2] и `aux`[3] в формате тензора PyTorch либо в формате кортежей тензоров PyTorch.

Пример результата работы фрагмента исходного кода `obj_example1.py` приводится ниже:

```
Визуализация полигональной сетки с помощью Open3D
Загрузка того же файла с помощью PyTorch3D
Тип vertices = <class 'torch.Tensor'>
Тип faces = <class 'pytorch3d.io.obj_io.Faces'>
Тип aux = <class 'pytorch3d.io.obj_io.Properties'>
vertices = tensor([[-0.5000, -0.5000,  0.5000],
                   [-0.5000, -0.5000, -0.5000],
                   [-0.5000, 0.5000, -0.5000],
                   [-0.5000, 0.5000,  0.5000],
                   [ 0.5000, -0.5000,  0.5000],
                   [ 0.5000, -0.5000, -0.5000],
                   [ 0.5000, 0.5000, -0.5000],
                   [ 0.5000, 0.5000,  0.5000]])
faces = Faces(verts_idx=tensor([
                   [0, 1, 2],
                   [5, 4, 7],
                   [6, 2, 1],
                   ...
                   [3, 4, 0],
                   [7, 2, 6],
                   [5, 0, 4]]),
               normals_idx=tensor([
                   [-1, -1, -1],
                   [-1, -1, -1],
                   [-1, -1, -1],
                   [-1, -1, -1],
                   ...
                   [-1, -1, -1],
                   [-1, -1, -1]]),
               textures_idx=tensor([
                   [-1, -1, -1],
                   [-1, -1, -1],
                   [-1, -1, -1],
                   ...
                   [-1, -1, -1],
                   [-1, -1, -1]]),
               materials_idx=tensor([0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0]))
aux = Properties(
           normals=None,
           verts_uvs=None,
```

[1] Вершины. – *Прим. перев.*

[2] Грани. – *Прим. перев.*

[3] Вспомогательная переменная. – *Прим. перев.*

```
material_colors={
  'Door': {'ambient_color': tensor([0.8000, 0.6000, 0.4000]),
  'diffuse_color': tensor([0.8000, 0.6000, 0.4000]),
  'specular_color': tensor([0.9000, 0.9000, 0.9000]),
  'shininess': tensor([0.])}},
texture_images={},
texture_atlas=None)
```

Здесь из приведенной выше распечатки мы понимаем, что возвращенная переменная `vertices` представляет собой тензор PyTorch с очертанием 8×3, в котором каждая строка – это вершина с координатами *x*, *y* и *z*. Возвращенная переменная `faces` представляет собой именованный кортеж из трех тензоров PyTorch, `verts_idx`, `normals_idx` и `textures_idx`. В данном примере все тензоры `normals_idx` и `textures_idx` являются недопустимыми, поскольку файл `cube.obj` не содержит определения нормали и текстур. В следующем ниже примере мы увидим, как определять нормали и текстуры в формате OBJ-файла. `verts_idx` – это индексы вершин по каждой грани. Обратите внимание, что в PyTorch3D индексы вершин индексируются от нуля, т. е. отсчет индексов начинается с 0. Однако индексы вершин в OBJ-файлах индексируются от единицы, т. е. отсчет индексов начинается с 1. Библиотека PyTorch3D уже провела конверсию между двумя способами индексации вершины за нас.

Возвращаемая переменная `aux` содержит некоторую дополнительную информацию о полигональной сетке. Обратите внимание на пустое поле `texture_image`[1] переменной `aux`. Изображения текстур используются в MTL-файлах для задания цвета на вершинах и гранях. Опять же, мы покажем способы применения этой функциональности в следующем примере.

Во втором примере мы будем использовать пример из OBJ-файла `cube_texture.obj`, чтобы осветить некоторые другие функциональные возможности OBJ-файла. Ниже приведен сам файл:

```
mtllib cube_texture.mtl
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -0.999999
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vt 1.000000 0.333333
vt 1.000000 0.666667
vt 0.666667 0.666667
vt 0.666667 0.333333
vt 0.666667 0.000000
vt 0.000000 0.333333
vt 0.000000 0.000000
vt 0.333333 0.000000
vt 0.333333 1.000000
```

[1] Текстурное изображение. – *Прим. перев.*

```
vt 0.000000 1.000000
vt 0.000000 0.666667
vt 0.333333 0.333333
vt 0.333333 0.666667
vt 1.000000 0.000000
vn 0.000000 -1.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 1.000000 0.000000 0.000000
vn -0.000000 0.000000 1.000000
vn -1.000000 -0.000000 -0.000000
vn 0.000000 0.000000 -1.000000
g main
usemtl Skin
s 1
f 2/1/1 3/2/1 4/3/1
f 8/1/2 7/4/2 6/5/2
f 5/6/3 6/7/3 2/8/3
f 6/8/4 7/5/4 3/4/4
f 3/9/5 7/10/5 8/11/5
f 1/12/6 4/13/6 8/11/6
f 1/4/1 2/1/1 4/3/1
f 5/14/2 8/1/2 6/5/2
f 1/12/3 5/6/3 2/8/3
f 2/12/4 6/8/4 3/4/4
f 4/13/5 3/9/5 8/11/5
f 5/6/6 1/12/6 8/11/6
```

Файл `cube_texture.obj` похож на файл `cube.obj`, за исключением следующих ниже различий:

- имеется несколько дополнительных строк, которые начинаются с `vt`. Каждая такая строка объявляет текстурную вершину с координатами `x` и `y`. В каждой текстурной вершине задается цвет, причем речь идет о пиксельном цвете на так называемом текстурном изображении, на котором местоположение пиксела – это *x*-координата ширины текстурной вершины *x* и *y*-координата высоты текстурной вершины *x*. Изображение текстуры будет определено в сопровождающем файле `cube_texture.mtl`;
- имеются дополнительные строки, которые начинаются с `vn`. В каждой такой строке объявляется вектор нормали – например, строка `vn 0.000000 -1.000000 0.000000` объявляет вектор нормали, указывающий на отрицательную ось *z*;
- каждая строка определения грани теперь содержит больше информации о каждой вершине. Например, строка `f 2/1/1 3/2/1 4/3/1` содержит определения трех вершин. Первая тройка, `2/1/1`, определяет первую вершину, вторая тройка, `3/2/1`, – вторую вершину, а третья тройка, `4/3/1`, – третью вершину. Каждая такая тройка – это индекс вершины, индекс текстурной вершины и индекс вектора нормали. Например, `2/1/1` определяет вершину, геометрическое местоположение которой определяется во второй строке, начинающейся с `v`, цвет определяется

в первой строке, начинающейся с vt, и вектор нормали определяется в первой строке, начинающейся с vn.

Сопровождающий файл cube_texture.mtl выглядит, как показано ниже. В нем начинающаяся с ключевого слова map_Kd строка объявляет текстурное изображение. Здесь wal67ar_small.jpg – это файл RGB-изображения 250×250 в той же папке, что и MTL-файл:

```
newmtl Skin
Ka 0.200000 0.200000 0.200000
Kd 0.827451 0.792157 0.772549
Ks 0.000000 0.000000 0.000000
Ns 0.000000
map_Kd ./wal67ar_small.jpg
```

Опять же, для загрузки находящейся в файле cube_texture.obj полигональной сетки можно использовать пакет Open3D и библиотеку PyTorch3D – например, используя следующий ниже файл obj_example2.py:

```
import open3d
from pytorch3d.io import load_obj
import torch

mesh_file = 'cube_texture.obj'

print('Визуализация полигональной сетки с помощью Open3D')
mesh = open3d.io.read_triangle_mesh(mesh_file)
open3d.visualization.draw_geometries([mesh],
                                     mesh_show_wireframe=True,
                                     mesh_show_back_face=True)
print('Загрузка того же файла с помощью PyTorch3D')
vertices, faces, aux = load_obj(mesh_file)
print('Тип vertices = ', type(vertices))
print('Тип faces = ', type(faces))
print('Тип aux = ', type(aux))
print('vertices = ', vertices)
print('faces = ', faces)
print('aux = ', aux)

texture_images = getattr(aux, 'texture_images')
print('texture_images type = ', type(texture_images))
print(texture_images['Skin'].shape)
```

Результат работы фрагмента исходного кода файла Python obj_example2.py должен выглядеть, как показано ниже:

```
Визуализация полигональной сетки с помощью Open3D
Загрузка того же файла с помощью PyTorch3D
Тип vertices = <class 'torch.Tensor'>
Тип faces = <class 'pytorch3d.io.obj_io.Faces'>
Тип aux = <class 'pytorch3d.io.obj_io.Properties'>
vertices = tensor([[ 1.0000, -1.0000, -1.0000],
                   [ 1.0000, -1.0000,  1.0000],
```

```
                  [-1.0000, -1.0000,  1.0000],
                  [-1.0000, -1.0000, -1.0000],
                  [ 1.0000, 1.0000, -1.0000],
                  [ 1.0000, 1.0000,  1.0000],
                  [-1.0000, 1.0000,  1.0000],
                  [-1.0000, 1.0000, -1.0000]])
faces = Faces(verts_idx=tensor([
                  [1, 2, 3],
                  [7, 6, 5],
                  [4, 5, 1],
                  [5, 6, 2],
                  [2, 6, 7],
                  [0, 3, 7],
                  [0, 1, 3],
                  ...
                  [3, 3, 3],
                  [4, 4, 4],
                  [5, 5, 5]]),
              textures_idx=tensor([
                  [ 0, 1, 2],
                  [ 0, 3, 4],
                  [ 5, 6, 7],
                  [ 7, 4, 3],
                  [ 8, 9, 10],
                  [11, 12, 10],
                  ...
                  [12, 8, 10],
                  [ 5, 11, 10]]),
              materials_idx=tensor([0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0]))
Aux = Properties(
           normals=tensor([[ 0., -1.,  0.],
                           [ 0., 1., 0.],
                           [ 1., 0., 0.],
                           [-0., 0., 1.],
                           [-1.,-0.,-0.],
                           [ 0., 0.,-1.]]),
           verts_uvs=tensor([[1.0000, 0.3333],
                             ...
                             [0.3333, 0.6667],
                             [1.0000, 0.0000]]),
           material_colors={
             'Skin': {'ambient_color': tensor([0.2000, 0.2000, 0.2000]),
             'diffuse_color': tensor([0.8275, 0.7922, 0.7725]),
             'specular_color': tensor([0., 0., 0.]),
             'shininess': tensor([0.])}},

          texture_images={'Skin': tensor([[[0.2078, 0.1765, 0.1020],
                                           [0.2039, 0.1725, 0.0980],
                                           [0.1961, 0.1647, 0.0902],
                                           ...,
                                           [0.2235, 0.1882, 0.1294]]])},
```

```
          texture_atlas=None)
texture_images type = <class 'dict'>
Skin
torch.Size([250, 250, 3])
```

Примечание

Это не полная распечатка; обратитесь к полной распечатке во время выполнения исходного кода.

По сравнению с распечаткой фрагмента исходного кода в файле `obj_example1.py` приведенная выше распечатка имеет следующие отличия.

- Поля `normals_idx` и `textures_idx` переменной `faces` теперь содержат валидные индексы вместо значения `-1`.
- Поле `normals` переменной `aux` теперь является тензором PyTorch, а не `None`.
- Поле `verts_uvs` переменной `aux` теперь является тензором PyTorch, а не `None`.
- Поле `texture_images` переменной `aux` больше не является пустым словарем. Словарь `texture_images` содержит одну запись с ключом `Skin` и тензором PyTorch с очертанием (250, 250, 3). Этот тензор точно такой же, как и содержащееся в файле `wal67ar_small.jpg` изображение, согласно определению в файле `mtl_texture.mtl`.

Мы научились использовать базовые форматы файлов 3D-данных, а также файлы PLY и OBJ. В следующем далее разделе вы изучите базовые понятия систем 3D-координат.

Понятие системы 3D-координат

В этом разделе вы познакомитесь с часто используемыми в библиотеке PyTorch3D системами координат. Данный раздел является адаптированной версией документации PyTorch по системам координат камеры: https://pytorch3d.org/docs/cameras. Для того чтобы понять и применять принятую в библиотеке PyTorch3D систему отрисовки, обычно необходимо знать эти системы координат и уметь их использовать. Как обсуждалось в предыдущих разделах, 3D-данные могут быть представлены точками, гранями и вокселами. Местоположение каждой точки может быть представлено набором координат *x*, *y* и *z* относительно определенной системы координат. Обычно требуется задать и использовать несколько систем координат, в зависимости от того, какая из них наиболее удобна.

Первая часто используемая система координат называется **мировой системой координат**. Эта система координат представляет собой систему 3D-координат, выбранную по отношению ко всем 3D-объектам, в результате чего местоположение 3D-объектов легко определяется. Обычно ось мировой системы координат не согласуется с ориентацией объекта или камеры. И поэтому между началом мировой системы координат и ориентациями объек-

та и камеры существуют ненулевые повороты и смещения. Ниже приведен рис. 1.3, показывающий мировую систему координат.

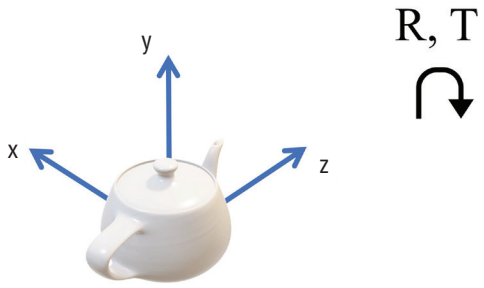


Рис. 1.2 ❖ Мировая система координат, начало координат и ось которой определяются независимо от позиций камеры

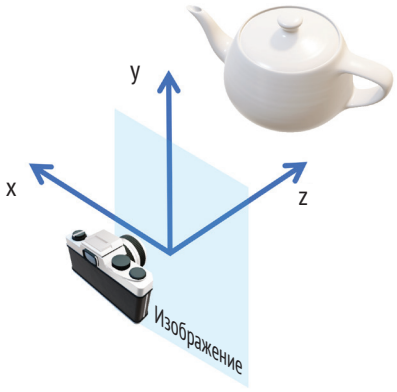


Рис. 1.3 ❖ Система координат поля зрения камеры, начало которой находится в центре проекции камеры, а три оси определяются в соответствии с плоскостью изображения

Поскольку ось мировой системы координат обычно не согласуется с ориентацией камеры, во многих ситуациях удобнее определять и использовать систему координат поля зрения камеры. В PyTorch3D система координат поля зрения камеры задается таким образом, что ее начало координат находится в точке проецирования камеры, ось *x* указывает влево, ось *y* указывает вверх, а ось *z* указывает вперед[1] (рис. 1.4).

Нормализованная координата устройства (NDC)[2] ограничивает объем, который камера может передавать. Значения координат *x* в пространстве NDC

---

[1] Иными словами, согласно документации PyTorch3D, система координат поля зрения камеры (Camera view coordinate system) – это система, начало которой находится в плоскости изображения, а ось *z* перпендикулярна плоскости изображения. – *Прим. перев.*

[2] От англ. *normalized device coordinate. – Прим. перев.*

варьируются в интервале от –1 до +1, как и значения координат *y*. Значения координат *z* варьируются в интервале от znear до zfar, где znear – это самая близкая глубина, а zfar – самая дальняя глубина. Любой объект вне этого диапазона между znear и zfar не будет передаваться камерой.

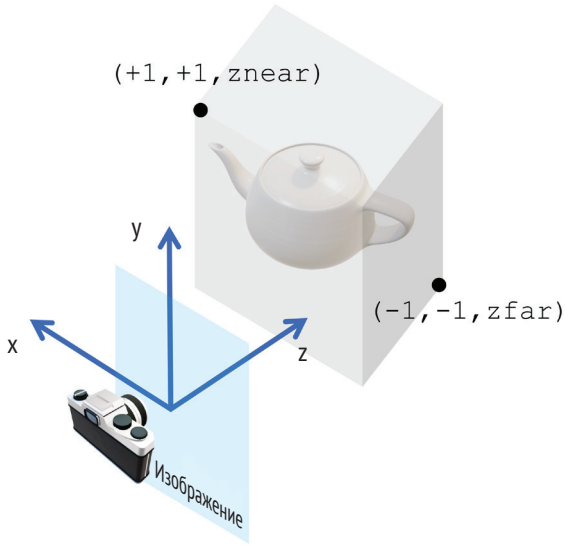


Рис. 1.4 ❖ Система координат NDC, где объем ограничен диапазонами, которые камера может передавать

Наконец, система координат экрана определяется с точки зрения того, как изображения отображаются на экранах. Данная система координат содержит координату *x* в виде столбцов пикселов, координату *y* в виде строк пикселов, а координата *z* соответствует глубине объекта.

В целях правильной отрисовки 3D-объекта на 2D-экране необходимо переключаться между этими системами координат. К счастью, такие конверсии легко выполняются с помощью применяемых в PyTorch3D моделей камеры. Мы обсудим конверсию координат подробнее после того, как рассмотрим модели камеры.

Понятие модели камеры

В этом разделе вы познакомитесь с моделями камеры. В трехмерном глубоком обучении 2D-изображения обычно используются для 3D-обнаружения. 3D-информация обнаруживается исключительно по 2D-изображениям, либо в целях получения высокой точности 2D-изображения смешиваются с глубиной. Тем не менее модели камеры необходимы для выстраивания соответствия между 2D-пространством и 3D-миром.

В библиотеке PyTorch3D есть две главнейшие модели камеры, ортографическая камера, определенная в классе `OrthographicCameras`, и перспективная камера, определенная в классе `PerspectiveCameras`. На следующем ниже рисунке показаны различия между этими двумя моделями камеры.

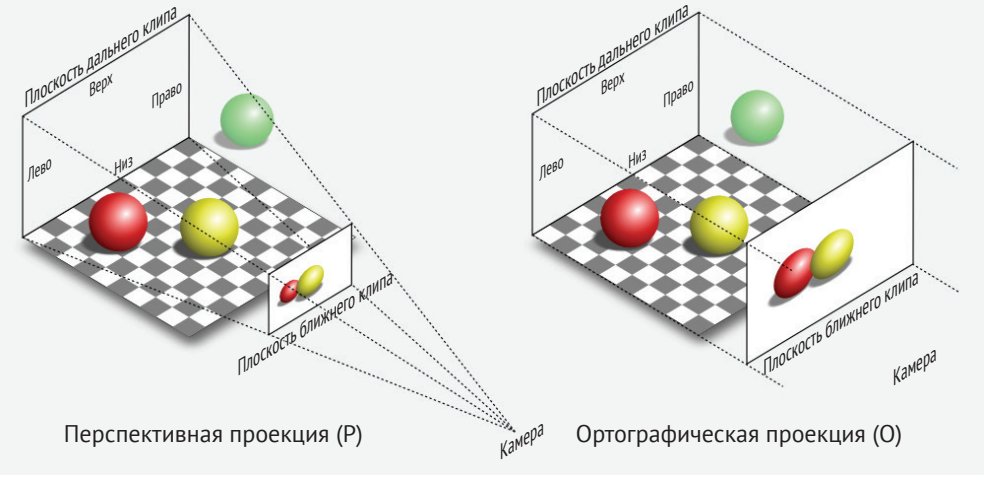


Рис. 1.5 ❖ Две реализованные в PyTorch3D главнейшие модели камеры: перспективная и ортографическая

В ортографических камерах используются ортографические проекции, чтобы отображать объекты 3D-мира на 2D-изображенияи, тогда как в перспективных камерах используются перспективные проекции, чтобы отображать объекты 3D-мира на 2D-изображения. Ортографические проекции отображают объекты на 2D-изображения, игнорируя глубину объекта. Например, как показано на рисунке, два объекта с одинаковым геометрическим размером на разных глубинах будут отображены на 2D-изображения одинакового размера. С другой стороны, в перспективных проекциях если объект отошел от камеры далеко, то на 2D-изображении он будет отображен на меньший размер.

Теперь, когда вы познакомились с базовым понятием моделей камеры, давайте рассмотрим несколько примеров программирования, чтобы понять, как создавать и использовать эти модели камеры.

Пример программирования моделей камеры и систем координат

В этом разделе мы применим все то, чему вы научились, чтобы разработать конкретную модель камеры и выполнять конверсию между разными системами координат, используя конкретный пример исходного кода, написанный на Python с использованием PyTorch3D.

1. Прежде всего будем использовать следующую ниже полигональную сетку, определенную в OBJ-файле `cube.obj`. По сути, полигональная сетка – это куб:

```
mtllib ./cube.mtl
o cube
# Список вершин
v -50 -50 20
v -50 -50 10
v -50 50 10
v -50 50 20
v 50 -50 20
v 50 -50 10
v 50 50 10
v 50 50 20

# Список точек/линий/граней
usemtl Door
f 1 2 3
f 6 5 8
f 7 3 2
f 4 8 5
f 8 4 3
f 6 2 1
f 1 3 4
f 6 8 7
f 7 2 6
f 4 5 1
f 8 3 7
f 6 1 5
# Конец файла
```

   Пример фрагмента исходного кода находится в файле Python `camera.py`, который можно скачать из репозитория книги на GitHub.
2. Далее импортируем все необходимые модули:

```
import open3d
import torch
import pytorch3d
from pytorch3d.io import load_obj
from scipy.spatial.transform import Rotation as Rotation

from pytorch3d.renderer.cameras import PerspectiveCameras
```

3. Загружаем и визуализируем полигональную сетку с использованием функции Open3D `draw_geometrics`:

```
# Загрузить полигональную сетку и визуализировать с помощью Open3D
mesh_file = "cube.obj»
print('Визуализация полигональной сетки с помощью Open3D')
mesh = open3d.io.read_triangle_mesh(mesh_file)
open3d.visualization.draw_geometries([mesh],
                                     mesh_show_wireframe=True,
                                     mesh_show_back_face=True)
```

4. Переменная `camera` определяется как объект PyTorch3D `PerspectiveCamera`. Камера здесь на самом деле мини-пакетирована. Например, матрица поворота, R, представляет собой тензор PyTorch с очертанием `[8, 3, 3]`, который фактически задает восемь камер, причем каждая с одной из восьми матриц поворота. Это касается всех других параметров камеры, таких как размеры изображений, фокусные расстояния и фокальные точки:

```
# Определить мини-пакет из 8 камер
image_size = torch.ones(8, 2)
image_size[:,0] = image_size[:,0] * 1024
image_size[:,1] = image_size[:,1] * 512
image_size = image_size.cuda()

focal_length = torch.ones(8, 2)
focal_length[:,0] = focal_length[:,0] * 1200
focal_length[:,1] = focal_length[:,1] * 300
focal_length = focal_length.cuda()

principal_point = torch.ones(8, 2)
principal_point[:,0] = principal_point[:,0] * 512
principal_point[:,1] = principal_point[:,1] * 256
principal_point = principal_point.cuda()

R = Rotation.from_euler('zyx', [
                    [n*5, n, n]  for n in range(-4, 4, 1)],
degrees=True).as_matrix()
R = torch.from_numpy(R).cuda()
T = [ [n, 0, 0] for n in range(-4, 4, 1)]
T = torch.FloatTensor(T).cuda()

camera = PerspectiveCameras(focal_length=focal_length,
                            principal_point=principal_point,
                            in_ndc=False,
                            image_size=image_size,
                            R=R,
                            T=T,
                            device='cuda')
```

5. После определения переменной `camera` можно вызвать метод `get_world_to_view_transform` класса, чтобы получить объект класса `Transform3D` под названием `world_to_view_transform`[1]. Затем можно применить метод `transform_points`, чтобы выполнить конвертацию из мировой системы координат в систему координат поля зрения камеры. Метод `get_full_projection_transform`[2] применяется точно так же, чтобы получить объект класса `Transform3D` под названием `world_to_screen_transform`[3], который

[1] Трансформанта мировой системы в систему координат обзора камеры. – *Прим. перев.*

[2] Получить трансформанту полной проекции. – *Прим. перев.*

[3] Трансформанта мировой системы в систему координат экрана. – *Прим. перев.*

предназначен для конвертации из мировой системы координат в систему координат экрана:

```
world_to_view_transform = camera.get_world_to_view_transform()
world_to_screen_transform = camera.get_full_projection_transform()

# Загрузить полигональные сетки с помощью PyTorch3D
vertices, faces, aux = load_obj(mesh_file)
vertices = vertices.cuda()

world_to_view_vertices = world_to_view_transform. \
    transform_points(vertices)
world_to_screen_vertices = world_to_screen_transform. \
    transform_points(vertices)
print('world_to_view_vertices = ', world_to_view_vertices)
print('world_to_screen_vertices = ', world_to_screen_vertices)
```

В примере исходного кода показаны базовые способы применения камер библиотеки PyTorch3D и проиллюстрирована легкость, с которой можно переключаться между разными системами координат с использованием данной библиотеки.

Резюме

В этой главе вы сначала научились формировать среду разработки. Затем мы поговорили о наиболее широко используемых представлениях 3D-данных. Затем вы проинспектировали несколько конкретных примеров представления 3D-данных, изучив форматы файлов 3D-данных, формат PLY и формат OBJ. Затем вы познакомились с базовыми понятиями систем 3D-координат и моделей камеры. В последней части главы вы научились создавать модели камеры и конвертировать разные системы координат с помощью практического примера программирования.

В следующей главе мы поговорим о более важных понятиях трехмерного глубокого обучения, таких как отрисовка, чтобы конвертировать 3D-модели в 2D-изображения, разнородное мини-пакетирование и несколько способов представления поворотов.

Глава 2

Введение в трехмерное компьютерное зрение и геометрию

В этой главе вы познакомитесь с несколькими базовыми понятиями трехмерного компьютерного зрения и геометрии, которые будут особенно полезны в последующих главах этой книги. Мы начнем с обсуждения того, что такое отрисовка, растеризация и затенение. Мы рассмотрим различные модели освещения и модели затенения, такие как точечные источники света, направленные источники света, окружающее освещение, рассеивание, блики и блеск. Мы рассмотрим пример исходного кода с отрисовкой сеточной модели с использованием различных моделей освещения и параметров.

Затем вы научитесь использовать библиотеку PyTorch для решения задач оптимизации. В частности, мы пройдемся по стохастическому градиентному спуску на разнородных мини-пакетах, что становится возможным при использовании библиотеки PyTorch3D. Вы также узнаете о различных форматах мини-пакетов, которые используются в бибилотеке PyTorch3D, в том числе о списковом, дополненном и упакованном форматах, и научитесь выполнять конверсию между разными форматами.

В последней части главы мы обсудим несколько часто используемых представлений поворотов и способы конверсии между этими представлениями.

В данной главе будут охвачены следующие ниже главные темы:

- ознакомление с базовыми понятиями отрисовки, растеризации и затенения,
- концепции модели затенения по Ламберту и модели освещения по Фонгу,
- ознакомление с тем, как задавать тензор PyTorch и оптимизировать его с помощью оптимизатора,
- ознакомление с техникой определения мини-пакета и разнородного мини-пакета в частности, а также с упакованным и дополненным тензорами,

- повороты и разные способы описания поворотов,
- экспоненциальное отображение и логарифмическое отображение в пространстве SE(3).

Технические требования

Для выполнения примеров исходного кода этой книги в идеале понадобится компьютер с графическим процессором. Тем не менее для выполнения фрагментов исходного кода вполне будет достаточно только центрального процессора(ов).

Рекомендуемая компьютерная конфигурация включает следующее:

- GPU, такой как серия GTX или серия RTX с не менее 8 Гб памяти,
- Python 3,
- библиотеки PyTorch и PyTorch3D.

Фрагменты исходного кода к этой главе находятся по адресу https://github.com/packtpublishing/3d-deep-learning-with-python.

Ознакомление с базовыми понятиями отрисовки, растеризации и затенения

Отрисовка, или рендеринг, – это процесс, который на входе принимает модели 3D-данных мира вокруг камеры и на выходе выдает изображения[1]. Это аппроксимация физического процесса, при котором изображения формируются внутри камеры в реальном мире. В типичном случае модели 3D-данных представляют собой полигональные сетки. И в этом случае отрисовка обычно выполняется с использованием трассировки лучей (рис. 2.1).

Пример обработки трассировки лучей показан на рис. 2.1. В указанном примере модель мира содержит одну 3D-сферу, которая представлена сеточной моделью. Для того чтобы сформировать изображение 3D-сферы, по каждому пикселу изображения генерируется один луч, начинающийся из начальной точки камеры и проходящий через пиксел изображения. Если один луч пересекается с одной гранью сетки, то мы знаем, что грань сетки может проецировать свой цвет на пиксел изображения. Кроме того, необходимо прослеживать глубину каждого пересечения, потому что грань с меньшей глубиной будет загораживать грани с большей глубиной.

Таким образом, процесс отрисовки обычно можно подразделить на два этапа – растеризацию и затенение. Процесс трассировки лучей – это ти-

[1] Для справки: отрисовка (rendering) – это синтез изображения, трактуемый как процесс генерирования изображения из 2D- или 3D-модели с помощью компьютерной программы. Иными словами, это воспроизведение формы, видимости и внешнего вида объектов, наблюдаемых с заданного ракурса. – *Прим. перев.*

пичный процесс растеризации, т. е. процесс отыскания соответствующих геометрических объектов по каждому пикселу изображения. Затенение, или шейдинг, – это процесс получения результатов растеризации и вычисления значения каждого пиксела изображения[1].

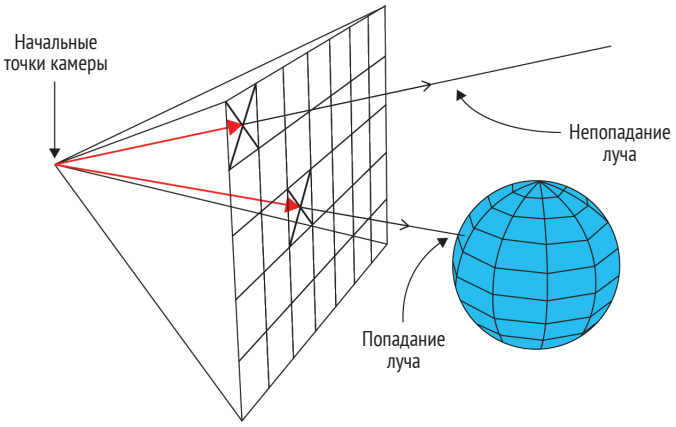


Рис. 2.1 ❖ Отрисовка с помощью трассировки лучей (лучи генерируются из начальной точки[2] камеры и проходят через пикселы изображения, чтобы отыскать грани соответствующей полигональной сетки)

Функция `pytorch3d.renderer.mesh.rasterize_meshes` в библиотеке PyTorch3D по каждому пикселу изображения обычно вычисляет следующие четыре величины:

- `pix_to_face` – список индексов граней, которые могут пересекаться лучом,
- `zbuf` – список значений глубины этих граней,
- `bary_coords` – список барицентрических координат точки пересечения каждой грани и луча,
- `pix_dists` – список расстояний со знаком между пикселами (*x* и *y*) и ближайшей точкой на всех гранях, в которых есть пересечение луча. Значения этого списка могут отрицательными, поскольку он содержит расстояния со знаком.

Обратите внимание, что обычно одна грань с наименьшей глубиной загораживает все грани полигональной сетки с большей глубиной. Таким образом, если требуется лишь это отрисованное изображение, то в этом списке

---

[1] Для справки: затенение (shading) – это метод *имитации освещения* в 3D-отрисовке и состоит в назначении освещения каждой грани объекта. Например, по Ламберту освещение пропорционально углу между нормалью к грани, которая оценивается как плоская, и направлением источника света, освещающего сцену. – *Прим. перев.*

[2] Или начала координат. – *Прим. перев.*

нужна только грань с наименьшей глубиной. Однако в условиях более продвинутой дифференцируемой отрисовки (которую мы рассмотрим в следующих главах этой книги) значения цвета пикселов обычно смешиваются на основе нескольких граней полигональной сетки.

Понятие барицентрических координат

Координаты каждой точки, компланарной к грани полигональной сетки, всегда можно записать как линейную комбинацию координат трех вершин грани полигональной сетки. Например, как показано на следующей ниже диаграмме, точка p может быть записана как $uA + vB + wC$, где A, B и C – это координаты трех вершин грани полигональной сетки. Таким образом, каждую такую точку можно представить коэффициентами u, v и w. Это представление называется барицентрическими координатами точки. Для точки, лежащей внутри треугольника грани сетки, $u + v + w = 1$, причем u, v, w – это положительные числа. Поскольку барицентрические координаты определяют любую точку внутри грани как функцию вершин грани, те же коэффициенты можно использовать для интерполяции других свойств по всей грани в зависимости от свойств, определенных в вершинах грани. Например, их можно использовать для затенения, как показано на рис. 2.2.

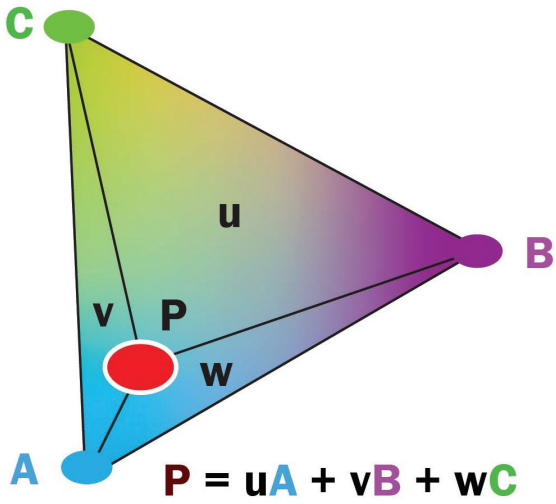


Рис. 2.2 ❖ Определение барицентрической системы координат

Имея список значений величин `pix_to_face`, `zbuf`, `bary_coords` и `dists`, процесс затенения будет имитировать физический процесс формирования изображения, как и в реальном мире. И следовательно, мы собираемся обсудить несколько физических моделей формирования цвета.

Модели источника света

Распространение света в реальном мире – технически сложный процесс. В целях снижения вычислительных затрат обычно в затенении используется несколько аппроксимаций источников света:

- первым допущением является окружающее освещение, при котором мы исходим из того, что после достаточного числа отражений имеется некоторое фоновое световое излучение, вследствие чего оно обычно исходит со всех направлений с почти одинаковой амплитудой во всех пикселах изображения;
- еще одно обычно используемое допущение состоит в том, что некоторые источники света можно считать точечными источниками света. Точечный источник света излучает свет из одной точки, и излучения во всех направлениях имеют одинаковый цвет и амплитуду;
- третье обычно используемое допущение заключается в том, что некоторые источники света можно моделировать как направленные. В таком случае направления света от источника света идентичны во всех трехмерных пространственных местоположениях. Направленное освещение является неплохой аппроксимационной моделью в случаях, когда источники света находятся далеко от отрисовываемых объектов – например, солнечного света.

Концепция модели затенения по Ламберту

Первая физическая модель, которую мы обсудим, будет закон косинусов Ламберта. Ламбертовы поверхности – это типы объектов, которые совсем не блестят, например бумага, необработанное дерево и неполированные камни:

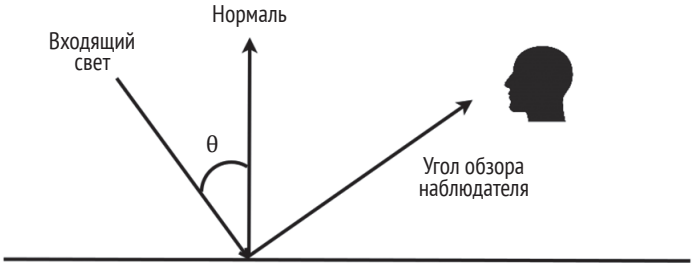


Рис. 2.3 ❖ Рассеяние света на ламбертовых поверхностях

На рис. 2.3 показан пример рассеяния света на ламбертовой поверхности. Одна из основных идей закона косинусов Ламберта состоит в том, что для ламбертовых поверхностей амплитуда отраженного света не зависит от угла обзора, а зависит только от угла θ между нормалью к поверхности и направлением падающего света. Точнее, интенсивность отраженного света *c* выглядит следующим образом:

$c = c_r c_l \cos(\theta)$.

Здесь c_r – это коэффициент отражения материала, а c_l – амплитуда падающего света. Если далее учесть окружающий свет, то амплитуда отраженного света будет следующей:

$c = c_r(c_a + c_l \cos(\theta))$.

Здесь c_a – это амплитуда окружающего света.

Концепция модели освещения по Фонгу

На блестящих поверхностях, таких как полированные кафельные полы и глянцевая краска, отраженный свет также содержит бликовую компоненту. Модель освещения по Фонгу часто используется для таких глянцевых компонент:

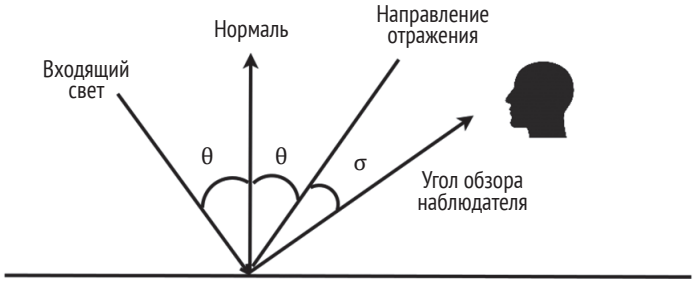


Рис. 2.4 ❖ Модель освещения по Фонгу

Пример модели освещения по Фонгу показан на рис. 2.4. Один из основных принципов фонговской модели освещения заключается в том, что яркая компонента света должна быть наиболее сильной в направлении отражения падающего света. Компонента становится слабее по мере увеличения угла *c* между направлением отражения и углом обзора.

Точнее, амплитуда блестящего светового компонента *c* равна следующему:

$c = c_r c_l c_p [\cos(\sigma)]^p$.

Здесь экспонента *p* является параметром модели и служит для контроля над скоростью, с которой блестящие компоненты затухают, когда угол обзора отклоняется от направления отражения.

Наконец, если учесть все три главнейших компоненты – окружающее освещение, рассеивание и блики, – окончательное уравнение амплитуды света будет следующим:

$c = c_r(c_a + c_l \cos(\theta) + c_l c_p [\cos(\sigma)]^p)$.

Обратите внимание, что приведенное выше уравнение применяется к каждой цветовой компоненте. Другими словами, у нас будет одно из этих

уравнений для каждого цветового канала (красного, зеленого и синего) с разным набором значений c_r, c_a, c_l:

Теперь, когда вы познакомились с базовыми понятиями отрисовки, растеризации и затенения, а также различными моделями источника света и моделями затенения/освещения, вы готовы выполнить несколько примеров программирования, чтобы применить эти источники света и модели затенения/освещения.

Пример программирования 3D-отрисовки

В этом разделе мы рассмотрим конкретный пример программирования с использованием библиотеки PyTorch3D на отрисовку сеточной модели. Исходный код примера находится в файле Python `rendering_basic.py` репозитория книги на GitHub. Вы научитесь задавать модель камеры и источник света с использованием PyTorch3D. Вы также научитесь изменять компоненты входящего света и свойства материала, чтобы отрисовывать более реалистичные изображения, управляя тремя компонентами света (окружающим, рассеивающим и глянцевым).

1. Вначале необходимо импортировать все требующиеся модули Python:

```
import open3d
import os
import sys
import torch

import matplotlib.pyplot as plt

# Функция-утилита для загрузки полигональных сеток
from pytorch3d.io import load_objs_as_meshes

# Структуры данных и функции для отрисовки
from pytorch3d.renderer import (
    look_at_view_transform,
    PerspectiveCameras,
    PerspectiveCameras,
    PointLights,
    Materials,
    RasterizationSettings,
    MeshRenderer,
    MeshRasterizer
)

from pytorch3d.renderer.mesh.shader import HardPhongShader

sys.path.append(os.path.abspath(''))
```

2. Затем загружаем используемую в примере полигональную сетку. OBJ-файл `cow.obj` содержит сеточную модель объекта игрушечной коровы:

```
# Задать пути
DATA_DIR = "./data»
obj_filename = os.path.join(DATA_DIR, "cow_mesh/cow.obj»)
device = torch.device('cuda')

# Загрузить OBJ-файл
mesh = load_objs_as_meshes([obj_filename], device=device)
```

3. Далее задаем камеры и источники света. Здесь используется функция `look_at_view_transform`, чтобы отобразить простые для понимания параметры, такие как расстояние от камеры, угол возвышения и угол азимута, чтобы получить матрицы поворота (R) и трансляции[1] (T). Переменные `R` и `T` определяют место, в котором мы собираемся разместить камеру. Переменная `lights` представляет собой точечный источник света, помещенный в `[0.0, 0.0, -3.0]` в качестве его местоположения:

```
R, T = look_at_view_transform(2.7, 0, 180)
cameras = PerspectiveCameras(device=device, R=R, T=T)
lights = PointLights(device=device, location=[[0.0, 0.0, -3.0]])
```

4. Затем задаем переменную `renderer`[2] типа `MeshRenderer`. Переменная `renderer` – это вызываемый объект, который на входе может принимать полигональную сетку и на выходе выводить отрисованные изображения. Обратите внимание, что переменная `renderer` при инициализации принимает два входных параметра – растеризатор и затенитель. В библиотеке PyTorch3D определено несколько разных типов растеризаторов и затенителей. Здесь мы будем использовать `MeshRasterizer`[3] и `HardPhongShader`[4]. Обратите внимание, что, помимо этого, можно задать настройку растеризатора. Здесь значение параметра `image_size` задано равным `512`, и, следовательно, отрисованные изображения будут иметь размер 512×512 пикселов. Значение параметра `blur_radius`[5] задано равным `0`, и значение параметра `face_per_pixel`[6] задано равным `1`. Параметры `blur_radius` и `face_per_pixel` наиболее полезны для дифференцируемой отрисовки, в которой параметр `blur_radius` должен быть больше `0`, а параметр `face_per_pixel` – больше `1`:

```
raster_settings = RasterizationSettings(
    image_size=512,
    blur_radius=0.0,
    faces_per_pixel=1,
)

renderer = MeshRenderer(
```

---

1 Син. прямолинейный перенос (объекта из одной позиции в другую). – *Прим. перев.*
2 Отрисовщик. – *Прим. перев.*
3 Сеточный растеризатор. – *Прим. перев.*
4 Плотный затенитель по Фонгу. – *Прим. перев.*
5 Радиус размытия. – *Прим. перев.*
6 Грани в расчете на пиксел. – *Прим. перев.*

```
        rasterizer=MeshRasterizer(
            cameras=cameras,
            raster_settings=raster_settings
        ),
        shader = HardPhongShader(
            device=device,
            cameras=cameras,
            lights=lights
        )
    )
```

5. Теперь все готово к тому, чтобы выполнить первую отрисовку, вызвав отрисовщика и передав сеточную модель. Отрисованное изображение показано на рис. 2.5:

```
images = renderer(mesh)

plt.figure(figsize=(10, 10))
plt.imshow(images[0, ..., :3].cpu().numpy())
plt.axis("off")
plt.savefig('light_at_front.png')
plt.show()
```

Рис. 2.5 ❖ Отрисовка изображения, когда источник света находится впереди

6. Далее изменим местоположение источника света на заднюю часть полигональной сетки и посмотрим, что получится. Отрисованноее изображение показано на рис. 2.6. В этом случае свет от точечного источника света не может пересекаться ни с какими обращенными к нам гранями сетки. И поэтому все цвета, которые мы здесь можем наблюдать, обусловлены окружающим светом:

```
lights.location = torch.tensor([0.0, 0.0, +1.0],
                               device=device)[None]
```

```
images = renderer(mesh, lights=lights, )

plt.figure(figsize=(10, 10))
plt.imshow(images[0, ..., :3].cpu().numpy())
plt.axis("off")
plt.savefig('light_at_back.png')
plt.show()
```

Рис. 2.6 ❖ Отрисовка изображения,
когда источник света находится позади игрушечной коровы.

7. В следующем далее эксперименте зададим структуру данных `materials`. Здесь мы изменяем конфигурацию так, чтобы окружающие компоненты были близки к `0` (на самом деле они равны `0.01`). Поскольку точечный источник света находится позади объекта, а окружающий свет также выключен, отрисованный объект теперь не отражает свет. Отрисованное изображение показано на рис. 2.7:

```
# Задать зеркальный цвет, блеск материала и
# цвет окружающих компонент
materials = Materials(
    device=device,
    specular_color=[[0.0, 1.0, 0.0]],
    shininess=10.0,
    ambient_color=((0.01, 0.01, 0.01),),
)

images = renderer(mesh, lights=lights, materials=materials)

plt.figure(figsize=(10, 10))
plt.imshow(images[0, ..., :3].cpu().numpy())
plt.axis("off")
plt.savefig('dark.png')
plt.show()
```

Рис. 2.7 ❖ Отрисованное изображение без окружающего света и точечного источника света за игрушечной коровой

8. В следующем эксперименте снова повернем камеру и переопределим положение источника света так, чтобы свет падал на морду коровы. Обратите внимание, что при определении материала мы устанавливаем значение параметра `shininess`[1] равным `10.0`. Параметр `shininess` в точности является параметром p в рамках фонговской модели освещения. Значение параметра `specular_color`[2] равно `[0.0, 1.0, 0.0]`, и, следовательно, поверхность блестит в основном в зеленой компоненте. Результаты отрисовки показаны на рис. 2.8:

```
R, T = look_at_view_transform(dist=2.7, elev=10, azim=-150)
cameras = PerspectiveCameras(device=device, R=R, T=T)
lights.location = torch.tensor([[2.0, 2.0, -2.0]], device=device)

# Изменить зеркальный цвет на зеленый и
# изменить блеск материала
materials = Materials(
    device=device,
    specular_color=[[0.0, 1.0, 0.0]],
    shininess=10.0
)

# Повторить отрисовку полигональной сетки,
# передав именованные аргументы для измененных компонент.
images = renderer(mesh, lights=lights,
                  materials=materials, cameras=cameras)

plt.figure(figsize=(10, 10))
```

[1] Блеск. – *Прим. перев.*

[2] Зеркальный цвет. – *Прим. перев.*

```
plt.imshow(images[0, ..., :3].cpu().numpy())
plt.axis("off")
plt.savefig('green.png')
plt.show()
```

Рис. 2.8 ❖ Отрисованное изображение с компонентами зеркального освещения

9. В следующем эксперименте мы поменяем значение параметра `specular_color` на красный и увеличим значение параметра `shininess`. Результаты показаны на рис. 2.9:

```
# Изменить зеркальный цвет на красный и
# изменить блеск материала
materials = Materials(
    device=device,
    specular_color=[[1.0, 0.0, 0.0]],
    shininess=20.0
)

# Снова повторить отрисовку полигональной сетки
images = renderer(mesh, lights=lights,
                  materials=materials, cameras=cameras)

plt.figure(figsize=(10, 10))
plt.imshow(images[0, ..., :3].cpu().numpy())
plt.savefig('red.png')
plt.axis("off")
plt.show()
```

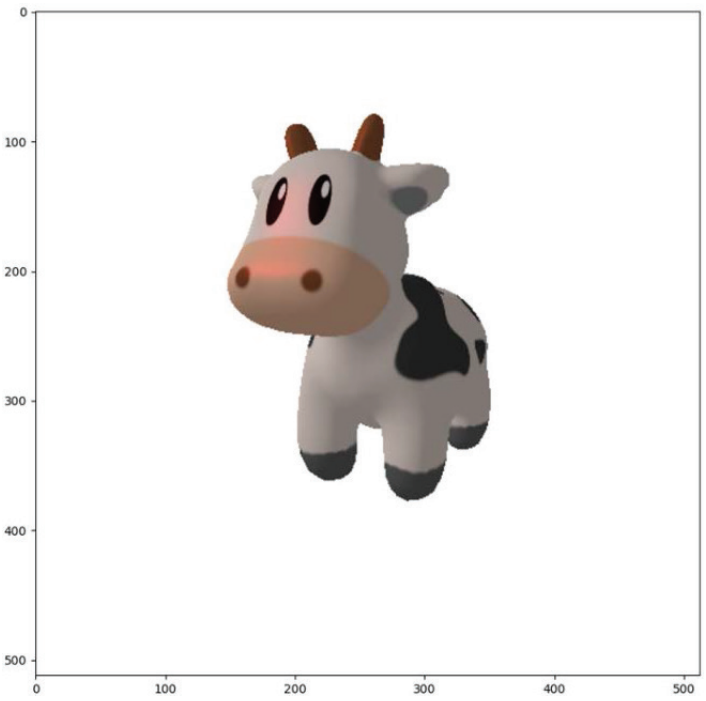


Рис. 2.9 ❖ Отрисованное изображение с красным зеркальным цветом

10. Наконец, мы отключаем блеск, и результаты показаны на рис. 2.10:

```
# Отключить блеск материала
materials = Materials(
    device=device,
    specular_color=[[0.0, 0.0, 0.0]],
    shininess=0.0
)

# Снова повторить отрисовку
images = renderer(mesh, lights=lights,
                  materials=materials, cameras=cameras)

plt.figure(figsize=(10, 10))
plt.imshow(images[0, ..., :3].cpu().numpy())
plt.savefig('blue.png')
plt.axis("off")
plt.show()
```

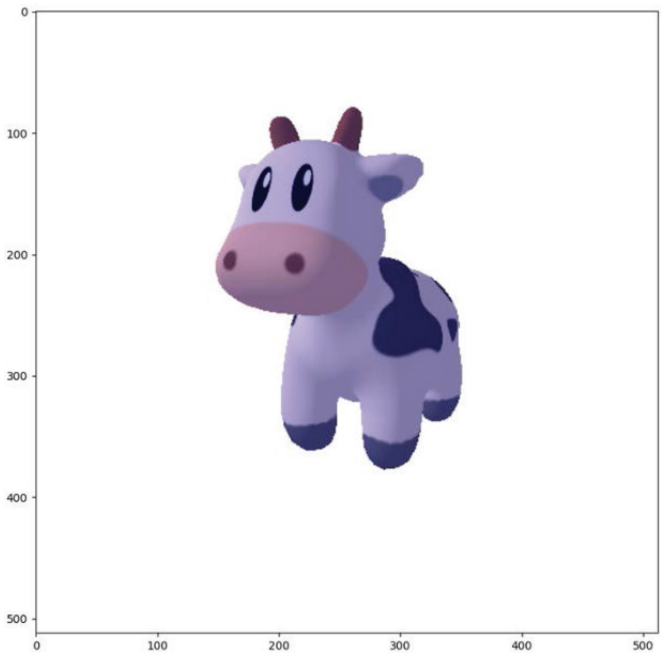


Рис. 2.10 ❖ Отрисованное изображение без зеркального отражения

В первой части этой главы мы в основном обсуждали отрисовку и затенение, которые очень важны для трехмерного компьютерного зрения. Далее мы обсудим еще одну очень важную для трехмерного глубокого обучения тему, а именно проблему оптимизации разнородных пакетов.

