КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ
И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ
Кафедра информационных систем

Ф.М. ГАФАРОВ, А.Ф. ГИЛЕМЗЯНОВ

НЕЙРОННЫЕ СЕТИ В PYTORCH

Учебное пособие

КАЗАНЬ

2024

УДК 004.032.26
ББК 32.973.2
Г24

Принято на заседании учебно-методической комиссии ИВМиИТ
(протокол № 8 от 29 марта 2024 года)

Рецензенты:
кандидат физико-математических наук, заведующий кафедрой прикладной математики и искусственного интеллекта КФУ **Д.М. Тумаков;**
кандидат технических наук, заместитель директора Института прикладной семиотики АН РТ **А.Р. Гатиятуллин**

Гафаров Ф.М.
Г24 Нейронные сети в PyTorch / Ф.М. Гафаров, А.Ф. Гилемзянов. – Казань: Казанский федеральный университет, 2024. – 106 с.

Учебное пособие посвящено изучению использования фреймворка PyTorch в работе с нейронными сетями. В пособии подробно рассмотрены основные концепции, такие как тензоры, слои, функции активации, в контексте фреймворка PyTorch. Рассмотрены базовые теоретические основы и особенности построения нейронных сетей различной архитектуры, таких как многослойные полносвязные нейронные сети перцептронного типа, рекуррентные нейронные сети, сверточные нейронные сети, графовые нейронные сети, а также нейронные сети архитектуры трансформер. На основе множества примеров рассмотрены основные этапы разработки нейронных сетей, начиная с определения архитектуры и заканчивая процессом обучения и оценки модели.

Адресовано в первую очередь студентам-бакалаврам, магистрантам направления «Информационные системы и технологии», а также широкому кругу читателей, интересующихся нейронными сетями и приложениями. Это комплексное методическое пособие предназначено для студентов, исследователей и практикующих специалистов, стремящихся освоить создание и обучение нейронных сетей с использованием фреймворка PyTorch.

УДК 004.032.26
ББК 32.973.2

© Гафаров Ф.М., Гилемзянов А.Ф., 2024
© Казанский федеральный университет, 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В последние десятилетия искусственные нейронные сети (ИНС) стали важным инструментом в области искусственного интеллекта, привнося в обработку данных и машинное обучение новые перспективы и возможности. Искусственные нейронные сети представляют собой математические модели, которые могут быть воплощены как программные или аппаратные системы. Они создаются с учетом принципов организации и функционирования биологических нейронных сетей, которые присутствуют в живых организмах. Термин этот возник в ходе изучения мозговых процессов с целью их моделирования. С точки зрения реализации искусственные нейронные сети представляют собой систему простых процессоров, искусственных нейронов, взаимодействующих и соединенных между собой. Подобные процессоры, в сравнении с теми, что используются в персональных компьютерах, обычно являются более простыми. Каждый из них работает исключительно с сигналами, которые он периодически получает, и сигналами, которые периодически посылает другим процессорам. При их объединении в обширную сеть с управляемым взаимодействием эти простые процессоры способны вместе справляться с сложными задачами.

Отличительной особенностью нейронных сетей является их способность к обучению, в отличие от традиционных алгоритмов, которые подразумевают программирование. Обучение заключается в настройке коэффициентов связей между нейронами. В ходе этого процесса нейронная сеть может выявлять сложные зависимости между входными и выходными данными, а также осуществлять обобщение. Успешное обучение позволяет сети давать верные результаты на основе данных, которые не входили в обучающую выборку, а также справляться с неполными и шумными данными, даже если они частично искажены.

Искусственные нейронные сети применяются для решения наиболее популярных типов задач, перечисленных ниже.

1. ***Распознавание образов и классификация.*** Образы могут представлять собой разнообразные объекты, такие как символы текста, изображения, образцы звуков и другие. В процессе обучения нейронной сети предоставляются разнообразные образцы, сопоставленные определенным классам. Обычно каждый образец представлен в виде вектора, содержащего значения различных признаков. В данной задаче количество нейронов в выходном слое обычно соответствует количеству уникальных классов. Устанавливается соответствие между выходами нейронной сети и классами, которые они представляют. Когда сети предъявляется конкретный образ, один из ее выходов должен сигнализировать о том, что данный образ принадлежит определенному классу.