Использование разнородных пакетов данных в библиотеке PyTorch3D и оптимизаторов PyTorch

В этом разделе вы научитесь использовать оптимизатор PyTorch на разнородных мини-пакетах библиотеки PyTorch3D. В глубоком обучении обычно задан список примеров данных наподобие следующих: $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_n, y_n)\}$. Здесь x_i – это наблюдения и y_i – значения предсказания. Например, x_i может быть каким-то изображением, а y_i – достоверным результатом классифицирования – например, «кошка» либо «собака». Затем глубокую нейронную сеть тренируют так, чтобы данные на выходе из нейронной сети были как можно ближе к y_i. Обычно функция потери между результатами на

выходе из нейронной сети и y_i определяется таким образом, чтобы значения функции потери уменьшались по мере приближения таких результатов нейронной сети к y_i.

Таким образом, тренировка сети глубокого обучения обычно выполняется путем минимизации функции потери, которая вычисляется на всех примерах тренировочных данных, x_i и y_i. Во многих алгоритмах оптимизации используется простой метод, который заключается в том, чтобы сначала вычислять градиенты, как показано в следующем ниже уравнении, а затем видоизменять параметры нейронной сети в направлении отрицательного градиента. В приведенном ниже уравнении f представляет нейронную сеть, которая на входе принимает x_i и имеет параметры θ; loss – это функция потери между данными на выходе их нейронной сети и достоверным предсказанием y_i:

$$\frac{\partial \sum_{i=1}^{n} \text{loss}(f(x_i;\theta), y_i)}{\partial \theta}.$$

Однако вычисление этого градиента обходится дорого, поскольку вычислительные затраты пропорциональны размеру тренировочного набора данных. На самом деле вместо изначального алгоритма градиентного спуска используется алгоритм **стохастического градиентного спуска** (SGD)[1]. В алгоритме SGD направление спуска вычисляется по следующему ниже уравнению:

$$\frac{\partial \sum_{d \in D} \text{loss}(f(x_k;\theta), y_k)}{\partial \theta}.$$

В приведенном выше уравнении так называемый мини-пакет D представляет собой малое подмножество всех примеров тренировочных данных. На каждой итерации мини-пакет D случайно отбирается из всех примеров тренировочных данных. Алгоритм SGD требует гораздо меньших вычислительных затрат, чем алгоритм градиентного спуска. По закону больших чисел рассчитанные направления спуска в SGD примерно близки к направлениям градиентного спуска. Также широко распространено мнение, что SGD вносит определенную неявную регуляризацию, которая способствует хорошим обобщающим свойствам глубокого обучения. Метод выбора размера мини-пакета является важным гиперпараметром, который необходимо тщательно учитывать. Так или иначе, алгоритм SGD и его варианты были предпочтительными методами тренировки моделей глубокого обучения.

Для многих типов данных, таких как изображения, данные можно легко делать однородными. Можно формировать мини-пакет изображений с одинаковой шириной, высотой и каналами. Например, мини-пакет из восьми изображений с тремя каналами (три цвета: красный, зеленый и синий), высотой 256 и шириной 256 можно преобразовать в тензор PyTorch с размерами 8×3×256×256. Обычно первая размерность тензора представляет индексы образцов данных в мини-пакете. Как правило, вычисления такого рода одно-

[1] От англ. *Stochastic Gradient Descent. – Прим.перев.*

родных данных можно эффективно выполнять с использованием графических процессоров.

С другой стороны, 3D-данные обычно неоднородны. Например, полигональные сетки внутри одного мини-пакета могут содержать разное число вершин и граней. И эффективная обработка таких разнородных данных на графических процессорах становится нетривиальной задачей. Программировать гетерогенную мини-пакетную обработку также бывает утомительно. К счастью, библиотека PyTorch3D способна очень эффективно обрабатывать разнородные мини-пакеты. В следующем далее разделе мы рассмотрим пример программирования, в котором предусмотрено применение этих возможностей библиотеки PyTorch3D.

Пример программирования разнородных мини-пакетов

В этом разделе вы научитесь использовать оптимизатор PyTorch и возможности разнородных мини-пакетов PyTorch3D, рассмотрев игрушечный пример. Исходный код примера находится в файле Python `optimization.py` репозитория книги на GitHub. В этом примере мы рассмотрим задачу, в которой камера глубины расположена в неизвестном местоположении, и мы хотим оценить это неизвестное местоположение, используя результаты съемки камеры. В целях упрощения задачи мы допустим, что ориентация камеры известна, и единственной неизвестной величиной является 3D-смещение.

Точнее, мы исходим из допущения, что камера наблюдает за тремя объектами в сцене, и мы знаем достоверные модели этих трех объектов. Давайте посмотрим на исходный код с использованием библиотек PyTorch и PyTorch3D, чтобы решить задачу, как показано ниже.

1. Первым делом импортируем все требующиеся пакеты:

```
import open3d
import os
import torch

from pytorch3d.io import load_objs_as_meshes
from pytorch3d.structures.meshes import join_meshes_as_batch

from pytorch3d.ops import sample_points_from_meshes
from pytorch3d.loss import chamfer_distance
import numpy as np
```

2. На следующем шаге определяем устройство `torch`, используя центральный процессор либо CUDA:

```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
else:
    device = torch.device("cpu»)
    print("ПРЕДУПРЕЖДЕНИЕ: только CPU, очень медленно!»)
```

3. Применяемые в этом игрушечном примере сеточные модели включены в репозиторий исходного кода и находятся в подпапке `data`. Мы собираемся использовать три сеточные модели, содержащиеся в OBJ-файлах `cube.obj`, `diamond.obj` и `dodecahedron.obj`. В следующем ниже фрагменте исходного кода мы используем библиотеку Open3D, чтобы загрузить эти сеточные модели и их визуализировать:

```
mesh_names = ['cube.obj', 'diamond.obj', 'dodecahedron.obj']
data_path = './data'

for mesh_name in mesh_names:
    mesh = open3d.io.read_triangle_mesh(os.path.join(data_path,
                                                     mesh_name))
    open3d.visualization.draw_geometries([mesh],
                                         mesh_show_wireframe=True,
                                         mesh_show_back_face=True)
```

4. Далее используем PyTorch3D, чтобы загрузить те же полигональные сетки и построить список полигональных сеток, назначив его переменной `mesh_list`:

```
mesh_list = list()
device = torch.device('cuda')

for mesh_name in mesh_names:
    mesh = load_objs_as_meshes([os.path.join(data_path,
                                             mesh_name)],
                                             device=device)
    mesh_list.append(mesh)
```

5. Наконец, теперь можно создать мини-пакет PyTorch3D полигональных сеток, используя функцию PyTorch3D `join_meshes_as_batch`[1]. Указанная функция принимает список полигональных сеток и возвращает мини-пакет полигональных сеток:

```
mesh_batch = join_meshes_as_batch(mesh_list,
                                  include_textures=False)
```

   В каждом мини-пакете PyTorch3D есть три способа представления вершин и граней.

   - **Списковый формат**: вершины представлены списком тензоров, в котором каждый тензор представляет вершины или грани одной полигональной сетки в мини-пакете.
   - **Дополненный формат**: все вершины представлены одним тензором, а данные меньших полигональных сеток дополнены нулями, в результате чего все полигональные сетки теперь имеют одинаковое число вершин и граней.
   - **Упакованный формат**: все вершины или грани упакованы в один тензор. На внутреннем уровне отслеживается, к какой полигональной сетке принадлежит каждая вершина или грань.

[1] Объединить сетки в мини-пакет. – *Прим. перев.*

Все три представления имеют свои плюсы и минусы. Тем не менее с помощью API библиотеки PyTorch3D указанные форматы можно эффективно конвертировать друг в друга.

6. В следующем ниже фрагменте исходного кода показан пример того, как возвращать вершины и грани из мини-пакета в списковом формате:

```
vertex_list = mesh_batch.verts_list()
print('vertex_list = ', vertex_list)
face_list = mesh_batch.faces_list()
print('face_list = ', face_list)
```

7. Для возврата вершин и граней в дополненном формате используется следующий ниже API библиотеки PyTorch3D:

```
vertex_padded = mesh_batch.verts_padded()
print('vertex_padded = ', vertex_padded)
face_padded = mesh_batch.faces_padded()
print('face_padded = ', face_padded)
```

8. Для получения вершин и граней в упакованном формате используется следующий ниже фрагмент исходного кода:

```
vertex_packed = mesh_batch.verts_packed()
print('vertex_packed = ', vertex_packed)
face_packed = mesh_batch.faces_packed()
print('face_packed = ', face_packed)

num_vertices = vertex_packed.shape[0]
print('num_vertices = ', num_vertices)
```

9. В этом примере программирования переменная `mesh_batch`[1] рассматривается в качестве достоверной сеточной модели трех объектов. Затем мы выполним симуляцию зашумленной и смещенной версии трех полигональных сеток. На первом этапе мы создадим клон достоверных сеточных моделей:

```
mesh_batch_noisy = mesh_batch.clone()
```

10. Затем мы определяем переменную `motion_gt`, которая будет представлять смещение между местоположением камеры и начальной точкой:

```
motion_gt = np.array([3, 4, 5])
motion_gt = torch.as_tensor(motion_gt)
print('достоверные данные перемещения = ', motion_gt)

motion_gt = motion_gt[None, :]
motion_gt = motion_gt.to(device)
```

11. Для выполнения симуляции шумных мерных данных камеры глубины мы генерируем случайный гауссов шум со средним значением, равным

[1] Пакет сеточных данных. – *Прим. перев.*

motion_gt. Шумы добавляются в mesh_batch_noisy с помощью функции PyTorch3D offset_verts:

```
noise = (0.1**0.5)*torch.randn(
    mesh_batch_noisy.verts_packed().shape).to(device)
motion_gt = np.array([3, 4, 5])
motion_gt = torch.as_tensor(motion_gt)
noise = noise + motion_gt
mesh_batch_noisy = mesh_batch_noisy.offset_verts(noise).detach()
```

12. Для того чтобы оценить неизвестное смещение между камерой и начальной точкой, мы формулируем задачу оптимизации. Прежде всего мы определим переменную оптимизации motion_estimate[1]. Функция torch.zeros создаст нулевой тензор PyTorch. Обратите внимание, что мы установили значение параметра require_grad равным True. Это означает, что при выполнении обратного распространения градиента от функции потери loss мы хотим, чтобы градиент этой переменной вычислялся за нас автоматически библиотекой PyTorch:

```
motion_estimate = torch.zeros(motion_gt.shape,
                              device=device,
                              requires_grad=True)
```

13. Далее мы определим оптимизатор PyTorch со скоростью обучения 0.1. Передача списка переменных оптимизатору задает для задачи оптимизации переменные оптимизации. Здесь такой переменной является переменная motion_estimate:

```
optimizer = torch.optim.SGD([motion_estimate], lr=0.1,
                            momentum=0.9)
```

14. Затем главная процедура оптимизации выглядит, как показано ниже. По сути, мы выполняем стохастический градиентный спуск из 200 итераций. После 200 итераций результирующая расчетная величина motion_estimate должна быть очень близкой к достоверной.
    Каждую итерацию оптимизации можно разделить на следующие ниже четыре шага.
    I. На первом шаге optimizer.zero_grad() сбрасывает все вычисленные на последней итерации значения градиента, устанавливая их равными нулю.
    II. На втором шаге вычисляется функция потери. Обратите внимание, что PyTorch поддерживает динамический вычислительный граф. Другими словами, все вычислительные процедуры по направлению к функции потери регистрируются и будут использоваться при обратном распространении.
    III. На третьем шаге метод loss.backward() вычисляет все градиенты от функции потерь до переменных оптимизации в оптимизаторе PyTorch.

[1] Расчетная величина перемещения. – *Прим. перев.*

IV. На четвертом и последнем шаге метод `optimizer.step()` перемещает все переменные оптимизации на один шаг в направлении уменьшения функции потери.

В процессе вычисления функции потери в случайном порядке отбирается 5000 точек из двух полигональных сеток и вычисляются их фасочные расстояния. Фасочное расстояние – это расстояние между двумя множествами точек[1,2]. Подробнее об этой функции расстояния вы узнаете в последующих главах:

```
for i in range(0, 200):
    optimizer.zero_grad()
    current_mesh_batch = mesh_batch.offset_verts(
        motion_estimate.repeat(num_vertices,1))

    sample_trg = sample_points_from_meshes(current_mesh_batch, 5000)
    sample_src = sample_points_from_meshes(mesh_batch_noisy, 5000)
    loss, _ = chamfer_distance(sample_trg, sample_src)

    loss.backward()
    optimizer.step()
    print('i = ', i, ', motion_estimation = ', motion_estimate)
```

Мы видим, что здесь процесс оптимизации очень быстро сходится к достоверному местоположению `[3,4,5]`.

В приведенном выше примере программирования вы научились использовать разнородные мини-пакеты PyTorch3D. Далее мы обсудим еще одно важное понятие трехмерного компьютерного зрения.

Понятия трансформации и поворота

В трехмерном глубоком обучении и компьютерном зрении обычно приходится работать с 3D-трансформациями, такими как повороты и жесткие перемещения[3] в 3D. Библиотека PyTorch3D обеспечивает высокоуровневую инкапсуляцию этих трансформаций в своем классе `pytorch3d.transforms.Transform3d`. Одним из преимуществ класса `Transform3d` является то, что он основан на мини-пакетах. Вследствие этого, как и зачастую требуется в трехмерном глубоком обучении, мини-пакет трансформаций можно применять к мини-пакету полигональных сеток, используя для этого всего несколько

---

1 Фасочное расстояние (Chamfer Distance), или расстояние по фаске, рассчитывается путем суммирования квадратов расстояний между ближайшими соседями двух облаков точек. Термин «фаска» используется вследствие того, что в расчете учитываются ближайшие соседи двух облаков точек. – *Прим. перев.*

2 Фаска – это переходная грань между двумя гранями объекта, полученная путем среза по двум смежным сторонам. – *Прим. перев.*

3 Жесткое перемещение (rigid motion), или перемещение объекта как единого целого, – это перемещение множества точек, при котором расстояние между точками не меняется. – *Прим. перев.*

строк исходного кода. Еще одним преимуществом класса `Transform3d` является то, что обратное распространение градиента может проходить прямо через `Transform3d`.

Библиотека PyTorch3D также предоставляет целый ряд низкоуровневых API для вычислений в группах Ли SO(3) и SE(3). Здесь SO(3) обозначает специальную ортогональную группу в 3D, а SE(3) – специальную евклидову группу в 3D. На неформальном языке SO(3) обозначает множество всех поворотных трансформаций, а SE(3) – множество всех жестких трансформаций в 3D. PyTorch3D предоставляет целый ряд низкоуровневых API-операций на группах SE(3) и SO(3).

В трехмерном компьютерном зрении существует несколько представлений поворота. Одним из представлений являются матрицы поворота *R*x. В указанном обозначении *x* – это трехмерный вектор, а R – матрица 3×3. Для того чтобы считаться матрицей поворота, R должна быть ортогональной матрицей и иметь детерминант +1. Следовательно, не все матрицы 3×3 могут быть матрицами поворота. Степень свободы у матриц поворота равна 3.

Трехмерный поворот также может быть представлен 3D-вектором *v*, в котором направление вектора *v* является осью поворота. То есть поворот будет удерживать *v* фиксированным и поворачивать все остальное вокруг *v*. Также для представления угла поворота принято использовать амплитуду вектора *v*.

Между двумя представлениями поворота существуют различные математические связи. Если взять характеризующийся постоянной скоростью поворот вокруг оси *v,* то матрицы поворота становятся матричнозначной функцией времени *t*, *R*(*t*). В этом случае градиент функции *R*(*t*) всегда представляет собой кососимметричную матрицу в форме, показанной ниже:

$$\begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}.$$

Здесь действует следующее:

$$v = [v_x \;\; v_y \;\; v_z]^{\mathrm{T}}.$$

Как видно из этих двух уравнений, кососимметричная матрица градиента однозначно определяется вектором *v*, и наоборот. Такое отображение из вектора *v* в его кососимметричную матричную форму обычно называют шляпным оператором[1].

Замкнутая формула отображения из градиента кососимметричной матрицы в матрицу поворота выглядит, как показано ниже. Указанное отображение называется экспоненциальным отображением для специальной ортогональной группы *S*O(3):

---

[1] От англ. *hat operator*; син. циркумфлексный оператор, оператор с диакритическим знаком «^». – *Прим. перев.*

$$R = \exp(A) = \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

Разумеется, существует и обратное отображение экспоненциального отображения:

$A = \log(R)$.

Указанное отображение называется логарифмическим отображением для специальной ортогональной группы $SO(3)$.

Все шляпные операции, обратно-шляпные, экспоненциальные и логарифмические операции уже реализованы в библиотеке PyTorch3D. В PyTorch3D также реализованы многие другие часто используемые 3D-операции, такие как операции на кватернионах и эйлеровы углы.

Примеры программирования трансформации и поворота

В этом разделе мы рассмотрим пример программирования, исходный код которого расположен в файле Python `transformation_and_rotation.py` репозитория книги на GitHub, чтобы научиться применять несколько низкоуровневых API библиотеки PyTorch3D.

1. Начнем с импорта необходимых пакетов:

```
import torch
from pytorch3d.transforms.so3 import (so3_exp_map,
                                      so3_log_map,
                                      hat_inv, hat)
```

2. Затем определяем устройство PyTorch, используя центральный процессор либо CUDA:

```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
else:
    device = torch.device("cpu»)
    print("ПРЕДУПРЕЖДЕНИЕ: только CPU, очень медленно!»)
```

3. Далее задаем мини-пакет из четырех поворотов. Здесь каждый поворот представлен одним 3D-вектором. Направление вектора представляет ось поворота, а амплитуда вектора – угол поворота:

```
log_rot = torch.zeros([4, 3], device = device)
log_rot[0, 0] = 0.001
log_rot[0, 1] = 0.0001
log_rot[0, 2] = 0.0002

log_rot[1, 0] = 0.0001
```

```
log_rot[1, 1] = 0.001
log_rot[1, 2] = 0.0002

log_rot[2, 0] = 0.0001
log_rot[2, 1] = 0.0002
log_rot[2, 2] = 0.001

log_rot[3, 0] = 0.001
log_rot[3, 1] = 0.002
log_rot[3, 2] = 0.003
```

4. Мини-пакет `log_rot` имеет очертание `[4, 3]`, где `4` – это размер пакета, а каждый поворот представлен 3D-вектором. Для того чтобы конвертировать их в представление в виде кососимметричной матрицы 3×3, можно применить шляпный оператор из PyTorch3D, как показано ниже:

```
log_rot_hat = hat(log_rot)
print('log_rot_hat shape = ', log_rot_hat.shape)
print('log_rot_hat = ', log_rot_hat)
```

5. Обратное преобразование из кососимметричной матричной формы в трехмерную векторную форму также возможно с использованием оператора `hat_inv`:

```
log_rot_copy = hat_inv(log_rot_hat)
print('log_rot_copy shape = ', log_rot_copy.shape)
print('log_rot_copy = ', log_rot_copy)
```

6. С помощью функции PyTorch3D `so3_exp_map` из матрицы градиентов можно вычислить матрицу поворота:

```
rotation_matrices = so3_exp_map(log_rot)
print('rotation_matrices = ', rotation_matrices)
```

7. Обратным преобразованием является `so3_log_map`, которое снова отображает матрицу поворота обратно в матрицу градиента:

```
log_rot_again = so3_log_map(rotation_matrices)
print('log_rot_again = ', log_rot_again)
```

В приведенном выше примере программирования показаны API библиотеки PyTorch3D, наиболее часто применяемые для трансформаций и поворотов. Указанные API бывают очень полезны в реальных проектах трехмерного компьютерного зрения.

Резюме

В этой главе вы познакомились с базовыми понятиями отрисовки, растеризации и затенения, включая модели источника света, модель затенения

по Ламберту и модель освещения по Фонгу. Вы научились реализовывать отрисовку, растеризацию и затенение с помощью библиотеки PyTorch3D. Вы также научились изменять параметры процесса отрисовки, такие как окружающее освещение, блеск и зеркальные цвета, и познакомились с тем, как эти параметры влияют на результаты отрисовки.

Затем вы научились применять оптимизатор библиотеки PyTorch. Мы рассмотрели пример исходного кода, в котором указанный оптимизатор использовался в мини-пакете PyTorch3D. В последней части главы вы научились использовать API библиотеки PyTorch3D, чтобы выполнять конверсию между разными представлениями либо поворотами и трансформациями.

В следующей главе вы познакомитесь с несколькими более продвинутыми методами использования деформируемых сеточных моделей, чтобы выполнять их подгонку к реальным 3D-данным.

Часть II

ТРЕХМЕРНОЕ ГЛУБОКОЕ ОБУЧЕНИЕ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ PYTORCH3D

В этой части будет рассмотрено несколько базовых методов обработки трехмерного компьютерного зрения с использованием библиотеки PyTorch3D. При использовании указанной библиотеки реализация алгоритмов трехмерного компьютерного зрения становится проще. Читатели получат большой практический опыт работы с полигональными сетками, облаками точек и подгонкой моделей данными изображений.

Эта часть включает в себя следующие главы:

- глава 3 «Подгонка деформируемых сеточных моделей к необработанным облакам точек»;
- глава 4 «Обнаружение и отслеживание позы объекта с помощью дифференцируемой отрисовки»;
- глава 5 «Понятие дифференцируемой объемометрической отрисовки»;
- глава 6 «Обследование полей нейронного излучения (NeRF)».

Глава 3

Подгонка деформируемых сеточных моделей к необработанным облакам точек

В этой главе мы обсудим проект по использованию деформируемых сеточных моделей с целью их подгонки к необработанным облачным мерным данным, потенциально поступающим из необработанных результатов съемки камерой глубины. Необработанные мерные данные облаков точек из камер глубины обычно имеют формат без какой-либо информации о том, как эти точки связаны между собой; т. е. облака точек не содержат информации о том, как из точек могут формироваться поверхности. Это идет вразрез с полигональной сеткой, в которой определенный на ней список граней показывает, как выглядят поверхности. Информация о том, как точки могут собираться в поверхности, важна для последующей постобработки, такой как устранение шума и обнаружение объектов. Например, если одна точка изолирована без какой-либо связи с какой-либо другой точкой, то эта точка может быть ложно обнаружена датчиком.

Таким образом, восстановление информации о поверхности из облаков точек обычно является стандартным шагом в конвейерах обработки 3D-данных. Существует большой объем предшествующей технической литературы по реконструкции 3D-поверхности из облаков точек, такой как пуассоновская реконструкция. Применение деформируемых сеточных моделей с целью реконструкции поверхности также является одним из часто используемых методов. Обсуждаемый в этой главе метод подгонки таких моделей к облакам точек является практичным и простым базовым методом.

Представленный в этой главе метод основан на принятой в PyTorch оптимизации. Этот метод – еще одна прекрасная демонстрация принципа работы оптимизация с использованием PyTorch. Мы предоставим достаточно подробное объяснение процедуры оптимизации, чтобы вы имели возможность получше в ней разобраться.

Функции потери очень важны в большинстве алгоритмов глубокого обучения. Здесь мы также затронем функции потери, подлежащие применению в описываемом примере, а также те, которые обычно содержатся в библиотеке PyTorch3D. К счастью, во многих современных 3D-средах и библиотеках глубокого обучения, таких как PyTorch3D, многие известные функции потери были уже реализованы. В этой главе вы познакомитесь с несколькими такими функциями потери.

В данной главе будут охвачены следующие ниже главные темы:

- задача подгонки полигональных сеток к облакам точек,
- формулировка задачи подгонки сеточной модели как задачи оптимизации,
- регуляризационные функции потери,
- реализация подгонки полигональной сетки с помощью библиотеки PyTorch3D.

Технические требования

Для выполнения примеров исходного кода этой книги в идеале понадобится компьютер с графическим процессором. Тем не менее для выполнения фрагментов исходного кода вполне будет достаточно только центрального процессора(ов).

Рекомендуемая компьютерная конфигурация включает следующее:

- GPU, такой как серия GTX или серия RTX с не менее 8 Гб памяти,
- Python 3,
- библиотеки PyTorch и PyTorch3D.

Фрагменты исходного кода к этой главе находятся по адресу https://github.com/packtpublishing/3d-deep-learning-with-python.

Задача подгонки полигональных сеток к облакам точек

Реальные камеры глубины, такие как LiDAR, времяпролетные камеры и стереоскопические камеры, обычно выводят изображения глубины либо облака точек. Например, в случае времяпролетных камер модулированный световой луч проецируется из камеры в мир, а глубина в каждом пикселе измеряется

по фазе отраженных световых лучей, полученных в пикселе. Таким образом, для каждого пиксела обычно можно получить одну мерную величину глубины и одну мерную величину амплитуды отраженного света. Однако, кроме выборочной информации о глубине, у нас обычно нет прямых мерных данных поверхностей. Например, мы не можем напрямую измерить гладкость или норму поверхности.

Точно так же в случае стереоскопических камер в каждом временном интервале камера может получать с пары камер два изображения RGB примерно в одно и то же время. Затем камера оценивает глубину, отыскивая соответствие пикселов между двумя изображениями. Таким образом, на выходе получается расчетная величина глубины каждого пиксела. Опять же, камера не способна давать прямых мерных данных поверхностей.

Однако во многих практических приложениях запрашивается информация о поверхности. Например, в роботизированных задачах подбора обычно требуется найти такие области на объекте, которые могут быть крепко схвачены роботизированными руками. В таком сценарии обычно желательно, чтобы области были крупными по размеру и достаточно плоскими.

Есть много других сценариев, в которых требуется выполнить подгонку (деформируемой) сеточной модели к облаку точек. Например, в некоторых приложениях машинного зрения будет сеточная модель промышленной детали, а облачные мерные данные из камеры глубины будут иметь неизвестную ориентацию и позу. В этом случае отыскание подгонки сеточной модели к облаку точек восстановит позу неизвестного объекта.

Вот еще один пример: при отслеживании лица человека иногда требуется выполнить подгонку деформируемой сеточной модели лица к облачным мерным данным, чтобы получить возможность восстановить личность человека и/или выражений его лица.

Функции потери являются центральными понятиями почти всех оптимизаций. По сути говоря, для того чтобы выполнить подгонку к облаку точек, необходимо разработать функцию потери, такую чтобы при минимизации функции потери полигональная сетка как переменная оптимизации вписывалась в облако точек.

На самом деле во многих практических проектах выбор правильной функции потери обычно является важным конструктивным решением. Различные варианты функции потери чаще всего приводят к существенно разной результативности систем. Требования к функции потери обычно включают как минимум следующие ниже свойства:

- функция потери должна иметь желательные числовые свойства, такие как гладкость, выпуклость, отсутствие проблем с исчезающими градиентами и т. д.;
- функция потери (и ее градиенты) должны легко вычисляться; например, они должны эффективно вычисляться на графических процессорах;
- функция потери должна быть хорошим показателем подгонки модели; т. е. минимизация функции потери приводит к удовлетворительной подгонке сеточной модели к входным облакам точек.

Помимо одной первичной функции потери, в таких задачах оптимизации подгонки модели обычно также нужны другие функции потери, чтобы выполнить регуляризацию подгонки модели. Например, если имеются некие предварительные знания о том, что поверхности должны быть гладкими, то обычно нужно вводить дополнительную регуляризационную функцию потери, чтобы подвергать негладкие полигональной сетки большему штрафу.

Пример облачных мерных данных с использованием пешехода показан на рис. 3.1. В последующих разделах этой главы мы собираемся обсудить подход к подгонке деформируемой сеточной модели к облакам точек. Используемое облако точек находится в PLY-файле `pedestrian.ply`, который можно скачать со страницы книги на GitHub. Указанное облако точек можно визуализировать с помощью фрагмента исходного кода в файле Python `vis_input.py`.

Рис. 3.1 ❖ Пример трехмерного облака точек на выходе камеры глубины; обратите внимание на относительно низкую плотность точек

Мы обсудили задачу подгонки полигональной сетки к облаку точек. Теперь давайте поговорим о том, как сформулировать задачу оптимизации.

Формулирование задачи подгонки деформируемой полигональной сетки в задачу оптимизации

В этом разделе мы поговорим о том, как сформулировать задачу подгонки полигональной сетки в задачу оптимизации. Одним из ключевых наблюдений здесь является то, что поверхности объектов, таких как пешеходы, всегда можно непрерывно деформировать в сферу. Таким образом, принятый здесь подход будет начинаться с поверхности сферы и деформировать поверхность, чтобы минимизировать функцию стоимости (также именуемую функцией потери).

Функция стоимости должна выбираться таким образом, чтобы она была хорошим показателем степени похожести облака точек на полигональную сетку. Здесь в качестве главной функции стоимости мы выбираем **фасочное расстояние** между множествами. Фасочное расстояние определяется между двумя множествами точек следующим образом:

$$CD(S_1, S_2) = \frac{1}{|S_1|}\sum_{x \in S_1} \min_{y \in S_2} \|x - y\|_2^2 + \frac{1}{|S_2|}\sum_{y \in S_2} \min_{x \in S_1} \|x - y\|_2^2.$$

Фасочное расстояние является симметричным и представляет собой сумму двух членов. В первом члене для каждой точки *x* в первом облаке точек отыскивается ближайшая точка *y* в другом облаке точек. Для каждой такой пары *x* и *y* вычисляется их расстояние, и расстояния всех пар суммируются. Во втором члене схожим образом для каждой точки *y* во втором облаке точек отыскивается точка *x* в первом облаке, и расстояния между такими парами *x* и *y* суммируются.

Вообще говоря, фасочное расстояние – это расстояние между двумя облаками точек. Если два облака точек идентичны или очень похожи, то фасочное расстояние будет нулевым или очень малым. Если два облака точек находятся далеко, то расстояние между ними будет большим.

Реализация фасочного расстояния в библиотеке PyTorch3D представлена в функции `pytorch3d.loss.champer_distance`. При этом обеспечивается не только прямое вычисление функции потери, но, используя эту реализацию, также можно легко вычислять градиенты для обратного их распространения.

При подгонке полигональных сеток к облакам точек сначала из сеточной модели берется случайная выборка нескольких точек, а затем фасочные расстояния между отобранными из сеточной модели точками и входным облаком точек оптимизируются. Случайная выборка осуществляется посредством функции `pytorch3d.ops.sample_points_from_meshes`. Опять же, на основе указанной функции можно вычислять градиенты для обратного их распространения.

Теперь у нас есть базовая версия задачи оптимизации. Однако нам все еще нужны некоторые функции потери для регуляризации указанной задачи. Мы рассмотрим эти вопросы подробнее в следующем далее разделе.

Функции потери для регуляризации

В предыдущем разделе мы успешно сформулировали задачу подгонки деформируемой полигональной сетки в задачу оптимизации. Однако подход к решению задачи путем прямой оптимизации этой главной функции потери будет проблематичным. Проблема заключается в возможности существования нескольких сеточных моделей, которые хорошо вписываются в одно и то же облако точек. Эти хорошо вписывающиеся сеточные модели могут содержать полигональные сетки, расположенные далеко от гладких полигональных сеток.

С другой стороны, обычно мы обладаем предварительными знаниями о пешеходах. Например, поверхности пешеходов – как правило, гладкие, нормы поверхностей – тоже гладкие. Таким образом, даже если негладкая полигональная сетка близка к входному облаку точек в терминах фасочного расстояния, мы с определенной степенью уверенности знаем, что она далека от достоверных данных.

В технической литературе по машинному обучению на протяжении нескольких десятилетий предлагались решения, предусматривающие исключение таких нежелательных негладких ситуаций. Одно из таких решений называется **регуляризацией**. По сути дела, подлежащая оптимизации потеря выбирается как сумма нескольких функций потери. Разумеется, первый член суммы будет первичным фасочным расстоянием. Другие члены предназначены для наложения штрафов за негладкость поверхности и за нормальную негладкость.

В следующих подразделах мы обсудим несколько таких функций потери, в том числе следующие:

- функцию потери с учетом лапласианова сглаживания полигональной сетки,
- функцию потери с учетом согласованности нормалей полигональной сетки,
- функцию потери с учетом длин ребер полигональной сетки.

Функция потери с учетом лапласианова сглаживания полигональной сетки

Сеточный лапласиан представляет собой дискретную версию известного оператора Лапласа–Бельтрами. Одна из его версий, обычно именуемая униформным лапласианом, выглядит следующим образом:

$$\sigma_i = \sum_{i,j - \text{это ребра}} v_i - v_j.$$

В приведенном выше определении лапласиан в i-й вершине – это просто сумма разностей, где каждая разность находится между координатами текущей вершины и координатами соседней вершины.

Лапласиан является мерой гладкости. Если i-я вершина и все ее соседи лежат в одной плоскости, то лапласиан должен быть равен нулю. Здесь мы используем униформную версию лапласиана, в котором вклад в сумму от каждого соседа имеет одинаковый вес. В более сложных версиях лапласиана предыдущие вклады взвешиваются по разным схемам.

В сущности, включение этой функции потери в оптимизацию привело бы к более плавным решениям. Реализация функции потери с учетом лапласианова сглаживания полигональной сетки (включая несколько других версий, кроме униформной) находится в функции `pytorch3d.loss.mesh_laplacian_smoothing`. Опять же, в ней задействованы вычисления градиента для обратного его распространения.

Функция потери с учетом согласованности нормалей полигональной сетки

Функция потери с учетом согласованности нормалей полигональной сетки служит для наложения штрафа на расстояния между смежными векторами нормалей на полигональной сетке. Ее реализация находится в функции `pytorch3d.loss.mesh_normal_consistency`.

Функция потери с учетом длин ребер полигональной сетки

Функция потери с учетом длин ребер полигональной сетки служит для наложения штрафа за длинные ребра в полигональных сетках. Например, в задаче подгонки сеточной модели, которую мы рассматриваем в этой главе, мы хотим в конечном итоге получить решение, чтобы полученная сеточная модель униформно вписывалась во входное облако точек. Другими словами, каждая локальная область облака точек покрывается малыми треугольниками полигональной сетки. В противном случае сеточная модель не сможет уловить мелкие детали медленно варьирующихся поверхностей, а это означает, что модель будет не такой точной или заслуживающей доверия.

Вышеупомянутой проблемы можно легко избежать, включив функцию потери с учетом длин ребер полигональной сетки в целевую функцию. По сути дела, данная функция потери представляют собой сумму длин всех ребер полигональной сетки. Реализация функции потери с учетом длин ребер полигональной сетки находится в функции `pytorch3d.loss.mesh_edge_loss`.

Теперь мы охватили все понятия и математический аппарат задачи подгонки полигональной сетки. Далее давайте рассмотрим подробнее то, как данная задача выражается программно с помощью Python и PyTorch3D.

Реализация подгонки полигональной сетки с помощью библиотеки PyTorch3D

Входное облако точек содержится в PLY-файле `pedestrian.ply`. Полигональная сетка визуализируется с помощью фрагмента исходного кода в файле Python `vis_input.py`. Главный фрагмент исходного кода подгонки сеточной модели к облаку точек содержится в файле Python `deform1.py`.

1. Начнем с импорта необходимых пакетов:

```
import os
import sys

import torch
from pytorch3d.io import load_ply, save_ply
from pytorch3d.io import load_obj, save_obj
from pytorch3d.structures import Meshes
from pytorch3d.utils import ico_sphere
from pytorch3d.ops import sample_points_from_meshes
from pytorch3d.loss import (
    chamfer_distance,
    mesh_edge_loss,
    mesh_laplacian_smoothing,
    mesh_normal_consistency,
)
import numpy as np
```

2. Затем объявляем устройство согласно библиотеке PyTorch. Если у вас есть графические процессоры, то устройство будет создано под использование графических процессоров. В противном случае устройство должно использовать центральный процессор(ы):

```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
else:
    device = torch.device("cpu»)
    print("ПРЕДУПРЕЖДЕНИЕ: только CPU, очень медленно!»)
```

3. Загружаем облако точек из PLY-файла `pedestrian.ply`. Функция PyTorch3D `load_ply` загружает файл `.ply` и определяет переменные `verts` и `faces`. В данном случае переменная `verts` – это тензор PyTorch. Переменная `faces` – это пустой тензор PyTorch, потому что на самом деле файл `pedestrian.ply` не содержит никаких граней. Функция-член `to`

перемещает тензоры на устройство device; если в device используется графический процессор, то переменные vert и face передаются в память графического процессора:

```
verts, faces = load_ply("pedestrian.ply")
verts = verts.to(device)
faces = faces.to(device)
```

4. Затем выполняем нормализацию и меняем очертания тензоров для последующей обработки:

```
center = verts.mean(0)
verts = verts - center
scale = max(verts.abs().max(0)[0])
verts = verts / scale
verts = verts[None, :, :]
```

5. На следующем шаге создаем сеточную переменную с именем src_mesh, используя функцию PyTorch3D ico_sphere. Функция ico_sphere, по существу, создает полигональную сетку, примерно представляющую собой сферу. Переменная src_mesh будет переменной оптимизации; она начнется как сфера, а затем будет оптимизирована, чтобы вписаться в облако точек:

```
src_mesh = ico_sphere(4, device)
```

6. На следующем шаге нужно определить переменную deform_verts. Переменная deform_verts – это тензор смещений вершин, где для каждой вершины в src_mesh есть смещение вершин трехмерного вектора. Мы собираемся оптимизировать deform_verts, чтобы найти оптимальную деформируемую полигональную сетку:

```
src_vert = src_mesh.verts_list()
deform_verts = torch.full(src_vert[0].shape, 0.0,
                          device=device, requires_grad=True)
```

7. Определяем SGD-оптимизатор, в котором переменная deform_verts является переменной оптимизации:

```
optimizer = torch.optim.SGD([deform_verts], lr=1.0, momentum=0.9)
```

8. Задаем набор весов для разных функций потери. Как уже упоминалось, нам нужно несколько функций потери, включая первичную и регуляризационные функции потери. Окончательная потеря будет представлять собой взвешенную сумму разных функций потери. Ниже мы задаем веса:

```
w_chamfer = 1.0
w_edge = 1.0
w_normal = 0.01
w_laplacian = 0.1
```

9. Теперь все готово к тому, чтобы перейти к главным итерациям оптимизации. Для вычисления функции потери, вычисления градиентов и движения вдоль направления градиентного спуска будет выполнено 2000 итераций. Каждая итерация начинается с вызова функции `optimizer.zero_grad()`, которая сбрасывает все градиенты в нулевое значение, затем вычисляется переменная `loss`, а далее в функции `loss.backward()` вычисляется обратное распространение градиента; движение вдоль направления градиентного спуска выполняется в функции `optimizer.step()`.
Для того чтобы вычислить фасочное расстояние, с помощью функции PyTorch3D `sample_points_from_meshes` из деформируемой сеточной модели во время каждой итерации берется случайная выборка нескольких точек. Обратите внимание, что функция `sample_points_from_meshes` поддерживает вычисления обратного распространения градиента.
Мы также используем три другие регуляризационные функции потери: `mesh_edge_loss`, `mesh_normal_consistency` и `mesh_laplacian_smooth`. Окончательная переменная `loss` фактически представляет собой взвешенную сумму четырех функций потери:

```
for i in range(0, 2000):
    print("i = ", i)

    # Инициализируем оптимизатор
    optimizer.zero_grad()
    #
    new_src_mesh = src_mesh.offset_verts(deform_verts)

    # Отбираем точки с поверхности каждой полигональной сетки
    sample_trg = verts
    sample_src = sample_points_from_meshes(new_src_mesh,
                                           verts.shape[1])

    # Сравниваем два множества облаков точек,
    # вычисляя (a) фасочную потерю
    loss_chamfer, _ = chamfer_distance(sample_trg, sample_src)

    # и (b) длину ребер предсказанной полигональной сетки
    loss_edge = mesh_edge_loss(new_src_mesh)

    # согласованность нормалей полигональной стеки
    loss_normal = mesh_normal_consistency(new_src_mesh)

    # лапласианово сглаживание полигональной сетки
    loss_laplacian = mesh_laplacian_smoothing(new_src_mesh,
                                              method=»uniform»)
    # Взвешенная сумма потерей
    loss = (
        loss_chamfer * w_chamfer
        + loss_edge * w_edge
        + loss_normal * w_normal
        + loss_laplacian * w_laplacian
    )
```

```
    # Шаг оптимизации
    loss.backward()
    optimizer.step()
```

10. Затем извлекаем полученные вершины и грани из переменной `new_src_mesh`, а затем восстанавливаем изначальное местоположение центра и масштаб:

```
final_verts, final_faces = new_src_mesh.get_mesh_verts_faces(0)
final_verts = final_verts * scale + center
```

11. Наконец, полученная сеточная модель сохраняется в PLY-файле `deform1.ply`:

```
# Сохранить предсказанную сетку с помощью save_ply
final_obj = os.path.join("./", "deform1.ply")
save_ply(final_obj, final_verts, final_faces, ascii=True)
```

Рис. 3.2 ❖ Оптимизированная деформируемая сеточная модель. Обратите внимание, что теперь гораздо больше точек, чем у изначального входного облака точек

Полученную полигональную сетку можно визуализировать на экране с помощью файла Python `vis1.py`. Снимок экрана полученной полигональной сет-

ки показан на рис. 3.2. Обратите внимание, что по сравнению с изначальным входным облаком точек оптимизированная сеточная модель на самом деле содержит гораздо больше точек (2500 по сравнению с 239). Полученные поверхности также выглядят более гладкими, чем изначальные входные точки.

Эксперимент без использования каких-либо регуляризационных функций потери

Что, если не использовать ни одну из этих регуляризационных функций потери? Мы проведем эксперимент, используя исходный код из файла Python `deform2.py`. Единственная разница между фрагментом исходного кода в файле `deform2.py` и в файле `deform1.py` заключается в следующих ниже строках:

```
w_chamfer = 1.0
w_edge = 0.0
w_normal = 0.00
w_laplacian = 0.0
```

Обратите внимание, что значения всех весов были установлены равными нулю, кроме одного, для функции фасочной потери. По сути, мы не используем никаких регуляризационных функций потери. Полученную полигональную сетку можно визуализировать на экране, выполнив исходный файла Python `vis2.py`. Снимок экрана показан на рис. 3.3:

Рис. 3.3 ❖ Полигональная сетка, полученная без использования каких-либо регуляризационных функций потери

Обратите внимание, что полученная полигональная сетка на рис. 3.3 не является гладкой и вряд ли будет близка к фактическим достоверным поверхностям.

Эксперимент с использованием только одной функции потери – потери с учетом длин ребер полигональной сетки

На этот раз мы будем использовать следующий ниже набор весов. Фрагмент исходного кода находится в файле Python `deform3.py`:

```
w_chamfer = 1.0
w_edge = 1.0
w_normal = 0.00
w_laplacian = 0.0
```

Полученная сеточная модель находится в PLY-файле `deform3.ply`. Полигональную сетку можно визуализировать на экране с помощью файла Python `vis3.py`. Снимок экрана полигональной сетки показан на рис. 3.4:

Рис. 3.4 ❖ Полученная полигональная сетка с использованием только регуляризации `mesh_edge_loss`

Из рис. 3.4 видно, что полученная сетка намного более гладкая, чем полигональная сетка на рис. 3.3. Тем не менее кажется, что есть несколько быстрых изменений в нормали к поверхности. На самом деле вы можете поработать самостоятельно и попробовать другие веса, чтобы увидеть, как эти функции потери влияют на окончательные результаты.

Резюме

В этой главе мы поговорили о подходе на основе подгонки деформируемой сеточной модели к облаку точек. Как уже отмечалось, получение полигональных сеток из облаков точек обычно является стандартным шагом во многих конвейерах трехмерного компьютерного зрения. Описанный в этой главе подход можно использовать на практике в качестве простого базового метода.

На базе этого метода подгонки деформируемой полигональной сетки вы научились использовать оптимизацию PyTorch. Вы также познакомились с несколькими функциями потери и их реализациях в PyTorch3D, в том числе с функцией потери с учетом фасочного расстояния, функцией потери с учетом длин ребер полигональной сетки, функции потери с учетом лапласианова сглаживания полигональной сетки и функции потери с учетом согласованности нормалей полигональной сетки.

Вы выяснили ситуации, когда эти функции потери следует использовать и для каких целей. Вы увидели несколько экспериментов, показывающих, как функции потери влияют на окончательный результат. Вам также было предложено провести собственные эксперименты с разными комбинациями функций потери и весов.

В следующей главе мы обсудим очень интересную технику трехмерного глубокого обучения, именуемую дифференцируемой отрисовкой. На самом деле в этой книге с дифференцируемой отрисовкой будет связано несколько глав. И следующая глава будет первой из таких глав.

Глава 4

Обнаружение и отслеживание позы объекта с помощью дифференцируемой отрисовки

В этой главе мы обследуем проект по обнаружению и отслеживанию позы объекта с использованием дифференцируемой отрисовки. При обнаружении позы объекта интересует обнаружение ориентации и местоположения определенного объекта. Например, дана модель камеры и сеточная модель объекта, и требуется оценить ориентацию и позицию объекта по одному изображению объекта. В описанном в этой главе подходе мы собираемся сформулировать такую задачу оценивания позы как задачу оптимизации, в которой поза объекта вписывается в изображение, полученное в результате наблюдения.

Тот же подход, что и вышеупомянутый, также можно использовать для отслеживания позы объекта в ситуациях, когда мы уже оценили позу объекта в интервалах времени 1, 2, ..., вплоть до $t - 1$ и хотим оценить позу объекта во временном интервале t, основываясь на одном изображении, полученном в результате наблюдения за объектом в момент времени t.

В этой главе мы будем использовать важный метод, который называется *дифференцируемой отрисовкой*[1], являющийся в настоящее время очень увлекательной темой исследования в области глубокого обучения. Например, в статье-победителе конкурса CVPR 2021 за лучшую научно-исследова-

[1] От англ. *Differentiable Rendering. – Прим. перев.*

тельскую работу «*GIRAFFE*: *представление сцен в виде композиционных полей генеративных нейронных признаков*»[1] дифференцируемая отрисовка используется в качестве одной из важных компонент в конвейере.

Отрисовка – это процесс *проецирования* физических 3D-моделей (сеточной модели объекта или модели камеры) в 2D-изображения. Это имитация физического процесса формирования изображения. Многие задачи трехмерного компьютерного зрения можно трактовать как процесс, обратный процессу отрисовки, т. е. во многих задачах компьютерного зрения требуется начинать с 2D-изображений и вычислять физические 3D-модели (сетки, сегментацию облака точек, поз объекта или позиций камеры).

Таким образом, весьма естественная идея, обсуждавшаяся в сообществе компьютерного зрения в течение нескольких десятилетий, состоит в том, что многие задачи трехмерного компьютерного зрения можно формулировать как задачи оптимизации, в которых переменными оптимизации являются трехмерные модели (полигональные сетки или вокселы облака точек), а целевыми функциями являются определенные меры сходства между отрисованными и наблюдаемыми изображениями.

В целях эффективного решения задачи оптимизации, такой как упомянутая выше, процесс отрисовки должен быть дифференцируемым. Например, если отрисовка является дифференцируемой, то к тренировке модели глубокого обучения решать задачу можно использовать сквозной подход. Однако, как будет более подробно обсуждаться в последних разделах, конвенциональные процессы отрисовки не дифференцируемы. И следовательно, необходимо изменить конвенциональные подходы так, чтобы придать им дифференцируемость. Мы подробно обсудим способы того, как это можно сделать, в следующем далее разделе.

Таким образом, в этой главе сначала мы рассмотрим задачу дифференцируемой отрисовки сначала как ответ на вопрос «зачем она нужна?», а затем как ответ на вопрос «как сделать отрисовку дифференцируемой?». Затем мы поговорим о том, какие задачи трехмерного компьютерного зрения можно решать с помощью дифференцируемой отрисовки. Мы посвятим значительную часть главы конкретному примеру использования дифференцируемой отрисовки для решения задачи оценивания позы объекта. И по ходу изложения представим примеры программирования.

В данной главе будут охвачены следующие ниже главные темы:

- зачем нужны дифференцируемые отрисовки,
- как сделать отрисовку дифференцируемой,
- какие задачи можно решать с помощью дифференцируемой отрисовки,
- задача оценивания позы объекта.

---

[1] От англ. *GIRAFFE: representing scenes as compositional generative neural feature fields*,. См. https://m-niemeyer.github.io/project-pages/giraffe/index.html и https://arxiv.org/abs/2011.12100. – *Прим. перев.*

Технические требования

Для выполнения примеров исходного кода этой книги в идеале понадобится компьютер с графическим процессором. Тем не менее для выполнения фрагментов исходного кода вполне будет достаточно только центрального процессора(ов).

Рекомендуемая компьютерная конфигурация включает следующее:

- GPU, такой как серия GTX или серия RTX с не менее 8 Гб памяти,
- Python 3,
- библиотеки PyTorch и PyTorch3D.

Фрагменты исходного кода к этой главе находятся по адресу https://github.com/packtpublishing/3d-deep-learning-with-python.

Зачем нужна дифференцируемая отрисовка

Физический процесс формирования изображения представляет собой отображение 3D-моделей в 2D-изображения. Как показано в примере на рис. 4.1, в зависимости от позиций красной и синей сфер в 3D (слева показаны две возможные конфигурации) можно получить разные 2D-изображения (изображения, соответствующие двум конфигурациям, показаны справа).

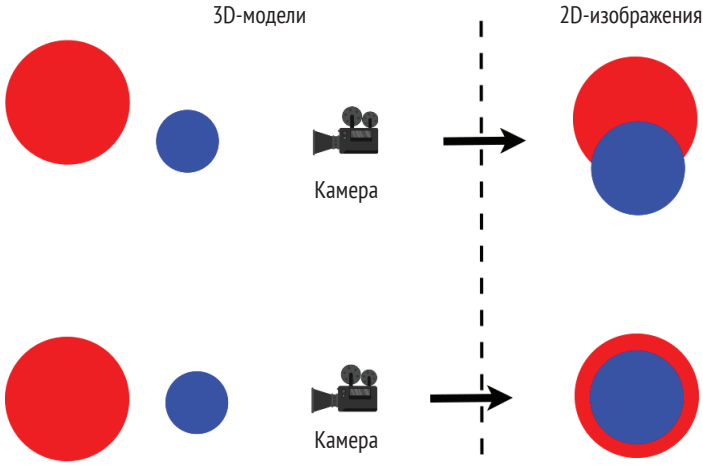


Рис. 4.1 ❖ Процесс формирования изображения представляет собой отображение 3D-моделей в 2D-изображения

Многие задачи трехмерного компьютерного зрения связаны с обратным формированием изображения. В этих задачах обычно даны 2D-изображения, и необходимо рассчитать 3D-модели по 2D-изображениям. Например, на

рис. 4.2 справа дано двумерное изображение, и возникает вопрос, какая трехмерная модель соответствует наблюдаемому изображению?

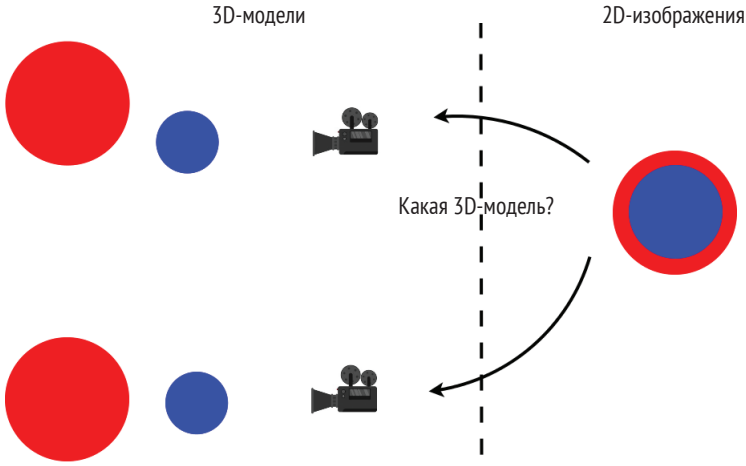


Рис. 4.2 ❖ Многие задачи трехмерного компьютерного зрения основаны на 2D-изображениях, заданных для вычисления 3D-моделей

Согласно некоторым идеям, которые впервые обсуждались в сообществе компьютерного зрения десятилетия назад, мы можем сформулировать задачу как задачу оптимизации. В данном случае переменными оптимизации здесь являются позиции двух 3D-сфер. Мы хотим оптимизировать два центра так, чтобы отрисованные изображения были похожи на предыдущее наблюдаемое 2D-изображение. В целях точного измерения сходства нужно использовать функцию стоимости – например, можно применить попиксельные среднеквадратические ошибки. Затем нужно вычислить градиент от функции стоимости к двум центрам сфер, в результате чего можно было бы итеративно минимизировать функцию стоимости, двигаясь в направлении градиентного спуска.

Однако мы можем вычислить градиент от функции стоимости к переменным оптимизации только при условии, что отображение из переменных оптимизации в функции стоимости является дифференцируемым, из чего следует, что процесс отрисовки также является дифференцируемым.

Как сделать отрисовку дифференцируемой

В этом разделе мы обсудим вопрос, почему конвенциональные алгоритмы отрисовки не являются дифференцируемыми. Мы обсудим подход, используемый в PyTorch3D, который придает отрисовке дифференцируемость.

Рендеринг – это имитация физического процесса формирования изображения. Сам этот физический процесс формирования изображения во многих

случаях является дифференцируемым. Предположим, что поверхность нормальна и все материальные свойства объекта – гладкие. Тогда цвет пиксела в примере является дифференцируемой функцией позиций сфер.

Однако бывают случаи, когда цвет пиксела не является гладкой функцией позиции. Например, такой случай может происходить на границах загораживания одного объекта другим[1]. Это показано на рис. 4.3, где синяя сфера находится в месте, которое загородило бы красную сферу в этом ракурсе, если бы синяя сфера немного сместилась вверх. Таким образом, перемещаемый в этот ракурс пиксел не является дифференцируемой функцией местоположения центра сферы.

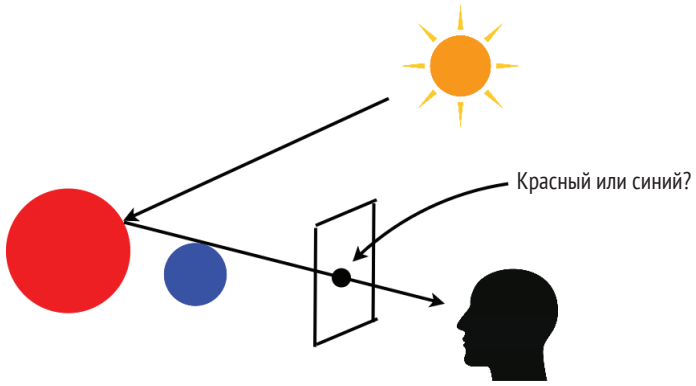


Рис. 4.3 ❖ Формирование изображения не является гладкой функцией на границах загораживания одного объекта другим

При использовании конвенциональных алгоритмов отрисовки информация о локальных градиентах теряется из-за дискретизации. Как мы обсуждали в предыдущих главах, растеризация – это этап отрисовки, на котором для каждого пиксела на плоскости изображения отыскивается наиболее подходящая грань полигональной сетки (либо принимается решение о том, что подходящую грань полигональной сетки отыскать невозможно).

В конвенциональной растеризации для каждого пиксела генерируется луч из центра камеры, проходящий через пиксел на плоскости изображения. Отыскиваются все грани полигональной сетки, которые пересекаются с этим лучом. При конвенциональном подходе растеризатор возвращает только ту грань полигональной сетки, которая находится к камере ближе всего. Затем на следующем этапе конвейера отрисовки возвращенная поверхность полигональной сетки передается затенителю. Затенитель в свою очередь применяется к одному из алгоритмов затенения (например, модели затенения по Ламберту или модели освещения по Фонгу), чтобы определить цвет пиксела. Этот этап выбора отрисовываемой полигональной сетки является недифференцируемым процессом, поскольку он математически моделируется как ступенчатая функция.

---

[1] Син. окклюзия. – *Прим. перев.*

В сообществе компьютерного зрения имеется большой объем технической литературы о том, как сделать отрисовку дифференцируемой. Дифференцируемая отрисовка, реализованная в библиотеке PyTorch3D, основана главным образом на подходе, описанном в статье «Мягкий растеризатор»[1] Лю, Ли, Чена и Ли.

Главная идея дифференцируемой отрисовки проиллюстрирована на рис. 4.4. На этапе растеризации вместо того, чтобы возвращать только одну подходящую грань полигональной сетки, отыскиваются все грани полигональной сетки таким образом, чтобы расстояние грани полигональной сетки до луча находилось в пределах определенного порога. В библиотеке PyTorch3D этот порог можно установить в атрибуте `RasterizationSettings.blur_radius`. Кроме того, можно контролировать максимальное число возвращаемых граней, установив атрибут `RasterizationSettings.faces_per_pixel`.

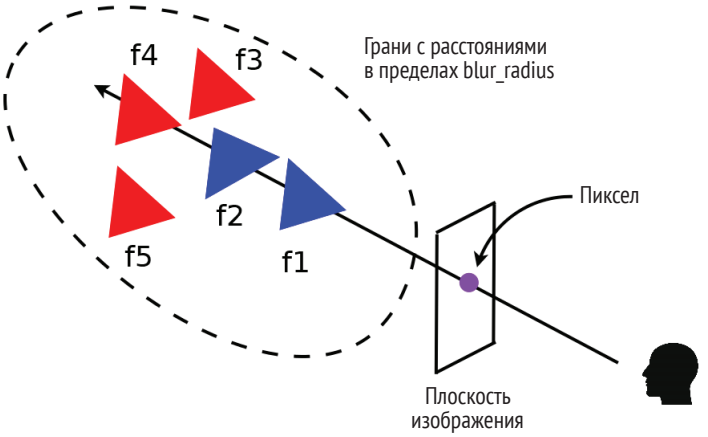


Рис. 4.4 ❖ Дифференцируемая отрисовка путем взвешенного усреднения всех подходящих граней полигональной сетки

Далее отрисовщик должен рассчитать вероятность, поставленную в соответствие каждой грани полигональной сетки, как показано ниже, где dist представляет расстояние между гранью полигональной сетки и лучом, а σ (сигма) – это гиперпараметр. В библиотеке Pytorch3D параметр `sigma` можно установить в атрибуте `BlendParams.sigma`. Проще говоря, соответствующая грани вероятность представляет собой вероятность того, что данная грань полигональной сетки покрывает данный пиксел изображения. Расстояние может быть отрицательным, если луч пересекает грань полигональной сетки.

$$D_j = sigmoid\left(\frac{-dist_j}{\sigma}\right).$$

Затем цвет пиксела определяется средневзвешенными значениями результатов затенения всех граней полигональной сетки, возвращаемых рас-

[1] См. Soft Rasterizer, Liu, Li, Chen, and Li (arXiv:1904.01786).

теризатором. Вес каждой грани полигональной сетки зависит от ее обратного значения глубины, z, и соответствующей грани вероятности, D, как показано в следующем ниже уравнении. Поскольку значение z является обратной глубиной, любые грани полигональной сетки, расположенные ближе к камере, имеют более крупные значения z, чем грани полигональной сетки, находящиеся далеко от камеры. Вес, w_b, – это малый вес цвета фона. Здесь параметр γ (гамма) – это гиперпараметр. В библиотеке PyTorch3D значение этого параметра может быть задано в атрибуте `BlendParams.gamma`:

$$W_j = D_j \exp\left(\frac{z_j}{\gamma}\right), \quad w_b = \exp\left(\frac{\epsilon}{\gamma}\right).$$

Таким образом, окончательный цвет пиксела можно определить с помощью следующего ниже уравнения:

$$I = \frac{\left(\sum_j w_j c_j\right) + w_b c_b}{\left(\sum_j w_j\right) + w_b}.$$

В реализации дифференцируемой отрисовки в библиотеке PyTorch3D для каждого пиксела изображения также вычисляется значение α (альфа). Указанное значение представляет степень вероятности того, что пиксел изображения находится на переднем плане и луч пересекает хотя бы одну грань полигональной сетки, как показано на рис. 4.4. Мы хотим вычислить это значение α и сделать его дифференцируемым. В мягком растеризаторе значение α вычисляется из соответствующих граням вероятностей, как показано ниже.

$$\alpha = 1 - \prod_j (1 - D_j).$$

Теперь, когда мы научились делать отрисовку дифференцируемой, мы посмотрим на то, как ее применять для различных целей.

Какие задачи можно решать с использованием дифференцируемой отрисовки

Как отмечалось ранее, дифференцируемая отрисовка обсуждалась в сообществе компьютерного зрения на протяжении десятилетий. В прошлом дифференцируемая отрисовка использовалась для одноракурсной реконструкции полигональной сетки, подгонки контура на основе изображения и многого другого. В следующих разделах этой главы мы собираемся показать конкретный пример применения дифференцируемой отрисовки для оценивания и отслеживания позы жесткого объекта.

Дифференцируемая отрисовка представляет собой технику, с помощью которой можно формулировать широко встречающиеся в трехмерном компьютерном зрении задачи оценивания в задачи оптимизации. Ее можно

применять для решения широкого круга задач. Что еще интереснее, один из недавних захватывающих трендов состоит в сочетании дифференцируемой отрисовки с глубоким обучением. Обычно дифференцируемая отрисовка используется в качестве генераторной части моделей глубокого обучения. И следовательно, весь конвейер можно тренировать в сквозном порядке, от начала до конца.

Задача оценивания поз объекта

В этом разделе мы собираемся показать конкретный пример применения дифференцируемой отрисовки для задач трехмерного компьютерного зрения. Задача будет заключаться в оценивании позы объекта по одному единственному изображению, полученному в результате наблюдения. В дополнение к этому мы будем исходить из допущения, что у нас трехмерная сеточная модель объекта.

Например, мы исходим из того, что дана трехмерная сеточная модель игрушечной коровы и чайника, как показано соответственно на рис. 4.5 и рис. 4.7. Теперь предположим, что мы сделали один снимок игрушечной коровы и чайника. Таким образом, у нас есть одно RGB-изображение игрушечной коровы, как показано на рис. 4.6, и одно силуэтное изображение чайника, как показано на рис. 4.8. Тогда задача состоит в том, чтобы оценить ориентацию и местоположение игрушечной коровы и чайника в моменты, когда эти изображения были сделаны.

Поскольку поворачивать и перемещать полигональные сетки затруднительно, вместо этого мы решили зафиксировать ориентацию и местоположение полигональных сеток и оптимизировать ориентации и местоположения камеры. Приняв в качестве допущения, что ориентация камеры всегда направлена на полигональные сетки, можно упростить задачу еще больше, сведя всю оптимизацию к местоположению камеры.

Таким образом, мы формулируем задачу оптимизации таким образом, чтобы переменными оптимизации были местоположения камеры. Используя дифференцируемую отрисовку, можно отрисовать RGB-изображения и силуэтные изображения двух полигоналных сеток. Отрисованные изображения сравниваются с наблюдаемыми изображениями, в результате чего можно рассчитать функции потери между отрисованными и наблюдаемыми изображениями. Здесь в качестве функции потери используются среднеквадратичные ошибки. Поскольку все дифференцируется, можно вычислить градиенты от функций потери до переменных оптимизации. Затем применить алгоритм градиентного спуска с целью отыскания наилучших позиций камеры таких, чтобы отрисованные изображения совпадали с наблюдаемыми изображениями.

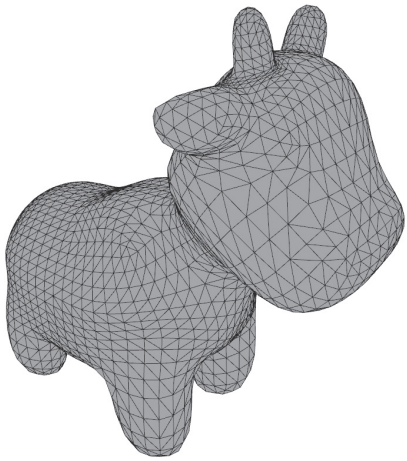


Рис. 4.5 ❖ Сеточная модель игрушечной коровы

На следующем ниже изображении показан RGB-результат для коровы:

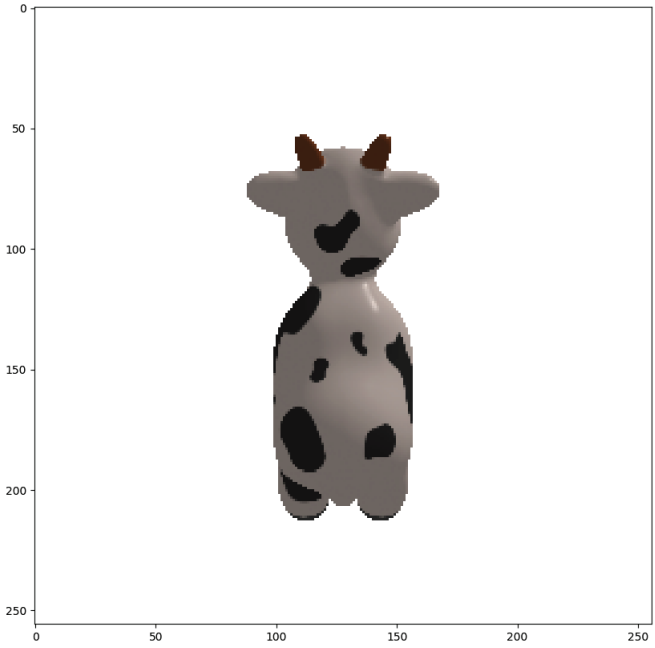


Рис. 4.6 ❖ Наблюдаемое RGB-изображение игрушечной коровы

На следующем ниже рисунке показана полигональная сетка чайника:

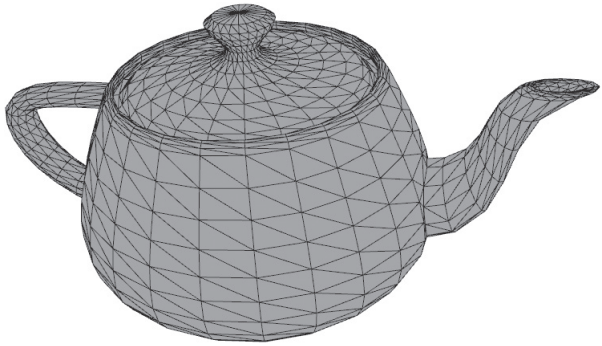


Рис. 4.7 ❖ Сеточная модель чайника

На следующем ниже рисунке показан силуэт чайника:

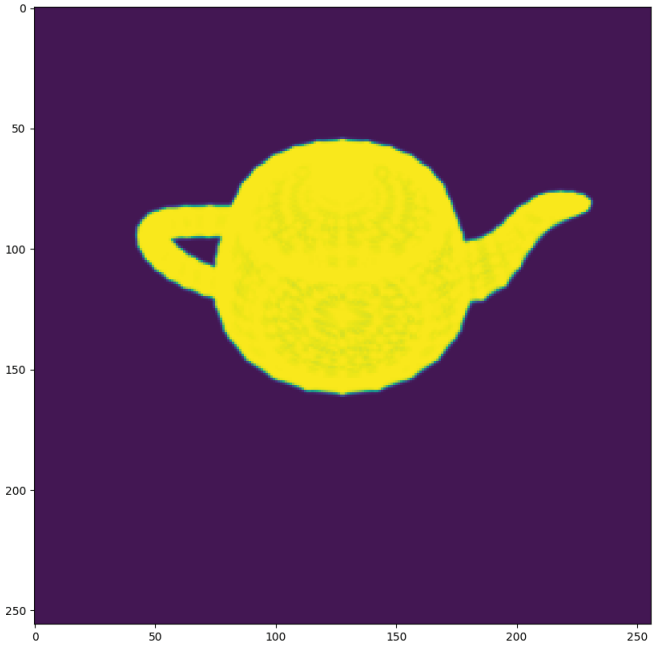


Рис. 4.8 ❖ Наблюдаемый силуэт чайника

Теперь, когда вы знаете задачу и то, как с ней работать, давайте в следующем разделе приступим к программированию.

Как это программируется

Исходный код находится в репозитории книги в файле Python `diff_render.py` папки `chap4`. Сеточная модель чайника находится в OBJ-файле `teapot.obj` подпапки `data`. Давайте пройдемся по исходному коду, как показано ниже.

1. Исходный код в файле Python `diff_render.py` начинается с импорта необходимых пакетов:

```
import os

import torch
import numpy as np
import torch.nn as nn
import matplotlib.pyplot as plt
from skimage import img_as_ubyte

from pytorch3d.io import load_obj
from pytorch3d.structures import Meshes
from pytorch3d.renderer import (
    FoVPerspectiveCameras, look_at_view_transform,
    look_at_rotation, RasterizationSettings,
    MeshRenderer, MeshRasterizer,
    BlendParams, SoftSilhouetteShader, HardPhongShader,
    PointLights, TexturesVertex,
)
```

2. На следующем шаге объявляем устройство PyTorch. Если у вас есть графический процессор(ы), то устройство `device` будет создано под использование графического процессора(ов). В противном случае устройство должно использовать центральный процессор(ы):

```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
else:
    device = torch.device("cpu»)
    print("ПРЕДУПРЕЖДЕНИЕ: только CPU, очень медленно!»)
```

3. Затем в следующей ниже строке задаем выходной каталог `output_dir`. При выполнении исходного кода файла Python `diff_render.py` указанный исходный код будет генерировать отрисованные изображения на каждой итерации оптимизации, предоставляя возможность видеть пошаговый ход выполнения оптимизации. Все сгенерированные исходным кодом отрисованые изображения будут помещаться в папку `output_dir`.

```
output_dir = './result_teapot'
```

4. Затем загружаем сеточную модель из OBJ-файла `./data/teapot.obj`. Поскольку эта сеточная модель не сопровождается текстурами (значениями материального цвета), мы создаем единый тензор и делаем его текстурой для сеточной модели. В итоге мы получаем сеточную модель с текстурами в качестве переменной `teapot_mesh`:

```
verts, faces_idx, _ = load_obj("./data/teapot.obj")
faces = faces_idx.verts_idx

verts_rgb = torch.ones_like(verts)[None]  # (1, V, 3)
textures = TexturesVertex(verts_features=verts_rgb.to(device))

teapot_mesh = Meshes(
    verts=[verts.to(device)],
    faces=[faces.to(device)],
    textures=textures
)
```

5. Далее в следующей строке задаем модель камеры.

```
cameras = FoVPerspectiveCameras(device=device)
```

6. На следующем шаге определяется дифференцируемый отрисовщик с именем `silhouette_renderer`. Каждый отрисовщик состоит в основном из двух компонент, таких как растеризатор для отыскания граней полигональной сетки, подходящих для каждого пиксела изображения, затенитель для определения значений цвета пикселов изображения и т. д. В данном конкретном примере мы используем мягкосилуэтный затенитель, который выводит значение альфа для каждого пиксела изображения. Значение альфа – это действительное число в диапазоне от 0 до 1, которое указывает, частью чего является этот пиксел изображения: переднего плана либо фона. Обратите внимание, что гиперпараметры затенителя определены в переменных `blend_params`, параметр `sigma` предназначен для вычисления соответствующих граням вероятностей, а `gamma` – для вычисления весов граней полигональной сетки. Здесь для растеризации используется `MeshRasterizer`. Обратите внимание, что параметр `blur_radius` – это пороговое значение для отыскания соответствующих граней полигональной сетки, а `faces_per_Pixel` – максимальное число граней полигональной сетки, которое будет возвращено для каждого пиксела изображения:

```
blend_params = BlendParams(sigma=1e-4, gamma=1e-4)

raster_settings = RasterizationSettings(
    image_size=256,
    blur_radius=np.log(1. / 1e-4 - 1.) * blend_params.sigma,
    faces_per_pixel=100,
)

silhouette_renderer = MeshRenderer(
    rasterizer=MeshRasterizer(
```

```
        cameras=cameras,
        raster_settings=raster_settings
    ),
    shader=SoftSilhouetteShader(blend_params=blend_params)
)
```

7. Затем определяем фонговский отрисовщик `phong_renderer`. Указанный отрисовщик в основном используется для визуализации процесса оптимизации. В сущности, на каждой итерации оптимизации мы будем отрисовывать одно RGB-изображение в соответствии с позицией камеры на этой итерации. Обратите внимание, что этот отрисовщик используется только для целей визуализации, поэтому он не является дифференцируемым отрисовщиком. На самом деле если обратить внимание на следующее ниже моменты, то можно понять, что фонговский отрисовщик `phong_renderer` не является дифференцируемым:
   - в нем используется затенитель `HardPhoneShader`, который на входе принимает только одну грань полигональной сетки по каждому пикселу изображения;
   - в нем используется сеточный отрисовщик `MeshRenderer` со значением `blur_radius`, равным `0.0`, и значением `face_per_pixel`, равным `1`.
8. Затем определяем источник света `lights` с координатами `2.0`, `2.0` и `-2.0`:

```
raster_settings = RasterizationSettings(
    image_size=256,
    blur_radius=0.0,
    faces_per_pixel=1,
)

lights = PointLights(
    device=device,
    location=((2.0, 2.0, -2.0),)
)

phong_renderer = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=cameras,
        raster_settings=raster_settings
    ),
    shader=HardPhongShader(
        device=device,
        cameras=cameras,
        lights=lights
    )
)
```

9. Далее задаем местоположение камеры и вычисляем соответствующий поворот `R` и смещение `T` камеры. Указанные поворот и смещение являются целевой позицией камеры, т. е. мы будем генерировать изображение из этой позиции камеры и использовать это изображение в задаче в качестве наблюдаемого изображения:

```
distance = 3
elevation = 50.0
azimuth = 0.0

R, T = look_at_view_transform(distance,
                              elevation,
                              azimuth,
                              device=device)
```

10. Теперь генерируем изображение `image_ref` из этой позиции камеры. Переменная `image_ref` имеет четыре канала: **RGBA** – **R** для красного, **G** для зеленого, **B** для синего и **A** для значений альфа. Переменная `image_ref` также сохраняется в виде PNG-файла `target_rgb.png` для последней проверки:

```
silhouette = silhouette_renderer(meshes_world=teapot_mesh,
                                 R=R, T=T)
image_ref = phong_renderer(meshes_world=teapot_mesh,
                           R=R, T=T)

silhouette = silhouette.cpu().numpy()
image_ref = image_ref.cpu().numpy()

plt.figure(figsize=(10, 10))
# Рисовать только альфа-канал RGBA-изображения
plt.imshow(silhouette.squeeze()[..., 3])
plt.grid(False)
plt.savefig(os.path.join(output_dir, 'target_silhouette.png'))
plt.close()

plt.figure(figsize=(10, 10))
plt.imshow(image_ref.squeeze())
plt.grid(False)
plt.savefig(os.path.join(output_dir, 'target_rgb.png'))
plt.close()
```

11. На следующем шаге определяем класс `Model`. Этот класс является производным от класса `torch.nn.Module.Model`, и, следовательно, как и во многих других моделях PyTorch, для класса `Model` можно активировать автоматические вычисления градиента.
Класс `Model` имеет функцию инициализации `__init__`, которая на входе принимает `meshes` для сеточных моделей, отрисовщик `renderer` и `image_ref` в качестве целевого изображения; экземпляр класса `Model` попытается выполнить его подгонку. Функция `__init__` создает буфер `image_ref` посредством функции `torch.nn.Module.register_buffer`. В качестве напоминания для тех, кто не очень знаком с этой частью PyTorch: буфер – это то, что можно сохранять как часть словаря состояний `state_dict` и перемещать на разные устройства в `cuda()` и `cpu()` с остальными

параметрами модели. Однако буфер не обновляется оптимизатором. Функция `__init__` также создает модельный параметр `camera_position`. Оптимизатор может обновлять переменную `camera_position` в качестве модельного параметра. Обратите внимание, что переменные оптимизации теперь становятся модельными параметрами.
Класс `Model` также имеет функцию-член `forward`, которая может выполнять прямое вычисление и обратное распространение градиента. Функция `forward` отрисовывает силуэтное изображение из текущей позиции камеры и вычисляет функцию потери между отрисованным изображением и `image_refer`, т. е. наблюдаемым изображением:

```
class Model(nn.Module):
    def __init__(self, meshes, renderer, image_ref):
        super().__init__()
        self.meshes = meshes
        self.device = meshes.device
        self.renderer = renderer

        image_ref = torch.from_numpy(
                (image_ref[..., :3].max(-1) != 1).astype(np.float32)
            )
        self.register_buffer('image_ref', image_ref)

        self.camera_position = nn.Parameter(
            torch.from_numpy(
                np.array([3.0, 6.9, +2.5],
                         dtype=np.float32)).to(meshes.device))

    def forward(self):
        R = look_at_rotation(self.camera_position[None, :],
                             device=self.device)  # (1, 3, 3)
        T = -torch.bmm(
            R.transpose(1, 2),
            self.camera_position[None, :, None])[:, :, 0]  # (1, 3)
        image = self.renderer(meshes_world=self.meshes.clone(),
                              R=R, T=T)
        loss = torch.sum((image[..., 3] - self.image_ref) ** 2)
        return loss, image
```

12. Теперь класс `Model` определен. Затем можно создать экземпляр данного класса и определить оптимизатор. Перед выполнением какой-либо оптимизации мы хотим отрисовать изображение, чтобы показать стартовую позицию камеры. Это силуэтное изображение в стартовой позиции камеры будет сохранено в PNG-файле `startup_silhouette.png`:

```
model = Model(meshes=teapot_mesh,
              renderer=silhouette_renderer,
              image_ref=image_ref).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.05)
```

```
_, image_init = model()
plt.figure(figsize=(10, 10))
plt.imshow(image_init.detach().squeeze().cpu().numpy()[..., 3])
plt.grid(False)
plt.title("Стартовый силуэт")
plt.savefig(os.path.join(output_dir, 'starting_silhouette.png'))
plt.close()
```

13. Наконец, можно выполнить итерации оптимизации. Во время каждой итерации будем сохранять отрисованное из позиции камеры изображение в файл в папке `output_dir`:

```
for i in range(0, 200):
    if i%10 == 0:
        print('i = ', i)

    optimizer.zero_grad()
    loss, _ = model()
    loss.backward()
    optimizer.step()

    if loss.item() < 500:
        break

    R = look_at_rotation(model.camera_position[None, :],
                         device=model.device)
    T = -torch.bmm(R.transpose(1, 2),
                   model.camera_position[None, :, None])[:, :, 0]  # (1, 3)
    image = phong_renderer(meshes_world=model.meshes.clone(),
                           R=R, T=T)
    image = image[0, ..., :3].detach().squeeze().cpu().numpy()
    image = img_as_ubyte(image)

    plt.figure()
    plt.imshow(image[..., :3])
    plt.title("iter: %d, loss: %0.2f" % (i, loss.data))
    plt.axis("off")
    plt.savefig(os.path.join(output_dir,
                             'fitting_' + str(i) + '.png'))
    plt.close()
```

На рис. 4.9 показан наблюдаемый силуэт объекта (в данном случае чайника).

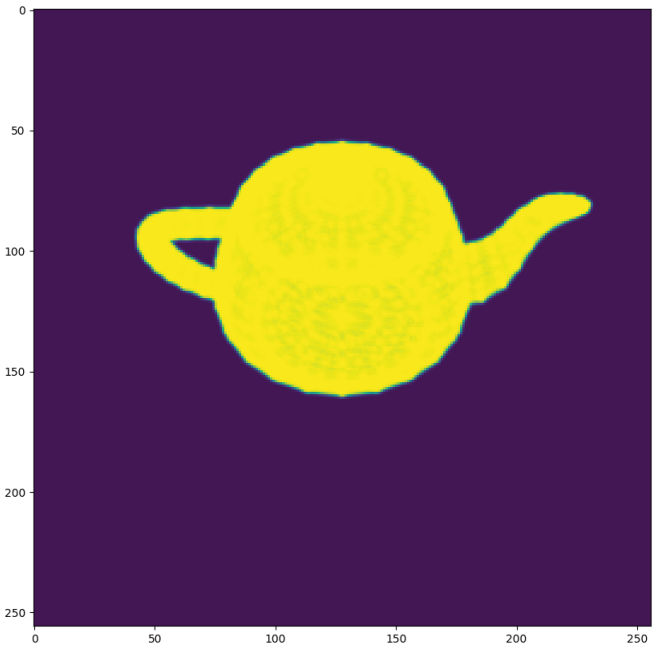


Рис. 4.9 ❖ Силуэт чайника

Формулируем задачу подгонки как задачу оптимизации. Первоначальная позиция чайника показана на рис. 4.10.

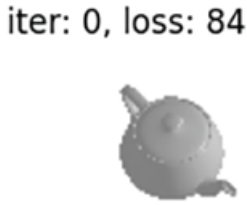


Рис. 4.10 ❖ Первоначальная позиция чайника

Окончательная оптимизированная позиция чайника показана на рис. 4.11.

Рис. 4.11 ❖ Окончательная позиция чайника

Пример оценивания позы объекта для подгонки силуэта и подгонки текстуры

В приведенном выше примере мы оценивали позу объекта путем подгонки силуэта. В этом разделе мы представим еще один пример оценивания позы объекта, но теперь с использованием как подгонки силуэта, так и подгонки текстуры. В трехмерном компьютерном зрении текстура обычно используется для обозначения значений цвета. Таким образом, в этом примере мы будем использовать дифференцируемую отрисовку для отрисовки RGB-изображений в соответствии с позициями камеры и для оптимизации позиции камеры. Соответствующий исходный код находится в файле Python `diff_render_texture.py`:

1. На первом шаге импортируем все необходимые пакеты:

```
import os

import torch
import numpy as np
import torch.nn as nn
import matplotlib.pyplot as plt
from skimage import img_as_ubyte

from pytorch3d.io import load_objs_as_meshes
from pytorch3d.renderer import (
    FoVPerspectiveCameras, look_at_view_transform,
    look_at_rotation, RasterizationSettings,
    MeshRenderer, MeshRasterizer, BlendParams,
    SoftSilhouetteShader, HardPhongShader,
    PointLights, SoftPhongShader
)
```

2. Далее создаем устройство PyTorch, используя графический процессор(ы) либо центральный процессор(ы):

```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    torch.cuda.set_device(device)
else:
    device = torch.device("cpu")
```

3. Задаем `result_cow` в качестве выходного каталога `output_dir`. Это будет папка, в которой будут сохраняться результаты подгонки:

```
output_dir = './result_cow'
```

4. Загружаем сеточную модель игрушечной коровы из OBJ-файла `cow.obj`:

```
obj_filename = "./data/cow_mesh/cow.obj"
cow_mesh = load_objs_as_meshes([obj_filename],
                               device=device)
```

5. Определяем камеры и источники света, как показано ниже:

```
cameras = FoVPerspectiveCameras(device=device)
lights = PointLights(device=device,
                     location=((2.0, 2.0, -2.0),))
```

6. Далее создаем силуэтный отрисовщик `renderer_silhouette`. Это дифференцируемый отрисовщик для отрисовки силуэтных изображений. Обратите внимание на числовые значения парметров `blur_radius` и `faces_per_pixel`. Этот отрисовщик используется главным образом в подгонке силуэтов:

```
blend_params = BlendParams(sigma=1e-4, gamma=1e-4)
raster_settings = RasterizationSettings(
    image_size=256,
    blur_radius=np.log(1. / 1e-4 - 1.) * blend_params.sigma,
    faces_per_pixel=100,
)

renderer_silhouette = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=cameras,
        raster_settings=raster_settings
    ),
    shader=SoftSilhouetteShader(blend_params=blend_params)
)
```

7. Далее создаем текстурированный отрисовщик `renderer_textured`. Это еще один дифференцируемый отрисовщик, в основном используемый для отрисовки RGB-изображений:

```
sigma = 1e-4
raster_settings_soft = RasterizationSettings(
    image_size=256,
    blur_radius=np.log(1. / 1e-4 - 1.)*sigma,
    faces_per_pixel=50,
)

renderer_textured = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=cameras,
        raster_settings=raster_settings_soft
    ),
    shader=SoftPhongShader(
        device=device,
        cameras=cameras,
        lights=lights
    )
)
```

8. Далее создаем фонговский отрисовщик `phong_renderer`. Этот отрисовщик используется главным образом для визуализации. Приведенные выше дифференцируемые отрисовщики тяготеют к созданию размытых изображений. Поэтому нам было бы неплохо иметь отрисовщик, который способен генерировать четкие изображения:

```
raster_settings = RasterizationSettings(
    image_size=256,
    blur_radius=0.0,
    faces_per_pixel=1,
)

phong_renderer = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=cameras,
        raster_settings=raster_settings
    ),
    shader=HardPhongShader(
        device=device,
        cameras=cameras,
        lights=lights
    )
)
```

9. Далее зададим позицию камеры, а также соответствующий поворот и позицию камеры. Это будет позиция камеры, в которой будет снято наблюдаемое изображение. Как и в предыдущем примере, вместо ориентации и позиции объекта мы оптимизируем ориентацию и позицию камеры. Кроме того, мы исходим из допущения, что камера всегда направлена на объект. Таким образом, требуется оптимизировать только позицию камеры:

```
distance = 3
elevation = 50.0
```

```
azimuth = 0.0
R, T = look_at_view_transform(distance,
                              elevation,
                              azimuth,
                              device=device)
```

10. Далее создаем наблюдаемые изображения и сохраняем их в PNG-файлах `target_silhouette.png` и `target_rgb.png`. Указанные изображения также хранятся в переменных `silhouette` и `image_ref`:

```
silhouette = renderer_silhouette(meshes_world=cow_mesh,
                                 R=R, T=T)
image_ref = phong_renderer(meshes_world=cow_mesh,
                           R=R, T=T)
silhouette = silhouette.cpu().numpy()
image_ref = image_ref.cpu().numpy()

plt.figure(figsize=(10, 10))
plt.imshow(silhouette.squeeze()[..., 3])
plt.grid(False)
plt.savefig(os.path.join(output_dir, 'target_silhouette.png'))
plt.close()

plt.figure(figsize=(10, 10))
plt.imshow(image_ref.squeeze())
plt.grid(False)
plt.savefig(os.path.join(output_dir, 'target_rgb.png'))
plt.close()
```

11. Теперь видоизменим определение класса `Model`, как показано ниже. Наиболее заметные изменения, по сравнению с предыдущим примером, заключаются в том, что теперь мы будем отрисовывать и изображение альфа-канала, и RGB-изображения и сравнивать их с наблюдаемыми изображениями. Среднеквадратические потери в альфа-канале и RGB-каналах взвешиваются, давая окончательное значение потери:

```
class Model(nn.Module):
    def __init__(self, meshes, renderer_silhouette,
                 renderer_textured, image_ref,
                 weight_silhouette, weight_texture):
        super().__init__()
        self.meshes = meshes
        self.device = meshes.device
        self.renderer_silhouette = renderer_silhouette
        self.renderer_textured = renderer_textured

        self.weight_silhouette = weight_silhouette
        self.weight_texture = weight_texture

        image_ref_silhouette = torch.from_numpy(
            (image_ref[..., :3].max(-1) != 1).astype(np.float32))
        self.register_buffer('image_ref_silhouette',
                             image_ref_silhouette)
```

```
        image_ref_textured = torch.from_numpy(
            (image_ref[..., :3]).astype(np.float32))
        self.register_buffer('image_ref_textured',
                             image_ref_textured)

        self.camera_position = nn.Parameter(
            torch.from_numpy(np.array(
                [3.0, 6.9, +2.5],
                dtype=np.float32)).to(meshes.device))

    def forward(self):
        # Отрисовать изображение с использованием обновленной
        # позиции камеры. На основе новой позиции камеры
        # вычисляется матрица поворота и трансляции.
        R = look_at_rotation(
            self.camera_position[None, :],
            device=self.device)  # (1, 3, 3)
        T = -torch.bmm(
            R.transpose(1, 2),
            self.camera_position[None, :, None])[:, :, 0]  # (1, 3)

        image_silhouette = self.renderer_silhouette(
            meshes_world=self.meshes.clone(), R=R, T=T)
        image_textured = self.renderer_textured(
            meshes_world=self.meshes.clone(), R=R, T=T)

        loss_silhouette = torch.sum(
            (image_silhouette[..., 3] - self.image_ref_silhouette) ** 2)
        loss_texture = torch.sum(
            (image_textured[..., :3] - self.image_ref_textured) ** 2)

        loss = self.weight_silhouette * loss_silhouette + \
            self.weight_texture * loss_texture
        return loss, image_silhouette, image_textured
```

12. Далее создаем экземпляр класса `Model` и создаем оптимизатор:

```
model = Model(meshes=cow_mesh,
              renderer_silhouette=renderer_silhouette,
              renderer_textured=renderer_textured,
              image_ref=image_ref, weight_silhouette=1.0,
              weight_texture=0.1).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.05)
```

13. Наконец, выполняем 200 итераций оптимизации. Отрисованные изображения сохраняются на каждой итерации:

```
for i in range(0, 200):
    if i%10 == 0:
        print('i = ', i)

    optimizer.zero_grad()
    loss, image_silhouette, image_textured = model()
```

```
loss.backward()
optimizer.step()

plt.figure()
plt.imshow(
    image_silhouette[..., 3].detach().squeeze().cpu().numpy())
plt.title("iter: %d, loss: %0.2f" % (i, loss.data))
plt.axis("off")
plt.savefig(os.path.join(output_dir,
                         'soft_silhouette_' + str(i) + '.png'))
plt.close()

plt.figure()
plt.imshow(
    image_textured.detach().squeeze().cpu().numpy())
plt.title("iter: %d, loss: %0.2f" % (i, loss.data))
plt.axis("off")
plt.savefig(os.path.join(output_dir,
                         'soft_texture_' + str(i) + '.png'))
plt.close()

R = look_at_rotation(model.camera_position[None, :],
                     device=model.device)
T = -torch.bmm(R.transpose(1, 2),
               model.camera_position[None, :, None])[:, :, 0] # (1, 3)
image = phong_renderer(meshes_world=model.meshes.clone(),
                       R=R, T=T)

plt.figure()
plt.imshow(
    image[..., 3].detach().squeeze().cpu().numpy())
plt.title("iter: %d, loss: %0.2f" % (i, loss.data))
plt.axis("off")
plt.savefig(os.path.join(output_dir,
                         'hard_silhouette_' + str(i) + '.png'))
plt.close()

image = image[0, ..., :3].detach().squeeze().cpu().numpy()
image = img_as_ubyte(image)

plt.figure()
plt.imshow(image[..., :3])
plt.title("iter: %d, loss: %0.2f" % (i, loss.data))
plt.axis("off")
plt.savefig(os.path.join(output_dir,
                         'hard_texture_' + str(i) + '.png'))
plt.close()

if loss.item() < 800:
    break
```

Наблюдаемое силуэтное изображение показано на рис. 4.12.

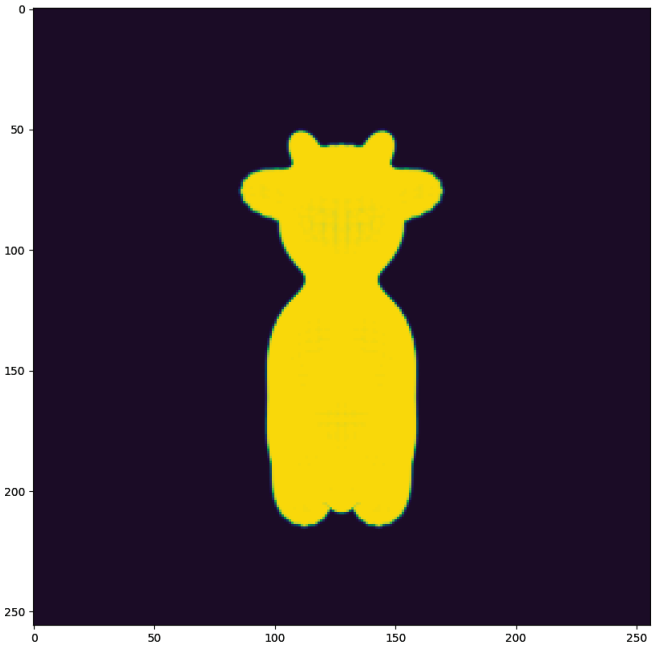


Рис. 4.12 ❖ Наблюдаемое силуэтное изображение

RGB-изображение показано на рис. 4.13:

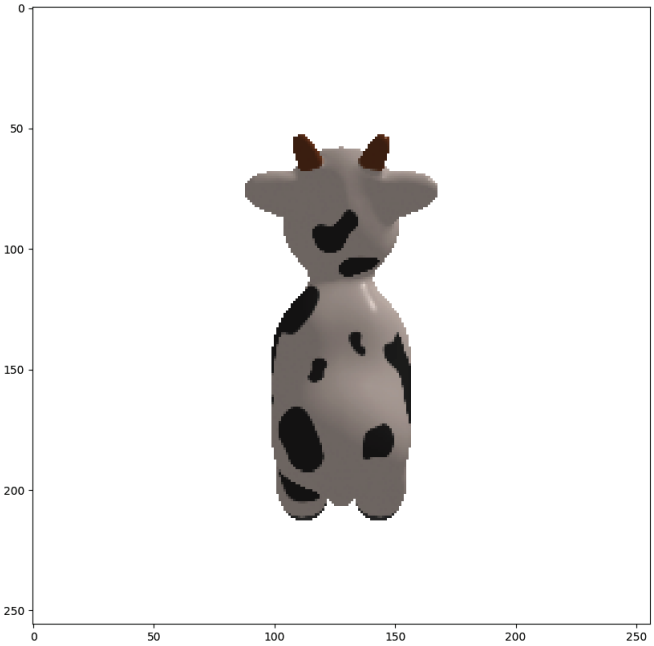


Рис. 4.13 ❖ Наблюдаемое RGB-изображение

Отрисованные RGB-изображения, соответствующие первоначальной и окончательной позициям камеры, показаны соответственно на рис. 4.14 и рис. 4.15.