2. ***Кластеризация.*** Кластеризация представляет собой процесс разделения набора входных сигналов на группы, при этом как количество, так и характеристики этих групп заранее неизвестны. После завершения обучения такой нейронной сети она способна идентифицировать, к какому конкретному классу принадлежит поступивший входной сигнал. Таким образом, данная сеть может выявлять новые классы сигналов, которые ранее были неизвестны. Установление соответствия между выделенными сетью классами и классами, существующими в предметной области, производится человеком. Примером инструмента для кластеризации может служить нейронная сеть Кохонена.

3. ***Прогнозирование.*** Задача прогнозирования с использованием нейронных сетей представляет собой важное направление в области искусственного интеллекта, где нейронные сети применяются для анализа данных и предсказания будущих событий. Применение нейронных сетей в задаче прогнозирования обусловлено их способностью автоматически выявлять сложные зависимости в данных и обобщать информацию из прошлого опыта для предсказания будущих событий. Например, в финансовой сфере нейронные сети могут использоваться для прогнозирования изменений цен на акции, в метеорологии – для предсказания погоды; кроме того, они могут быть использованы и в здравоохранении.

4. ***Аппроксимация функций.*** Нейронные сети могут аппроксимировать непрерывные функции. Применение нейронных сетей в задаче аппроксимации представляет собой мощный инструмент для приближения сложных и нелинейных функций. Аппроксимация в данном контексте означает создание модели, которая приближает неизвестную функцию на основе доступных данных.

5. ***Принятие решений и управление.*** Эта задача схожа с задачей классификации. В рамках классификации на вход нейронной сети поступают ситуации с их характеристиками, и сеть генерирует выходной признак, представляющий принятое ею решение. В данном случае в качестве входных сигналов используются разнообразные критерии, описывающие состояние контролируемой системы.

6. ***Сжатие данных и ассоциативная память.*** Нейросети обладают способностью выявлять взаимосвязи между различными параметрами, что предоставляет возможность более компактного представления данных большой размерности, особенно в случае тесных взаимосвязей между ними. Процесс, обратный этому, – восстановление полного набора данных из части информации, называется ассоциативной памятью. Ассоциативная память также дает возможность восстановления исходного сигнала или образа из исходных данных, даже если они зашумлены или повреждены.

7. *Генеративные задачи.* Применение нейронных сетей в генеративных задачах представляет собой важное направление в области искусственного интеллекта, где модели способны создавать новые данные, визуальные изображения, звуки и другие выразительные контенты. Генеративные задачи, в основе которых лежит использование нейронных сетей, имеют значительный потенциал в различных областях искусства, развлечений, дизайна и науки, предоставляя возможность творческого и инновационного подхода к созданию данных и контента.

Практическое изучение нейронных сетей будет основано на фреймворке PyTorch, который является одним из ведущих фреймворков для построения, обучения и применения ИНС. Он стал популярным среди исследователей и разработчиков благодаря своей гибкости, интуитивному интерфейсу и активному сообществу. Разработка нейронных сетей в PyTorch предоставляет уникальные преимущества, облегчая процесс создания и оптимизации моделей. Одной из ключевых особенностей PyTorch является динамический вычислительный граф, который облегчает отладку и экспериментирование с различными архитектурами нейронных сетей. Это особенно важно при выполнении исследовательских задач, когда требуется быстрая адаптация и изменение структуры модели.

В данном пособии мы рассмотрим основные этапы разработки нейронных сетей в PyTorch, начиная с определения архитектуры и заканчивая процессом обучения и оценки модели. Разберем основные концепции, такие как тензоры, слои, функции активации, а также рассмотрим практические примеры использования PyTorch для создания эффективных и мощных нейронных сетей различных архитектур. Программные коды всех примеров, рассмотренных в данном пособии, опубликованы в GitHub репозитории по адресу: https://github.com/fgafarov1977/pytorch_nn_tutorial.

ГЛАВА 1. Основы PyTorch. Работа с тензорами в PyTorch

PyTorch – это фреймворк машинного обучения, построенный на основе языка программирования Python. В настоящее время язык Python является одним из наиболее популярных языков программирования при построении проектов машинного обучения. Популярность этого языка программирования связана с его удобством и простотой, а также наличием огромного количества разнообразных библиотек и модулей для решения задач в различных областях. PyTorch является проектом с открытым исходным кодом, поэтому любой желающий может участвовать в развитии этого проекта. Изначально PyTorch, который назывался просто Torch, был разработан на языке Lua, и его первый релиз был 2002 г. Потом появилась версия на языке Python, который был назван PyTorch. PyTorch используется для решения различных задач, таких как машинное обучение, нейронные сети, компьютерное зрение, обработка естественного языка и звука, и многих других.