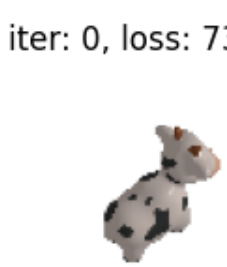


Рис. 4.14 ❖ Изображение, соответствующее первоначальной позиции камеры

Изображение, соответствующее окончательной позиции, выглядит следующим образом:

Рис. 4.15 ❖ Изображение, соответствующее подогнанной позиции камеры

Резюме

В этой главе мы начали с вопроса о том, зачем нужна дифференцируемая отрисовка. Ответ на этот вопрос заключается в том, что отрисовку можно трактовать как отображение 3D-сцен (сеток или облаков точек) в 2D-изображения. Если отрисовку сделать дифференцируемой, то появляется возможность оптимизировать 3D-модели напрямую с правильно выбранной функцией стоимости между отрисованными и наблюдаемыми изображениями.

Затем мы обсудили реализованный в библиотеке PyTorch3D подход, позволяющий придавать отрисовке дифференцируемость. Далее мы обсудили два конкретных примера оценивания позы объекта, сформированной как задача оптимизации, в которой поза объекта оптимизируется напрямую с целью минимизации среднеквадратичных ошибок между отрисованными и наблюдаемыми изображениями.

Мы также рассмотрели примеры исходного кода, в которых библиотека PyTorch3D используется для решения задач оптимизации.

В следующей главе мы обследуем другие варианты дифференцируемой отрисовки и способы ее применения.

Глава 5

Понятие дифференцируемой объемометрической отрисовки

В этой главе мы обсудим новый способ дифференцируемой отрисовки. Мы собираемся использовать представление 3D-данных в виде вокселов, в отличие от представления 3D-данных в виде полигональной сетки, которое мы использовали в предыдущей главе. Представление 3D-данных в виде вокселов имеет определенные преимущества по сравнению с сеточными моделями. Например, оно гибче и хорошо структурированно.

В целях понимания объемометрической отрисовки сперва нужно понять несколько важных понятий, таких как отбор лучей, объемы, отбор объемов и лучевая маршировка. Все эти понятия имеют соответствующие реализации в библиотеке PyTorch3D. Мы обсудим каждое из этих понятий, используя объяснения и примеры программирования.

После того как вы поймете приведенные выше базовые понятия объемометрической отрисовки, вы легко увидите, что все упомянутые операции уже дифференцируемы. Объемометрическая отрисовка естественным образом дифференцируема. И следовательно, к тому времени вы уже будете готовы применить дифференцируемую объемометрическую отрисовку в нескольких практических приложениях. Мы рассмотрим пример программирования реконструкции трехмерных воксельных моделей по нескольким изображениям с использованием дифференцируемой объемометрической отрисовки.

Сначала мы рассмотрим объемометрическую отрисовку в общем плане. Затем углубимся в базовые понятия, такие как отбор лучей, объемы, отбор объемов и лучевая маршировка. Затем мы представим пример программирования реконструкции форм 3D-объектов по коллекции изображений, снятых под разным углом зрения на объект.

В данной главе будут охвачены следующие ниже главные темы:

- высокоуровневое описание объемометрической отрисовки,
- понятие отбора лучей,
- использование отбора объемов,
- понятие лучевой маршировки,
- реконструкция 3D-объектов и значений цвета по многоракурсным изображениям.

Технические требования

Для выполнения примеров исходного кода этой книги в идеале понадобится компьютер с графическим процессором. Тем не менее для выполнения фрагментов исходного кода вполне будет достаточно только центрального процессора(ов).

Рекомендуемая компьютерная конфигурация включает следующее:

- GPU, такой как серия GTX или серия RTX с не менее 8 Гб памяти,
- Python 3,
- библиотеки PyTorch и PyTorch3D.

Фрагменты исходного кода к этой главе находятся по адресу https://github.com/packtpublishing/3d-deep-learning-with-python.

Общий обзор объемометрической отрисовки

Объемометрическая отрисовка – это набор технических приемов, используемых для генерирования 2D-проекции дискретных 3D-данных. Дискретные 3D-данные могут быть коллекцией изображений, воксельным представлением или любым другим дискретным представлением данных. Главная цель объемометрической отрисовки состоит в отрисовке 2D-проекции 3D-данных, поскольку это то, что наши глаза могут воспринимать на плоском экране. Этот метод генерирует такие проекции без какой-либо явной конверсии в геометрическое представление (такое как полигональные сетки). Объемометрическая отрисовка обычно используется в ситуациях, когда генерирование поверхностей затруднено или может приводить к ошибкам. Ее также можно использовать в ситуациях, когда имеет важность содержимое (а не только геометрия и поверхность) объема. Она обычно используется для визуализации данных. Например, в сканировании мозга обычно очень важна визуализация содержимого внутренней части мозга.

В этом разделе мы рассмотрим объемометрическую отрисовку объема. Вы получите обобщенный обзор объемометрической отрисовки, как показано на рис. 5.1.

1. Прежде всего мы представляем 3D-пространство и объекты в нем с помощью **объема**, который является не чем иным, как трехмерной ре-

шеткой регулярно расположенных узлов. Каждый узел имеет два свойства: плотность и цвет. Плотность обычно колеблется в интервале от 0 до 1. Плотность также можно трактовать как вероятность занятости, т. е. с какой степенью уверенности мы считаем, что узел занят определенным объектом. В некоторых случаях вероятностью также может быть непрозрачность.

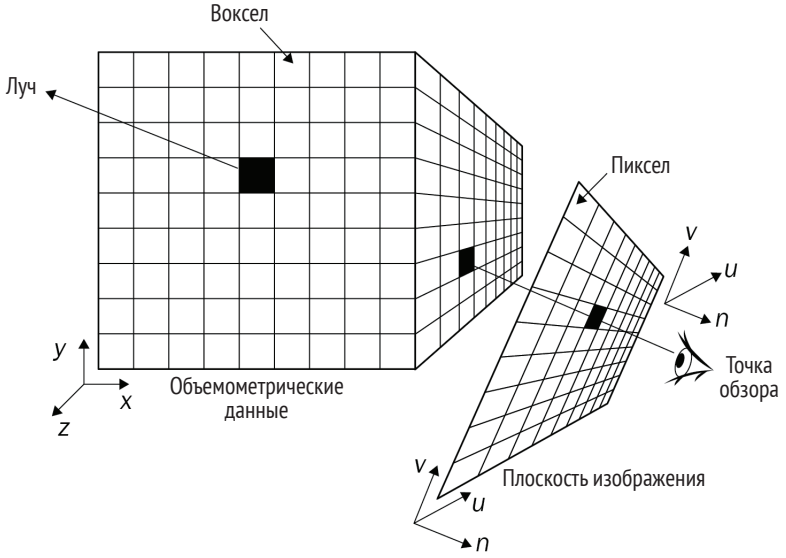


Рис. 5.1 ❖ Объемометрическая отрисовка

2. Нам нужно определить одну или несколько камер. Отрисовка – это процесс, который определяет, что именно камеры могут наблюдать со своего ракурса.
3. В целях определения RGB-значений в каждом пикселе указанных выше камер из центра проекции генерируется луч, проходящий через каждый пиксел камер. При этом нужно проверить вероятность занятости или непрозрачности и значений цвета вдоль этого луча, чтобы определить RGB-значения пиксела. Обратите внимание, что на каждом таком луче бесконечно много точек. Следовательно, необходимо иметь схему отбора, чтобы отбирать определенное число точек вдоль этого луча. Эта операция отбора называется **отбором лучей**[1].
4. Обратите внимание, что у нас есть плотности и цвета, определенные в узлах объема, но не в точках на лучах. Следовательно, требуется способ конвертировать плотности и цвета объемов в точки на лучах. Эта операция называется **отбором из объемов**[2].
5. Наконец, из плотностей и значений цвета лучей необходимо определить RGB-значения каждого пиксела. В этом процессе требуется вы-

[1] От англ. *ray sampling. – Прим. перев.*

[2] От англ. *volume sampling. – Прим. перев.*

числить, сколько падающих световых сигналов может достичь каждой точки вдоль луча и сколько световых сигналов отразится на пиксел изображения. Этот процесс называется **лучевой маршировкой**[1].

Поняв базовый процесс объемометрической отрисовки, давайте рассмотрим подробнее первое понятие: отбор лучей.

Понятие отбора лучей

Отбор лучей – это процесс испускания лучей из камеры, которые проходят через пикселы изображения и отобранные точки вдоль этих лучей. Схема отбора лучей зависит от варианта использования. Например, иногда бывает необходимо делать случайный отбор лучей, проходящих через некоторое случайное подмножество пикселов изображения. Как правило, отборщик[2] необходимо использовать во время тренировки, поскольку требуется только репрезентативная выборка из полных данных. В таких случаях можно использовать лучевой отборщик Pytorch3D `MonteCarloRaysampler`. В других случаях требуется получить пиксельные значения по каждому пикселу изображения и поддерживать пространственный порядок. Это удобно для отрисовки и визуализации. Для таких случаев использования библиотека PyTorch3D предоставляет лучевой отборщик `NDCMultiNomialRaysampler`.

Далее мы продемонстрируем применение одного из лучевых отборщиков PyTorch3D `NDCGridRaysampler`. Он похож на лучевой отборщик `NDCMultiNomialRaysampler`, в котором пикселы отбираются вдоль решетки. Соответствующий исходный код находится в репозитории книги на GitHub в файле Python `understand_ray_sampling.py`.

1. Сначала необходимо импортировать все модули Python, включая определение лучевого отборщика `NDCGridRaysampler`:

```
import torch
import math
import numpy as np

from pytorch3d.renderer import (
    FoVPerspectiveCameras,
    PointLights,
    look_at_view_transform,
    NDCGridRaysampler,
)
```

2. Настраиваем устройство для использования в следующих далее шагах. Если у вас имеются графические процессоры, то будем использовать первый графический процессор. В противном случае будем использовать центральный процессор:

---

[1] От англ. *ray marching*. – *Прим. перев.*

[2] Син. семплер. – *Прим. перев.*

```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    torch.cuda.set_device(device)
else:
    device = torch.device("cpu")
```

3. Определяем пакет из 10 камер. Здесь `num_views` – это число ракурсов, в данном случае 10. Переменная `elev` обозначает угол возвышения, а `azim` – угол азимута. Указанные переменные позволяют определить поворот `R` и трансляцию `T` с помощью функции PyTorch3D `look_at_view_transform`. Тогда 10 камер можно определить, используя повороты и трансляции. Все 10 камер направлены на объект, расположенный в центре мировых координат:

```
num_views: int = 10
azimuth_range: float = 180

elev = torch.linspace(0, 0, num_views)
azim = torch.linspace(-azimuth_range,
                      azimuth_range,
                      num_views) + 180.0
lights = PointLights(device=device,
                     location=[[0.0, 0.0, -3.0]])
R, T = look_at_view_transform(dist=2.7,
                              elev=elev,
                              azim=azim)
cameras = FoVPerspectiveCameras(device=device,
                                R=R, T=T)
```

4. Наконец, можно определить лучевой отборщик, т. е. переменную `raysampler`. Далее, нужно указать размер изображения в камере. Кроме того, требуется указать минимальную и максимальную глубины, в которых луч колеблется. Входной параметр `n_pts_per_ray` – это число точек вдоль луча:

```
image_size = 64
volume_extent_world = 3.0

raysampler = NDCGridRaysampler(
    image_width=image_size,
    image_height=image_size,
    n_pts_per_ray=50,
    min_depth=0.1,
    max_depth=volume_extent_world,
)
```

5. На предыдущем шаге мы уже определили лучевой отборщик. Для того чтобы лучевой отборщик делал выборку вдоль луча подлежащих использованию точек, необходимо ему сообщить, где находятся камеры и в каком направлении они указывают. Этого делается легко путем

передачи в raysampler камер, определенных на шаге 3. На выходе мы получаем переменную ray_bundle[1]:

```
ray_bundle = raysampler(cameras)
```

6. Переменная ray_bundle содержит коллекцию разных тензоров PyTorch, которые определяют отобранные лучи и точки. Эти переменные можно распечатать, чтобы проверить их очертание и содержимое:

```
print('Очертание тензора источников в ray_bundle = ',
      ray_bundle.origins.shape)
print('Очертание тензора направлений в ray_bundle = ',
      ray_bundle.directions.shape)
print('Очертание тензора длин в ray_bundle = ',
      ray_bundle.lengths.shape)
print('Очертание тензора xy-местоположений в ray_bundle = ',
      ray_bundle.xys.shape)
```

7. Исходный код должен отработать и вывести следующую ниже информацию.
   - Мы видим, что ray_bundle.origins – это тензор источников лучей, и размер пакета равен 10. Поскольку размер изображения равен 64×64, размер второй и третьей размерностей равен 64. У каждого источника должно быть три числа, чтобы указывать его 3D-местоположение.
   - ray_bundle.directions – это тензор направлений луча. Опять же, размер пакета равен 10, и размер изображения – 64×64. Они объясняют размер первых трех размерностей тензора. Должно быть три числа, чтобы указывать направление в 3D-пространстве.
   - ray_bundle.lengths – это тензор глубины каждой точки на лучах. Это лучи 10×64×64, и на каждом луче 50 точек.
   - ray_bundle.xys – это тензор местоположений *x* и *y* на плоскости изображения, соответствующей каждому лучу. Это лучи 10×64×64. Должно быть одно число для представления местоположения *x* и одно число для представления местоположения *y*:

```
Очертание тензора источников в ray_bundle = torch.Size([10, 64, 64, 3])
Очертание тензора направлений в ray_bundle = torch.Size([10, 64, 64, 3])
Очертание тензора длин в ray_bundle = torch.Size([10, 64, 64, 50])
Очертание тензора xy-местоположений в ray_bundle = torch.Size([10, 64, 64, 2])
```

8. Наконец, сохраняем ray_bundle в PT-файл ray_sampling.pt. Эти лучи пригодятся в примерах программирования в следующих разделах:

```
torch.save({'ray_bundle': ray_bundle}, 'ray_sampling.pt')
```

К настоящему времени вы уже должны разбираться в том, что делают лучевые отборщики. Лучевые отборщики дают пакет лучей и дискретных точек

[1] Пучок лучей. – *Прим. перев.*

на лучах. Однако у нас еще нет определений плотностей и значений цвета на этих точках и лучах. Далее вы научитесь получать эти плотности и цвета из объемов.

Применение отбора объемов

Отбор объемов – это процесс получения информации о цвете и занятости точек, предоставленных выборами лучей. Мы работаем с дискретным представлением объема. Поэтому точки, определенные на шаге отбора лучей, возможно, не будут попадать точно в точку. Узлы решеток объемов и точки на лучах обычно имеют разные пространственные местоположения. Нужна схема интерполяции, чтобы интерполировать плотности и значения цвета в точках лучей из плотностей и значений цвета в объемах. Это делается с помощью реализованного в PyTorch3D объемного отборщика `VolumeSampler`. Следующий ниже исходный код находится в репозитории книги на GitHub в файле Python `understand_volume_sampler.py`:

1. Импортируем нужные модули Python:

```
import torch
from pytorch3d.structures import Volumes
from pytorch3d.renderer.implicit.renderer import VolumeSampler
```

2. Настраиваем устройство:

```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    torch.cuda.set_device(device)
else:
    device = torch.device("cpu")
```

3. Загружаем вычисленный в предыдущем разделе `ray_bundle`:

```
checkpoint = torch.load('ray_sampling.pt')
ray_bundle = checkpoint.get('ray_bundle')
```

4. Затем определяем объем. Тензор плотностей имеет очертание `[10, 1, 64, 64, 50]`, в котором имеется пакет из 10 объемов, и каждый объем представляет собой решетку из 64×64×50 узлов. Каждый узел имеет одно число, которое представляет плотность в узле. С другой стороны, тензор значений цвета имеет очертание `[10, 3, 64, 64, 50]`, потому что для представления RGB-значений каждому цвету требуется три числа:

```
batch_size = 10
densities = torch.zeros(
    [batch_size, 1, 64, 64, 64]).to(device)
colors = torch.zeros(
    batch_size, 3, 64, 64, 64).to(device)
voxel_size = 0.1
```

```
volumes = Volumes(
    densities=densities,
    features=colors,
    voxel_size=voxel_size
)
```

5. Теперь необходимо определить отборщик `volume_sampler` на основе объемов. Здесь для отбора объемов используется билинейная интерполяция. Затем плотности и значения цвета точек на лучах можно легко получить, передав пучок лучей `ray_bundle` в отборщик `volume_sampler`:

```
volume_sampler = VolumeSampler(volumes=volumes,
                               sample_mode="bilinear")
rays_densities, rays_features = volume_sampler(ray_bundle)
```

6. Распечатаем очертание плотностей и значений цвета:

```
print('Очертание rays_densities = ', rays_densities.shape)
print('Очертание rays_features = ', rays_features.shape)
```

7. Результат показан ниже. Первое, на что стоит обратить внимание, – это то, что у нас пакет имеет размер 10, т. е. состоит из 10 камер, что объясняет первую размерность тензоров. Далее, для каждого пиксела изображения у нас имеется один луч, а разрешающая способность изображения камеры составляет 64×64. Число точек на каждом луче равно 50, что объясняет четвертую размерность тензоров. Каждая плотность может быть представлена одним числом, а каждому цвету требуется три числа, чтобы представлять RGB-значения:

```
Очертание rays_densities = torch.Size([10, 64, 64, 50, 1])
Очертание rays_features = torch.Size([10, 64, 64, 50, 3])
```

8. Наконец, давайте сохраним плотности и цвета, потому что они потребуются в следующем разделе:

```
torch.save({'rays_densities': rays_densities,
            'rays_features': rays_features
}, 'volume_sampling.pt')
```

Теперь у вас есть четкое понимание отбора объемов. Вы знаете, что это такое и в чем его польза. В следующем разделе вы научитесь применять эти плотности и цвета, чтобы генерировать значения RGB-изображения для группы камер.

Обследование лучевого маршировщика

Теперь, когда у нас есть значения цвета и плотности всех точек, отобранных с помощью отборщика лучей, нужно выяснить, как его использовать для окончательной отрисовки значения пиксела на проецируемом изображении.

В этом разделе мы обсудим процесс конвертирования плотностей и значений цвета в точках лучей в RGB-значения на изображениях. Этот процесс моделирует физический процесс формирования изображения.

В данном разделе мы обсудим очень простую модель, в которой RGB-значение каждого пиксела изображения представляет собой взвешенную сумму значений цвета в точках соответствующего луча. Если трактовать плотности как вероятности занятости или непрозрачности, то интенсивность падающего света в каждой точке луча равна произведению $a = \prod_{i=1}(1 - p_i)$, где p_i – это плотности. Учитывая вероятность того, что эта точка занята определенным объектом с p_i, ожидаемая интенсивность отраженного от этой точки света равна $w_i = ap_i$. Мы используем w_i просто в качестве весов для взвешенной суммы значений цвета. Обычно веса нормализуются, применяя операцию мягкого максимума (softmax), чтобы все веса в сумме составляли единицу.

Библиотека PyTorch3D содержит несколько реализаций лучевых маршировщиков. Следующий ниже исходный код находится в репозитории книги на GitHub в файле Python `understand_ray_marcher.py`.

1. На первом шаге импортируем все необходимые пакеты:

```
import torch
from pytorch3d.renderer.implicit.raymarching import \
    EmissionAbsorptionRaymarcher
```

2. Далее загружаем плотности и значения цвета на лучах из последнего раздела:

```
checkpoint = torch.load('volume_sampling.pt')
rays_densities = checkpoint.get('rays_densities')
rays_features = checkpoint.get('rays_features')
```

3. Определяем лучевого маршировщика `ray_marcher` и передаем плотности и цвета лучей в `ray_marcher`. В результате получаем признаки изображения `image_features`, т. е. точно отрисованные RGB-значения:

```
ray_marcher = EmissionAbsorptionRaymarcher()
image_features = ray_marcher(rays_densities=rays_densities,
                             rays_features=rays_features)
```

4. Распечатываем очертание признаков изображения:

```
print('Очертание image_features = ', image_features.shape)
```

5. Как и ожидалось, очертание имеет вид `[10, 64, 64, 4]`, где 10 – это размер пакета, а 64 – ширина и высота изображения. Выходы имеют четыре канала, первые три – это RGB. Последний – это альфа-канал, который представляет местоположение пиксела: на переднем либо заднем плане:

```
Очертание image_features = torch.Size([10, 64, 64, 4])
```

Мы прошлись по нескольким базовым компонентам объемометрической отрисовки. Обратите внимание, что процесс вычисления, начиная от

плотностей объемов и значений цвета и вплоть до RGB-значений пикселов изображения, уже дифференцируем. И следовательно, объемометрическая отрисовка естественным образом является дифференцируемой. Учитывая, что все переменные в приведенных выше примерах являются тензорами PyTorch, есть возможно вычислять для них градиенты.

В следующем разделе вы познакомитесь с дифференцируемой объемометрической отрисовкой. Мы рассмотрим пример применения дифференцируемой объемометрической отрисовки с целью реконструкции 3D-моделей по многоракурсным изображениям.

Дифференцируемая объемометрическая отрисовка

В то время как стандартная объемометрическая отрисовка используется для отрисовки 2D-проекций 3D-данных, дифференцируемая объемометрическая отрисовка используется для противоположного: конструирования 3D-данных по 2D-изображениям. И вот как это работает: форма и текстура объекта представляется параметрической функцией. Эту функцию можно использовать для генерирования 2D-проекций. Но, имея 2D-проекции (обычно это несколько ракурсов 3D-сцены), существует возможность оптимизировать параметры этих неявных функций формы и текстуры, чтобы их проекции были многоракурсными 2D-изображениями. Возможность такой оптимизации обусловлена тем, что процесс отрисовки полностью дифференцируем, и используемые неявные функции тоже дифференцируемы.

Реконструкция 3D-моделей по многоракурсным изображениям

В этом разделе мы покажем пример использования дифференцируемой объемометрической отрисовки с целью реконструкции 3D-моделей по многоракурсным изображениям. Задача реконструкции 3D-моделей встречается довольно часто. Обычно прямые способы измерения 3D-мира имеют свои сложности и дорогостоящи, например в типичной ситуации LiDAR и Radar обходятся дорого. С другой стороны, 2D-камеры имеют гораздо меньшую стоимость, что делает реконструкцию 3D-мира по 2D-изображениям невероятно привлекательной. Разумеется, для того чтобы реконструировать трехмерный мир, нужно иметь несколько изображений с разных ракурсов.

Следующий ниже исходный код в файле Python `volume_render.py` находится в репозитории книги на GitHub и является видоизмененной версией исходного кода из практического руководства по библиотеке PyTorch3D. Мы будем использовать этот пример исходного кода, чтобы показать, каким

может быть реальное применение дифференцируемой объемометрической отрисовки.

1. Сначала необходимо импортировать все модули Python:

```
import os
import sys
import time
import json
import glob

import torch
import math
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

from pytorch3d.structures import Volumes
from pytorch3d.renderer import (
    FoVPerspectiveCameras, VolumeRenderer,
    NDCGridRaysampler, EmissionAbsorptionRaymarcher
)
from pytorch3d.transforms import so3_exp_map

from plot_image_grid import image_grid
from generate_cow_renders import generate_cow_renders
```

2. Далее настраиваем устройство:

```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    torch.cuda.set_device(device)
else:
    device = torch.device("cpu")
```

3. Используя функцию из практического руководства по PyTorch3D, мы создаем 40 камер, изображения и силуэтные изображения под разными углами. Мы будем рассматривать эти изображения как заданные достоверные изображения и выполним подгонку трехмерной объемометрической модели к этим наблюдаемым достоверным изображениям:

```
target_cameras, target_images, target_silhouettes = \
    generate_cow_renders(num_views=40)
```

4. Далее определяем отборщик лучей. Как отмечалось в предыдущих разделах, отборщик лучей предназначен для отбора лучей и точек на каждом луче:

```
render_size = 128
volume_extent_world = 3.0
raysampler = NDCGridRaysampler(
    image_width=render_size,
```

```
    image_height=render_size,
    n_pts_per_ray=150,
    min_depth=0.1,
    max_depth=volume_extent_world,
)
```

5. Далее создаем лучевого маршировщика, как и раньше. Обратите внимание, на этот раз мы определяем переменную `renderer` типа `VolumeRenderer`. Класс `VolumeRenderer` – это просто удобный интерфейс, в котором отборщики лучей и лучевые маршировщики выполняют под капотом всю тяжелую работу:

```
raymarcher = EmissionAbsorptionRaymarcher()
renderer = VolumeRenderer(
    raysampler=raysampler,
    raymarcher=raymarcher,
)
```

6. Затем определяем класс `VolumeModel`. Этот класс предназначен только для инкапсуляции объема, чтобы иметь возможность вычислять градиенты в функции `forward`, а плотности и цвета объема обновлять оптимизатором:

```
class VolumeModel(torch.nn.Module):
    def __init__(self, renderer,
                 volume_size=[64] * 3,
                 voxel_size=0.1):
        super().__init__()
        self.log_densities = torch.nn.Parameter(
            -4.0 * torch.ones(1, *volume_size))
        self.log_colors = torch.nn.Parameter(
            torch.zeros(3, *volume_size))
        self._voxel_size = voxel_size
        self._renderer = renderer

    def forward(self, cameras):
        batch_size = cameras.R.shape[0]
        densities = torch.sigmoid(self.log_densities)
        colors = torch.sigmoid(self.log_colors)
        volumes = Volumes(
            densities=densities[None].expand(
            batch_size, *self.log_densities.shape),
            features=colors[None].expand(
            batch_size, *self.log_colors.shape),
            voxel_size=self._voxel_size,
        )
        return self._renderer(cameras=cameras,
                              volumes=volumes)[0]
```

7. Определяем функцию потери Хьюбера. Функция потери Хьюбера – это устойчивая функция потери, не дающая малому числу выбросов оття-

гивать оптимизацию от истинных оптимальных решений. Минимизация этой функции потери приближает *x* к *y*:

```
def huber(x, y, scaling=0.1):
    diff_sq = (x - y) ** 2
    loss = ((1 + diff_sq / (scaling ** 2)
             ).clamp(1e-4).sqrt() - 1) * float(scaling)
    return loss
```

8. Перемещаем все на нужное устройство:

```
target_cameras = target_cameras.to(device)
target_images = target_images.to(device)
target_silhouettes = target_silhouettes.to(device)
```

9. Определяем экземпляр класса `VolumeModel`:

```
volume_size = 128
volume_model = VolumeModel(
    renderer,
    volume_size=[volume_size] * 3,
    voxel_size=volume_extent_world / volume_size,
).to(device)
```

10. Теперь настраиваем оптимизатор. Значение скорости обучения `lr` установлено равным `0.1`. Мы используем оптимизатор Adam, и число итераций оптимизации будет равно `300`:

```
lr = 0.1
optimizer = torch.optim.Adam(volume_model.parameters(),
                             lr=lr)
batch_size = 10
n_iter = 300
```

11. Далее идет главный цикл оптимизации. Отрисовываются плотности и цвета объема, а полученные цвета и силуэты сравниваются с наблюдаемыми многоракурсными изображениями. Потеря Хьюбера между отрисованными изображениями и наблюдаемыми достоверными изображениями минимизирована:

```
for iteration in range(n_iter):
    if iteration == round(n_iter * 0.75):
        print('Уменьшаем LR в 10 раз...')
        optimizer = torch.optim.Adam(
            volume_model.parameters(), lr=lr * 0.1
        )

    optimizer.zero_grad()
    batch_idx = torch.randperm(len(target_cameras))[:batch_size]

    # Произвести выборку мини-пакета камер.
    batch_cameras = FoVPerspectiveCameras(
        R=target_cameras.R[batch_idx],
```

```
            T=target_cameras.T[batch_idx],
            znear=target_cameras.znear[batch_idx],
            zfar=target_cameras.zfar[batch_idx],
            aspect_ratio=target_cameras.aspect_ratio[batch_idx],
            fov=target_cameras.fov[batch_idx],
            device=device,
        )

        rendered_images, rendered_silhouettes = volume_model(
            batch_cameras).split([3, 1], dim=-1)

        sil_err = huber(
            rendered_silhouettes[..., 0],
            target_silhouettes[batch_idx],
            ).abs().mean()

        color_err = huber(
            rendered_images,
            target_images[batch_idx],
            ).abs().mean()

        loss = color_err + sil_err

        # Сделать шаг оптимизации.
        loss.backward()
        optimizer.step()
```

12. После завершения оптимизации берем итоговую результирующую объемометрическую модель и отрисовываем изображения под новыми углами:

```
with torch.no_grad():
   rotating_volume_frames = generate_rotating_volume(
      volume_model, n_frames=7 * 4)

image_grid(rotating_volume_frames.clamp(0., 1.).cpu().numpy(),
                                        rows=4, cols=7,
                                        rgb=True, fill=True)
plt.savefig('rotating_volume.png')
plt.show()
```

13. Новые отрисованные изображения показаны на рис. 5.2.

К настоящему моменту у вас есть общее понимание нескольких главных понятий дифференцируемой объемометрической отрисовки. Вы также научились применять дифференцируемую объемометрическую отрисовку с целью реконструкции 3D-моделей по многоракурсным изображениям на конкретном примере. У вас есть все необходимое для того, чтобы овладеть навыками применения данной техники и уметь ее использовать для решения своих собственных задач.

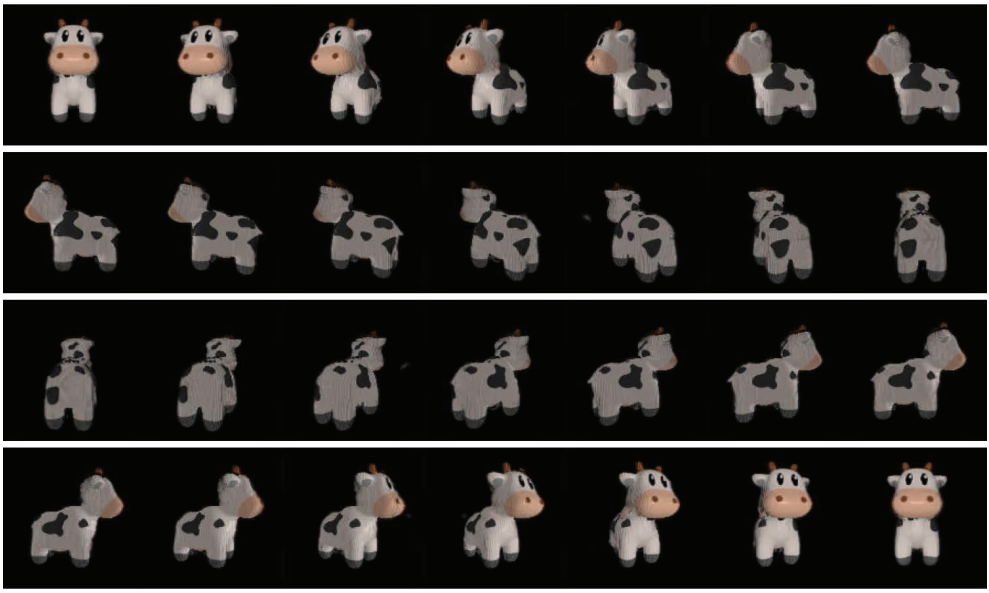


Рис. 5.2 ❖ Отрисованные изображения подогнанной 3D-модели

Резюме

В этой главе мы начали с обобщенного описания дифференцируемой объемометрической отрисовки. Затем мы углубились в несколько важных понятий объемометрической отрисовки, включая отбор лучей, отбор объемов и лучевую маршировку, с пояснениями и примерами программирования. Мы рассмотрели пример исходного кода с использованием дифференцируемой объемометрической отрисовки с целью реконструкции 3D-моделей по многоракурсным изображениям.

Использование объемов для трехмерного глубокого обучения в последние годы стало интересным направлением. Поскольку в этом направлении появляется много инновационных идей, появляется и много прорывов. Одно из открытий под названием «нейронные поля яркости излучения» (NeRF) станет темой следующей главы.

Глава 6

Обследование нейронных полей яркости излучения (NeRF)

В предыдущей главе вы познакомились с дифференцируемой объемометрической отрисовкой, в которой вы реконструировали 3D-объем по нескольким многоракурсным изображениям. С помощью этой техники вы смоделировали объем, состоящий из N × N × N вокселов. Следовательно, потребность в пространстве для хранения этой объемной шкалы составляет $O(N^3)$. Это нежелательно, в особенности если мы хотим передавать эту информацию по сети. Другие методы могут преодолевать такие большие требования к дисковому пространству, но они склонны к сглаживанию геометрии и текстуры. Поэтому их нельзя использовать для надежного моделирования очень сложных или текстурированных сцен.

В этой главе мы обсудим новый революционный подход к представлению 3D-сцен, который называется **нейронными полями яркости излучения** (**NeRF**)[1]. Это один из первых методов моделирования 3D-сцены, который требует меньше постоянного дискового пространства и в то же время улавливает точную геометрию и текстуру сложных сцен.

В этой главе вы познакомитесь со следующими ниже темами:

- концепцией нейронных полей яркости излучения (NeRF),
- тренировкой модели NeRF,
- пониманием архитектуры модели NeRF,
- концепцией объемометрической отрисовки с полями яркости излучения.

---

[1] От англ. *Neural Radiance Fields. – Прим. перев.*

Технические требования

Для выполнения примеров исходного кода этой книги в идеале понадобится компьютер с графическим процессором. Выполнение фрагментов исходного кода с использованием центрального процессора(ов) невозможно и будет чрезвычайно медленным. Рекомендуемая конфигурация компьютера приводится ниже:

- устройство GPU, например серия GTX или серия RTX с не менее 8 Гб памяти,
- Python 3.7+,
- библиотеки PyTorch и PyTorch3D.

Фрагменты исходного кода к этой главе находятся по адресу https://github.com/packtpublishing/3d-deep-learning-with-python.

Концепция нейронных полей яркости излучения (NeRF)

Синтез ракурсов, или точек обзора, – это давняя задача в трехмерном компьютерном зрении. Ее трудность состоит в том, что приходится синтезировать новые ракурсы 3D-сцены, используя малое число доступных 2D-снимков сцены. Это особенно сложно, потому что ракурс сложной сцены зависит от большого числа факторов, таких как артефакты объекта, источники света, отражения, непрозрачность, текстура поверхности объекта и загораживания[1]. Любое хорошее представление должно улавливать эту информацию в явной или неявной форме. Кроме того, многие объекты имеют сложную структуру, которая видна не полностью под определенным ракурсом. Задача состоит в том, чтобы построить полную информацию о мире, имея неполную и зашумленную информацию.

Как следует из названия, в методе NeRF используются нейронные сети, чтобы моделировать мир. Как вы узнаете позже в этой главе, в методе NeRF нейронные сети используются весьма неконвенциональным способом. Эта концепция была впервые разработана группой исследователей из Калифорнийского университета в Беркли, исследовательского центра Google Research и Калифорнийского университета в Сан-Диего. Из-за их неконвенционального использования нейронных сетей и качества усваиваемых моделей данный метод породил целый ряд новых изобретений в области синтеза ракурсов изображения, определения глубины и 3D-реконструкции. Таким образом, это очень полезная концепция, в которой следует хорошо разобраться, пока вы будете читать оставшуюся часть этой главы и книги.

В этом разделе сначала мы обследуем смысл идеи полей яркости излучения и способы использования нейронной сети для представления полей яркости излучения.

---

[1] Син. окклюзии. – *Прим. перев.*

Что такое поле яркости излучения?

Прежде чем перейти к методу NeRF, давайте сначала разберемся, что такое поля яркости излучения. Вы видите объект, когда свет от этого объекта обрабатывается сенсорной системой вашего тела. Свет от объекта может либо генерироваться самим объектом, либо от него отражаться.

Яркость излучения, или сияние, – это стандартная метрика измерения величины света, который проходит сквозь или излучается из области внутри определенного телесного угла. Для наших целей яркость излучения можно трактовать как интенсивность точки в пространстве при обзоре в том или ином направлении. При улавливании этой информации в формате RGB яркость излучения будет состоять из трех компонент, соответствующих красному, зеленому и синему цветам. Яркость излучения точки в пространстве может зависеть от многих факторов, включая следующие:

- источники света, освещающие эту точку,
- наличие поверхности (или плотности объема), которая может отражать свет в этой точке,
- текстурные свойства поверхности.

На следующем ниже рисунке показано значение яркости излучения в определенной точке на 3D-сцене при просмотре под определенным углом. Поле яркости излучения – это просто коллекция этих значений яркости во всех точках и углах обзора в 3D-сцене:

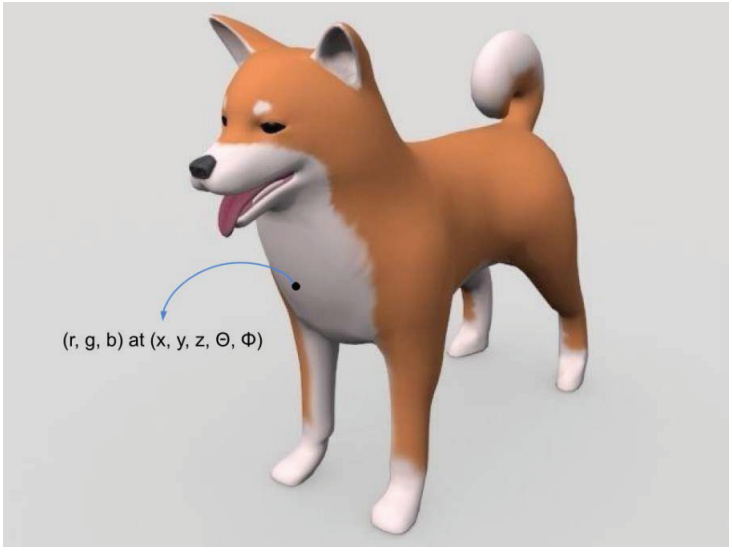


Рис. 6.1 ❖ Яркость излучения (r, g, b) в точке (x, y, z) при просмотре под определенным углом обзора (θ, Ø)

Если знать яркость излучения всех точек сцены во всех направлениях, то у нас будет вся необходимая визуальная информация о сцене. Это множе-

ство значений яркости излучения составляет поле яркости излучения. Информацию о поле яркости излучения можно хранить как объем в структуре данных, реализованной в виде трехмерной воксельной решетки. Вы встречали ее в предыдущей главе, когда мы обсуждали тему объемометрической отрисовки.

Представление полей яркости излучения с помощью нейронных сетей

В этом разделе мы рассмотрим новый способ использования нейронных сетей. В типичных задачах компьютерного зрения нейронные сети применяются для соотнесения входных данных в пиксельном пространстве с выходными данными. В случае дискриминативной модели результатом является метка класса. В случае генеративной модели результат все также находится в пиксельном пространстве. Модель NeRF не является ни тем, ни другим.

В модели NeRF нейронная сеть используется для представления объемометрической функции сцены. Нейронная сеть принимает 5-мерные входные данные. Это три пространственных местоположения (x, y, z) и два угла обзора (θ, Ø). Ее выходными данными являются объемная плотность σ в точке (x, y, z) и излучаемый цвет (r, g, b) точки (x, y, z) при наблюдении под углом обзора (θ, Ø). Излучаемый цвет – это косвенный индикатор, используемый для оценивания яркости излучения в этой точке. На практике вместо прямого использования (θ, Ø) для представления угла обзора в NeRF применяется единичный вектор направления d в трехмерной декартовой системе координат. Это представление эквивалентно углу обзора.

Таким образом, модель отображает любую точку 3D-сцены и угол обзора в плотность объема и яркость излучения в этой точке. Тогда эта модель может использоваться для синтеза ракурсов, опрашивая 5D-координаты вдоль лучей камеры и используя технику объемометрической отрисовки, с которой вы познакомились в предыдущей главе, чтобы проецировать выходные значения цвета и плотности объемов в изображение.

На рис. 6.2 давайте разберемся, как использовать нейронную сеть для предсказывания плотности и яркости излучения в определенной точке (x, y, z) при просмотре в определенном направлении (θ, Ø).

Обратите внимание, что это полносвязная сеть – обычно ее называют **многослойным персептроном** (**MLP**)[1]. Что еще важнее, это не сверточная нейронная сеть. Эта модель называется моделью NeRF. Одна модель NeRF оптимизируется на множестве изображений из одной сцены. Следовательно, каждая модель знает только ту сцену, на которой она оптимизирована. Такой подход к использованию нейронной сети отличается своей нестандартностью – обычно модель нужна для обобщения ранее неизвестных изображений, в случае же модели NeRF сеть нужна для того, чтобы хорошо обобщать ранее неизвестные точки обзора.

---

[1] От англ. *Multi-Layer Perceptron. – Прим. перев.*

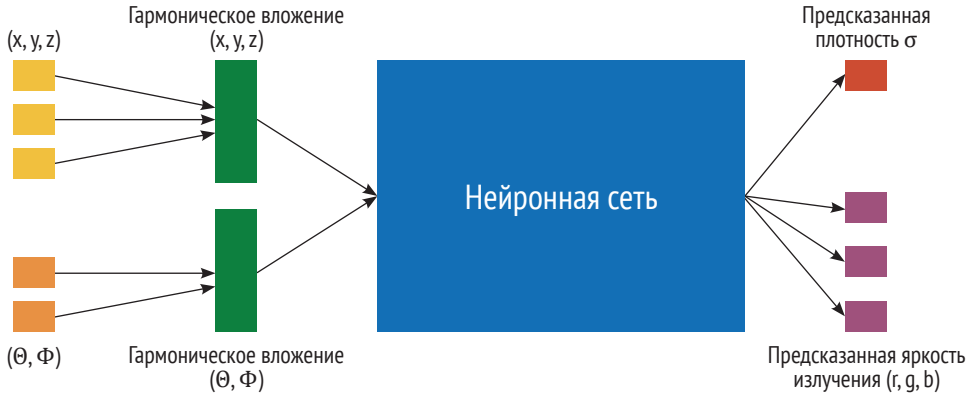


Рис. 6.2 ❖ В начале входные данные (x, y, z, θ и Ø) используются для создания отдельных гармонических вложений для пространственного местоположения и угла обзора, формируя данные, подаваемые на вход в нейронную сеть, и затем эта нейронная сеть выводит предсказанные плотность и яркость излучения

Теперь, когда вы знаете, что такое модель NeRF, давайте взглянем на способы применения данного метода для отрисовки новых ракурсов.

Тренировка модели NeRF

В этом разделе мы натренируем простую модель NeRF на изображениях, созданных из модели синтетической коровы. Мы создадим только экземпляр модели NeRF, не беспокоясь о том, как она устроена. Детали реализации описаны в следующем далее разделе. Одна нейронная сеть (модель NeRF) тренируется представлять одну 3D-сцену. Следующий ниже исходный код находится в файле Python `train_nerf.py` этой главы книги в репозитории на GitHub. Он представляет собой видоизмененную версию примера из практического руководства по библиотеке PyTorch3D. Давайте пройдемся по исходному коду тренировки модели NeRF на сцене синтетической коровы.

1. Сначала импортируем стандартные модули:

```
import torch
import matplotlib.pyplot as plt
```

2. Далее импортируем используемые для отрисовки функции и классы. Это структуры данных `pytorch3d`:

```
from pytorch3d.renderer import (
    FoVPerspectiveCameras,
    NDCMultinomialRaysampler,
    MonteCarloRaysampler,
    EmissionAbsorptionRaymarcher,
    ImplicitRenderer,
)
```

```
from utils.helper_functions import (generate_rotating_nerf,
                                    huber,
                                    show_full_render,
                                    sample_images_at_mc_locs)

from nerf_model import NeuralRadianceField
```

3. Далее необходимо настроить устройство:

```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    torch.cuda.set_device(device)
else:
    device = torch.device("cpu")
```

4. Теперь импортируем функции-утилиты, которые позволят генерировать синтетические тренировочные данные и визуализировать изображения:

```
from utils.plot_image_grid import image_grid
from utils.generate_cow_renders import generate_cow_renders
```

5. Теперь можно применить эти функции-утилиты, чтобы сгенерировать ракурсы камеры, изображения и силуэты синтетической коровы под разными углами зрения. В результате будет распечатано число сгенерированных изображений, силуэтов и ракурсов камеры:

```
target_cameras, target_images, target_silhouettes = \
    generate_cow_renders(num_views=40, azimuth_range=180)
print(f'Сгенерировано {len(target_images)} изображений/силуэтов/камер.')
```

6. Как и в предыдущей главе, теперь определим отборщика лучей. Мы будем использовать `MonteCarloRaysampler`. Он генерирует лучи из случайного подмножества пикселов из плоскости изображения. Здесь нужен случайный отборщик, так как мы хотим использовать алгоритм мини-пакетного градиентного спуска, чтобы оптимизировать модель. Это стандартный метод оптимизации нейронной сети. Систематический отбор лучей может приводить к смещению оптимизации на каждом этапе оптимизации, а это в свою очередь – к ухудшению модели и увеличению времени тренировки модели. Отборщик лучей извлекает выборку точек равномерно вдоль луча:

```
render_size = target_images.shape[1] * 2

volume_extent_world = 3.0

raysampler_mc = MonteCarloRaysampler(
    min_x = -1.0,
    max_x = 1.0,
    min_y = -1.0,
    max_y = 1.0,
    n_rays_per_image=750,
    n_pts_per_ray=128,
```

```
    min_depth=0.1,
    max_depth=volume_extent_world,
)
```

7. Далее определим лучевого маршировщика. В нем используются объемные плотности и цвета отобранных вдоль луча точек, и он отрисовывает значение пиксела для этого луча. В качестве лучевого маршировщика мы используем `EmissionAbsorbRaymarcher`. В нем реализован классический алгоритм испускательно-поглотительной лучевой маршировки[1]:

```
raymarcher = EmissionAbsorptionRaymarcher()
```

8. Теперь создадим экземпляр отрисовщика `ImplicitRenderer`. Он компонует отборщика лучей и лучевого маршировщика в единую структуру данных:

```
renderer_mc = ImplicitRenderer(raysampler=raysampler_mc,
                               raymarcher=raymarcher)
```

9. Давайте посмотрим на функцию потери Хьюбера. Она определена в `utils.helper_functions.huber`. Это устойчивая альтернатива функции на базе среднеквадратичной ошибки и менее чувствительная к выбросам:

```
def huber(x, y, scaling=0.1):
    diff_sq = (x - y) ** 2
    loss = ((1 + diff_sq / (scaling**2)).clamp(1e-4).sqrt() - 1) * \
          float(scaling)
    return loss
```

10. Теперь посмотрим на вспомогательную функцию, определенную в `utils.helper_functions.sample_images_at_mc_loss`, которая используется для извлечения достоверных значений пикселов из целевых изображений. Лучевой отборщик `MonteCarloRaysampler` производит выборку лучей, проходящих через определенные координаты *x* и *y* на изображении. Это **нормализованные координаты устройства** (**NDC**)[2]. Однако нам нужно брать выборку из координат пикселов изображения. Для этого мы используем функцию `torch.nn.functional.grid_sample`. При этом используются методы интерполяции в фоновом режиме, чтобы предоставлять точные значения пикселов. Это лучше, чем простое соотнесение координат NDC с координатами пикселов и затем отбор одного пиксела, который соответствует значению координаты NDC. В системе координат NDC значения *x* и *y* имеют диапазон [–1, 1]. Например, (*x*, *y*) = (–1, –1) соответствует верхнему левому углу изображения:

```
def sample_images_at_mc_locs(target_images, sampled_rays_xy):
    ba = target_images.shape[0]
    dim = target_images.shape[-1]
```

[1] От англ. *Emission-Absorption ray marching algorithm.* – *Прим. перев.*

[2] От англ. *Normalized Device Coordinates.* – *Прим. перев.*

```
    spatial_size = sampled_rays_xy.shape[1:-1]
    images_sampled = torch.nn.functional.grid_sample(
            target_images.permute(0, 3, 1, 2),
            # обратите внимание на инверсию знака
            -sampled_rays_xy.view(ba, -1, 1, 2),
            align_corners=True
    )
    return images_sampled.permute(0, 2, 3, 1).view(ba,
                                                   *spatial_size,
                                                   dim)
```

11. Во время тренировки модели всегда полезно визуализировать результат, получаемый на выходе из модели. Среди многих других применений это поможет нам корректировать курс, если мы увидим, что данные на выходе из модели не меняются с течением времени. До сих пор мы использовали лучевой отборщик `MonteCarloRaysampler`, который очень полезен при тренировке модели, но он бесполезен, когда требуется визуализировать полные изображения, так как он отбирает лучи в случайном порядке. Для того чтобы просмотреть полное изображение, надо систематически отбирать лучи, соответствующие всем пикселам выходного кадра. Для этого мы будем использовать лучевой отборщик `NDCMultinomialRaysampler`:

```
render_size = target_images.shape[1] * 2
volume_extent_world = 3.0
raysampler_grid = NDCMultinomialRaysampler(
    image_height=render_size,
    image_width=render_size,
    n_pts_per_ray=128,
    min_depth=0.1,
    max_depth=volume_extent_world
)
```

12. Теперь создадим экземпляр неявного отрисовщика:

```
renderer_grid = ImplicitRenderer(
    raysampler=raysampler_grid,
    raymarcher=raymarcher)
```

13. Для того чтобы визуализировать промежуточные результаты тренировки, давайте определим вспомогательную функцию, которая на входе берет параметры модели и камеры и сравнивает их с целевым изображением и соответствующим ему силуэтом. Если отрисованное изображение очень большое, то может оказаться невозможным уложить все лучи одновременно в памяти графического процессора. Поэтому нужно разбить их на пакеты и выполнить несколько прямых проходов по модели, чтобы получить результат. Далее необходимо объединить результат рендеринга в одно целостное изображение. Для простоты мы здесь просто импортируем функцию, но полный исходный код находится в папке главы книги репозитория на GitHub:

```
from utils.helper_function import show_full_render
```

14. Теперь создадим экземпляр модели NeRF. Для простоты мы не приводим здесь определение модельного класса. Его определение находится в папке главы книги репозитория на GitHub. Поскольку структура модели имеет большую важность, мы обсудим ее подробнее в отдельном разделе:

```
from nerf_model import NeuralRadianceField

neural_radiance_field = NeuralRadianceField()
```

15. Теперь давайте подготовимся к тренировке модели. Для того чтобы воспроизводить тренировку в дальнейшем, необходимо задать фиксированное значение используемого в `torch` случайного начального числа. Затем нужно отправить все переменные на используемое для обработки устройство. Поскольку эта задача является вычислительно-ресурсоемкой, в идеале необходимо выполнять ее на машине с поддержкой графического процессора(ов). Ее выполнение на центральном процессоре(ах) требует очень много времени и не рекомендуется:

```
torch.manual_seed(1)

renderer_grid = renderer_grid.to(device)
renderer_mc = renderer_mc.to(device)
target_cameras = target_cameras.to(device)
target_images = target_images.to(device)
target_silhouettes = target_silhouettes.to(device)
neural_radiance_field = neural_radiance_field.to(device)
```

16. Теперь определим используемые для тренировки модели гиперпараметры. Они таковы: `lr` – это скорость обучения, `n_iter` – число итераций (или шагов) тренировки, и `batch_size` – число используемых в мини-пакете случайных камер. Размер пакета здесь выбирается в соответствии с имеющейся в распоряжении памяти графического процессора(ов). Если вы обнаружите, что вам не хватает памяти графического процессора(ов), то следует выбрать меньшее значение размера пакета:

```
lr = 1e-3
optimizer = torch.optim.Adam(
    neural_radiance_field.parameters(),
    lr=lr)
batch_size = 6
n_iter = 3000
```

17. Теперь все готово к тому, чтобы натренировать модель. Во время каждой итерации нужно в случайном порядке отбирать мини-пакет камер:

```
loss_history_color, loss_history_sil = [], []
for iteration in range(n_iter):
    if iteration == round(n_iter * 0.75):
        print('Уменьшаем LR в 10 раз...')
        optimizer = torch.optim.Adam(
            neural_radiance_field.parameters(), lr=lr * 0.1
        )

    optimizer.zero_grad()
    batch_idx = torch.randperm(len(target_cameras))[:batch_size]

    batch_cameras = FoVPerspectiveCameras(
        R = target_cameras.R[batch_idx],
        T = target_cameras.T[batch_idx],
        znear = target_cameras.znear[batch_idx],
        zfar = target_cameras.zfar[batch_idx],
        aspect_ratio = target_cameras.aspect_ratio[batch_idx],
        fov = target_cameras.fov[batch_idx],
        device = device)
```

18. Во время каждой итерации сначала нужно получить отрисованные значения пикселов и отрисованные силуэты на случайно отобранных камерах с использованием модели NeRF. Все они являются предсказанными значениями. Это шаг прямого распространения. Мы хотим сравнить эти предсказания с достоверными данными, чтобы выяснить потерю во время тренировки. Потеря представляет собой смесь двух функций потери: A, потеря Хубера на предсказанном силуэте и достоверном силуэте, и B, потеря Хубера на предсказанном цвете и достоверном цвете. После получения значения потери можно выполнить обратный проход через модель NeRF, выполнив обратное распространение потери, и пошагово пройти через оптимизатор:

```
rendered_images_silhouettes, sampled_rays = renderer_mc(
    cameras=batch_cameras,
    volumetric_function=neural_radiance_field
)

rendered_images, rendered_silhouettes = (
    rendered_images_silhouettes.split([3, 1], dim=-1)
)

silhouettes_at_rays = sample_images_at_mc_locs(
    target_silhouettes[batch_idx, ..., None],
    sampled_rays.xys
)

sil_err = huber(
```

```
        rendered_silhouettes,
        silhouettes_at_rays,
    ).abs().mean()
    colors_at_rays = sample_images_at_mc_locs(
        target_images[batch_idx],
        sampled_rays.xys
    )

    color_err = huber(
        rendered_images,
        colors_at_rays,
    ).abs().mean()

    loss = color_err + sil_err
    loss_history_color.append(float(color_err))
    loss_history_sil.append(float(sil_err))

    loss.backward()
    optimizer.step()
```

19. Давайте будем визуализировать результативность модели после каждых 100 итераций. Это поможет отслеживать ход выполнения модели и позволит ее прервать, если произойдет что-то неожиданное. В результате будут созданы изображения в той же папке, откуда выполняется исходный код:

```
# Визуализировать полные результаты отрисовки каждые 100 итераций.
if iteration % 100 == 0:
    show_idx = torch.randperm(len(target_cameras))[:1]
    fig = show_full_render(
        neural_radiance_field,
        FoVPerspectiveCameras(
            R = target_cameras.R[show_idx],
            T = target_cameras.T[show_idx],
            znear = target_cameras.znear[show_idx],
            zfar = target_cameras.zfar[show_idx],
            aspect_ratio = target_cameras.aspect_ratio[show_idx],
            fov = target_cameras.fov[show_idx],
            device = device,
        ),
        target_images[show_idx][0],
        target_silhouettes[show_idx][0],
        renderer_grid,
        loss_history_color,
        loss_history_sil,
    )
    fig.savefig(f'intermediate_{iteration}')
```

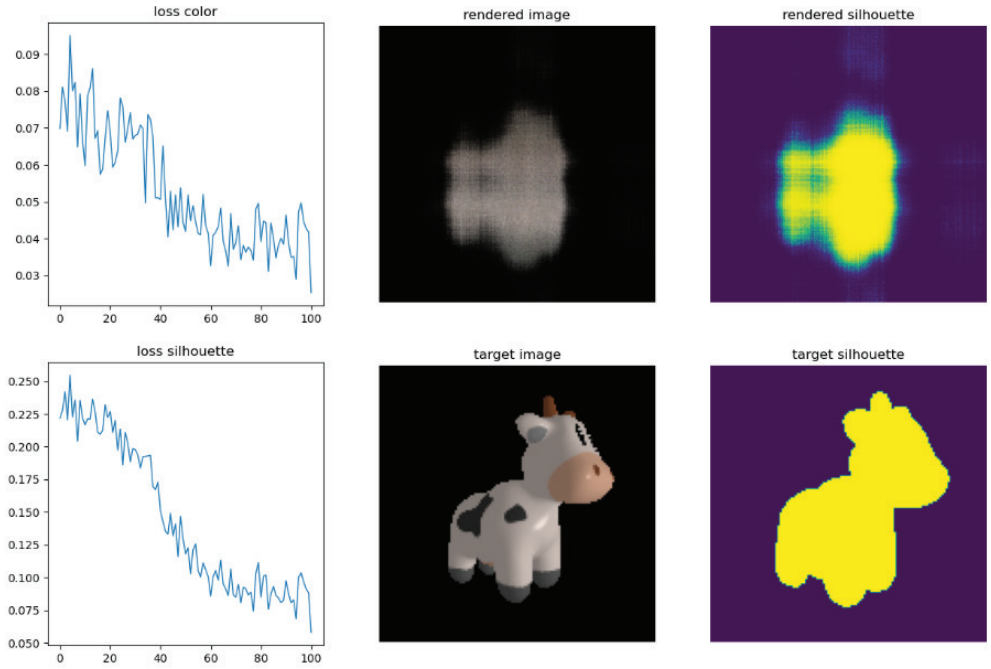


Рис. 6.3 ❖ Промежуточная визуализация с целью отслеживания тренировки модели

20. После завершения оптимизации берем итоговую результирующую объемометрическую модель и отрисовываем изображения под новыми углами обзора:

```
from utils import generate_rotating_nerf

with torch.no_grad():
    rotating_nerf_frames = generate_rotating_nerf(
        neural_radiance_field, n_frames=3*5)

image_grid(rotating_nerf_frames.clamp(0., 1.).cpu().numpy(),
           rows=3, cols=5, rgb=True, fill=True)
plt.show()
```

Наконец, новые отрисованные изображения показаны на рис. 6.4.

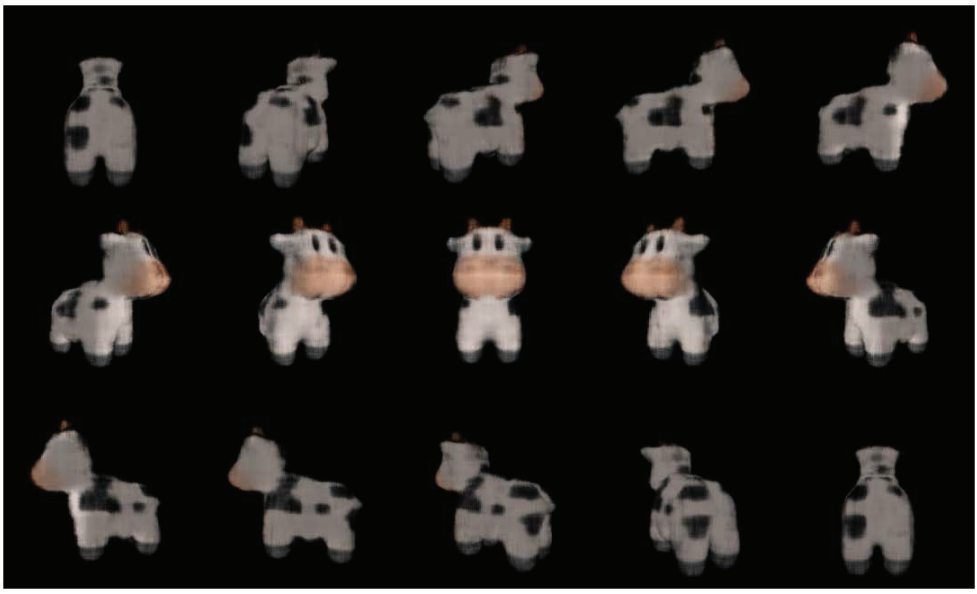


Рис. 6.4 ❖ Отрисованные изображения сцены с синтетической коровой, которую усвоила модель NeRF

В этом разделе мы натренировали модель NeRF на сцене с синтетической коровой. В следующем разделе вы познакомитесь с деталями реализации модели NeRF, подробнее пройдясь по ее исходному коду.

Понимание архитектуры модели NeRF

До сих пор мы использовали класс модели NeRF, не зная полностью, как он выглядит. В этом разделе мы сначала визуализируем внешний вид нейронной сети, а затем подробно рассмотрим исходный код и разберемся в том, как он реализован.

Нейронная сеть принимает на входе гармоническое вложение пространственного местоположения (x, y, z) и гармоническое вложение угла обзора (θ, $\emptyset$) и на выходе выводит предсказанную плотность σ и предсказанный цвет (r, g, b). На рис. 6.5 показана сетевая архитектура, которую мы собираемся реализовать в этом разделе.

Примечание

Модельная архитектура, которую мы собираемся реализовать, отличается от изначальной архитектуры модели NeRF. В этой реализации мы применяем упрощенную ее версию. Такую упрощенную архитектуру быстрее и легче тренировать.

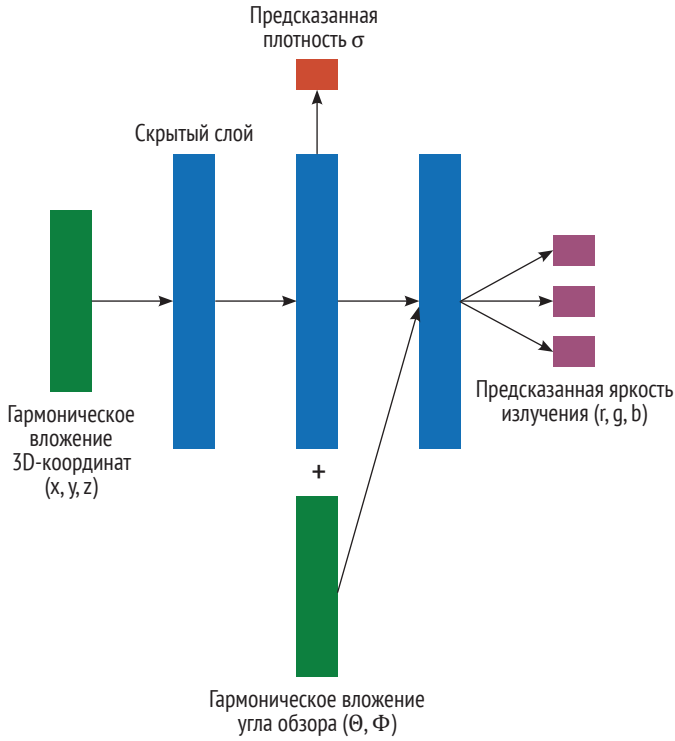


Рис. 6.5 ❖ Упрощенная архитектура модели NeRF

Давайте приступим к определению класса `NeuralRadianceField`, останавливаясь подробнее на разных частях определения этого класса. Полное определение класса находится в исходном коде файла Python `nerf_model.py` папке главы книги репозитория на GitHub.

1. Каждая входная точка представляет собой 5-мерный вектор. Было обнаружено, что тренировка модели непосредственно на этих входных данных плохо работает при представлении высокочастотной вариации цвета и геометрии. Это связано с тем, что нейронные сети известны своей склонностью к усвоению низкочастотных функций. Хорошим решением этой проблемы является отображение входного пространства в пространство более высокой размерности и его использование для тренировки. Эта функция отображения представляет собой набор синусоидных функций с фиксированными, но уникальными частотами:

   $\gamma(p) = (\sin(2^0\pi p), \cos(2^0\pi p), \ldots, \sin(2^{L-1}\pi p), \cos(2^{l-1}\pi p))$.

2. Эта функция применяется к каждой компоненте входного вектора:

```
class NeuralRadianceField(torch.nn.Module):
    def __init__(self,
                 n_harmonic_functions=60,
                 n_hidden_neurons=256):
```

```
        super().__init__()
        # Слой гармонического вложения конвертирует
        # входные 3D-координаты в представление, более
        # подходящее для обработки глубокой нейронной сетью.
        self.harmonic_embedding = HarmonicEmbedding(n_harmonic_functions)
```

3. Нейронная сеть состоит из несущего каркаса в виде MLP. На входе указанный каркас принимает вложения местоположений (x, y, z). Это полносвязная сеть, и в ней используется активационная функция `softplus`. Функция `softplus` – это более плавная версия активационной функции ReLU. На выходе каркас выдает вектор размером `n_hidden_neurons`:

```
        # Размерность гармонического вложения.
        embedding_dim = n_harmonic_functions * 2 * 3

        # self.mlp - это простой двухслойный персептрон,
        # который конвертирует входные поточечные гармонические
        # вложения в латентное представление.
        # Обратите внимание, что вместо активаций ReLU
        # используются активации Softplus.
        self.mlp = torch.nn.Sequential(
            torch.nn.Linear(embedding_dim, n_hidden_neurons),
            torch.nn.Softplus(beta=10.0),
            torch.nn.Linear(n_hidden_neurons, n_hidden_neurons),
            torch.nn.Softplus(beta=10.0),
        )
```

4. Определяем слой цвета. Данный слой принимает полученные на выходе из MLP-каркаса вложения вместе с входными вложениями направления луча и выводит RGB-цвет входных данных. Мы объединяем эти входные данные, потому что цветовые выходные данные сильно зависят от местоположения точки и от направления обзора, и поэтому, для того чтобы извлечь пользу из этой нейронной сети, важно обеспечить более короткие пути:

```
        # С учетом предсказанных персептроном self.mlp
        # признаков self.color_layer отвечает за
        # предсказание трехмерного поточечного вектора,
        # представляющего RGB-цвет точки.
        self.color_layer = torch.nn.Sequential(
            torch.nn.Linear(n_hidden_neurons + embedding_dim,
                            n_hidden_neurons),
            torch.nn.Softplus(beta=10.0),
            torch.nn.Linear(n_hidden_neurons, 3),
            torch.nn.Sigmoid(),
            # Для того чтобы цвета правильно варьировались
            # в диапазоне [0-1], слой завершается сигмоидным слоем.
        )
```

5. Далее определяем слой плотности `density`. Плотность точки зависит только от ее местоположения:

```
        # Слой плотности конвертирует признаки персептрона
        # self.mlp в 1D-значение плотности, представляющее
        # предварительную непрозрачность каждой точки.
        self.density_layer = torch.nn.Sequential(
            torch.nn.Linear(n_hidden_neurons, 1),
            torch.nn.Softplus(beta=10.0),
            # Активация Sofplus обеспечивает, чтобы сырая
            # непрозрачность была неотрицательным числом.
        )
```

6. Теперь требуется некая функция, которая будет получать данные на выходе из слоя плотности `density_layer` и предсказывать предварительную плотность:

```
    def _get_densities(self, features):
        "»»
        Эта функция принимает предсказанные персептроном
        self.mlp признаки features и конвертирует их в
        предварительные плотности raw_densities с помощью
        слоя self.density_layer.
        В дальнейшем сырые плотности отображаются в
        диапазон [0-1] посредством 1 - взаимообратная
        экспонента сырых плотностей raw_densities.
        """
        raw_densities = self.density_layer(features)
        return 1 - (-raw_densities).exp()
```

7. То же самое проделываем, чтобы получить значения цвета в определенной точке с учетом направления луча. Сначала к входным данным направления луча необходимо применить функцию позиционного кодирования. Затем нужно соединить их с данными, полученными на выходе из MLP-каркаса:

```
    def _get_colors(self, features, rays_directions):
        "»»
        Эта функция принимает предсказанные персептроном
        self.mlp поточечные признаки features и оценивает
        цветовую модель для того, чтобы прикрепить к каждой
        точке 3D-вектор ее RGB-цвета.

        Для того чтобы представить зависящие от точки обзора
        эффекты, перед вычислением слоя self.color_layer,
        нейронное поле NeuralRadianceField конкатенирует с
        признаками features гармоническое вложение параметра
        ray_directions, который представляет собой поточечные
        направления точечных лучей, выраженные как
        l2-нормализованные 3D-векторы в мировых координатах.
        """
        spatial_size = features.shape[:-1]

        # Нормализовать параметр ray_directions
        # в единичную l2-норму.
```

```
        rays_directions_normed = torch.nn.functional.normalize(
           rays_directions, dim=-1
        )

        # Получить гармоническое вложение
        # нормализованных направлений лучей.
        rays_embedding = self.harmonic_embedding(
            rays_directions_normed
        )

        # Расширить тензор направлений лучей
        # таким образом, чтобы его пространственный
        # размер был равен размеру признаков.
        rays_embedding_expand = rays_embedding[..., None, :].expand(
            *spatial_size, rays_embedding.shape[-1]
        )

        # Конкатенировать вложения направлений лучей
        # с признаками и оценить цветовую модель.
        color_layer_input = torch.cat(
            (features, rays_embedding_expand),
            dim=-1
        )
        return self.color_layer(color_layer_input)
```

8. Определяем функцию прямого распространения. Сначала получаем вложения. Затем пропускаем их через MLP-каркас, чтобы получить набор признаков. Затем используем его для получения плотностей. Мы используем признаки и направления лучей, чтобы получить цвет. Возвращаем плотности и цвета:

```
    def forward(
            self,
            ray_bundle: RayBundle,
            **kwargs,
            ):
        """
        Функция forward принимает параметризации трехмерных
        точек, отобранных вдоль лучей проекции. Прямое
        прохождение отвечает за прикрепление 3D-вектора и
        1D-мерного скаляра, представляющих соответственно
        RGB-цвет и непрозрачность точки.
        """
        # Сначала мы конвертируем параметризации лучей в мировые
        # координаты с помощью функции ray_bundle_to_ray_points.
        rays_points_world = ray_bundle_to_ray_points(ray_bundle)
        # очертание = [minibatch x ... x 3]

        # По каждой мировой 3D-координате мы получаем
        # ее гармоническое вложение.
        embeds = self.harmonic_embedding(
            rays_points_world
        )
```

```
        # очертание = [minibatch x ... x self.n_harmonic_functions*6]

        # Персептрон self.mlp отображает каждое гармоническое
        # вложение в латентное пространство признаков.
        features = self.mlp(embeds)
        # очертание = [minibatch x ... x n_hidden_neurons]

        # Наконец, получив поточечные признаки,
        # исполнить ветви плотности и цвета.

        rays_densities = self._get_densities(features)
        # очертание = [minibatch x ... x 1]

        rays_colors = self._get_colors(features,
                                       ray_bundle.directions)
        # очертание = [minibatch x ... x 3]

        return rays_densities, rays_colors
```

9. Эта функция используется для обеспечения эффективной по потреблению памяти обработки входных лучей. Входные лучи сначала разбиваются на куски `n_batches` и пропускаются через функцию `self.forward` по одному в цикле `for`. В сочетании с отключением применяемого в PyTorch кеширования градиента (`torch.no_grad()`) это позволяет за один прямой проход отрисовывать крупные пакеты лучей, которые не все помещаются в память графического процессора. В нашем случае `batched_forward` используется для экспорта полноразмерного результата отрисовки поля яркости излучения для целей визуализации:

```
    def batched_forward(
            self,
            ray_bundle: RayBundle,
            n_batches: int = 16,
            **kwargs,
            ):
        "»»
        Эта функция служит для эффективной по
        потреблению памяти обработки входных лучей.
        "»»

        # Выделить очертания, необходимые для изменения
        # очертаний тензоров в этой функции.
        n_pts_per_ray = ray_bundle.lengths.shape[-1]
        spatial_size = [*ray_bundle.origins.shape[:-1], n_pts_per_ray]

        # Разбить лучи на n_batches пакетов.
        tot_samples = ray_bundle.origins.shape[:-1].numel()
        batches = torch.chunk(torch.arange(tot_samples), n_batches)
```

10. По каждому пакету нужно сначала выполнить прямое прохождение, а затем отдельно извлечь `ray_densities` и `ray_colors`, которые будут возвращены на выходе:

```
        # По каждому пакету исполнить
        # стандартный прямой проход.
        batch_outputs = [
            self.forward(
                RayBundle(
                    origins=ray_bundle.origins.view(
                        -1, 3)[batch_idx],
                    directions=ray_bundle.directions.view(
                        -1, 3)[batch_idx],
                    lengths=ray_bundle.lengths.view(
                        -1, n_pts_per_ray)[batch_idx],
                    xys=None,
                )
            ) for batch_idx in batches
        ]

        # Конкатенировать попакетные плотности лучей
        # rays_densities и цвета лучей rays_colors и
        # изменить очертание в соответствии с размерами
        # входных данных.
        rays_densities, rays_colors = [
            torch.cat(
                [batch_output[output_i]
                 for batch_output
                 in batch_outputs], dim=0
            ).view(*spatial_size, -1) for output_i in (0, 1)
        ]
        return rays_densities, rays_colors
```

В этом разделе мы рассмотрели реализацию модели NeRF. Для того чтобы получить полное представление о модели NeRF, также необходимо обследовать теоретические концепции, лежащие в основе его использования при отрисовке объемов. В следующем далее разделе мы рассмотрим эту тему подробнее.

Понимание объемной отрисовки с использованием полей яркости излучения

Объемная отрисовка позволяет создавать 2D-проекцию 3D-изображения или 3D-сцены. В данном разделе вы познакомитесь с отрисовкой 3D-сцены с разных точек обзора. Для целей этого раздела будем исходить из допущения, что модель NeRF натренирована полностью и точно отображает входные координаты (x, y, z, d_x, d_y, d_z) в выходные (r, g, b, σ). Ниже приведены определения этих входных и выходных координат:

- $\boldsymbol{(x, y, z)}$: точка в 3D-сцене в мировых координатах,
- $\boldsymbol{(d_x, d_y, d_z)}$: единичный вектор, представляющий направление, вдоль которого мы обозреваем точку (x, y, z),

- **(*r*, *g*, *b*)**: значение яркости излучения (или испускаемый цвет) точки (x, y, z),
- **σ**: объемная плотность в точке (x, y, z).

В предыдущей главе вы разобрались в понятиях, лежащих в основе объемометрической отрисовки. Вы использовали метод отбора лучей, чтобы получать из объема объемные плотности и цвета. Мы назвали этот процесс отбором объемов. В этой главе мы будем использовать отбор лучей в поле яркости излучения, чтобы получить объемные плотности и цвета. Затем мы сможем выполнить лучевую маршировку, чтобы получить интенсивности цвета этой точки. Использованный в предыдущей главе метод лучевой маршировки и тот, который используется в этой главе, концептуально схожи. Разница состоит в том, что 3D-вокселы – это дискретные представления 3D-пространства, тогда как поля яркости излучения – его непрерывное представление (поскольку для кодирования этого представления мы используем нейронную сеть). Это немного меняет способ накопления интенсивности цвета вдоль луча.

Проецирование лучей на сцену

Представьте, что вы помещаете камеру в точку обзора и направляете ее на интересующую вас 3D-сцену. Это сцена, на которой тренируется модель NeRF. Для того чтобы синтезировать 2D-проекцию сцены, мы сначала отправляем луч на 3D-сцену, исходящий из точки обзора.

Луч может быть параметризован следующим образом:

$$\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}.$$

Здесь r – это луч, выходящий из начальной точки o и идущий в направлении d. Он параметризуется параметром t, который можно варьировать, чтобы перемещать в разные точки на луче. Обратите внимание, что r является 3D-вектором, представляющим точку в пространстве.

Накопление цвета луча

Для отрисовки цвета луча можно использовать хорошо известные классические методы цветопередачи. Перед тем как это сделать, давайте познакомимся с несколькими стандартными определениями.

- Допустим, что мы хотим накопить цвет луча между tn (ближней границей) и tf (дальней границей). Нас не интересует луч вне этих границ.
- Объемную плотность $\sigma(r(t))$ можно трактовать как вероятность того, что луч заканчивается в бесконечно малой точке вблизи $r(t)$.
- Выражение $c(\boldsymbol{r}(t), d)$ можно трактовать как цвет в точке $r(t)$ на луче при обзоре в направлении d.
- $\int_{tn}^{t} \sigma(r(s))ds$ будет измерять накопленную объемную плотность между tn и некоторой точкой t.

- $T(t) = \exp\left(-\int_{tn}^{t} \sigma(r(s))ds\right)$ будет давать представление о накопленной пропускаемости вдоль луча из tn в некоторую точку t. Чем выше накопленная объемная плотность, тем ниже накопленная пропускаемость до точки t.
- Ожидаемый цвет луча теперь можно определить следующим образом:

$$C(\boldsymbol{r}) = \int_{tn}^{tf} T(t) * \sigma(\boldsymbol{r}(s)) * c(\boldsymbol{r}(t), d) * dt.$$

Важное примечание

Объемная плотность $\sigma(r(t))$ является функцией от точки $r(t)$. В частности, она не зависит от вектора направления d. Это связано с тем, что объемная плотность является функцией физического местоположения точки. Цвет $c(\boldsymbol{r}(t), d)$ является функцией как от точки $r(t)$, так и от направления луча d. Это связано с тем, что при обзоре с разных направлений одна и та же точка в пространстве может иметь разный цвет.

Наша модель NeRF – это непрерывная функция, представляющая поле яркости излучения для сцены. Ее можно использовать для получения $c(r(t), d)$ и $\sigma(r(t))$ в разных точках луча. Существует целый ряд методов численного оценивания интеграла $C(r)$. Во время тренировки и при визуализации данных, получаемых на выходе из модели NeRF, для накопления яркости излучения вдоль луча мы использовали стандартный испускательно-поглотительный метод `EmissionAbsorbRaymarcher`.

Резюме

В этой главе вы познакомились с техникой применения нейронной сети для моделирования и представления 3D-сцены. Эта нейронная сеть называется моделью NeRF. Затем мы натренировали простую модель NeRF на синтетической 3D-сцене. Затем мы углубились в архитектуру модели NeRF и ее реализацию в исходном коде. Вы также познакомились с главными компонентами модели. Затем вы разобрались в принципах отрисовки объемов с помощью модели NeRF.

Модель NeRF используется для захвата одной сцены. Разработав эту модель, мы сможем применять ее для отрисовки этой 3D-сцены под разными углами. Вполне будет логичным задаться вопросом, есть ли способ захватывать несколько сцен с помощью одной модели и есть ли возможность предсказуемо манипулировать определенными объектами и атрибутами в сцене. Это наша тема изучения в следующей главе, где мы рассмотрим модель GIRAFFE.

Часть III

СОВРЕМЕННОЕ ТРЕХМЕРНОЕ ГЛУБОКОЕ ОБУЧЕНИЕ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ PYTORCH3D

Эта часть книги будет посвящена использованию библиотеки PyTorch3D для реализации передовых 3D-моделей и алгоритмов глубокого обучения. Технологии трехмерного компьютерного зрения в последнее время быстро набирают оборот, и вы научитесь реализовывать и использовать эти передовые 3D-модели глубокого обучения самым наилучшим образом.

Данная часть содержит следующие главы:

- глава 7 «Обследование контролируемых нейронных полей признаков»;
- глава 8 «Моделирование человеческого тела в 3D»;
- глава 9 «Сквозной синтез ракурсов с помощью модели SynSin»;
- глава 10 «Модель Mesh R-CNN».

Глава 7

Обследование контролируемых нейронных полей признаков

В предыдущей главе вы научились представлять 3D-сцену с помощью нейронных полей яркости излучения (NeRF). Мы натренировали одну нейронную сеть на постановочных[1] многоракурсных изображениях 3D-сцены, чтобы усвоить ее неявное представление. Затем мы использовали модель NeRF для отрисовки 3D-сцены с других ракурсов и углов обзора. В той модели мы исходили из допущения, что объекты и фон неизменны.

Вместе с тем будет справедливым задаться вопросом, можно ли генерировать вариации 3D-сцены. Можно ли контролировать число объектов, их позы и фон сцены? Можно ли узнать о 3D-природе вещей без постановочных изображений и без понимания параметров камеры?

К концу этой главы вы узнаете, что все это действительно возможно. В частности, вы будете хорошо понимать совершенно новый метод контролируемого синтеза 3D-изображений под названием GIRAFFE. В нем объединены идеи из областей синтеза изображений и неявного усвоения 3D-представления с использованием NeRF-подобных моделей. Это станет ясно, когда мы рассмотрим следующие ниже темы:

- концепцию синтеза изображений на основе GAN-сети,
- введение в композиционный 3D-информированный синтез изображений,
- генерирование полей признаков,
- отображение полей признаков в изображения,
- обследование контролируемой генерации сцен,
- тренировку модели GIRAFFE.

---

[1] То есть с объектами, предварительно поставленными в определенной позиции или позе. – *Прим. перев.*

Технические требования

Для выполнения примеров исходного кода этой книги в идеале понадобится компьютер с графическим процессором. Выполнение фрагментов исходного кода с использованием центрального процессора(ов) невозможно и будет чрезвычайно медленным. Рекомендуемая конфигурация компьютера приводится ниже:

- устройство GPU, например серия GTX или серия RTX с не менее 8 Гб памяти,
- Python 3.7+,
- Anaconda3.

Фрагменты исходного кода к этой главе находятся по адресу https://github.com/packtpublishing/3d-deep-learning-with-python.

Концепция синтеза изображений на основе GAN-сети

Было показано, что глубокие генеративные модели создают фотореалистичные 2D-изображения при тренировке на распределении из определенной области значений. **Генеративно-состязательные сети** (**GAN**)[1] – одна из наиболее широко используемых сред для этой цели. Они могут синтезировать высококачественные фотореалистичные изображения с разрешающей способностью 1024×1024 и выше. Например, они использовались для генерирования реалистичных лиц:

Рис. 7.1 ❖ Рандомно сгенерированные лица в виде высококачественных 2D-изображений с использованием StyleGAN2

GAN-сеть можно тренировать генерировать похожие изображения из любого распределения данных. Та же модель StyleGAN2 при тренировке на на-

[1] От англ. *Generative Adversarial Network. – Прим. перев.*

боре данных *Cars* может генерировать изображения автомобилей с высокой разрешающей способностью:

Рис. 7.2 ❖ Рандомно сгенерированные автомобили в виде 2D-изображений с использованием StyleGAN2

GAN-сети основаны на теоретико-игровом сценарии, в котором генераторная нейронная сеть генерирует изображение. Однако для достижения успеха она должна обманывать дискриминатора в его классифицировании изображения как реалистичного. Это перетягивание каната между двумя нейронными сетями (т. е. генератором и дискриминатором) в конечном счете приводит к генератору, производящему фотореалистичные изображения. Генераторная сеть делает это, создавая в многомерном латентном пространстве распределение вероятностей такое, что точки этого распределения представляют собой реалистичные изображения из области значений тренировочных изображений. Для того чтобы сгенерировать новое изображение, просто нужно выбрать точку в латентном пространстве и дать генератору создать из нее изображение:

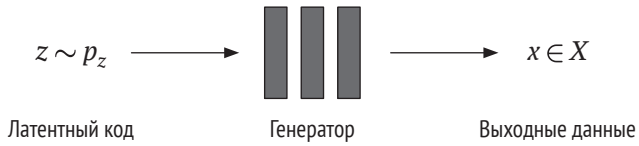


Рис. 7.3 ❖ Каноническая GAN-сеть

Синтез фотореалистичных изображений с высокой разрешающей способностью – все это здорово, но это не единственное желательное свойство генеративной модели. Если процесс генерации распутан и контролируется простым и предсказуемым образом, то открывается еще больше реально-практических приложений. Что еще важнее, требуется, чтобы атрибуты, такие как форма объекта, размер и поза, были как можно более распутанными, чтобы их можно было варьировать, не изменяя другие атрибуты изображения.

Существующие GAN-ориентированные подходы к генерированию изображений генерируют 2D-изображения без полного понимания лежащей в их основе 3D-природы изображения. Поэтому нет встроенных явных средств контроля для варьирования атрибутов, таких как позиция, форма, размер и поза объекта. Это приводит к тому, что GAN-сети имеют запутанные между

собой атрибуты. Для простоты подумайте о примере GAN-модели, которая генерирует реалистичные лица, где изменение позы головы также меняет воспринимаемый пол сгенерированного лица. Это может произойти, если атрибуты пола и позы головы запутаны между собой. Это нежелательно для большинства практических случаев использования. Нам нужно иметь возможность варьировать один атрибут, не затрагивая другие.

В следующем далее разделе мы проведем обобщенный обзор модели, которая может генерировать 2D-изображения с неявным пониманием 3D-природы базовой сцены.

Введение в композиционный 3D-информированный синтез изображений

Наша цель – контролируемый синтез изображений. Нам нужен контроль над числом объектов на изображении, их позицией, формой, размером и позой. Модель GIRAFFE – одна из первых, в которой реализованы все эти желаемые свойства, а также генерируются фотореалистичные изображения с высокой разрешающей способностью. Для того чтобы иметь контроль над этими атрибутами, модель должна иметь некую информированность о 3D-природе сцены.

Теперь давайте посмотрим, как модель GIRAFFE строится поверх других устоявшихся идей с целью достижения этой цели. Она извлекает пользу из следующих ниже высокоуровневых концепций.

- **Усвоение 3D-представления**: наличие NeRF-подобной модели для усвоения неявного 3D-представления и полей признаков. В отличие от стандартной модели NeRF эта модель выдает поле признаков вместо интенсивности цвета. Данная NeRF-подобная модель используется для обеспечения 3D-согласованности в сгенерированных изображениях.
- **Композиционный оператор**: наличие беспараметрического композиционного оператора для компоновки одного поля признаков из полей признаков нескольких объектов. Он способствует созданию изображений с нужным числом объектов в них.
- **Модель нейронной отрисовки**: она использует скомпонованное поле признаков для создания изображения. Это двумерная **сверточная нейронная сеть** (CNN)[1], которая повышает частоту отбора[2] из поля объекта, чтобы создавать выходное изображение более высокой разрешающей способности.
- **GAN-сеть**: в модели GIRAFFE используется модельная архитектура GAN-сети, чтобы генерировать новые сцены. Предыдущие три компоненты образуют генератор. Модель также состоит из нейросетевого

[1] От англ. *Convolutional Neural Network. – Прим. перев.*

[2] От англ. *upsample. – Прим. перев.*

дискриминатора, который отличает поддельные изображения от реальных. Благодаря наличию модели NeRF вместе с композиционным оператором эта модель делает процесс генерации изображений как композиционным, так и 3D-информированным.

Генерирование изображения представляет собой двухэтапный процесс.

1. Объемно отрисовать поле признаков с учетом угла обзора камеры и некоторой информации об объектах, которые требуется отрисовать. Информация об объекте представляет собой абстрактные векторы, о которых вы узнаете в следующих разделах.
2. Применить модель нейронной отрисовки, чтобы отобразить поле признаков в изображение высокой разрешающей способности.

Было обнаружено, что этот двухэтапный метод лучше подходит для генерирования изображений высокой разрешающей способности по сравнению с прямым генерированием RGB-значений из данных, получаемых на выходе из модели NeRF. Из предыдущей главы вы знаете, что модель NeRF тренируется на изображениях из одной и той же сцены. Натренированная модель может генерировать изображение только из той же самой сцены. Это было одним из больших ограничений модели NeRF.

Напротив, модель GIRAFFE тренируется на непостановочных изображениях из разных сцен. Натренированная модель может генерировать изображения из того же распределения, на котором она тренировалась. Как правило, эта модель тренируется на одних и тех же данных. То есть распределение тренировочных данных происходит из одной области значений. Например, если тренировать модель на наборе данных *Cars*, то можно ожидать, что сгенерированные этой моделью изображения будут какой-то версией автомобиля. Она не способна генерировать изображения из совершенно незнакомого ей распределения, такого как лица. Хотя это все-таки ограничение в том, что модель способна делать, она гораздо менее ограничена по сравнению со стандартной моделью NeRF.

Реализованные в модели GIRAFFE фундаментальные концепции, которые мы обсуждали до сих пор, представлены на рис. 7.4.

Генераторная модель использует выбранную позу камеры и N объектов (включая фон) и соответствующее число кодов форм и внешнего вида вместе с аффинными преобразованиями, чтобы сначала синтезировать поля признаков. Отдельные поля признаков, соответствующие отдельным объектам, затем компонуются вместе, чтобы сформировать совокупное поле признаков. Затем она объемно отрисовывает поле признаков вдоль луча, используя стандартные принципы отрисовки объемов. После этого нейронная сеть отрисовки трансформирует это поле признаков в пиксельное значение в пространстве изображения.

В этом разделе вы получили очень широкое представление о модели GIRAFFE. Теперь давайте рассмотрим отдельные ее компоненты, чтобы получить более глубокое ее понимание.

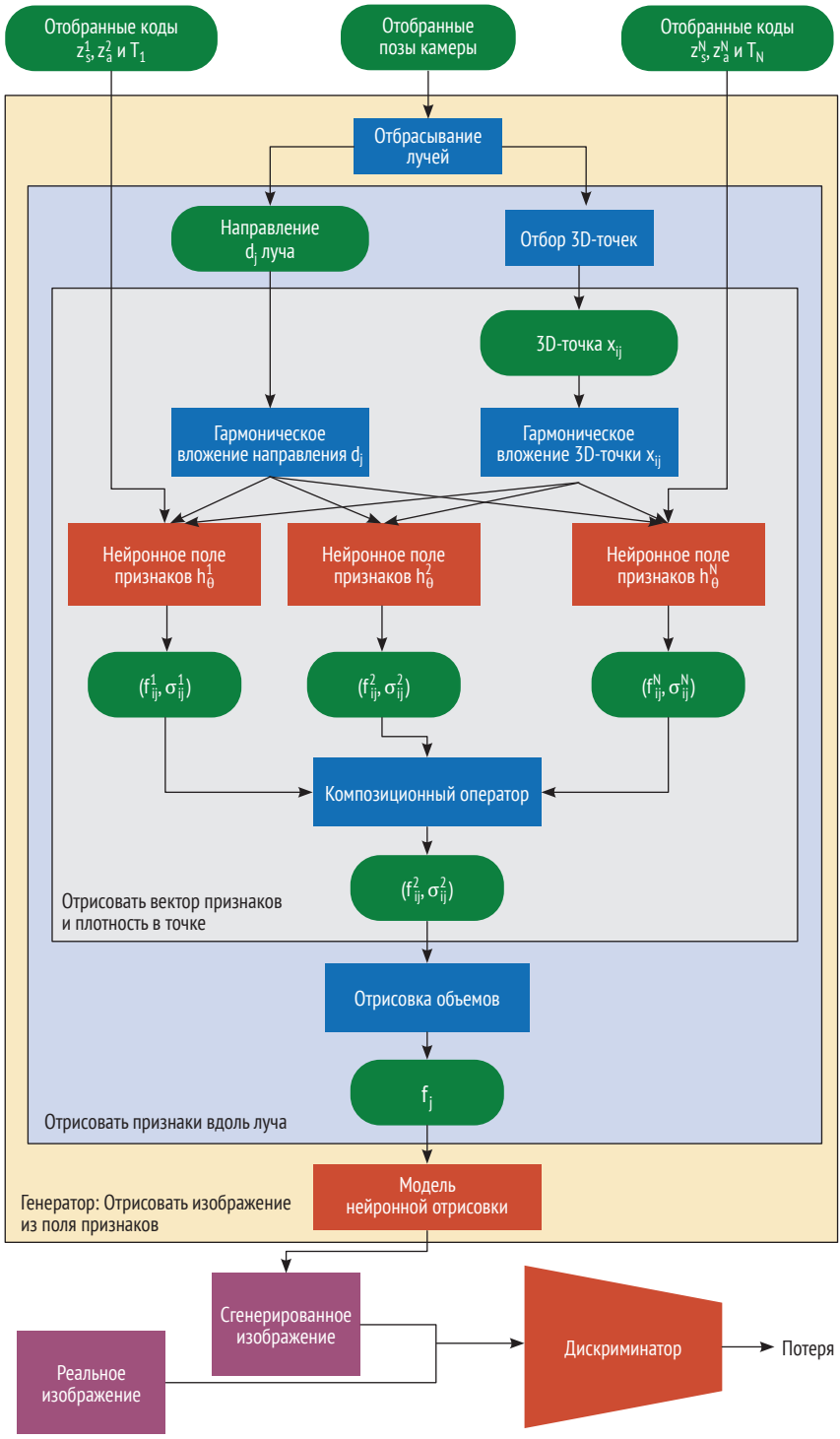


Рис. 7.4 ❖ Модель GIRAFFE

Генерирование полей признаков

Первым шагом процесса генерации сцены является генерирование поля признаков. Оно аналогично генерированию RGB-изображения в модели NeRF. В модели NeRF на выходе из модели получается поле признаков, которое оказывается изображением, состоящим из значений RGB. Однако поле признаков может быть любым абстрактным понятием изображения. Это обобщение матрицы изображения. Разница здесь в том, что вместо генерирования трехканального RGB-изображения модель GIRAFFE генерирует более абстрактное изображение, которое называется полем признаков с размерностями H_V, W_V и M_f, где H_V – это высота поля признаков, W_V – его ширина, а M_f – число каналов в поле признаков.

В этом разделе мы будем исходить из допущения, что у нас имеется натренированная модель GIRAFFE. Она была натренирована на некоем предопределенном наборе данных, о котором мы пока не будем говорить. Для того чтобы сгенерировать новое изображение, нужно сделать следующие три вещи.

1. Задать позу камеры: она определяет угол обзора камеры. В качестве этапа предобработки эта поза камеры используется для того, чтобы направить луч на сцену и сгенерировать вектор направления (d_j) вместе с отобранными точками (x_{ij}). Мы проецируем в сцену много таких лучей.
2. Отобрать 2N латентных кодов: мы отбираем два латентных кода, соответствующих каждому объекту, который мы хотим видеть на отрисованном выходном изображении. Один латентный код соответствует форме объекта, а другой латентный код – его внешнему виду. Эти коды отбираются из стандартного нормального распределения.
3. Задать N аффинных преобразований: это соответствует позе объекта в сцене.