Перечислим основные особенности библиотеки PyTorch.

Во-первых, это простой и удобный в использовании программный интерфейс API (application programming interface). Благодаря этом интерфейсу разрабатывать программный код во фреймворке PyTorch довольно просто, наглядно и удобно. Пользователь может без особых сложностей построить модели самых разнообразных нейронных сетей, эффективно организовать процесс их обучения и применять обученные модели уже на практике в самых разных областях.

Вторая важная особенность – это использование языка программирования Python. Благодаря этому разработчики проектов машинного обучения в PyTorch фреймворке могут использовать все возможности и функции, предлагаемые средой Python.

Третья важная особенность – это использование вычислительных графов, которые, по сути, являются основой этой библиотеки. PyTorch предоставляет собой платформу, которая содержит в себе необходимые инструменты для работы с динамическими вычислительными графами и инструменты автоматического вычисления градиентов и производных на их основе.

К настоящему времени вокруг этого фреймворка выстроена довольно обширная экосистема, состоящая из различных библиотек, разрабатываемых сторонними разработчиками и командами, которые расширяют возможности PyTorch, упрощают и ускоряют процесс обучения моделей. Наиболее известные из них перечислены ниже.

1. TorchVision. Эта библиотека предоставляет набор инструментов и датасетов для компьютерного зрения, включая функции для работы с изображениями, аугментации данных и предобученные модели.

2. **TorchText.** Библиотека для обработки текстовых данных, включающая в себя функции токенизации, представления текста и другие утилиты для работы с естественным языком.

3. **TorchAudio.** Данная библиотека содержит инструменты для обработки аудиоданных, включая функции извлечения признаков, преобразования и предварительно обученные модели.

4. **TorchHub.** Библиотека, которая позволяет легко обмениваться и использовать предобученные модели и компоненты, обеспечивая удобный доступ к моделям, опубликованным сообществом.

5. **Ignite**. Библиотека для управления обучением и оценкой моделей. Предоставляет инструменты для организации цикла обучения, визуализации результатов и других операций.

6. **PyTorch Lightning.** Высокоуровневая библиотека для обучения глубоких нейронных сетей, предоставляющая удобный интерфейс и автоматизированный цикл обучения.

7. **Catalyst.** Библиотека для управления обучением, предоставляющая набор инструментов для обучения и оценки моделей.

8. **PyTorch Geometric**. Библиотека, предназначенная для работы с графовыми данными и графовыми нейронными сетями.

PyTorch содержит мощные и удобные инструменты и методы для работы с тензорами. Тензоры PyTorch похожи на массивы numpy, однако имеют дополнительные возможности, с помощью которых их можно эффективно и удобно использовать в проектах машинного обучения. Для построения различных типов нейронных сетей имеется большое количество различных готовых слоев, которые легко можно использовать. В настоящее время PyTorch содержит десятки различных слоев, среди которых есть линейные слои, разнообразные сверочные слои, рекуррентные слои, слои нормализации, дропаут слои, и многие другие.

Возможность использования графических процессоров для проведения высокопроизводительных вычислений в настоящее чрезвычайно необходима для обучения и использования нейронных сетей. Большинство «боевых» проектов машинного обучения использует графические процессоры. PyTorch позволяет в полной мере использовать все возможности, предоставляемые графическими процессорами на основе платформы CUDA. Благодаря использованию GPU вычисления можно проводить в десятки и сотни раз быстрее, чем на обычных процессорах CPU. Проведение вычислений с тензорами PyTorch на устройствах CUDA заложено в самом ядре фреймворка, поэтому даже без глубоких теоретических знаний касательно вычислений на CUDA можно легко и эффективно его использовать.

Перечислим основные модули, которые непосредственно используются при организации проектов машинного обучения.

Модуль Autograd. Этот модуль используется для автоматического вычисления производных в вычислительном графе. На основе возможностей этого модуля производится сохранение в памяти истории последовательных вычислений, произведенных в прямом направлении. Затем на их основе производится воспроизведение этих вычислений в обратном порядке для автоматического вычисления градиентов. Этот модуль очень удобен и эффективен при построении и обучении нейронных сетей.

Модуль Optim. Этот модуль, реализует основные алгоритмы оптимизации, которые используются при обучении нейронных сетей. В нем реализовано большинство наиболее часто используемых методов оптимизации, таких как стохастический градиентный спуск, метод Adama, метод Adagrad, и многие другие.