Генераторная часть модели делает следующее:

- по каждому ожидаемому объекту в сцене нужно применить код формы, код внешнего вида, информацию о позе объекта (т. е. аффинное преобразование), вектор направления обзора и точку в сцене (x_{ij}), чтобы сгенерировать поле признаков (вектор) и объемную плотность для этой точки. Это модель NeRF в действии;
- применить композиционный оператор, чтобы скомпоновать эти поля признаков и плотности в одно поле признаков и значение плотности этой точки. Здесь композиционный оператор делает следующее:

$$\sigma = \sum_{i=1}^{N} \sigma_i,$$

$$f = \frac{1}{\sigma} \sum_{i=1}^{N} \sigma_i * f_i.$$

Объемную плотность в точке можно просто суммировать. Поле признаков усредняется путем назначения важности, пропорциональной объ-

емной плотности объекта в этой точке. Одним из важных преимуществ такого простого оператора является его дифференцируемость. Следовательно, его можно вводить внутрь нейронной сети, так как градиенты могут проходить через этот оператор на этапе тренировки модели;

- мы используем объемную отрисовку, чтобы отрисовывать поле признаков по каждому лучу, сгенерированному для входной позы камеры, путем агрегирования значений полей признаков вдоль луча. Это делается для многочисленных лучей, чтобы создать полное поле признаков размерности $H_V \times W_V$. Здесь значение V обычно является малым. Таким образом, мы создаем поле признаков низкой разрешающей способности.

Поля признаков

Поле признаков – это абстрактное понятие изображения. Они не являются RGB-значениями и в типичной ситуации имеют низкие пространственные размерности (например, 16×16 или 64×64), но большие канальные размерности. Нам нужно изображение с высокой пространственной размерностью (например, 512×512), но в трех каналах (RGB). Давайте посмотрим, как это сделать с помощью нейронной сети.

Отображение полей признаков в изображения

После того как было сгенерировано поле признаков размерностью $H_V \times W_V \times M_f$, необходимо его отобразить в изображение размерности $H \times W \times 3$. Как правило, $H_V < H$, $W_V < W$ и $M_f > 3$. В модели GIRAFFE используется двухэтапный подход, поскольку абляционный анализ показал, что он лучше, чем использование одноэтапного подхода непосредственного генерирования изображения.

Операция отображения является параметрической функцией, которую можно усваивать из данных, и применение двумерной CNN-сети подходит для этой задачи лучше всего, поскольку эта функция относится к области значений изображения. Указанную функцию можно трактовать как нейронную сеть с повышением частоты отбора, такой как декодировщик в автокодировщике. Результатом работы этой нейронной сети является отрисованное изображение, которое можно увидеть, понять и оценить. Данную сеть можно определить математически следующим образом:

$$\pi_\theta^{neural}: \mathbb{R}^{H_v \times W_v \times M_f} \rightarrow \mathbb{R}^{H \times W \times 3},$$

указанная нейронная сеть состоит из серии повышающих частоту отбора слоев, выполненных с использованием n блоков повышения частоты отбора ближайших соседей, за которыми следует свертка 3×3 и негерметичный ReLU[1]. За счет этого создается серия из n разных пространственных разреше-

[1] Негерметичный выпрямленный линейный блок (Leaky Rectified Linear Unit, ReLU) – это тип активационной функции, основанной на ReLU, но данная функция вместо плоского наклона имеет небольшой наклон с отрицательными значениями. – *Прим. перев.*

ний поля признаков. Однако в каждом пространственном разрешении поле признаков отображается в трехканальное изображение того же пространственного разрешения посредством свертки 3×3. При этом изображения из предыдущего пространственного разрешения подвергаются обработке с повышением частоты отбора с помощью непараметрического билинейного оператора повышения частоты отбора и добавляются в изображение нового пространственного разрешения. Это повторяется до тех пор, пока не будет достигнуто желаемое пространственное разрешение $H \times W$.

Пропущенные соединения, ведущие из поля признаков в изображение аналогичной размерности, способствуют сильному потоку градиента в поля признаков в каждом пространственном разрешении. В интуитивном плане за счет этого обеспечивается прочное понимание моделью нейронной отрисовки изображения в каждом пространственном разрешении. Вдобавок пропущенные соединения обеспечивают, чтобы генерируемое окончательное изображение представляло собой комбинацию понимания изображения в различных разрешениях.

Эта концепция становится очень ясной на следующей ниже диаграмме модели нейронной отрисовки:

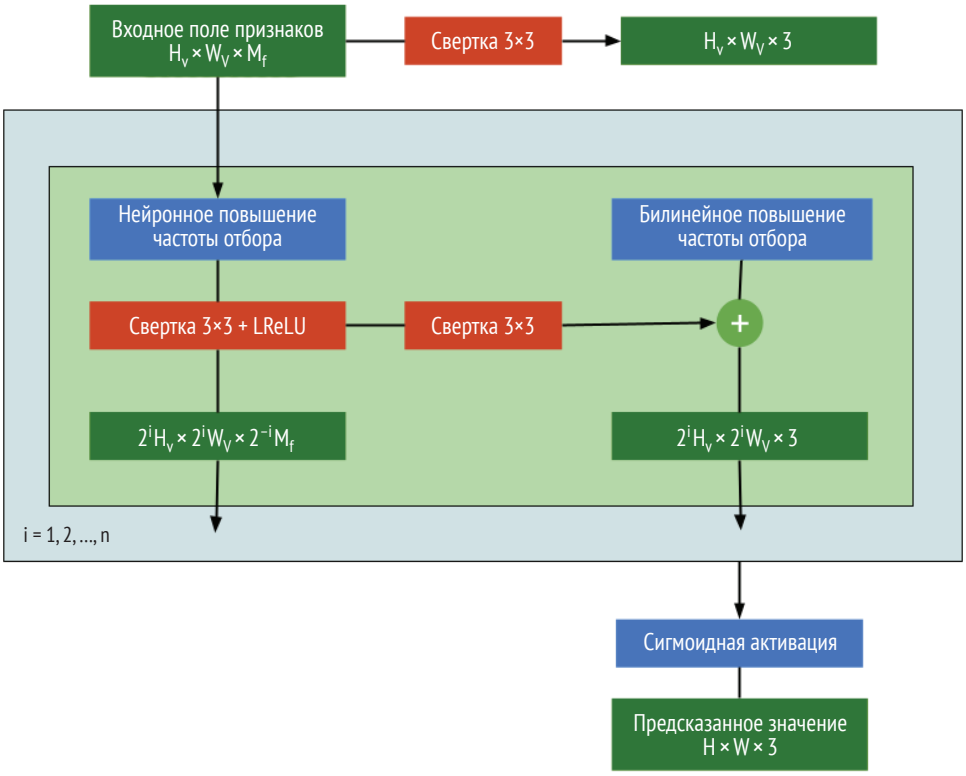


Рис. 7.5 ❖ Модель нейронной отрисовки;
это двумерная CNN-сеть с серией операторов повышения частоты отбора ближайших соседей и параллельным отображением в область значений RGB-изображения

Модель нейронной отрисовки берет данные поля признаков на выходе из предыдущего этапа и генерирует RGB-изображение высокой разрешающей способности. Поскольку поле признаков генерируется с помощью NeRF-ориентированного генератора, оно должно понимать 3D-природу сцены, объекты на ней, а также их позицию, позу, форму и внешний вид. А так как мы используем композиционный оператор, поле признаков также кодирует число объектов в сцене.

В следующем разделе вы узнаете, как контролировать процесс генерации сцены и какими механизмами контроля мы располагаем для этого.

Обследование контролируемой генерации сцен

Для того чтобы по-настоящему оценить и узнать, что именно модель компьютерного зрения генерирует, необходимо визуализировать данные, получаемые на выходе из натренированной модели. Поскольку мы имеем дело с генеративным подходом, это можно сделать легко, просто визуализируя сгенерированные моделью изображения. В этом разделе мы рассмотрим предварительно натренированные модели GIRAFFE и посмотрим, насколько хорошо они могут генерировать контролируемые сцены. Мы будем использовать предварительно натренированные контрольные точки, предоставленные создателями модели GIRAFFE. Продемонстрированные в этом разделе инструкции основаны на репозитории с открытым исходным кодом на GitHub по адресу https://github.com/autonomousvision/giraffe. С помощью следующих ниже команд создайте среду Anaconda под названием `giraffe`:

```
$ cd chap7/giraffe
$ conda env create -f environment.yml
$ conda activate giraffe
```

После активации среды `giraffe` можно начать отрисовывать изображения для различных наборов данных, используя соответствующие предварительно натренированные контрольные точки. Создатели модели GIRAFFE поделились предварительно натренированными моделями из пяти разных наборов данных.

- **Набор данных Cars**: состоит из 136 726 изображений 196 классов автомобилей.
- **Набор данных CelebA-HQ**: состоит из 30 000 изображений лиц. Все изображения имеют высокую разрешающую способность и отобраны из изначального набора данных *CelebA*.
- **Набор данных церквей LSUN**: содержит около 126 227 изображений церквей.
- **Набор данных CLEVR**: этот набор данных в основном используется для визуальных вопросно-ответных исследований. Он состоит из 54 336 изображений объектов разных размеров, форм и позиций.
- **Набор данных Flickr-Faces-HQ**: состоит из 70 000 высококачественных изображений лиц, полученных с Flickr.

Мы займемся обследованием результатов работы моделей на двух разных наборах данных, чтобы получить о них представление.

Обследование контролируемой генерации автомобилей

В этом подразделе мы обследуем модель, натренированную на наборе данных *Cars*. Передаваемые в модель коды внешнего вида и формы будут генерировать автомобили, поскольку модель натренирована именно на них. Для того чтобы сгенерировать образцы изображений, надо выполнить следующую ниже команду:

```
$ python render.py configs/256res/cars_256_pretrained.yaml
```

Здесь файл `config` задает путь к выходной папке, в которой хранятся сгенерированные изображения. Скрипт Python `render.py` автоматически загружает контрольные точки модели GIRAFFE и использует их для отрисовки изображений. Выходные изображения сохраняются в папке `out/cars256_pretrained/rendering`. Эта папка будет иметь следующие ниже подпапки:

```
- out
  - cars256_pretrained
    - rendering
      - interpolate_app
      - interpolate_shape
      - translation_object_depth
      - interpolate_bg_app
      - rotation_object
      - translation_object_horizontal
```

Каждая папка содержит изображения, полученные при изменении определенных данных, подаваемых в модель GIRAFFE. Например, взгляните на следующее ниже:

- `interpolate_app`: набор изображений для демонстрации того, что происходит, когда мы медленно варьируем код внешнего вида объекта,
- `interpolate_bg_app`: демонстрирует, что происходит, когда мы варьируем код внешнего вида фона,
- `interpolate_shape`: демонстрирует, что происходит, когда мы варьируем код формы объекта,
- `translation_object_depth`: демонстрирует, что происходит, когда мы меняем глубину объекта. Это часть кода матрицы аффинного преобразования, которая является частью входных данных,
- `translation_object_horizontal`: демонстрирует, что происходит, когда мы хотим переместить объект вбок на изображении. Это часть кода матрицы аффинного преобразования, которая является частью входных данных,

- `rotate_object`: демонстрирует, что происходит, когда мы хотим изменить позу объекта. Это часть кода матрицы аффинного преобразования, которая является частью входных данных.

Давайте посмотрим на изображения внутри папки `rotate_object` и проанализируем их:

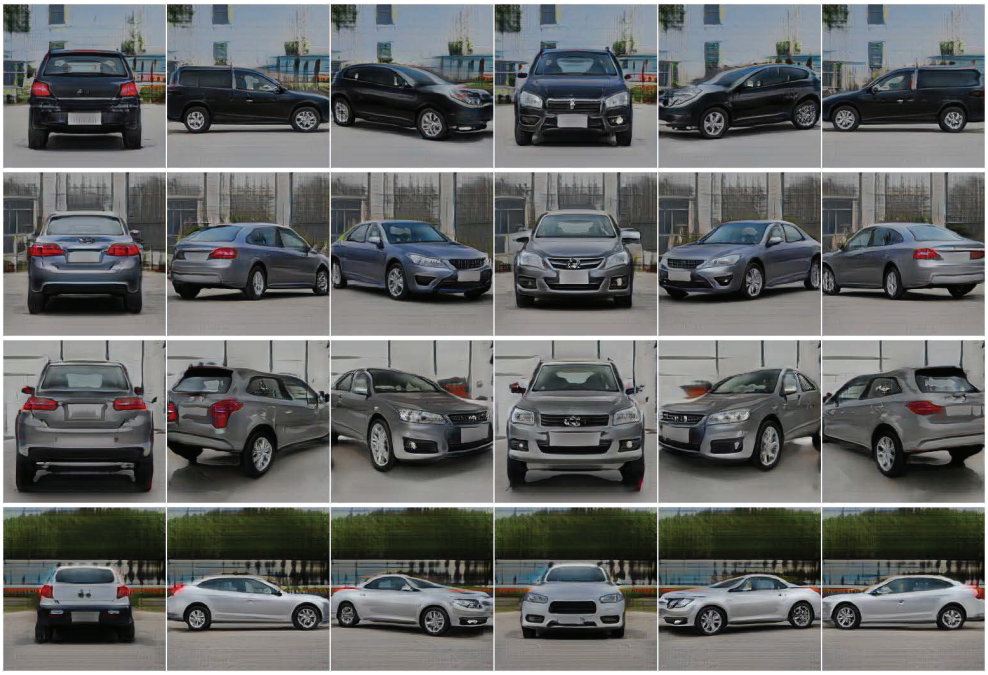


Рис. 7.6 ❖ Модельные изображения в папке rotation_object

Изображения в каждом ряду были получены сначала путем выбора кода внешнего вида и формы и варьирования матрицы аффинного преобразования, чтобы просто поворачивать объект. Горизонтальные и глубинные части кода аффинного преобразования оставались фиксированными. Код фонового объекта, код внешнего вида и формы объекта также оставались фиксированными. Разные ряды были получены с использованием разных кодов внешнего вида и формы. Ниже приводится несколько наблюдений.

- Фон всех изображений не меняется на изображениях одного и того же объекта. Это говорит о том, что мы успешно распутали фон для остальных частей изображения.
- Цвет, отражение и тени: по мере поворота объекта цвет изображения и отражение оставались достаточно стабильными, как и ожидается при повороте физического объекта. Это типичная ситуация вследствие применения NeRF-подобной модельной архитектуры.
- Левосторонняя-правосторонняя согласованность: левое и правое изображения автомобиля согласованы.

- Имеется несколько неестественных артефактов, таких как размытые края объектов и нечеткий фон. Высокочастотные вариации изображения не очень хорошо улавливаются моделью GIRAFFE.

Теперь вы можете обследовать другие папки, чтобы понять согласованность модели и качество сгенерированного изображения в ситуациях, когда транслировался объект или изменялся фон.

Обследование контролируемой генерации лиц

В этом подразделе мы обследуем модель, натренированную на наборе данных *CelebA-HQ*. Передаваемые в модель коды внешности (внешнего вида) и форм будут генерировать лица, поскольку модель натренирована именно на них. Для того чтобы сгенерировать образцы изображений, надо выполнить следующую ниже команду:

```
$ python render.py configs/256res/celebahq_256_pretrained.yaml
```

Файл `config` задает путь к выходной папке, в которой хранятся сгенерированные изображения. Выходные изображения сохраняются в папке `out/celebahq_256_pretrained/rendering`. Эта папка будет иметь следующие ниже подпапки:

```
- out
  - celebahq_256_pretrained
    - rendering
      - interpolate_app
      - interpolate_shape
      - rotation_object
```

Давайте посмотрим на изображения внутри папки `interpolate_app` и проанализируем их (рис. 7.7).

Изображения в каждом ряду были получены путем выбора кода формы и варьирования кода внешности, чтобы просто изменить внешность лица. Код матрицы аффинного преобразования также оставался фиксированным. Разные ряды были получены с использованием разных кодов формы. Ниже приведено несколько наблюдений.

- Форма сгенерированного лица в значительной степени фиксировано для одного ряда лиц. Это говорит о том, что код формы устойчив к изменениям в коде внешности.
- Внешность лица (такие признаки, как оттенок кожи, блеск кожи, цвет волос, цвет бровей, цвет глаз, выражение губ и форма носа) изменяется при изменении кода внешности. Это говорит о том, что в коде внешности кодируются признаки (черты) внешности лица.
- В коде формы кодируется воспринимаемый пол лица. Это в значительной степени имеет смысл, поскольку в наборе тренировочных данных существует большая воспринимаемая разница между формой лица мужских и женских изображений.

Давайте посмотрим на изображения внутри папки `interpolate_shape` и проанализируем их (рис. 7.8).

Рис. 7.7 ❖ Изображения в папке interpolate_app

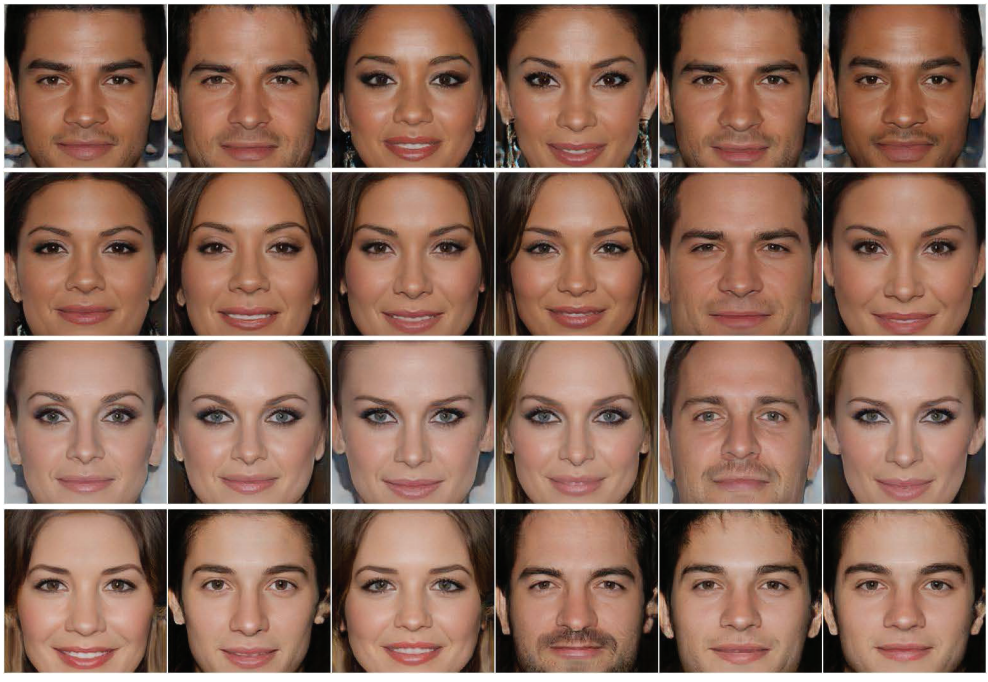


Рис. 7.8 ❖ Изображения в папке interpolate_shape

Изображения в каждом ряду были получены путем выбора кода внешности и варьирования кода формы, чтобы просто изменить форму лица. Код матрицы аффинного преобразования также оставался фиксированным. Разные ряды были получены с использованием разных кодов внешности. Ниже приведено несколько наблюдений.

- Внешность лица (такие признаки, как оттенок кожи, блеск кожи, цвет волос, цвет бровей, цвет глаз, выражение губ и форма носа) в основном остается таким же, как и код формы. Это говорит о том, что код внешности устойчив к изменениям признаков (черт) лица.
- Форма сгенерированного лица изменяется при варьировании кода формы. Это говорит о том, что в коде формы правильно кодируются признаки формы лица.
- В коде формы кодируется воспринимаемый пол лица. Это в значительной степени имеет смысл, поскольку в наборе тренировочных данных существует большая воспринимаемая разница между формой лица мужских и женских изображений.

В этом разделе мы обследовали контролируемую генерацию 3D-сцен с использованием модели GIRAFFE. Мы сгенерировали автомобили, используя модель, натренированную на наборе данных *Cars*. Кроме того, мы сгенерировали лица, используя модель, натренированную на наборе данных *CelebA-HQ*. В каждом из этих случаев вы увидели, что входные параметры модели очень хорошо распутаны. Мы использовали предварительно натренированную модель, предоставленную создателями модели GIRAFFE. В следующем разделе вы подробнее узнаете о том, как тренировать такую модель на новом наборе данных.

Тренировка модели GIRAFFE

На данный момент в этой главе вы разобрались в том, как работает натренированная модель GIRAFFE. Вы поняли разные компоненты, составляющие генераторную часть модели.

Но для тренировки модели есть еще одна часть, которую мы пока не рассматривали, а именно дискриминатор. Как и в любой другой модели GAN, эта дискриминаторная часть модели не используется во время синтеза изображений, но является жизненно важной для тренировки модели компонентой. В этой главе мы рассмотрим ее подробнее, и вы получите представление об используемой функции потери. Мы будем тренировать новую модель с нуля, используя тренировочный модуль, предоставленный авторами GIRAFFE.

Генератор принимает на входе различный латентный код, соответствующий повороту объекта, повороту фона, возвышению камеры, трансляции по горизонтали и глубине, а также размеру объекта. Все это используется, чтобы сначала сгенерировать поле признаков, а затем отобразить его в RGB-пикселы с помощью модуля нейронной отрисовки. Это генератор. На вход дискриминатору подаются два изображения: одно – реальное изображение из тренирвочного набора данных, а другое – изображение, сгенерированное

генератором. Цель дискриминатора состоит в том, чтобы классифицировать реальное изображение как реальное, а сгенерированное изображение как поддельное. В этом цель GAN-сети.

Важное примечание

Тренировочный набор данных не помечен. Параметры позы, глубины и позиции объекта на изображении не аннотированы. Однако по каждому набору данных мы приблизительно знаем такие параметры, как скорость поворота объекта, диапазон поворота фона, диапазон возвышения камеры, горизонтальная трансляция, диапазон трансляции глубины и диапазон масштаба объекта. Во время тренировки входные данные отбираются рандомно из диапазона значений, исходя из допущения о равномерном распределении в пределах диапазона.

Дискриминатор представляет собой двумерную CNN-сеть, которая на входе принимает изображение и на выходе выводит баллы уверенности относительно реальных и поддельных изображений.

Начальное расстояние Фреше

Для того чтобы оценить качество сгенерированных изображений, мы используем **начальное расстояние Фреше** (**FID**)[1]. Это мера расстояния между признаками, извлеченными из реальных и сгенерированных изображений. Это не показатель одного изображения. Как раз наоборот, это статистика по всей совокупности изображений. Ниже приводится процедура вычисления балла FID.

1. Вначале используется модель InceptionV3 (популярный несущий каркас глубокого обучения, используемый во многих реально-практических приложениях), чтобы извлечь из изображения вектор признаков. Как правило, это последний слой модели перед слоем классифицирования. Этот вектор признаков резюмирует изображение в низкоразмерном пространстве.
2. Векторы признаков извлекаются для всей коллекции реальных и сгенерированных изображений.
3. Отдельно для набора реальных и сгенерированных изображений вычисляется среднее значение и ковариация этих векторов признаков.
4. Статистики среднего значения и ковариации используются в формуле расстояния, чтобы получить метрику расстояния.

Тренировка модели

Давайте посмотрим, как инициируется тренировки модели на наборе данных *Cars*:

```
python train.py .yaml configs/256res/celebahq_256.yaml
```

[1] От англ. *Frechet Inception Distance. – Прим. перев.*

Параметры тренировки можно понять, заглянув в конфигурационный файл `configs/256res/celebahq_256.yaml`:

- **data**: в этой секции конфигурационного файла задается путь к используемому тренировочному набору данных:

```
data:
  path: data/comprehensive_cars/images/*.jpg
  fid_file: data/comprehensive_cars/fid_files/comprehensiveCars_256.npz
  random_crop: True
  img_size: 256
```

- **model**: здесь задаются параметры моделирования:

```
model:
  background_generator_kwargs:
    rgb_out_dim: 256
  bounding_box_generator_kwargs:
    scale_range_min: [0.2, 0.16, 0.16]
    scale_range_max: [0.25, 0.2, 0.2]
    translation_range_min: [-0.22, -0.12, 0.]
    translation_range_max: [0.22, 0.12, 0.]
  generator_kwargs:
    range_v: [0.41667, 0.5]
    fov: 10
  neural_renderer_kwargs:
    input_dim: 256
    n_feat: 256
  decoder_kwargs:
    rgb_out_dim: 256
```

- **training**: здесь задаются параметры тренировки, такие как выходные данные `d`, путь к каталогу и скорость обучения среди прочего:

```
training:
  out_dir:  out/cars256
  learning_rate: 0.00025
```

Важное примечание

Тренировка модели требует больших вычислительных ресурсов. Полная тренировка модели на одном графическом процессоре, скорее всего, займет от 1 до 4 дней, в зависимости от используемого графического процессора.

Резюме

В этой главе вы обследовали контролируемый 3D-информированный синтез изображений с использованием модели GIRAFFE. Эта модель заимствует концепции из модели NeRF, GAN-сети и двумерной CNN-сети, чтобы создавать контролируемые 3D-сцены. Вначале мы освежили ваши знания о GAN-сети. Затем углубились в модель GIRAFFE с описанием процесса генерации полей

признаков и того, как эти поля признаков затем трансформируются в RGB-изображения. Далее, вы обследовали данные, получаемые на выходе из этой модели, и разобрались в ее свойствах и ограничениях. Наконец, мы кратко коснулись темы тренировки этой модели.

В следующей главе мы собираемся обследовать относительно новую технику, используемую для генерирования реалистичных человеческих тел в трех измерениях, именуемую моделью SMPL. Примечательно, что модель SMPL – одна из немногих моделей, в которых не используются глубокие нейронные сети. Вместо них для достижения своих целей в ней используются более классические статистические методы, такие как анализ главных компонент. Вы узнаете о важности правильной формулировки математических задач при разработке моделей с использованием классических технических приемов.

Глава 8

Моделирование человеческого тела в 3D

В предыдущих главах мы рассмотрели идеи моделирования 3D-сцены и объектов в ней. Большинство смоделированных нами объектов были статичными и неизменными, но многие приложения компьютерного зрения в реальной жизни сосредоточены вокруг людей в их естественной среде обитания. Мы хотим моделировать их взаимодействие с другими людьми и объектами в сцене.

Для этого есть несколько приложений. Фильтры Snapchat[1], программа FaceRig[2], виртуальная примерка[3] и технология захвата движения[4] в Голливуде – все это выигрывает от точного 3D-моделирования тела. Например, рассмотрим автоматизированную технологию оформления покупки. Здесь розничный магазин оборудован несколькими камерами съемки глубины. Они могут обнаруживать ситуации, когда человек извлекает объект, и соответствующим образом изменять корзину покупок. Для такого приложения и многих других потребуется точное моделирование человеческого тела.

Оценивание позы человека является краеугольным камнем задачи моделирования человеческого тела. Такая модель может предсказывать местоположение суставов, таких как плечи, бедра и локти, создавая скелет человека на изображении. Затем они используются в нескольких нижестоящих приложениях, таких как распознавание действий и взаимодействие человека

---

[1] Фильтры Snapchat трансформируют ракурс и окружающий мир при создании снимков, добавляя 3D-эффекты, объекты, персонажей и трансформации. – *Прим. перев.*

[2] Программа, которая позволяет воплощать удивительных персонажей с полной свободой выражения лица и эмоций в реальном времени, используя только вебкамеру. – *Прим. перев.*

[3] Функциональность, которая позволяет потребителям примерять товары, такие как одежда, украшения и косметика, виртуально, используя свои телефоны или планшеты. – *Прим. перев.*

[4] Техника записи шаблонов движения в цифровом формате, в особенности запись движений актера с целью анимации цифрового персонажа в фильме или видеоигре. – *Прим. перев.*

с объектом. Однако моделирование человеческого тела как набора суставов имеет свои ограничения:

- человеческие суставы не видны и никогда не взаимодействуют с физическим миром. И следовательно, мы не можем полагаться на них в отношении точного моделирования взаимодействия человека с объектом;
- суставы не моделируют топологию, объем и поверхность тела. В некоторых приложениях, таких как моделирование того, как сидит одежда, суставы сами по себе бесполезны.

В связи с этим можно прийти к соглашению, что модели поз человека функциональны для некоторых приложений, но определенно нереалистичны. Как смоделировать реалистично человеческое тело? Устранит ли модель эти ограничения? Какие еще приложения можно в результате раскрыть? В данной главе мы отвечаем на все эти вопросы. В частности, мы рассмотрим следующие ниже темы:

- постановку задачи 3D-моделирования,
- концепцию техники линейно-переходного кожного покрова Linear Blend Skinning,
- концепцию модели SMPL,
- применение модели SMPL,
- оценивание 3D-позы и формы человека с помощью метода SMPLify,
- обследование метода SMPLify.

Технические требования

Требования к вычислениям исходного кода в этой главе довольно низкие. Однако рекомендуется выполнять его в среде Linux, поскольку она лучше поддерживает определенные библиотеки. И все же его можно выполнять в других средах. В разделах программирования мы подробно описываем процедуру настройки среды для успешного выполнения исходного кода. В этой главе потребуются следующие ниже технические компоненты:

- Python 3.7,
- библиотеки opendr, matplotlib, opencv и numpy.

Фрагменты исходного кода к этой главе находятся по адресу https://github.com/packtpublishing/3d-deep-learning-with-python.

Постановка задачи 3D-моделирования

Популярный афоризм в статистике гласит: «*Все модели ошибочны, но некоторые из них полезны*». Он говорит о том, что часто бывает трудно смоделировать все мельчайшие детали задачи математически. Модель всегда будет приближением к реальности, но некоторые модели более точны и, следовательно, более полезны, чем другие.

В области машинного обучения задача моделирования обычно предусматривает следующие две компоненты:

- математическую формулировку задачи,
- разработку алгоритмов для решения этой задачи с учетом ограничений и границ этой формулировки.

Хорошие алгоритмы, используемые в плохо сформулированных задачах, часто приводят к субоптимальным моделям. Однако менее мощные алгоритмы, примененные к хорошо сформулированной модели, иногда могут приводить к отличным решениям. Это понимание особенно актуально для создания моделей 3D-тела человека.

Целью этой задачи моделирования является создание реалистичных анимированных человеческих тел. Что еще важнее, моделирование должно представлять реалистичные формы тела и естественным образом его деформировать в соответствии с изменениями в позе тела и улавливать движения мягких тканей. Моделирование человеческого тела в 3D – сложная задача. Человеческое тело состоит из массы костей, органов, кожи, мышц и воды, и они сложным образом взаимодействуют друг с другом. Для того чтобы точно смоделировать человеческое тело, потребуется смоделировать поведение всех этих отдельных компонент и их взаимодействие друг с другом. Это сложная задача, и в некоторых практических приложениях такой уровень точности не нужен. В этой главе мы будем моделировать поверхность и форму человеческого тела в 3D в качестве замены моделированию всего человеческого тела. Нам не нужна точная модель; нам просто нужно, чтобы она имела реалистичный внешний вид. За счет этого решение задачи станет более достижимым.

Определение подходящего представления

Цель состоит в том, чтобы точно представить человеческое тело с помощью низкоразмерного представления. Модели суставов – это низкоразмерные представления (обычно от 17 до 25 точек в 3D-пространстве), но они не несут много информации о форме и текстуре человека. С другой стороны, еще можно рассмотреть представление в виде воксельной решетки. Воксельная решетка может моделировать 3D-форму и текстуру тела, но она чрезвычайно многомерна и, естественно, не подходит для моделирования динамики тела вследствие изменения позы.

Следовательно, требуется представление, которое может совместно представлять суставы и поверхности тела, содержащее информацию об объеме тела. Есть несколько возможных представлений поверхностей; одним таким представлением является полигональная сетка вершин. В **линейной модели покрытых кожной оболочкой людей** (**SMPL**)[1] это представление используется. После ее задания полигональная сетка вершин будет описывать 3D-форму человеческого тела.

---

[1] От англ. *Skinned Multi-Person Linear model. – Прим. перев.*

Поскольку у этой задачи богатая история, мы обнаружим, что многие художники в индустрии анимации персонажей работали над созданием хороших полигональных сеток тела. В модели SMPL такие экспертные наработки используются для разработки подходящего первоначального шаблона полигональной сетки тела. Это важный первый шаг, потому что некоторые части тела имеют высокочастотные вариации (например, лицо и руки). Описание таких высокочастотных вариаций нуждается в более плотно упакованных точках, но точные описания частей тела с более низкой частотой вариаций (например, бедра) нуждаются в менее плотных точках. Такая первоначальная созданная вручную полигональная сетка поможет снизить размерность задачи, оставляя при этом необходимую информацию для ее точного моделирования. Эта полигональная сетка в модели SMPL является гендерно-нейтральной, при этом существует возможность разрабатывать вариации для мужчин и женщин отдельно.

Рис. 8.1 ❖ Шаблонная полигональная сетка модели SMPL в позе покоя

Если переходить к деталям, то первоначальная шаблонная полигональная сетка состоит из 6890 точек в 3D-пространстве, представляя поверхность человеческого тела. После ее векторизации вектор этой шаблонной полигональной сетки имеет длину 6 890 × 3 = 20 670. Любое человеческое тело может быть получено путем искажения этого вектора шаблонной полигональной сетки, чтобы уложиться в поверхность тела.

На бумаге эта концепция кажется удивительно простой, но число конфигураций 20 670-мерного вектора чрезвычайно велико. Множество представляющих реальное человеческое тело конфигураций является чрезвычайно малым подмножеством всех возможностей. Тогда возникает задача определения метода получения правдоподобной конфигурации, представляющей реальное человеческое тело.

Прежде чем разбираться во внутреннем устройстве модели SMPL, необходимо познакомиться с моделями облачения в кожную оболочку[1]. В следующем разделе мы рассмотрим одну из самых простых техник облачения в кожную оболочку: технику **линейно-переходного кожного покрова**[2]. Это важно тем, что модель SMPL построена именно на основе указанной техники.

Концепция техники линейно-переходного кожного покрова

Получив хорошее представление данных для 3D-тела человека, теперь возникает задача смоделировать его внешний вид в разных позах. Это имеет особую важность для анимации персонажей. Идея состоит в том, что **процесс облачения в кожную оболочку**, или скиннинг, предусматривает покрытие лежащего в основе скелета кожей (поверхностью), которая передает внешний вид анимируемого объекта. Термин «скиннинг» используется главным образом в анимационной индустрии. В типичной ситуации это представление принимает форму вершин, которые затем используются для определения связанных многоугольников, чтобы формировать поверхность.

Модель линейно-переходного кожного покрова Linear Blend Skinning берет кожу в позе покоя и трансформирует ее в кожу в любой произвольной позе, используя простую линейную модель. Это настолько эффективно для отрисовки, что многие игровые двигатели поддерживают эту технику, и она также используется для отрисовки игровых персонажей в реальном времени.

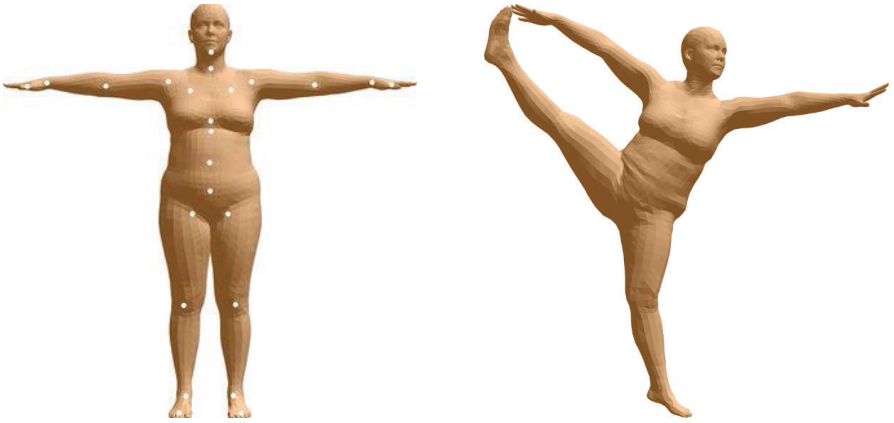


Рис. 8.2 ❖ Первоначальная переходная форма (слева) и деформированная полигональная сетка, сгенерированная с помощью техники линейного-переходного кожного покрова (справа)

[1] От англ. *skinning model. – Прим. перев.*

[2] От англ. *Linear Blend Skinning. – Прим. перев.*

Теперь давайте разберемся в устройстве данной техники. Указанная техника представляет собой модель, в которой используются следующие ниже параметры:

- шаблонная полигональная сетка T с N вершинами. В этом случае N = 6890;
- K суставов, представленных вектором J. Эти суставы соответствуют суставам человеческого тела (например, плечам, запястьям и лодыжкам). Таких суставов 23 (K = 23);
- переходные веса W. Обычно это матрица размера $N \times K$, которая улавливает взаимосвязь между N вершинами поверхности и K суставами тела;
- параметры позы, θ. Это поворотные параметры каждого сустава из числа K. Этих параметров $3K$. В нашем случае их 72, причем 69 из них соответствуют 23 суставам и 3 соответствуют совокупному повороту тела.

Функция кожного облачения принимает на входе полигональную сетку позы покоя, местоположения суставов, переходные веса и параметры позы и на выходе производит вершины:

$W(T, J, W, \vec{\theta}) \rightarrow$ вершины.

В технике линейно-переходного кожного покрова указанная функция принимает форму простой линейной функции вершин трансформированного шаблона, как описано в следующем ниже уравнении:

$$\bar{t}_i' = \sum_{k=1}^{K} w_{k,i} G_k'(\vec{\theta}, J).$$

Смысл этих членов таков:

- t_i – это вершины изначальной полигональной сетки в позе покоя,
- $G(\theta, J)$ – это матрица, трансформирующая сустав k из позы покоя в целевую позу,
- $w_{k,i}$ – это элементы переходных весов, W. Они представляют величину влияния сустава k на вершину i.

Хотя это простая в использовании линейная модель, которая очень широко применяется в индустрии анимации, она явно не сберегает объем. Поэтому трансформации могут выглядеть неестественно.

В целях решения этой проблемы художники-аниматоры адаптируют шаблонную полигональную сетку так, чтобы при применении модели кожного облачения результат выглядел естественным и реалистичным. Такие линейные деформации, применяемые к шаблонной полигональной сетке для получения реалистично выглядящей трансформированной полигональной сетки, называются переходными формами[1]. Эти разработанные художниками-аниматорами переходные формы предназначены для самых разных поз, которые может принимать анимированный персонаж, и это очень трудоемкий процесс.

---

[1] От англ. *blend shape*; син. блендшейп, переходное очертание. – *Прим. перев.*

Как мы увидим позже, модель SMPL вычисляет переходные формы автоматически перед применением к ним модели кожного облачения. В следующем далее разделе вы узнаете о том, как модель SMPL создает такие переходные формы в зависимости от позы и как данные используются для управления ими.

Концепция модели SMPL

Как следует из аббревиатуры SMPL, это усвоенная линейная модель, натренированная на данных тысяч людей. Указанная модель базируется на концепциях модели линейно-переходного кожного покрова. Это неконтролируемая и генеративная модель, которая генерирует 20 670-мерный вектор, используя предоставляемые входные параметры, которыми можно управлять. Данная модель рассчитывает переходные формы, необходимые для получения правильных деформаций при варьирующихся входных параметрах. Нам нужно, чтобы эти входные параметры имели следующие важные свойства:

- должны соответствовать реальному осязаемому атрибуту человеческого тела;
- признаки должны быть низкоразмерными по своей природе. Это позволит легко контролировать генеративный процесс;
- признаки должны быть распутаны и контролируемы предсказуемым образом. То есть варьирование одного параметра не должно изменять выходных характеристик, приписываемых другим параметрам.

Помня об этих требованиях, создатели модели SMPL придумали два наиболее важных входных атрибута: некое понятие идентичности тела и позы тела. Модель SMPL разлагает окончательную полигональную 3D-сетку тела на форму, основанную на идентичности, и форму, основанную на позе (форма, основанная на идентичности, также называется формой на основе формы, потому что телесная форма привязана к личности человека). Это дает модели желаемое свойство распутанности признаков. Есть и другие важные факторы, такие как дыхание и динамика мягких тканей (когда тело находится в движении), которые мы не объясняем в этой главе, но которые являются частью модели SMPL.

Самое главное, что модель SMPL является аддитивной моделью деформаций. То есть желаемый выходной вектор телесной формы получается путем прибавления деформаций в изначальный вектор шаблонного тела. Аддитивное свойство делает эту модель очень интуитивно понятной и легко оптимизируемой.

Определение модели SMPL

Модель SMPL строится поверх стандартных моделей кожного покрова. В них вносятся следующие изменения:

- вместо использования стандартного шаблона позы покоя используется шаблонная полигональная сетка, которая является функцией от телесной формы и поз;
- расположение суставов является функцией от телесной формы.

Заданная моделью SMPL функция принимает следующий вид:

$$M(\vec{\beta}, \vec{\theta}) = W(T_v(\vec{\beta}, \vec{\theta})J(\vec{\beta}), \vec{\theta}, W).$$

Ниже приводится значение членов в приведенном выше определении:

- β – вектор, представляющий идентичность (также именуемую формой) тела. Позже вы узнаете подробнее о том, что именно он представляет;
- θ – параметр позы, соответствующий целевой позе;
- W – переходной вес из модели линейно-переходного кожного покрова Linear Blend Skinning.

Эта функция очень похожа на модель линейно-переходного кожного покрова. В этой функции шаблонная полигональная сетка является функцией от параметров формы и позы, а местоположение суставов является функцией от параметров формы. И в этом отличие от модели линейно-переходного кожного покрова.

Форма и шаблонная полигональная сетка в зависимости от позы

Переопределенная шаблонная полигональная сетка представляет собой аддитивную линейную деформацию стандартной шаблонной полигональной сетки:

$$T_v(\vec{\beta}, \vec{\theta}) = \overline{T} + B_s(\vec{\beta}) + B_p(\vec{\theta}).$$

Здесь мы видим следующее:

- $B_s(\vec{\beta})$ – это аддитивная деформация от параметров телесной формы β. Это усвоенная деформация, моделируемая как первые несколько главных компонент смещений формы. Эти главные компоненты получены из тренировочных данных с участием разных людей в одной и той же позе покоя;
- $B_p(\vec{\theta})$ – это аддитивный член деформации позы. Он параметризуется тетой (θ). Это тоже линейная функция, полученная из набора данных о людях в разных позах.

Суставы в зависимости от формы

Поскольку местоположения суставов зависят от телесной формы, они переопределяются как функция от телесной формы. В модели SMPL это определяется следующим образом:

$$j(\vec{\beta}; J, \overline{T}, S) = J(\overline{T} + B_s(\vec{\beta}; S)).$$

Здесь $B_s(\vec{\beta})$ – это аддитивная деформация от параметров телесной формы β, а J – матрица, которая трансформирует остальные вершины шаблона в остальные позы шаблона. Параметры матрицы J тоже усваиваются из данных.

Применение модели SMPL

Теперь, когда у вас есть общее понимание модели SMPL, мы рассмотрим способы ее применения для создания 3D-моделей людей. В этом разделе мы начнем с нескольких базовых функций; это поможет обследовать модель SMPL. Мы загрузим модель SMPL и отредактируем параметры формы и позы, чтобы создать новое 3D-тело. Затем мы сохраним его как OBJ-файл и потом визуализируем.

Исходный код изначально был создан авторами модели SMPL для Python версии 2. Поэтому необходимо создать отдельную среду `python2`. Ниже приведены соответствующие инструкции:

```
cd chap8
conda create -n python2 python=2.7 anaconda
source activate python2
pip install -r requirements.txt
```

В результате будет создана и активирована среда conda Python 2.7 и установлены необходимые модули. Python 2.7 устарел, поэтому при попытке его использования вы, возможно, увидите предупреждающие сообщения. Для того чтобы создать трехмерное человеческое тело со случайными параметрами формы и позы, выполните следующую ниже команду.

```
$ python render_smpl.py
```

Появится окно, показывающее визуализацию человеческого тела в 3D. Ваш результат, скорее всего, будет другим, так как файл Python `render_smpl.py` создает человеческое тело со случайными параметрами позы и формы (рис. 8.3).

Теперь, когда вы выполнили исходный код и получили визуальный результат, давайте посмотрим, что происходит в файле `render_smpl.py`.

1. Импортируем все необходимые модули:

```
import cv2
import numpy as np
from opendr.renderer import ColoredRenderer
from opendr.lighting import LambertianPointLight
from opendr.camera import ProjectPoints
from smpl.serialization import load_model
```

2. Загружаем модельные веса базовой модели SMPL. Здесь мы загружаем нейронную модель тела:

```
# Загрузить модель SMPL (здесь загружается нейронная модель)
m = load_model('../smplify/code/models/basicModel_neutral'+
               '_lbs_10_207_0_v1.0.0.pkl')
```

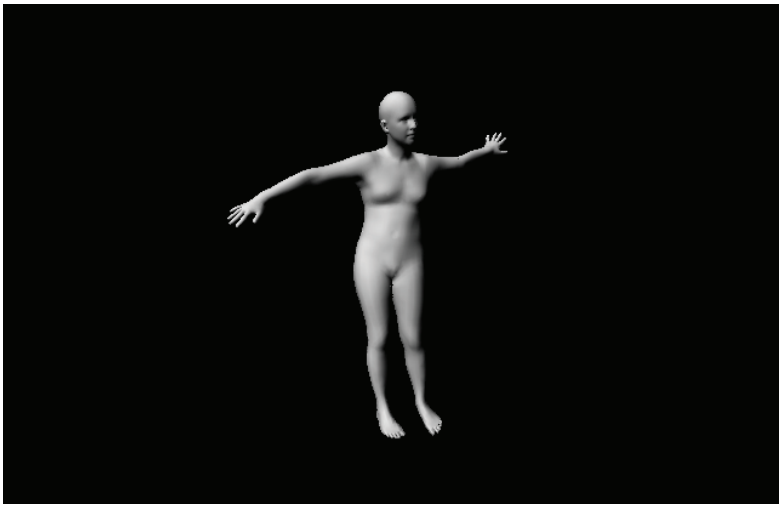


Рис. 8.3 ❖ Пример отрисовки OBJ-файла hello_smpl.obj, созданного с помощью файла Python explore_smpl.py

3. Далее задаем случайные параметры позы и формы. Следующие ниже параметры позы и формы определяют, как в конечном итоге будет выглядеть полигональная сетка 3D-тела:

```
# Задать рандомные параметры позы и формы
m.pose[:] = np.random.rand(m.pose.size) * .2
m.betas[:] = np.random.rand(m.betas.size) * .03
m.pose[0] = np.pi
```

4. Теперь создаем отрисовщик и назначаем ему атрибуты, а также конструируем источник света. По умолчанию используем отрисовщик OpenDR, но вы можете переключиться на использование отрисовщика и источника света из библиотеки PyTorch3D. Прежде чем это сделать, проверьте, чтобы были устранены все проблемы несовместимости с Python 3+.

```
# Создать отрисовщик OpenDR
rn = ColoredRenderer()

# Назначить отрисовщику атрибуты
w, h = (640, 480)

rn.camera = ProjectPoints(v=m,
                          rt=np.zeros(3),
                          t=np.array([0, 0, 2.]),
                          f=np.array([w,w])/2.,
                          c=np.array([w,h])/2.,
                          k=np.zeros(5))
rn.frustum = {'near': 1., 'far': 10.,
              'width': w, 'height': h}
rn.set(v=m, f=m.f, bgcolor=np.zeros(3))
```

```
# Сконструировать точечный источник света
rn.vc = LambertianPointLight(
    f=m.f,
    v=rn.v,
    num_verts=len(m),
    light_pos=np.array([-1000,-1000,-2000]),
    vc=np.ones_like(m)*.9,
    light_color=np.array([1., 1., 1.]))
```

5. Теперь можно визуализировать полигональную сетку и отобразить ее в окне OpenCV:

```
cv2.imshow('render_SMPL', rn.r)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Мы применили модель SMPL, чтобы сгенерировать случайное трехмерное человеческое тело. На самом деле вас, возможно, заинтересует генерирование более управляемых 3D-форм. Мы рассмотрим весь процесс в следующем далее разделе.

Оценивание позы и формы человека в 3D с помощью метода SMPLify

В предыдущем разделе вы обследовали модель SMPL и применили ее, чтобы сгенерировать трехмерное человеческое тело произвольной формы и позы. Вполне естественно задаться вопросом, можно ли использовать модель SMPL, чтобы выписывать 3D-тело человека в человека на 2D-изображении. Такая возможность имеет целый ряд практических применений, как, например, понимание действий человека или создание анимации из 2D-видео. И это действительно возможно. В данном разделе главы мы рассмотрим эту идею подробнее.

Представьте, что дано одно RGB-изображение человека без какой-либо информации о позе тела, параметрах камеры или параметрах формы. Наша цель – вычислительно вывести 3D-форму и позу только по этому единственному изображению. Оценивание 3D-формы по 2D-изображению не всегда безошибочно. Это сложная задача из-за сложности человеческого тела, артикуляции, загораживаний (окклюзии), одежды, освещения и присущей неоднозначности в вычислительном выведении 3D из 2D (поскольку при проецировании несколько 3D-поз могут иметь одну и ту же 2D-позу). Нам также нужен автоматический способ оценивания без особого ручного вмешательства. Он также должен работать над сложными позами на естественных изображениях с разнообразным фоном, условиями освещения и параметрами камеры.

Один из самых лучших подходов к решению этой задачи был изобретен исследователями из Института интеллектуальных систем Макса Планка (где

была изобретена модель SMPL), исследователями из Microsoft, из Университета Мэриленда и Университета Тюбингена. Этот подход называется SMPLify. Давайте рассмотрим этот подход подробнее.

Подход SMPLify состоит из следующих двух этапов.

1. Автоматически обнаруживать 2D-суставы с использованием устоявшихся моделей обнаружения поз, таких как OpenPose или DeepCut. Вместо них можно использовать любые 2D-детекторы суставов при условии, что они предсказывают те же суставы.
2. Использовать модель SMPL для генерирования 3D-формы. Оптимизировать параметры модели SMPL напрямую, чтобы модельные суставы модели SMPL проецировались на 2D-суставы, предсказанные на предыдущем этапе.

Мы знаем, что SMPL улавливает формы только по суставам. И поэтому с помощью модели SMPL можно получать информацию о телесной форме только по суставам. В модели SMPL параметры телесной формы характеризуются бетой (β). Это коэффициенты главных компонент в PCA-модели формы. Поза параметризуется относительным поворотом и тетой (θ) 23 суставов в кинематическом дереве. Нам нужно выполнить подгонку этих параметров, β и θ, чтобы минимизировать целевую функцию.

Определение целевой функции оптимизации

Любая целевая функция должна улавливать намерение минимизировать некоторое понятие ошибки. Чем точнее вычисление ошибки, тем точнее будут результаты шага оптимизации. Сначала мы рассмотрим всю целевую функцию целиком, затем остановимся на каждой отдельной компоненте этой функции и объясним причину, по которой каждая из них необходима:

$$E_i(\beta, \theta, K, J_{est}) + \lambda_\theta E_\theta(\theta) + \lambda_a E_a(\theta) + \lambda_{sv} E_{sv}(\theta; \beta) + \lambda_\beta E_\beta(\beta).$$

Мы хотим минимизировать эту целевую функцию, оптимизируя параметры β и θ. Она состоит из четырех членов и соответствующих коэффициентов λ_θ, λ_a, λ_{sv} и λ_β, являющихся гиперпараметрами процесса оптимизации. Ниже приводится значение каждого отдельного члена.

- $E_i(\beta, \theta, K, J_{est}) = \sum_{\text{сустав } i} w_i(\pi_K(R_\theta(J((\beta))_i)) - J_{est,i})$ – это основанный на суставах член, который штрафует расстояние между 2D-спроекцированным суставом модели SMPL и предсказанным местоположением сустава из 2D-детектора суставов (например, DeepCut или OpenPose). При загораживании сустава балл уверенности будет низким. Естественно, не следует придавать большого значения таким загороженным суставам.
- $E_a(\theta) = \sum_i \exp(\theta_i)$ – это поза, которая штрафует большие углы между суставами. Например, за счет этого обеспечивается, чтобы локти и колени не сгибались неестественным образом.
- $E_a(\theta)$ – это модель на основе гауссовой смеси, подогнанная на естественных позах, полученных из очень большого набора данных. Этот набор данных называется базой данных захвата движений CMU Gra-

phics Lab[1] и состоит почти из миллиона точек данных. Указанный управляемый данными член функции оптимизации обеспечивает, чтобы параметры позы были близки к тому, что мы наблюдаем в реальности.

- $E_{sv}(\theta; \beta)$ – это ошибка самопроникновения. Когда авторы оптимизировали целевую функцию без этого члена ошибки, они увидели неестественные самопроникновения, такие как скручивание локтей и рук и их проникновение через желудок. Это физически невозможно. Однако после добавления этого члена ошибки они получили сообразующиеся с реальностью качественные результаты. Указанный член ошибки состоит из частей тела, которые аппроксимируются набором сфер. Они определяют несовместимые сферы и штрафуют пересечение этих несовместимых сфер.
- $E_{\beta}(\beta) = \beta^T \sum_{\beta}^{-1} \beta$ – это полученная из модели SMPL форма. Обратите внимание, что матрица главных компонент является частью модели SMPL, которая была получена за счет тренировки на тренировочном наборе данных SMPL.

Подводя итог всему вышеперечисленному, целевая функция состоит из пяти компонент, которые вместе дают решение этой целевой функции в виде набора параметров позы и формы (тета и бета), которые обеспечивают минимизацию расстояний 2D-проекций суставов, одновременно обеспечивая отсутствие больших углов суставов, неестественных самопроникновений и соответствие параметров позы и формы априорному распределению, которое мы видим в большом наборе данных, состоящем из естественных телесных поз и форм.

Обследование метода SMPLify

Теперь, когда вы получили широкий обзор того, как оценивать 3D-форму человеческого тела на двумерном RGB-изображении, давайте попрактикуемся в работе с исходным кодом. В частности, мы собираемся выполнить подгонку 3D-формы тела к двум 2D-изображениям из набора данных **спортивных поз Leeds** (**LSP**)[2]. Этот набор данных содержит 2000 собранных с Flickr изображений с аннотациями, в основном спортсменов. Сначала мы пройдемся по исходному коду и создадим эти подогнанные телесные формы и только потом углубимся в исходных код. Весь используемый в этом разделе исходный код был адаптирован из реализации, опубликованной в исследовательской статье «*Будь SMPL: автоматическое оценивание 3D-позы и формы человека по одному изображению*»[3]. Мы только адаптировали его таким образом, чтобы

---

[1] От англ. *CMU Graphics Lab Motion Capture Database. – Прим. перев.*

[2] От англ. *Leeds Sports Pose. – Прим. перев.*

[3] См. «Keep it SMPL: Automatic Estimation of 3D Human Pose and Shape from Single Image». – *Прим. перев.*

помочь вам, учащимся, быстро запускать этот исходный код и самостоятельно визуализировать результаты.

Указанный исходный код изначально был создан авторами метода SMPLify для Python версии 2. Поэтому необходимо использовать ту же среду `python2`, которую мы использовали при обследовании модели SMPL. Перед тем как выполнять какой-либо исходный код, давайте бегло пробежимся по его структуре:

```
chap8
  -- smplify
    -- code
      -- fit3d_utils.py
      -- run_fit3d.py
      -- render_model.py
      -- lib
      -- models
  -- images
  -- results
```

Выполнение исходного кода

Главный файл Python, с которым вы будете напрямую взаимодействовать, называется `run_fir3d.py`. Папка с именем `images` будет содержать несколько примеров изображений из набора данных LSP. Однако, прежде чем выполнять исходный код, проверьте, чтобы путь `PYTHONPATH` был задан правильно. Он должен указывать на расположение папки `chap8`. Вы можете выполнить для нее следующую ниже команду:

```
export PYTHONPATH=$PYTHONPATH:<конкретная-папка-пользователя>/3D-Deep-
Learning-with-Python/chap8/
```

Перейдите в нужную папку:

```
cd smplify/code
```

И теперь можете запустить следующую ниже команду, чтобы выполнить подгонку 3D-тела к изображениям в папке `images`:

```
python run_fit3d.py --base_dir ../ --out_dir .
```

В этом прогоне не будет использоваться ошибка взаимопроникновения, так как в этом случае итерации оптимизации будут проходить быстрее. В конце концов мы будем выполнять подгонку телесно-нейтральной формы. Вы можете визуализировать проецируемую позу 3D-тела по мере того как она вписывается в человека на фотоснимке. После завершения оптимизации вы увидите следующие ниже два изображения:

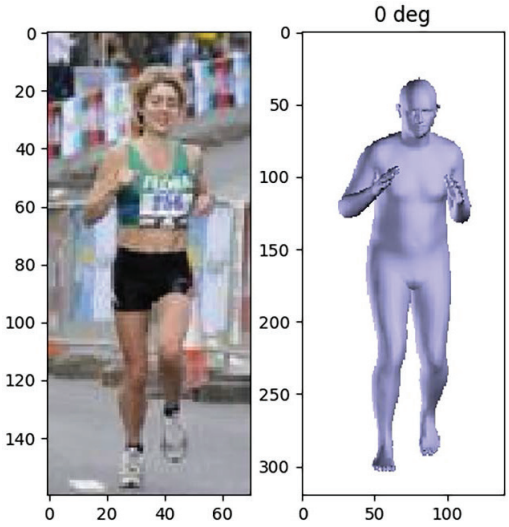


Рис. 8.4 ❖ Изображение бегущего человека из набора данных LSP (слева) и 3D-форма тела, вписанная в это изображение (справа)

Еще один результат выглядит следующим образом:

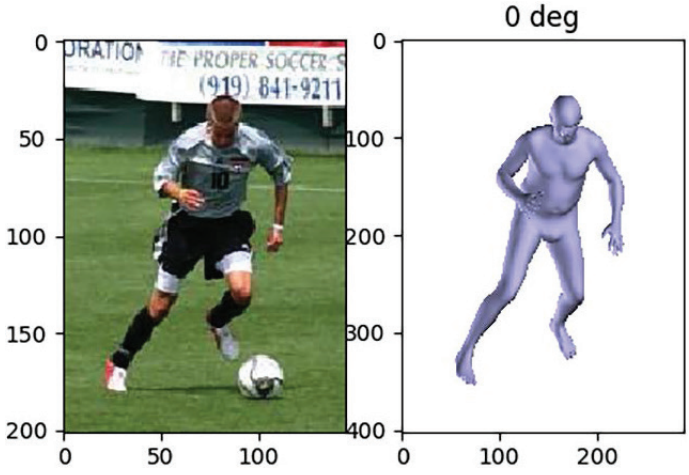


Рис. 8.5 ❖ Изображение футболиста в действии из набора данных LSP (слева) и 3D-форма тела, вписанная в это изображение (справа)

Обследование исходного кода

Теперь, когда вы выполнили исходный код подгонки к людям на 2D-изображениях, давайте рассмотрим его подробнее, чтобы понять некоторые главные его компоненты, необходимые для достижения этой цели. Все компо-

ненты находятся в файле Python `run_fit3d.py` папки `smplify/code` главы книги в репозитории на GitHub. Необходимо выполнить следующие ниже шаги.

1. Сначала импортируем все требующиеся модули:

```
from os.path import join, exists, abspath, dirname
from os import makedirs
import cPickle as pickle
from glob import glob

import cv2
import numpy as np
import matplotlib.pyplot as plt
import argparse

from smpl.serialization import load_model
from smplify.code.fit3d_utils import run_single_fit
```

2. Теперь определим место, где находится модель SMPL. Это делается посредством следующего ниже фрагмента исходного кода:

```
MODEL_DIR = join(abspath(dirname(__file__)), 'models')
MODEL_NEUTRAL_PATH = join(
    MODEL_DIR, 'basicModel_neutral_lbs_10_207_0_v1.0.0.pkl')
```

3. Давайте зададим несколько необходимых для метода оптимизации параметров и определим каталоги, в которых находятся изображения и результаты. Папка результатов будет содержать оценки суставов по всем изображениям набора данных. Параметр `viz` имеет значение `True`, чтобы активировать визуализацию. Мы используем модель SMPL с 10 параметрами (т. е. в ней используется 10 главных компонент, чтобы моделировать форму тела). Параметр `flength` относится к фокусному расстоянию камеры; во время оптимизации он остается фиксированным. Параметр `pix_thsh` относится к порогу (в пикселах). Если расстояние между плечевыми суставами в 2D меньше `pix_thsh`, то ориентация тела неоднозначна. Это может произойти, когда человек стоит перпендикулярно камере. В результате этого трудно сказать, смотрит ли он влево либо вправо. И поэтому подгонка выполняется как на расчетном теле, так и на его противоположности:

```
viz = True
n_betas = 10
flength = 5000.0
pix_thsh = 25.0
img_dir = join(abspath(base_dir), 'images/lsp')
data_dir = join(abspath(base_dir), 'results/lsp')

if not exists(out_dir):
    makedirs(out_dir)
```

4. Далее необходимо загрузить эту гендерно-нейтральную модель SMPL в память:

```
model = load_model(MODEL_NEUTRAL_PATH)
```

5. Затем нужно загрузить оценки суставов для изображений набора данных LSP. Сам набор данных LSP содержит оценки суставов и соответствующие суставные баллы уверенности по всем изображениям набора данных. Мы будем просто использовать их напрямую. Также можно предоставить свои собственные суставные оценки либо использовать неплохие оценщики, такие как из OpenPose или DeepCut, чтобы получить суставные оценки:

```
est = np.load(join(data_dir, 'est_joints.npz'))['est_joints']
```

6. Далее неоходимо загрузить изображения в набор данных и получить соответствующие суставные оценки и баллы уверенности:

```
img_paths = sorted(glob(join(img_dir, '*[0-9].jpg')))
for ind, img_path in enumerate(img_paths):
    out_path = '%s/%04d.pkl' % (out_dir, ind)
    img = cv2.imread(img_path)

    joints = est[:2, :, ind].T
    conf = est[2, :, ind]
```

7. По каждому изображению в наборе данных используем функцию `run_single_fit`, чтобы выполнить подгонку параметров бета и тета. Следующая ниже функция возвращает эти параметры после выполнения оптимизации целевой функции, аналогичной целевой функции метода SMPLify, которую мы обсуждали в предыдущем разделе:

```
params, vis = run_single_fit(img, joints, conf, model, regs=None,
                             n_betas=n_betas, flength=flength,
                             pix_thsh=pix_thsh, scale_factor=2,
                             viz=viz, do_degrees=do_degrees)
```

   Пока целевая функция оптимизируется, эта функция создает окно `matplotlib`, в котором зелеными кружками отмечены (предоставленые) оценки 2D-суставов из модели обнаружения 2D-суставов. Красными кружками помечены спроецированные суставы 3D-модели SMPL, вписываемой подгонкой в 2D-изображение (рис. 8.6).
8. Далее визуализируем подогнанное 3D-тело человека вместе с двумерным RGB-изображением. Для этого мы используем библиотеку `matplotlib`. Следующий ниже фрагмент исходного кода открывает интерактивное окно, в котором можно сохранять изображения на диск:

```
    if viz:
        plt.ion()
        plt.show()
        plt.subplot(121)
        plt.imshow(img[:, :, ::-1])
        if do_degrees is not None:
            for di, deg in enumerate(do_degrees):
                plt.subplot(122)
                plt.cla()
```

```
            plt.imshow(vis[di])
            plt.draw()
            plt.title('%d deg' % deg)
            plt.pause(1)
    raw_input('Нажмите любую клавишу, чтобы продолжить...')
```

9. Затем с помощью следующего ниже фрагмента исходного кода можно сохранить эти параметры и визуализацию на диск:

```
with open(out_path, 'w') as outf:
    pickle.dump(params, outf)

# Эти строки сохраняют только первую отрисовку.
if do_degrees is not None:
    cv2.imwrite(out_path.replace('.pkl', '.png'), vis[0])
```

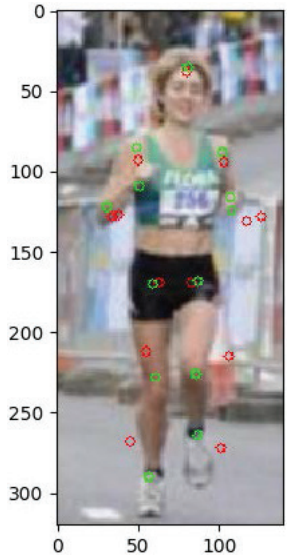


Рис. 8.6 ❖ Визуализация предоставленных 2D-суставов (зеленый) и спроецированных суставов (красный) модели SMPL, вписываемой подгонкой в 2D-изображение

В приведенном выше исходном коде самой важной функцией является функция `run_single_fit`. Вы можете обследовать эту функцию подробнее в файле Python `fit3d_utils.py` подпапки `code` папки `smplify` главы книги в репозитории на GitHub.

Здесь важно отметить, что точность подгонки 3D-тела зависит от точности 2D-суставов. Поскольку 2D-суставы предсказываются с использованием модели обнаружения суставов (такой как OpenPose или DeepCut), точность таких моделей предсказания суставов становится для этой задачи очень важной и актуальной. Оценивание 2D-суставов особенно подвержено ошибкам в следующих ниже сценариях.

- Суставы, которые не полностью видны, трудно предсказывать. Это может происходить по разным причинам, включая самозатенение, загораживание другими объектами, необычную одежду и т. д.
- Легко спутать левый и правый суставы (например, левое запястье с правым запястьем). Это особенно верно, когда человек смотрит в камеру сбоку.
- Обнаружение суставов в необычных позах затруднено, если модель не натренирована распознавать эти позы. Это зависит от разнообразия набора данных, используемого для тренировки детектора суставов.

В более широком смысле система, состоящая из нескольких моделей машинного обучения, взаимодействующих друг с другом последовательно (т. е. когда данные на выходе из одной модели подаются на вход другой модели), будет страдать от каскадных ошибок. Малые ошибки в одной компоненте будут приводить к большим ошибкам в данных на выходе из последующих компонент. Такая проблема обычно решается сквозной тренировкой системы. Однако в настоящее время эту стратегию использовать здесь не получится, поскольку в исследовательском сообществе нет достоверных данных, которые отображают входное 2D-изображение напрямую в 3D-модель.

Резюме

В этой главе мы провели общий обзор математической формулировки моделирования человеческих тел в 3D. Вы поняли мощь хорошего представления и применили простые методы, такие как линейно-переходный кожный покров Linear Blend Skinning, на мощном представлении, чтобы получить реалистичные результаты. Затем вы получили краткое обзор модели SMPL и применили ее для создания рандомного 3D-тела человека. После этого мы прошлись по используемому для его создания исходному коду. Затем мы взглянули на способы применения метода SMPLify подгонки формы 3D-тела человека к фигуре человека на двухмерном RGB-изображении. Вы узнали о том, как модель SMPL используется в этом методе в фоновом режиме. Более того, мы выполнили подгонку формы человеческого тела к двум изображениям из набора данных LSP и разобрались в исходном коде, который использовали для этой цели. Благодаря всему этому вы получили обобщенный обзор моделирования человеческого тела в 3D.

В следующей главе мы рассмотрим модель SynSin, которая обычно используется для 3D-реконструкции. Ваша цель в следующей главе будет состоять в том, чтобы понять, как реконструировать изображение с другого ракурса, имея только одно изображение.

Глава 9

Сквозной синтез ракурсов с помощью модели SynSin

Эта глава посвящена новейшей из современных моделей синтеза ракурсов под названием SynSin. Синтез ракурсов – это одно из главных направлений трехмерного глубокого обучения, которое можно использовать в различных областях, таких как дополненная реальность (AR)[1], виртуальная реальность (VR)[2], игры и др. Его цель состоит в создании модели для заданного на входе изображения и реконструкции нового изображения с другого ракурса.

В этой главе вы сначала обследуете задачу синтеза ракурсов и существующие подходы к решению этой задачи. Мы обсудим все преимущества и недостатки этих методов.

Потом вы познакомитесь с архитектурой модели SynSin подробнее. Это сквозная модель, состоящая из трех главных модулей. Мы обсудим каждый из них, и вы поймете, как эти модули помогают решать задачу синтеза ракурсов без каких-либо 3D-данных.

Разобравшись во всей структуре модели, вы перейдете к практической части, в которой мы настроим модель и поработаем с ней, чтобы вы могли получше понять весь процесс синтеза ракурсов. Мы будем тренировать и тестировать модель, а также использовать предварительно натренированные модели для формирования вычислительного вывода.

В этой главе мы рассмотрим следующие главные темы:

- краткий обзор синтеза ракурсов,
- сетевую архитектуру модели SynSin,
- тренировку и тестирование модели на практике.

---

[1] От англ. *Augmented Reality. – Прим. перев.*

[2] От англ. *Virtual Reality. – Прим. перев.*

Технические требования

Для выполнения примеров исходного кода этой книги в идеале понадобится компьютер с графическим процессором. Тем не менее для выполнения фрагментов исходного кода вполне будет достаточно только центрального процессора.

Рекомендуемая компьютерная конфигурация включает следующее:

- GPU, такой как серия GTX или серия RTX с не менее 8 Гб памяти,
- Python 3,
- библиотеки PyTorch и PyTorch3D.

Фрагменты исходного кода к этой главе находятся по адресу https://github.com/packtpublishing/3d-deep-learning-with-python.

Общий обзор синтеза ракурсов

Одним из самых популярных направлений исследований в области трехмерного компьютерного зрения является синтез ракурсов. Идея этого направления исследований заключается в генерировании нового изображения с учетом данных и точки обзора и его отрисовки с другой точки обзора.

Синтез ракурсов сопряжен с двумя трудностями. Модель должна понимать 3D-структуру и семантическую информацию изображения. Под 3D-структурой подразумевается, что при изменении точки обзора мы приближаемся к одним объектам и удаляемся от других. Хорошая модель должна с этим справляться, отрисовывая изображения, в которых при изменении ракурса некоторые объекты становятся больше, а некоторые – меньше. Под семантической информацией подразумевается, что модель должна различать объекты и понимать, какие объекты представлены на изображении. Это важно в связи с тем, что некоторые объекты могут быть включены в изображение частично; поэтому во время реконструкции модель должна понимать семантику объекта, чтобы знать, как реконструировать продолжение этого объекта. Например, имея изображение автомобиля с одной стороны, на котором мы видим только два колеса, мы знаем, что с другой стороны автомобиля есть еще два колеса. Модель должна содержать эту семантику во время реконструкции (рис. 9.1).

В связи с этим необходимо решить целый ряд проблем. Моделям трудно понимать 3D-сцену по изображению. Существует несколько методов синтеза ракурсов:

- **синтез ракурсов по нескольким изображениям**: глубокие нейронные сети можно использовать для усвоения глубины нескольких изображений, а затем реконструировать новые изображения с другого ракурса. Однако, как упоминалось ранее, из этого следует, что у нас есть несколько изображений со слегка отличающимися ракурсами, и такие данные иногда трудно получить;

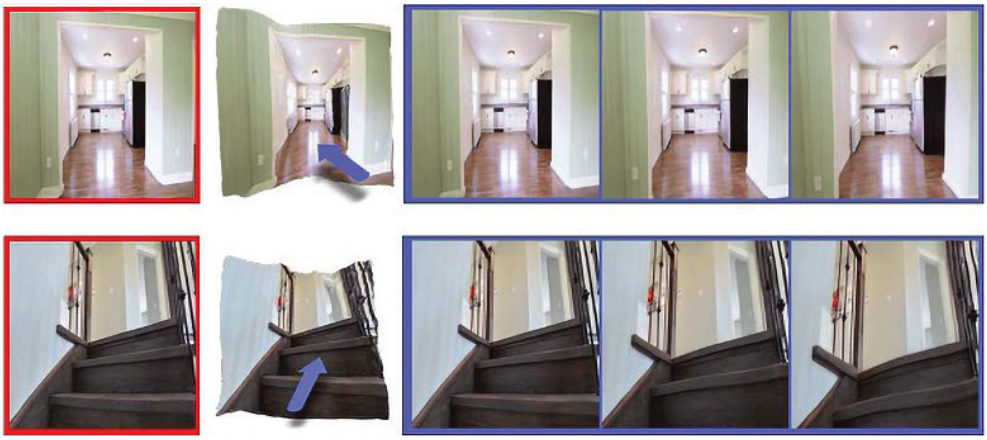


Рис. 9.1 ❖ Фотографии в красной рамке показывают изначальное изображение, а фотографии в синей рамке – новые сгенерированные изображения; это пример синтеза ракурсов с использованием методологии SynSin

- **синтез ракурсов с использованием достоверной глубины**: предусматривает группу техник, в которых рядом с изображением используется маска достоверности, представляющая глубину изображения и семантику. Хотя в некоторых случаях эти типы моделей могут давать хорошие результаты, собирать данные в больших масштабах сложно и обходится дорого, в особенности когда речь идет об уличных сценах. Кроме того, аннотировать такие данные в больших масштабах дорого и отнимает много времени;
- **синтез ракурсов по одному изображению**: это более реалистичная конфигурация; у нас есть только одно изображение, и мы стремимся реконструировать изображение с нового ракурса. Однако, используя только одно изображение, труднее получить более точные результаты. И метод SynSin принадлежит к группе методов, которые обеспечивают успешную работу синтеза ракурсов на передовом уровне.

Итак, мы провели краткий обзор синтеза ракурсов. Теперь обследуем метод SynSin, рассмотрим сетевую архитектуру поглубже и проинспектируем процессы тренировки и тестирования модели.

Сетевая архитектура модели SynSin

Идея SynSin состоит в решении задачи синтеза ракурсов с помощью сквозной модели при использовании только одного изображения во время тестирования. Эта модель не нуждается в аннотациях к 3D-данным и обеспечивает очень хорошую точность по сравнению с базовым уровнем:

Рис. 9.2 ❖ Структура сквозного метода SynSin

Модель тренируется в сквозном порядке и состоит из трех разных модулей:

- сетей пространственных признаков и глубин,
- нейронного отрисовщика облака точек,
- модуля уточнения[1] и дискриминатора.

Давайте рассмотрим каждый из них подробнее, чтобы лучше понять архитектуру.

Сети пространственных признаков и глубин

Если увеличить масштаб первой части рис. 9.2, то мы увидим две разные сети, в которые подается одно и то же изображение. Это сеть пространственных признаков (**f**) и сеть глубин (**d**) (рис. 9.3):

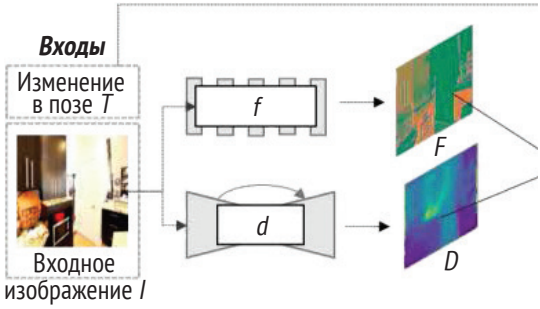


Рис. 9.3 ❖ Входные и выходные данные сетей пространственных признаков и глубин

С учетом опорного изображения и желаемого изменения позы (T) мы хотим сгенерировать изображение, как если бы это изменение позы было применено к опорному изображению. В первой части мы используем только опорное изображение и передаем его в две сети. Сеть пространственных признаков предназначена для усвоения карт признаков, т. е. изображений с более высокой разрешающей способностью. Эта часть модели отвечает за усвоение семантической информации об изображении. Она состоит из восьми блоков ResNet и выводит 64-мерные карты признаков по каждому

[1] От англ. *refinement*. – *Прим. перев.*

пикселу изображения. Результат на выходе имеет ту же разрешающую способность, что и у изначального изображения.

Далее, сеть глубин ориентирована на усвоение 3D-структуры изображения. 3D-структура не будет точной, так как мы не используем точные 3D-аннотации. Однако в дальнейшем модель будет ее улучшать. В этой сети используется UNet с восемью слоями понижающей и повышающей частоты отбора, за которыми следует сигмоидный слой. Опять же, результат на выходе имеет ту же разрешающую способность, что и у изначального изображения.

Как вы могли заметить, обе модели поддерживают высокую разрешающую способность для выходных каналов. Это в дальнейшем поможет реконструировать более точные и качественные изображения.

Нейронный отрисовщик облака точек

Следующим шагом является создание 3D-облака точек, которое затем можно использовать с трансформантой точки обзора для отрисовки нового изображения с новой точки обзора. Для этого используется комбинированный результат на выходе из сети пространственных признаков и сети глубин.

На следующем шаге идет отрисовка изображения с другой точки обзора. В большинстве сценариев будет использоваться наивный отрисовщик. Он проецирует 3D-точки на один пиксел или небольшую область в новом ракурсе. В наивном отрисовщике используется z-буфер, который поддерживает все расстояния от точки до камеры. Проблемой наивного отрисовщика является его недифференцируемость, и, стало быть, нельзя использовать градиенты для обновления сетей глубин и пространственных признаков. Более того, мы хотим отрисовывать признаки, а не RGB-изображения. Это означает, что наивный отрисовщик работать для этой техники не будет:

Рис. 9.4 ❖ Трансформация позы в нейронном отрисовщике облака точек

Почему бы просто не дифференцировать наивные отрисовщики? Здесь мы сталкиваемся с двумя трудностями.

- **Малые окрестности**: как отмечалось ранее, каждая точка появляется только на одном или нескольких пикселах отрисованного изображения. Поэтому для каждой точки существует всего несколько градиен-

тов. В этом недостаток локальных градиентов, который снижает результативность сети, опирающейся на обновления градиентов.

- **Жесткий z-буфер**: z-буфер удерживает для отрисовки изображения только ближайшую точку. Если новые точки окажутся ближе, то результат неожиданно резко изменится.

В целях преодоления представленных выше трудностей модель пытается их смягчить за счет применения техники, которая называется **нейронным отрисовщиком облака точек**. С этой целью вместо того, чтобы назначать пиксел точке, отрисовщик создает пятна с варьирующимся влиянием. За счет этого решается проблема малых окрестностей. Что касается жесткого z-буфера, то происходит накопление эффекта ближайших точек, а не только самой ближайшей точки:

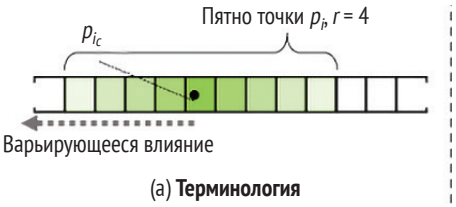


Рис. 9.5 ❖ Проецирование точки методом разбрызгивания

3D-точка проецируется и разбрызгивается с радиусом r (рис. 9.5). Затем влияние 3D-точки на этот пиксел измеряется евклидовым расстоянием между центром разбрызганной точки и этой точкой:

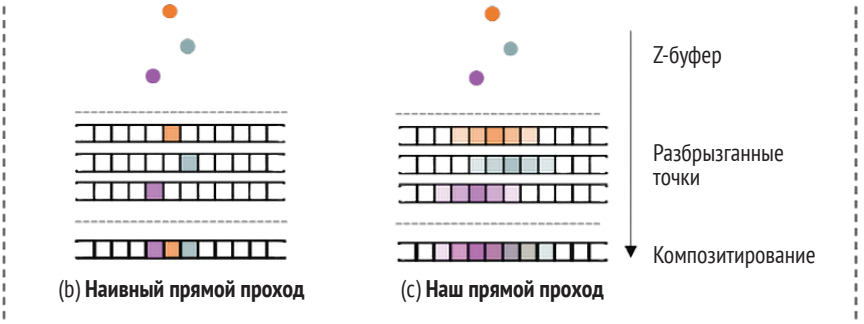


Рис. 9.6 ❖ Эффект прямого и обратного распространения нейронным отрисовщиком облака точек на примере наивной отрисовки (b) и принятой в SynSin отрисовки (c)

Как видно на предыдущем рисунке, каждая точка разбрызгивается, что помогает не терять слишком много информации и преодолевать перечисленные выше трудности.

Преимущество этого подхода состоит в том, что он позволяет собирать больше градиентов для одной 3D-точки, что улучшает процесс усвоения в сети как для сети пространственных признаков, так и для сети глубин:

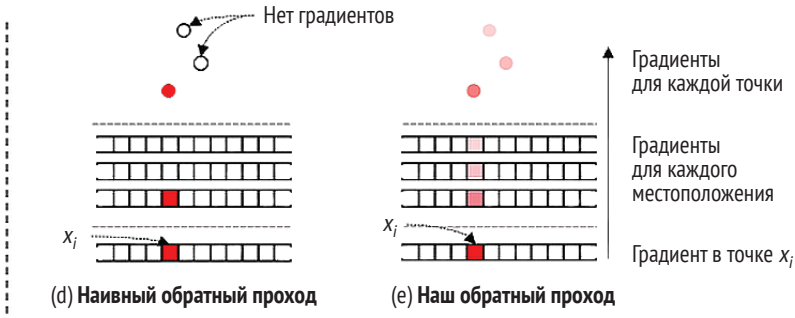
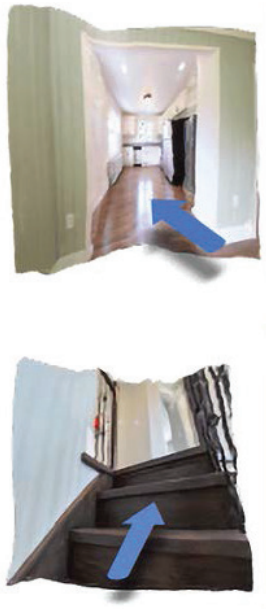


Рис. 9.7 ❖ Обратное распространение в наивном отрисовщике и в нейронном отрисовщике облака точек

Наконец, нам нужно собрать и накопить точки в z-буфере. Сначала мы сортируем точки по их удаленности от новой камеры, а затем для накопления точек используются K-ближайшие соседи с альфа-композитированием[1]:



Усвоенное облако 3D-точек
с наложенной траекторией

Рис. 9.8 ❖ Вывод облака 3D-точек

Как видно из рис. 9.8, облако 3D-точек выводит неуточненный новый ракурс. Затем этот результат должен быть подан на вход в модуль уточнения.

[1] От англ. *alpha compositing*; техника совмещения двух и более разных изображений, созданных независимо друг от друга. – *Прим. перев.*

Модуль уточнения и дискриминатор

И последнее, но не менее важное: модель состоит из модуля уточнения. У этого модуля две задачи: во-первых, повысить точность проецируемого признака, а во-вторых, заполнить невидимую часть изображения с нового ракурса. Он должен выводить семантически значимые и геометрически точные изображения. Например, если на изображении видна только одна часть стола, а в новом ракурсе изображение должно содержать более крупную его часть, то данный модуль должен семантически понимать, что это стол, и при реконструкции должен поддерживать линии новой части геометрически правильными (например, прямые должны оставаться прямыми). Модель усваивает эти свойства из набора данных реальных изображений:

Рис. 9.9 ❖ Модуль уточнения

Модуль уточнения (**g**) получает данные на выходе из нейронного отрисовщика облака точек, а затем на выходе выдает окончательное реконструированное изображение. Затем он используется в целевых функциях потери, чтобы улучшать процесс тренировки.

Указанная задача решается с помощью генеративных моделей. Используется ResNet с восемью блоками, а для поддержания хорошей разрешающей способности изображения также использовались блоки понижающей и повышающей частоты отбора. Мы используем GAN-сеть с двумя многослой-

ными дискриминаторами и функцией потери сочетаемости признаков на дискриминаторе[1].

Окончательная потеря модели состоит из L1-потери, потери содержимого и дискриминаторной потери между сгенерированным и целевым изображениями:

$$\mathcal{L} = \lambda_{GAN}\mathcal{L}_{GAN} + \lambda_{l1}\mathcal{L}_{l1} + \lambda_c\mathcal{L}_c.$$

Затем функция потери используется, как обычно, для оптимизации модели.

Именно таким образом в модели SynSin компонуются различные модули, чтобы создавать сквозной процесс синтеза ракурсов всего по одному изображению. Далее мы рассмотрим практическую реализацию модели.

Тренировка и тестирование модели на практике

Исследовательская группа Facebook Research выпустила репозиторий модели SynSin на GitHub, который позволяет тренировать модель и использовать предварительно натренированную модель для формирования вычислительного вывода. В этом разделе мы обсудим процесс тренировки и вычислительный вывод с применением предварительно натренированных моделей.

1. Но сначала необходимо настроить модель. Нужно клонировать репозиторий GitHub, создать среду и установить все требующиеся пакеты:

   ```
   git clone https://github.com/facebookresearch/synsin.git
   cd synsin/
   conda create --name synsin_env --file requirements.txt
   conda activate synsin_env
   ```

   Если с помощью приведенной выше команды установить требующиеся пакеты невозможно, то их всегда можно установить вручную. В случае ручной установки следуйте инструкциям файла `synsin/INSTALL.md`.
2. Модель была натренирована на трех разных наборах данных:
   1) `RealEstate10K` (набор данных о недвижимости),
   2) `MP3D` (набор данных панорамных ракурсов сцен со зданиями),
   3) `KITTI` (набор эталонных фотоснимков автомобилей и пешеходов на шоссе).

[1] Сочетание признаков (Feature Matching) – это регуляризирующая целевая функция в генераторе GAN-сетей, которая не дает его тренировать излишне много на текущем дискриминаторе. – *Прим. перев.*

Данные можно скачать с веб-сайтов наборов данных для проведения тренировки модели. В этой книге мы будем использовать набор данных KITTI; однако не стесняйтесь попробовать и другие.
Инструкции по скачиванию набора данных KITTI находятся в репозитории SynSin по адресу https://github.com/facebookresearch/synsin/blob/main/KITTI.md.
Сначала необходимо скачать набор данных с веб-сайта и сохранить файлы в `${KITTI_HOME}/dataset_kitti`, где `KITTI_HOME` – это путь, по которому будет расположен набор данных.

3. Далее нужно обновить файл Python `./options/options.py`, в котором нужно добавить путь к набору данных KITTI на локальной машине:

```
elif opt.dataset == 'kitti':
    opt.min_z = 1.0
    opt.max_z = 50.0
    opt.train_data_path = (
        './DATA/dataset_kitti/'
    )
    from data.kitti import KITTIDataLoader
    return KITTIDataLoader
```

Если вы собираетесь использовать другой набор данных, то следует найти загрузчик `DataLoader` для других наборов данных и добавить путь к этому набору данных.

4. Перед проведением тренировки необходимо скачать предварительно натренированные модели, выполнив следующую ниже команду:

```
bash ./download_models.sh
```

Если вы откроете и заглянете внутрь файла, то увидите, что он содержит все предварительно натренированные модели для всех трех наборов данных. Поэтому при запуске команды она создаст три разные папки для каждого набора данных и скачает все предварительно натренированные модели для этого набора данных. Мы можем их использовать как для тренировки, так и для формирования вычислительного вывода. Если вы не хотите, чтобы они скачивались все, то всегда можете скачать их вручную, просто выполнив следующую ниже команду:

```
wget https://dl.fbaipublicfiles.com/synsin/checkpoints/realestate/synsin.pth
```

Эта команда выполнит предварительно натренированную модель SynSin для набора данных о недвижимости. Для получения дополнительной информации о предварительно натренированных моделях можно скачать файл `readme.txt`, как показано ниже:

```
wget https://dl.fbaipublicfiles.com/synsin/checkpoints/readme.txt
```

5. Для тренировки необходимо выполнить файл Python `train.py`. Его можно выполнить из командной оболочки, используя Bash-скрипт `./train.sh`. Если открыть файл `train.sh`, то в нем можно увидеть команды для

трех разных наборов данных. Для набора данных KITTI по умолчанию используется пример, который выглядит следующим образом:

```
python train.py --batch-size 32 \
                --folder 'temp' --num_workers 4 \
                --resume --dataset 'kitti' --use_inv_z \
                --use_inverse_depth \
                --accumulation 'alphacomposite' \
                --model_type 'zbuffer_pts' \
                --refine_model_type 'resnet_256W8UpDown64' \
                --norm_G 'sync:spectral_batch' \
                --gpu_ids 0,1 --render_ids 1 \
                --suffix '' --normalize_image --lr 0.0001
```

Можно поэкспериментировать с параметрами и наборами данных и попытаться просимулировать результаты изначальной модели. После завершения процесса тренировки новую модель можно использовать для оценивания.

6. Для оценивания сначала необходимо сгенерировать достоверные изображения. Для этого нужно выполнить следующие ниже команды:

```
export KITTI=${KITTI_HOME}/dataset_kitti/images
python evaluation/eval_kitti.py --old_model ${OLD_MODEL}
                                --result_folder ${TEST_FOLDER}
```

Необходимо задать путь к папке, в которой будут сохраняться результаты, заменив `TEST_FOLDER` на используемое вами имя папки.
Первая строка экспортирует новую переменную с именем `KITTI` с путем `images` к набору данных. Следующий затем скрипт создает сгенерированные и достоверные пары по каждому изображению:

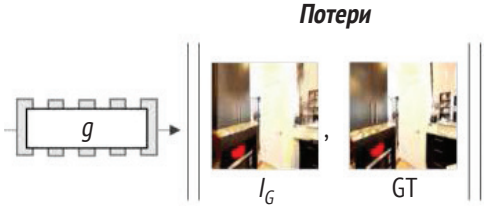


Рис. 9.10 ❖ Пример результата работы скрипта Python eval_kitti.py

Первое изображение является входным изображением, и второе – достоверным изображением. Третье изображение – это результат на выходе из сети. Как можно заметить, камера немного сдвинулась вперед, и в этом конкретном случае данные на выходе из модели выглядят очень хорошо.

7. Однако нам нужно некое числовое представление, чтобы понять, насколько хорошо работает сеть. Именно по этой причине необходимо выполнить файл Python `evaluation/evaluate_perceptualsim.py`, который рассчитывает точность:

```
python evaluation/evaluate_perceptualsim.py \
    --folder ${TEST_FOLDER} \
    --pred_image im_B.png \
    --target_image im_res.png \
    --output_file kitti_results
```

Приведенная выше команда поможет оценить модель при наличии пути к тестовым изображениям, одно из которых является предсказанным изображением, а другое – целевым.
Результат моего теста выглядит следующим образом:

```
Perceptual similarity for ./DATA/dataset_kitti/test_folder/:  2.0548
PSNR for /DATA/dataset_kitti/test_folder/:  16.7344
SSIM for /DATA/dataset_kitti/test_folder/:  0.5232
```

Одной из используемых при оценивании метрик является сходство восприятия, которое измеряет расстояние в пространстве признаков VGG. Чем оно ближе к нулю, тем выше сходство между изображениями. Следующая метрика, PSNR, служит для измерения реконструкции изображения. Она вычисляет отношение между максимальной мощностью сигнала и мощностью искажающего шума, которым в нашем случае является реконструированное изображение. Наконец, **индекс структурного подобия** (**SSIM**)[1] – это показатель, который количественно определяет ухудшение качества изображения.

8. Далее можно применить предварительно натренированную модель для формирования вычислительного вывода. Нам нужно входное изображение, которое мы будем использовать для вычислительного выведения результата (рис. 9.11).
9. Далее применяем модель `realestate`, чтобы сгенерировать новое изображение. Первым делом нужно настроить модель. Исходный код настройки модели находится в файле Python `set_up_model_for_inference.py` папки главы книги в репозитории на GitHub.
   Для настройки модели сначала нужно импортировать все необходимые пакеты:

```
import torch
import torch.nn as nn

import sys
sys.path.insert(0, './synsin')

import os
os.environ['DEBUG'] = '0'

from synsin.models.networks.sync_batchnorm import convert_model
from synsin.models.base_model import BaseModel
from synsin.options.options import get_model
```

[1] От англ. *Structural Similarity Index. – Прим. перев.*

Рис. 9.11 ❖ Входное изображение для формирования вычислительного вывода

10. Далее создадим функцию, которая на входе принимает путь к модели и на выходе выдает модель, готовую для формирования вычислительного вывода. Мы разобьем всю функцию на более мелкие фрагменты, чтобы лучше понять исходный код. Однако полная функция находится в папке главы книги в репозитории на GitHub:

```
torch.backends.cudnn.enabled = True

opts = torch.load(model_path)['opts']
opts.render_ids = [1]

torch_devices = [int(gpu_id.strip())
                 for gpu_id in opts.gpu_ids.split(",")]

device = 'cuda:' + str(torch_devices[0])
```

Здесь мы активируем пакет `cudnn` и определяем устройство, на котором модель будет работать. Кроме того, во второй строке импортируется модель, позволяя ей получить доступ ко всем заданным для тренировки параметрам, которые при необходимости можно изменить. Обратите внимание, что список `render_ids` относится к ИД графического процессора, который в некоторых случаях может отличаться у пользователей с разными настройками оборудования.

11. Далее определяем модель:

```
model = get_model(opts)

if 'sync' in opts.norm_G:
```

```
        model = convert_model(model)
        model = nn.DataParallel(model,
                                torch_devices[0:1]).cuda()
    else:
        model = nn.DataParallel(model,
                                torch_devices[0:1]).cuda()
```

Функция `get_model` импортируется из файла Python `options.py`, которая загружает веса и возвращает окончательную модель. Затем на основе `options` проверяем наличие синхронизированной модели, т. е. мы выполняем модель на разных машинах. Если она у нас есть, то выполняем функцию `convert_model`, которая берет модель и заменяет все модули `BatchNorm` модулями `SunchronizedBatchNorm`.

12. Наконец, загружаем модель:

```
# Загрузить изначальную модель для тестирования
model_to_test = BaseModel(model, opts)
model_to_test.load_state_dict(torch.load(model_path)['state_dict'])
model_to_test.eval()

print("Модель загружена")
```

Функция `BaseModel` устанавливает окончательный режим. В зависимости от режима тренировки либо тестирования она может устанавливать оптимизатор и инициализировать веса. В нашем случае она настроит модель на тестовый режим.

Весь этот код резюмирован в одной функции с именем `synsin_model`, которую мы импортируем для формирования вычислительного вывода. Следующий ниже исходный код взят из файла Python `inference_unseen_image.py`. Мы напишем функцию, которая на входе принимает путь к модели, тестовое изображение и параметры трансформации в новый ракурс и на выходе выдает новое изображение с нового ракурса. Если задать параметр `save_path`, то она автоматически будет сохранять выходное изображение.

13. Опять же, сначала импортируем все модули, необходимые для формирования вычислительного вывода:

```
import matplotlib.pyplot as plt

import quaternion
import torch
import torch.nn as nn
import torchvision.transforms as transforms

from PIL import Image
from set_up_model_for_inference import synsin_model
```

14. Далее настраиваем модель и создаем трансформацию данных для предобработки:

```
model_to_test = synsin_model(path_to_model)
```

```
# Загрузить изображение
transform = transforms.Compose([
    transforms.Resize((256,256)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

if isinstance(test_image, str):
    im = Image.open(test_image)
else:
    im = test_image
    im = transform(im)
```

15. Теперь нужно задать параметры трансформации ракурса:

```
# Параметры трансформации
theta = theta
phi = phi
tx = tx
ty = ty
tz = tz

RT = torch.eye(4).unsqueeze(0)

# Задать поворот
RT[0,0:3,0:3] = torch.Tensor(
    quaternion.as_rotation_matrix(
        quaternion.from_rotation_vector([phi, theta, 0])))

# Задать трансляцию
RT[0,0:3,3] = torch.Tensor([tx, ty, tz])
```

Здесь нам нужно указать параметры поворота и трансляции. Обратите внимание, что `theta` и `phi` отвечают за поворот, а `tx`, `ty` и `tz` используются в трансляции.

16. Далее используем загруженное изображение и новую трансформацию, чтобы получить выходные данные сети:

```
batch = {
    'images' : [im.unsqueeze(0)],
    'cameras' : [{
        'K' : torch.eye(4).unsqueeze(0),
        'Kinv' : torch.eye(4).unsqueeze(0)
    }]
}

# Сгенерировать новый ракурс новой трансформации
with torch.no_grad():
    pred_imgs = model_to_test.model.module.forward_angle(
        batch, [RT])
    depth = nn.Sigmoid()(
        model_to_test.model.module.pts_regressor(
            batch['images'][0].cuda()))
```

Здесь `pred_imgs` – это данные на выходе из модели, которые представляют собой новое изображение, а `depth` – предсказанная моделью 3D-глубина.

17. Наконец, используем выходные данные сети, чтобы визуализировать изначальное изображение, новое предсказанное изображение и выходное изображение:

```
fig, axis = plt.subplots(1,3, figsize=(10,20))
axis[0].axis('off')
axis[1].axis('off')
axis[2].axis('off')

axis[0].imshow(im.permute(1,2,0) * 0.5 + 0.5)
axis[0].set_title('Входное изображение')
axis[1].imshow(pred_imgs[0].squeeze().cpu().permute(1,2,0).numpy() * 0.5 + 0.5)
axis[1].set_title('Сгенерированное изображение')
axis[2].imshow(depth.squeeze().cpu().clamp(max=0.04))
axis[2].set_title('Предсказанная глубина')
```

Для визуализации результата используется библиотека `matplotlib`. Вот результат приведенного выше исходного кода:

Рис. 9.12 ❖ Результат вычислительного вывода

Как мы видим, мы получили новый ракурс, и модель очень хорошо реконструирует новый угол обзора. Теперь можно поэкспериментировать с параметрами трансформации, чтобы сгенерировать изображения с другого ракурса.

18. Если немного изменить `theta` и `phi`, то можно получить еще одну трансформацию ракурса. Теперь реконструируем правую часть изображения:

```
# Параметры трансформации
theta = 0.15
phi = 0.1
tx = 0
ty = 0
tz = 0.1
```

Результат выглядит следующим образом:

Рис. 9.13 ❖ Результат вычислительного вывода

Изменение всех параметров трансформации сразу или их изменение с большим шагом может привести к ухудшению точности.

19. Теперь вы научились создавать изображение с нового ракурса. Далее напишем краткий исходный код для последовательного создания изображений и небольшого видео:

```
from inference_unseen_image import inference
from PIL import Image
import numpy as np
import imageio

def create_gif(model_path, image_path, save_path,
               theta = -0.15, phi = -0.1, tx = 0,
               ty = 0, tz = 0.1, num_of_frames = 5):
    '''
```

```
    Эта функция создает последовательный
    gif из входного изображения и сохраняет
    результат в количестве num_of_frames
    кадров реконструированных изображений
    '''
    im = inference(model_path, test_image=image_path,
                   theta=theta, phi=phi, tx=tx, ty=ty, tz=tz)
    frames = []
    for i in range(num_of_frames):
        im = Image.fromarray((im * 255).astype(np.uint8))
        frames.append(im)
        im = inference(model_path, im, theta=theta,
                   phi=phi, tx=tx, ty=ty, tz=tz)

    imageio.mimsave(save_path, frames,  duration=1)

# Пример вызова функции
MODEL = './synsin/modelcheckpoints/realestate/zbufferpts.pth'
IMAGE = 'appartement.JPG'
SAVE = 'test_gif.gif'
create_gif(MODEL, IMAGE, SAVE)
```

Функция в этом фрагменте исходного кода на входе принимает изображение и для заданного числа кадров генерирует последовательные изображения. Под последовательным мы подразумеваем, что каждый результат на выходе из модели подается на вход генерации следующего изображения:

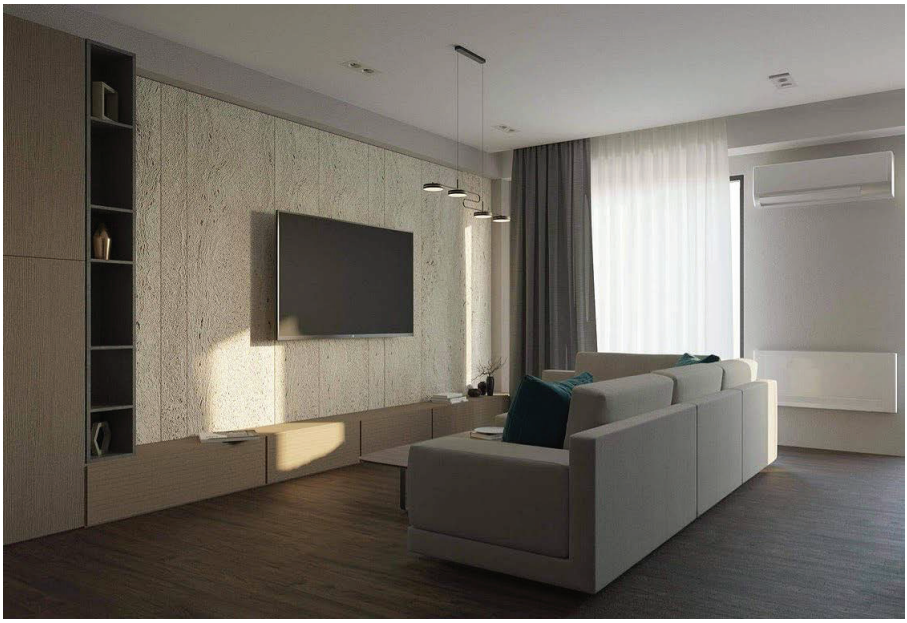


Рис. 9.14 ❖ Последовательный синтез ракурса

На приведенном выше рисунке имеется четыре последовательных кадра. Как видите, при попытке подавать в нее параметры с большим шагом модели все труднее и труднее генерировать хорошие изображения. Теперь самое время поэкспериментировать с гиперпараметрами модели, различными настройками камеры и размерами шага, чтобы увидеть, как это может улучшить или уменьшить точность данных на выходе из модели.

Резюме

В начале главы мы рассмотрели структуру модели SynSin, и вы получили глубокое понимание сквозного процесса модели. Как отмечалось ранее, одним из интересных подходов, применявшихся при создании модели, был дифференцируемый отрисовщик, который используется в качестве составной части тренировки. Кроме того, вы увидели, что модель помогает решать проблемы отсутствия огромного аннотированного набора данных и отсутствия нескольких изображений для тестирования. Именно по этой причине указанную передовую модель гораздо проще использовать в реальных сценариях. Мы рассмотрели плюсы и минусы модели. Кроме того, мы рассмотрели процедуру инициализации модели, тренировки, тестирования и применения новых изображений для формирования вычислительного вывода.

В следующей главе мы рассмотрим модель Mesh R-CNN, которая объединяет две разные задачи (обнаружение объектов и построение 3D-модели) в одну модель. Вы обследуете модельную архитектуру и протестируете результативность модели на случайном изображении.

Глава 10

Модель Mesh R-CNN

Эта глава посвящена передовой модели под названием Mesh R-CNN, целью которой является комбинирование двух разных, но важных задач в одну сквозную модель. Это комбинация хорошо известной модели сегментации изображения Mask R-CNN и новой модели предсказания 3D-структуры. Эти две задачи исследовались довольно много и глубоко, каждая в отдельности.

Модель Mask R-CNN базируется на алгоритме обнаружения объектов и сегментации экземпляров, получившем самые высокие баллы прецизионности в эталонных тестовых наборах данных. Она принадлежит к семейству R-CNN-сетей и представляет собой двухэтапную сквозную модель обнаружения объектов.

Модель Mesh R-CNN выходит за рамки задачи обнаружения 2D-объектов, вдобавок выдавая на выходе трехмерную полигональную сетку обнаруженных объектов. Если люди видят окружающий мир в 3D, то, стало быть, объекты – трехмерны. Так, почему бы не иметь модель обнаружения, которая тоже выводит объекты в 3D?

В этой главе вы разберетесь в принципе работы модели Mesh R-CNN. Кроме того, мы углубимся в понимание разных элементов и методов, используемых в моделях, таких как вокселы, полигональные сетки, графовые сверточные сети и операторы кубификации.

Далее мы обследуем репозиторий GitHub, предоставленный авторами исследовательской работы по Mesh R-CNN. Мы испытаем демонстрационный пример на своем изображении и визуализируем результаты предсказания.

Наконец, мы обсудим вопрос воспроизведения процессов тренировки и тестирования модели Mesh R-CNN и разберемся в результатах эталонного тестирования модельной точности.

В этой главе мы рассмотрим следующие ниже главные темы:

- понимание сеточных и воксельных структур,
- понимание структуры модели,
- понятие графовой свертки,
- испытание демонстрационного примера модели Mesh R-CNN,
- понимание процесса тренировки и тестирования модели.

Технические требования

Для выполнения примеров исходного кода этой книги в идеале понадобится компьютер с графическим процессором. Тем не менее вполне можно выполнять фрагменты исходного кода только с центральным процессором.

Рекомендуемая компьютерная конфигурация включает следующее:

- GPU, такой как серия GTX или серия RTX с не менее 8 Гб памяти,
- Python 3,
- библиотеки PyTorch и PyTorch3D,
- Detectron2,
- репозиторий модели Mesh R-CNN, который находится по адресу https://github.com/facebookresearch/meshrcnn.

Фрагменты исходного кода к этой главе находятся по адресу https://github.com/packtpublishing/3d-deep-learning-with-python.

Общий обзор полигональных сеток и вокселов

Как упоминалось ранее в этой книге, полигональные сетки и вокселы – это два разных представления 3D-данных. В модели Mesh R-CNN используются оба представления, чтобы получать более качественные предсказания 3D-структуры.

Полигональная сетка, или меш, – это поверхность 3D-модели, представленная в виде многоугольников (полигонов), в которой каждый многоугольник может быть представлен в виде треугольника. Полигональные сетки состоят из вершин, соединенных ребрами. Соединение ребра и вершины создает грани, которые обычно имеют треугольную форму. Это представление хорошо подходит для более быстрых трансформаций и отрисовки:

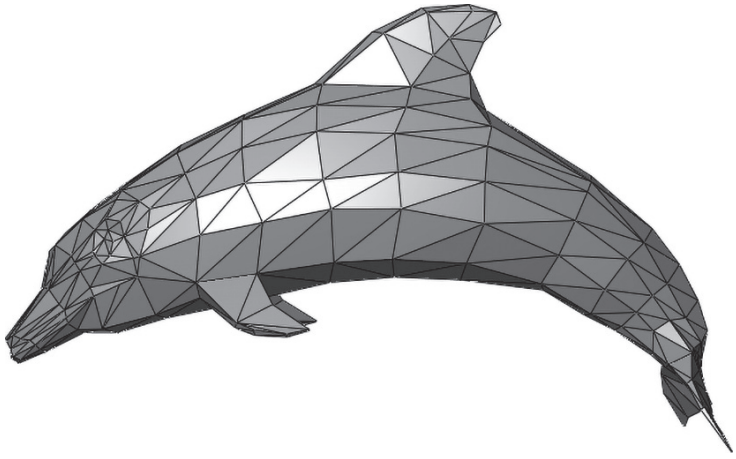


Рис. 10.1 ❖ Пример полигональной сетки

Вокселы – это 3D-аналоги 2D-пикселов. Поскольку каждое изображение состоит из 2D-пикселов, логично использовать ту же идею и для представления 3D-данных. Каждый воксел – это куб, а каждый объект – это группа кубов, одна часть которых видима извне, а другая расположена внутри объекта. Визуализировать 3D-объекты проще с помощью вокселов, но это не единственный вариант их использования. В задачах глубокого обучения вокселы можно использовать на входе в трехмерные сверточные нейронные сети:

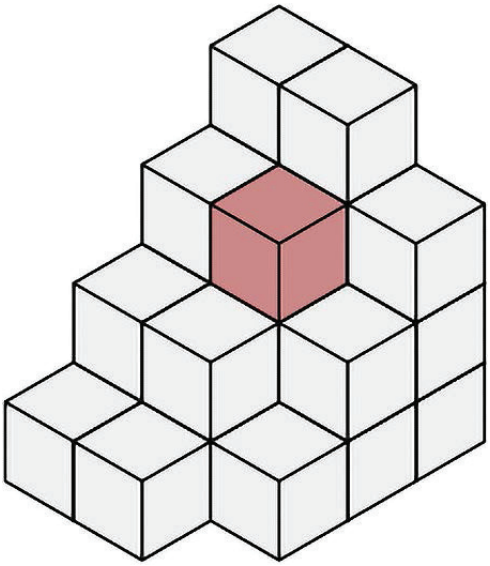


Рис. 10.2 ❖ Пример воксела

В модели Mesh R-CNN используются оба типа представлений 3D-данных. Эксперименты показали, что предсказание вокселов и последующая их трансформация в полигональную сетку, а затем уточнение полигональной сетки помогает сети учиться лучше.

Далее мы рассмотрим архитектуру модели Mesh R-CNN, чтобы увидеть, как вышеупомянутые представления 3D-данных создаются по входным изображениям.

Архитектура модели Mesh R-CNN

Обнаружение 3D-форм привлекло внимание многих исследователей. В результате было разработано много моделей, которые получили хорошую точность, но они в основном были ориентированы на синтетические эталонные тесты и изолированные объекты:

Рис. 10.3 ❖ Примеры 3D-объектов набора данных ShapeNet

В то же время быстро развивались задачи обнаружения 2D-объектов и сегментации изображений. Многие модели и архитектуры решают эту задачу с высокой точностью и скоростью. Существуют технические решения по локализации объектов и обнаружению ограничительных рамок и масок. Одно из них называется Mask R-CNN и представляет собой модель обнаружения объектов и сегментации экземпляров. Эта передовая модель имеет целый ряд реальных применений.

И все же мы видим мир в 3D. И авторы исследовательской работы Mesh R-CNN решили объединить эти два подхода в единое решение: модель, которая обнаруживает объект на реалистичном изображении и выводит трехмерную полигональную сетку вместо маски. В новой модели используется передовая модель обнаружения объектов, которая на входе принимает RGB-изображение и на выходе выдает метку класса, маску сегментации и трехмерную полигональную сетку объектов. Авторы добавили в модель Mask R-CNN новую ветвь, которая отвечает за предсказание треугольных сеток высокой разрешающей способности:

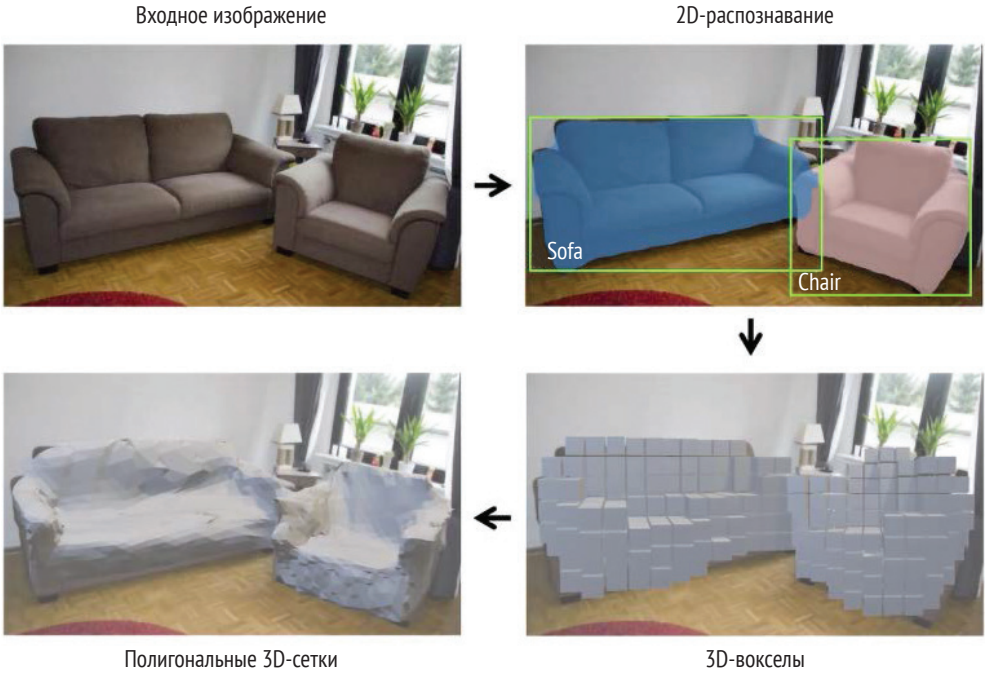


Рис. 10.4 ❖ Общая структура модели Mesh R-CNN

Авторы стремились создать одну модель, пригодную для тренировки в сквозном порядке. Именно по этой причине они взяли передовую модель Mask R-CNN и добавили новую ветвь, чтобы предсказывать полигональную сетку. Перед тем как углубляться в составную часть модели, служащую для предсказания полигональной сетки, давайте проведем краткий обзор модели Mask R-CNN:

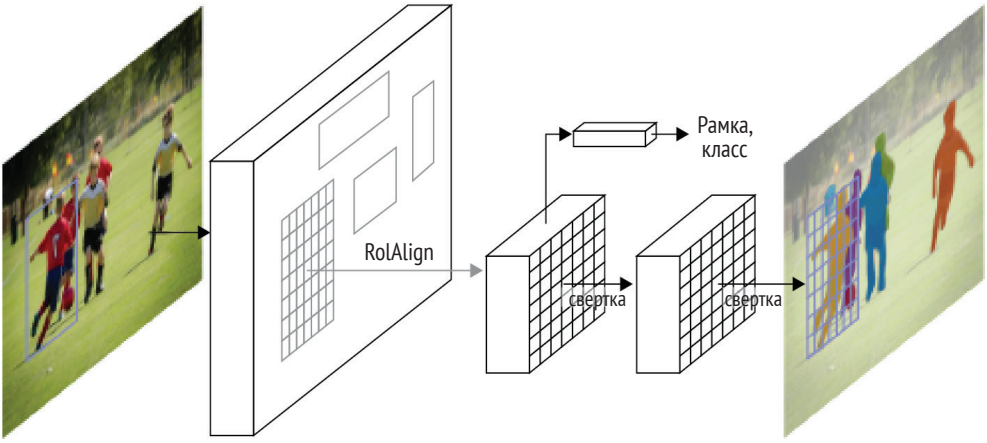


Рис. 10.5 ❖ Структура модели Mask R-CNN
(ссылка: https://arxiv.org/abs/1703.06870)

Модель Mask R-CNN на входе принимает RGB-изображение и на выходе выдает ограничительные рамки, метки категорий и маски сегментации экземпляров. Сначала изображение проходит через каркасную сеть, которая обычно основана на ResNet, – например, ResNet-50-FPN. Каркасная сеть выдает на выходе карту признаков, которая подается на вход следующей сети: **сети предложений участков** (**RPN**)[1]. Эта сеть выдает на выходе предложения. Затем ветви классифицирования объектов и предсказания масок обрабатывают предложения и выдают на выходе соответственно классы и маски.

Эта структура модели Mask R-CNN одинакова и для модели Mesh R-CNN. Однако в конце был добавлен предсказатель полигональной сетки. Предсказатель полигональной сетки – это новый модуль, состоящий из двух ветвей: ветви вокселов и ветви уточнения полигональной сетки.

Ветвь вокселов на входе принимает предложенные и выровненные признаки и на выходе выдает грубые предсказания вокселов. Затем они передаются на вход ветви уточнения полигональной сетки, которая на выходе выдает окончательную полигональную сетку. Потери воксельной ветви и ветви уточнения полигональной сетки добавляются к потерям рамки и маски, и модель тренируется в сквозном порядке от начала до конца:

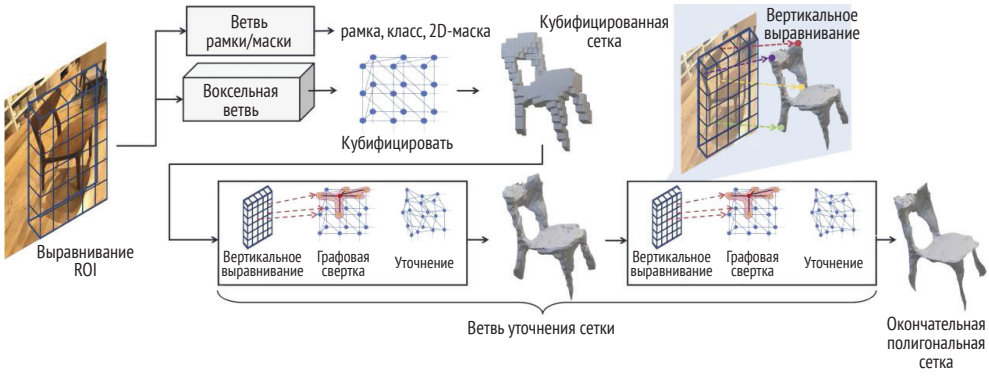


Рис. 10.6 ❖ Архитектура модели Mesh R-CNN

Графовые свертки

Прежде чем рассматривать структуру предсказателя полигональной сетки, давайте разберемся в том, что такое графовая свертка и как она работает.

Ранние варианты нейронных сетей были адаптированы под структурированные евклидовы данные. Однако в реальном мире большинство данных неевклидовы и имеют графовую структуру. В связи с этим в последнее время разработчики стали адаптировать многие варианты нейронных сетей под графовые данные, и одним из таких вариантов являются сверточные сети, именуемые **графовыми сверточными сетями** (**GCN**)[2].

---

[1] От англ. *Region Proposal Network. – Прим. перев.*

[2] От англ. *Graph Convolutional Networks. – Прим. перев.*

Полигональные сетки имеют такую графовую структуру, поэтому GCN-сети применимы в задачах предсказания трехмерной структуры. Базовой операцией CNN-сети является свертка, которая выполняется с помощью фильтров. Для свертки используется метод скользящего окна, а фильтры содержат веса, которые модель должна усваивать. В GCN-сетях для свертки используется аналогичный метод, хотя главное их отличие состоит в том, что число узлов может варьироваться, а узлы не упорядочены:

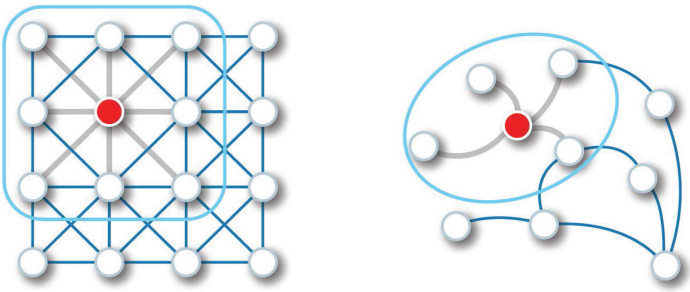


Рис. 10.7 ❖ Пример операции свертки в евклидовых и графовых данных (источник: https://arxiv.org/pdf/1901.00596.pdf)

На рис. 10.8 показан пример сверточного слоя. Входными данными сети являются граф и матрица смежности, которая представляет ребра между узлами при прямом распространении. Сверточный слой инкапсулирует в себе информацию по каждому узлу путем агрегирования информации из его окрестности. После этого применяется нелинейное преобразование. Позже данные на выходе из этой сети можно использовать в различных задачах, таких как классифицирование:

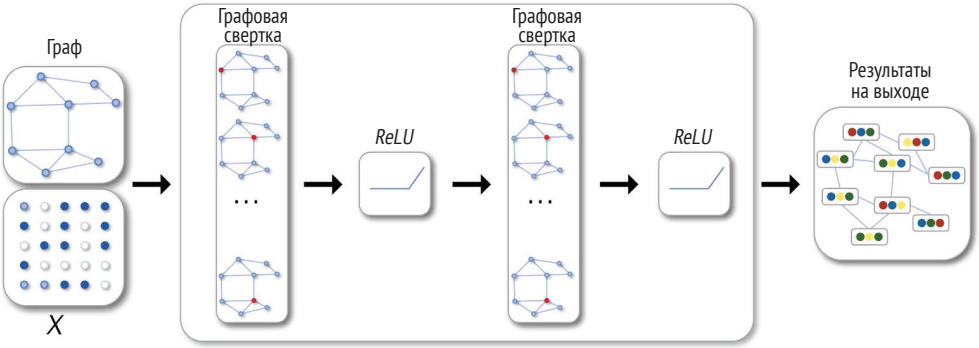


Рис. 10.8 ❖ Пример сверточной нейронной сети (источник: https://arxiv.org/pdf/1901.00596.pdf)

Предсказатель полигональной сетки

Модуль предсказания полигональной сетки предназначен для обнаружения 3D-структуры объекта. Он является логическим продолжением модуля `RoIAlign` и отвечает за предсказание и выдачу окончательной полигональной сетки.

Поскольку мы получаем трехмерные полигональные сетки из реальных изображений, мы не можем использовать фиксированные сеточные шаблоны с фиксированными сеточными топологиями. Именно по этой причине предсказатель полигональной сетки состоит из двух ветвей. Комбинация воксельной ветви и ветви уточнения полигональной сетки помогает смягчать проблему фиксированных топологий.

Воксельная ветвь аналогична масочной ветви из модели Mask R-CNN. Она берет из `ROIAlign` выровненные признаки и выдает решетку G×G×G вероятностей занятости вокселов. Далее используется операция кубифицирования[1]. В ней используется порог бинаризации занятости воксела. Каждый занятый воксел заменяется кубоидно-треугольной сеткой с 8 вершинами, 18 ребрами и 12 гранями.

Воксельная потеря представляет собой двоичную перекрестную энтропию, которая минимизирует предсказанные вероятности занятости воксела достоверными занятостями.

Ветвь уточнения полигональной сетки представляет собой последовательность из трех разных операций: выравнивания вершин, графовой свертки и уточнения вершин. Выравнивание вершин аналогично выравниванию ROI; по каждой вершине полигональной сетки оно выдает признак, выровненный по изображению.

Графовая свертка берет выровненные по изображению признаки и распространяет информацию по ребрам полигональной сетки. Уточнение вершин обновляет позиции вершин. Оно ориентировано на обновление геометрии вершин, оставляя топологию в фиксированном виде:

Рис. 10.9 ❖ Ветвь уточнения полигональной сетки

Как показано на рис. 10.9, может иметься несколько этапов уточнения. Каждый этап состоит из операций выравнивания вершин, графовой свертки

[1] От англ. *cubify*. – *Прим. перев.*

и уточнения вершин. В итоге мы получаем более точную трехмерную полигональную сетку.

Последней важной частью модели является функция сеточной потери. Для этой ветви используются фасочная и нормальная потери. Однако для этих методов требуются точки, отобранные из предсказанной и достоверной полигональных сеток.

В связи с этим используется следующий метод отбора из полигональной сетки: при заданных вершинах и гранях точки отбираются равномерно из вероятностного распределения поверхности полигональной сетки. Вероятность каждой грани пропорциональна ее площади.

Используя эти методы отбора, из достоверных данных отбирается облако точек Q и из предсказания – облако точек P. Далее вычисляется $\Delta_{P,Q}$, т. е. множество пар (p, q), где q – это ближайший сосед точки p в Q.

Фасочное расстояние рассчитывается между P и Q:

$$L_{cham}(\mathrm{P},\mathrm{Q}) = |P|^{-1} \sum_{(p,q)\in\Delta_{\mathrm{P,Q}}} \|p-q\|^2 + |Q|^{-1} \sum_{(q,p)\in\Delta_{\mathrm{Q,P}}} \|p-q\|^2.$$

Далее вычисляется абсолютное нормальное расстояние:

$$L_{norm}(\mathrm{P},\mathrm{Q}) = -|P|^{-1} \sum_{(p,q)\in\Delta_{\mathrm{P,Q}}} \|u_p \cdot u_q\|^2 - |Q|^{-1} \sum_{(q,p)\in\Delta_{\mathrm{Q,P}}} \|u_p \cdot u_q\|^2.$$

Здесь u_p и u_q – это единицы нормали соответственно к точкам p и q.

Однако эти две потери приводили к вырождению полигональной сетки. И именно по этой причине для качественного продуцирования полигональной сетки был добавлен регуляризатор формы, который назвали реберной потерей:

$$L_{edge}(\mathrm{V},\mathrm{E}) = \frac{1}{|E|} \sum_{(v,v')\in\mathrm{E}} \|v-v'\|^2, \quad \text{где } \mathrm{E} \subseteq \mathrm{V}\times\mathrm{V}.$$

Окончательная сеточная потеря представляет собой средневзвешенное значение трех представленных потерь: фасочной потери, нормальной потери и реберной потери.

В плане обучения были проведены два типа экспериментов. Первый заключался в проверке ветви предсказателя полигональной сетки. Здесь использовался набор данных ShapeNet, включающий 55 общих категорий классов. Он широко используется в эталонном тестировании предсказаний 3D-форм; однако он включает модели CAD, которые имеют отдельные фоны. В связи с этим сеточная предсказательная модель достигла статуса передовой. Кроме того, она преодолевает трудности, связанные с объектами, имеющими отверстия, которые предыдущие модели не могли хорошо обнаруживать:

Рис. 10.10 ❖ Предсказатель полигональной сетки на наборе данных ShapeNet

Третий ряд представляет данные на выходе предсказателя полигональной сетки. Мы видим, что он предсказывает трехмерную форму и очень хорошо оперирует топологией и геометрией объектов:

Рис. 10.11 ❖ Результат сквозной модели Mesh R-CNN

Следующим шагом является проведение экспериментов с реальными изображениями. Для этого использовался набор данных Pix3D, включающий 395 уникальных 3D-моделей, помещенных на 10 069 реальных изображений. В этом случае результаты эталонных тестов недоступны, поскольку авторы испытали эту технику впервые. Однако выходные результаты тренировки можно проверить на рис. 10.10.

Итак, мы обсудили архитектуру модели Mesh R-CNN, и теперь можно приступить к практической работе и применить модель Mesh R-CNN для отыскания объектов на тестовых изображениях.

Демонстрация модели Mesh R-CNN с помощью PyTorch3D

В этом разделе мы будем использовать репозиторий модели Mesh R-CNN для выполнения демонстрационного примера. Мы испытаем модель на нашем изображении и отрисуем выходной файл `.obj`, чтобы увидеть, как модель предсказывает 3D-форму. Кроме того, мы обсудим процесс тренировки модели.

Установка исходников Mesh R-CNN довольно проста. Сначала нужно установить библиотеки Detectron2 и PyTorch3D, а затем собрать модель Mesh R-CNN. `Detectron2` – это библиотека исследовательской группы Facebook Research, предоставляющая передовые модели обнаружения и сегментации. Она также содержит модель Mask R-CNN, на которой была построена модель Mesh R-CNN. Библиотека `detectron2` устанавливается следующей ниже командой:

```
python -m pip install 'git+https://github.com/facebookresearch/detectron2.git'
```

Если она у вас не работает, то обратитесь к веб-сайту, где описаны альтернативные способы установки. Далее нужно установить PyTorch3D, как описано ранее в этой книге. После того как оба предварительных требования были соблюдены, надо просто собрать модель Mesh R-CNN:

```
git clone https://github.com/facebookresearch/meshrcnn.git
cd meshrcnn && pip install -e .
```

Демонстрационный пример

Репозиторий содержит файл Python `demo.py`, который используется для демонстрации принципа работы сквозного процесса модели Mesh R-CNN. Указанный файл находится в `meshrcnn/demo/demo.py`. Давайте посмотрим на исходный код, чтобы понять механизм демонстрации. Данный файл содержит класс `VisualizationDemo`, состоящий из двух главных методов: `run_on_image` и `visualize_prediction`. Названия методов говорят сами за себя: первый

на входе берет изображение и на выходе выдает модельные предсказания, а другой визуализирует обнаружение маски, а затем сохраняет окончательную полигональную сетку и изображение с предсказаниями и уверенностью:

```
python demo/demo.py \
    --config-file configs/pix3d/meshrcnn_R50_FPN.yaml \
    --input /path/to/image \
    --output output_demo \
    --onlyhighest MODEL.WEIGHTS meshrcnn://meshrcnn_R50.pth
```

Для демонстрации надо просто выполнить приведенную выше команду из терминала. Указанная команда имеет следующие ниже опции:

- `--config-file` задает путь к конфигурационному файлу, который находится в каталоге `configs`,
- `--input` задает путь к входному изображению,
- `--output` задает путь к каталогу, в котором должны сохраняться предсказания,
- `--onlyhighest`, если `True`, то выводит только одну полигональную сетку и маску с наивысшей уверенностью.

Теперь давайте выполним команду и проверим результат.

Для демонстрации мы будем использовать изображение квартиры, которое брали в предыдущей главе:

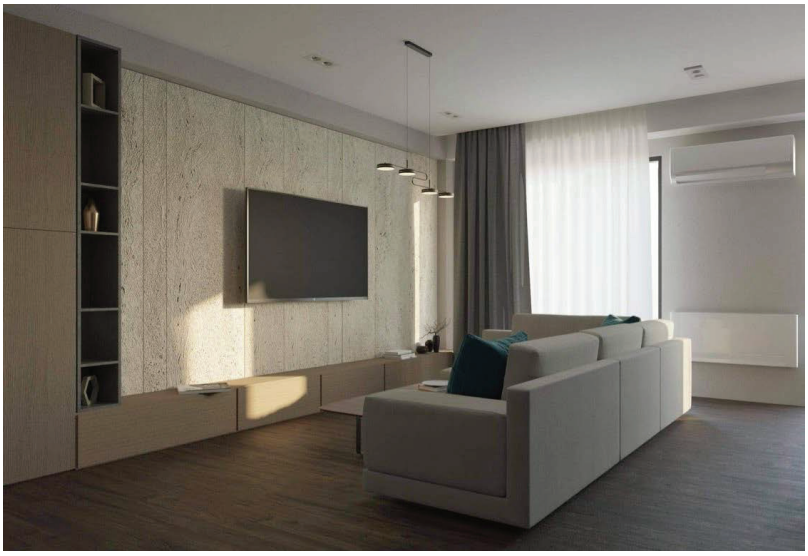


Рис. 10.12 ❖ Входное изображение для сети

Мы передаем путь к этому изображению в файл `demo.py`. После предсказания получаем визуализацию маски и полигональную сетку изображения. Поскольку использовался аргумент `--onlyhighest`, мы получили только одну маску, которая представляет собой предсказание объекта *диван*. Оно имеет

балл уверенности 88.7 %. Предсказание маски верно – она закрывает почти весь диван:

Рис. 10.13 ❖ Результат работы файла demo.py

Помимо маски, в том же каталоге мы также получили полигональную сетку, т. е. файл `.obj`. Теперь необходимо отрисовать изображения по 3D-объекту.

Следующий ниже исходный код взят из файла Python `chapt10/viz_demo_results.py` главы книги в репозитарии на GitHub.

1. Для начала импортируем все используемые в исходном коде библиотеки:

```
import torch
import numpy as np
import matplotlib.pyplot as plt

from pytorch3d.io import load_obj
from pytorch3d.structures import Meshes
from pytorch3d.renderer import (
    FoVPerspectiveCameras, look_at_view_transform, look_at_rotation,
    RasterizationSettings, MeshRenderer, MeshRasterizer, BlendParams,
    SoftSilhouetteShader, HardPhongShader, PointLights, TexturesVertex,
)

import argparse
```

2. Далее определяем аргументы для выполения исходного кода:

```
parser = argparse.ArgumentParser()
parser.add_argument('--path_to_mesh',
```

```
                    default="./demo_results/0_mesh_sofa_0.887.obj")
parser.add_argument('--save_path',
                    default='./demo_results/sofa_render.png')
parser.add_argument('--distance',
                    default=1,
                    help='distance from camera to the object')
parser.add_argument('--elevation',
                    default=150.0,
                    help='angle of elevation in degrees')
parser.add_argument('--azimuth',
                    default=-10.0,
                    help='rotation of the camera')

args = parser.parse_args()
```

Для файла `demo.py` требуется путь к полигональной сетке `path_to_mesh`, т. е. выходной файл `.obj`. Кроме того, нужно указать путь, по которому должны сохраняться отрисованные результаты, затем указать расстояние от камеры, угол возвышения и поворот.

3. Далее загружаем и инициализируем сеточный объект. Сначала с помощью функции `load_obj` из `pytorch3d` загружаем файл `.obj`. Затем делаем вершины белыми. Для создания сеточного объекта будем использовать структуру `Meshes` из `pytorch3d`:

```
# Загрузить obj-файл и проигнорировать текстуры и материалы.
verts, faces_idx, _ = load_obj(args.path_to_mesh)
faces = faces_idx.verts_idx

# Инициализировать каждую вершину белым цветом.
verts_rgb = torch.ones_like(verts)[None]  # (1, V, 3)
textures = TexturesVertex(verts_features=verts_rgb.to(device))

# Создать объект Meshes для дивана.
# Здесь в пакете только одна сетка.
sofa_mesh = Meshes(
    verts=[verts.to(device)],
    faces=[faces.to(device)],
    textures=textures
)
```

4. На следующем шаге инициализируем перспективную камеру. Затем задаем параметры смешивания, которые будут использоваться для смешивания граней. Параметр `sigma` управляет непрозрачностью, тогда как `gamma` управляет резкостью краев:

```
cameras = FoVPerspectiveCameras(device=device)

# Для смешивания 100 граней задаем несколько параметров,
# которые контролируют непрозрачность и резкость краев.
# Обратитесь к blending.py за подробностями.
blend_params = BlendParams(sigma=1e-4, gamma=1e-4)
```

5. Далее определяем параметры растеризации и затенения. Задаем размер выходного изображения равным 256×256 и параметр `faces_per_pixel` равным 100, который будет смешивать 100 граней в расчете на один пиксел. Затем применяем настройки растеризации, чтобы создать силуэтный сеточный отрисовщик путем компоновки растеризатора и затенителя:

```
raster_settings = RasterizationSettings(
    image_size=256,
    blur_radius=np.log(1. / 1e-4 - 1.) * blend_params.sigma,
    faces_per_pixel=100,
)

silhouette_renderer = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=cameras,
        raster_settings=raster_settings
    ),
    shader=SoftSilhouetteShader(blend_params=blend_params)
)
```

6. Теперь нужно создать еще один объект `RasterizationSettings`, так как мы также будем использовать фонговский отрисовщик. Нужно будет смешивать только одну грань на пиксел. Опять же, размер выходного изображения будет 256. Затем добавляем точечный источник света перед объектом. Наконец, инициализируем фонговский отрисовщик:

```
raster_settings = RasterizationSettings(
    image_size=256,
    blur_radius=0.0,
    faces_per_pixel=1,
)

lights = PointLights(device=device,
                     location=((2.0, 2.0, -2.0),))

phong_renderer = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=cameras,
        raster_settings=raster_settings
    ),
    shader=HardPhongShader(device=device,
                           ameras=cameras,
                           lights=lights)
)
```

7. Теперь необходимо создать позицию камеры на основе сферических углов. Мы будем использовать функцию `look_at_view_transform` и добавим упомянутые ранее параметры расстояния, высоты и азимута. Наконец, необходимо получить отрисованные данные на выходе из силуэтного и фонговского отрисовщика, сперва передав им на вход полигональную сетку и позицию камеры:

```
# Выбрать точку обзора, используя сферические углы
distance = args.distance   # растояние от камеры до объекта
elevation = args.elevation # угол возвышения в градусах
azimuth = args.azimuth     # поворот камеры

R, T = look_at_view_transform(distance, elevation, azimuth,
                              device=device)

# Отрисовать диван, предоставив значения R и T.
silhouette = silhouette_renderer(meshes_world=sofa_mesh, R=R, T=T)
image_ref = phong_renderer(meshes_world=sofa_mesh, R=R, T=T)
```

8. На последнем шаге визуализируем результаты. Для построения обоих отрисованных изображений будем использовать библиотеку `matplotlib`:

```
plt.figure(figsize=(10, 10))
plt.subplot(1, 2, 1)
# построить только альфа-канал RGBA-изображения
plt.imshow(silhouette.squeeze()[..., 3])
plt.grid(False)
plt.subplot(1, 2, 2)
plt.imshow(image_ref.squeeze())
plt.grid(False)

plt.savefig(args.save_path)
print('Отрисованное изображение сохранено.')
```

Результатом приведенного выше исходного кода будет изображение `.png`, которое будет сохранено в указанной в аргументах папке `save_path`. Для этих параметров и представленного здесь изображения отрисованная полигональная сетка будет выглядеть так:

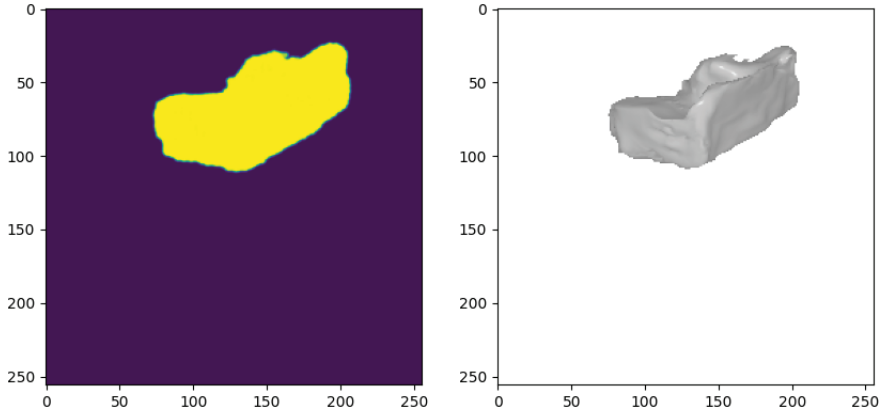


Рис. 10.14 ❖ Отрисованные 3D-данные на выходе из модели

Как видно с этого угла обзора, полигональная сетка очень похожа на диван, если не считать некоторых дефектов на невидимых частях. Теперь можно

поэкспериментировать с позицией камеры и освещением, чтобы отрисовать изображение объекта с другой точки обзора.

Репозиторий также предоставляет возможность выполнять и воспроизводить эксперименты, описанные в исследовательской статье по Mesh R-CNN. Он позволяет выполнять эксперимент на данных Pix3D и на данных ShapeNet.

Как упоминалось ранее, данные Pix3D содержат реальные изображения различной мебели IKEA. Эти данные использовались для оценивания всей модели Mesh R-NN от начала до конца.

Для того чтобы скачать эти данные, нужно выполнить следующую ниже команду:

```
datasets/pix3d/download_pix3d.sh
```

Данные содержат две секции с именами S1 и S2, и репозиторий предоставляет веса для обеих секций. После скачивания данных можно воспроизвести тренировку, выполнив следующую ниже команду:

```
python tools/train_net.py \
    --config-file configs/pix3d/meshrcnn_R50_FPN.yaml \
    --eval-only MODEL.WEIGHTS /path/to/checkpoint_file
```

При этом нужно быть осмотрительными с конфигурационными опциями. Изначальная модель распространялась и тренировалась на 8 Гб памяти графического процессора. Если у вас нет такой большой емкости, то она, вероятно, не достигнет такой же точности, поэтому следует отрегулировать гиперпараметры, чтобы повысить точность.

Вы можете использовать свои натренированные веса либо просто выполнить оценивание на предоставленных авторами предварительно натренированных моделях:

```
python tools/train_net.py \
    --config-file configs/pix3d/meshrcnn_R50_FPN.yaml \
    --eval-only MODEL.WEIGHTS /path/to/checkpoint_file
```

Приведенная выше команда выполнит оценивание модели для указанного файла контрольной точки. Контрольные точки находятся в репозитории модели на GitHub.

Далее, если вы хотите выполнить эксперимент на данных ShapeNet, то необходимо скачать данные. Это делается следующей ниже командой:

```
datasets/shapenet/download_shapenet.sh
```

Она скачает тренировочные, валидационные и тестовые наборы. Авторы также предоставили для набора данных ShapeNet исходный код предобработки. Предобработка сократит время загрузки. Следующая ниже команда выведет архивированные данные, что удобно для проведения тренировки в кластерах:

```
python tools/preprocess_shapenet.py \
    --shapenet_dir /path/to/ShapeNetCore.v1 \
```

```
    --shapenet_binvox_dir /path/to/ShapeNetCore.v1.binvox \
    --output_dir ./datasets/shapenet/ShapeNetV1processed \
    --zip_output
```

Далее, для того чтобы воспроизвести эксперимент, нужно просто выполнить файл Python `train_net_shapenet.py` с соответствующими конфигурационными опциями. Опять же, будьте осмотрительны при конфигурировании процесса обучения под свои аппаратные возможности:

```
python tools/train_net_shapenet.py --num-gpus 8 \
    --config-file configs/shapenet/voxmesh_R50.yaml
```

Наконец, с помощью следующей ниже команды всегда можно оценить свою модель или предоставленные авторами контрольные точки:

```
python tools/train_net_shapenet.py --num-gpus 8 \
    --config-file configs/shapenet/voxmesh_R50.yaml
```

Вы можете сравнить свои результаты с результатами, представленными в статье. На следующей ниже таблице показаны нормализованные по шкале запротоколированные авторами результаты обучения:

category	#instances	chamfer	normal	F1(0.1)	F1(0.3)	F1(0.5)
bench	8712	0.120899	0.657536	42.4005	86.0036	95.128
chair	32520	0.183693	0.712362	31.6906	79.8275	92.0139
lamp	11122	0.413965	0.672992	30.5048	70.3449	84.5068
speaker	7752	0.253796	0.730829	24.8335	74.6606	88.237
firearm	11386	0.168323	0.621439	47.2251	85.271	93.8171
table	40796	0.148357	0.75642	42.249	86.2039	94.1623
watercraft	9298	0.224168	0.642812	30.0589	75.5332	89.9764
plane	19416	0.187465	0.684285	39.009	80.998	92.1069
cabinet	7541	0.111294	0.75122	34.8227	86.9346	95.371
car	35981	0.107605	0.647857	29.6397	85.7925	96.2938
monitor	5256	0.218032	0.779365	27.2531	77.2979	90.904
couch	15226	0.144279	0.72302	27.5734	81.684	94.3294
cellphone	5045	0.121504	0.850437	42.9168	88.9888	96.1367
total	210051	0.184875	0.710044	34.629	81.5031	92.5372
per-instance	210051	0.171189	0.70275	34.9372	82.4107	93.1323

Рис. 10.15. Результаты оценивания на наборе данных ShapeNet

Таблица содержит название категории, число экземпляров в каждой категории, фасочную потерю, нормальную потерю и баллы F1.

Резюме

В этой главе мы представили новый взгляд на задачу обнаружения объектов. 3D-мир требует решений, которые работают соответствующим образом, и это один из первых подходов к достижению данной цели. Вы познакомились с принципом работы модели Mesh R-CNN, разобравшись в ее архитектуре и структуре. Мы подробно остановились на нескольких применяемых в модели интересных операциях и методах, таких как графовые сверточные сети, операция кубификации, структура сеточного предсказателя и многое другое. Наконец, вы узнали о том, как эту модель использовать на практике для обнаружения объектов на изображении, которые сеть никогда раньше не видела. Мы оценили результаты путем отрисовки 3D-объекта.

В этой книге мы рассмотрели понятия и концепции трехмерного глубокого обучения от основ до более продвинутых технических решений. Сначала вы познакомились с разными типами и структурами 3D-данных. Затем углубились в различные типы моделей, которые решают различные типы задач, такие как обнаружение полигональной сетки, синтез ракурсов и многое другое. Кроме того, вы добавили пакет PyTorch3D в свой набор инструментов компьютерного зрения. Завершив чтение этой книги, вы должны быть готовы к тому, чтобы решать реальные задачи, связанные с трехмерным компьютерным зрением и многим другим.

Тематический указатель

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

P

C

Т

У

Ф

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
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: **http://www.galaktika-dmk.com/**.

Ксудонг Ма, Вишах Хегде, Лилит Йольян

Трехмерное глубокое обучение на Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Логунов А. В.*
Корректор *Абросимова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 18,36. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**