Модуль nn. Это базовый класс для всех модулей нейронной сети, все модели, создаваемые пользователями, также должны наследоваться от этого класса. В этом модуле также содержаться все слои, которые уже имеются в PyTorch.

Для того чтобы начать работать в PyTorch, необходимо его сначала установить. Для этого необходимо зайти на сайт https://pytorch.org/get-started/locally/. На этой странице есть удобный интерфейс (см. рис. 1), в котором вы можете указать необходимые параметры, такие как операционная система, тип установщика, использование CUDA, и т. д. И в зависимости от этих параметров система предложит вам команду для установки PyTorch, которую будет необходимо выполнить в командной строке.

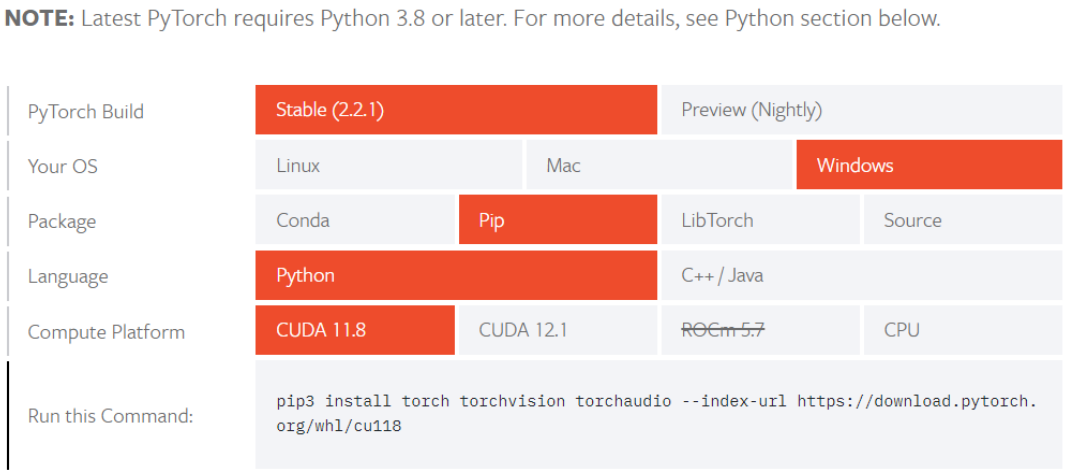


Рис. 1. Установка PyTorch

Если же вы работаете в системе Google Collab, то устанавливать PyTorch нет необходимости, так как он там уже установлен по умолчанию. Подключить библиотеку PyTorch можно написав **import torch** в самом начале программного скрипта.

В PyTorch имеется особый тип данных, который называется тензором. Тензоры – это специализированная структура данных, очень похожая на массивы и матрицы NumPy. Эти тензоры используются для кодирования входных и выходных данных модели, а также хранения параметров модели. Например, при подаче данных на вход нейронных сетей эти данные обязательно преобразуются в тензоры. При последовательном прохождении информации через различные слои нейронной сети от одного слоя другому фактически передаются тензоры разной формы. И еще одна важная особенность этих тензоров заключается в том, что они могут работать на графических процессорах или другом специализированном оборудовании для ускорения вычислений. На рисунке 2 показаны тензоры в трех представлениях.

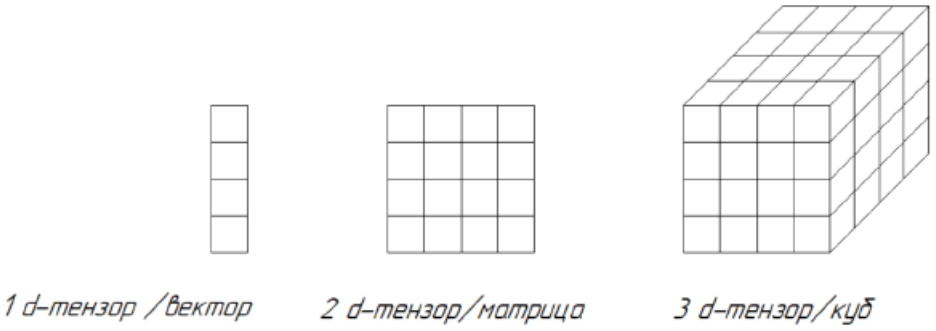


Рис. 2. Тензоры в PyTorch

Рассмотрим более подробно инструментарий для работы с тензорами. В самом начале тензоры необходимо каким-либо образом создавать и инициализировать какими-либо значениями. Тензоры можно создавать напрямую из имеющихся структур данных. В этом случае тип данных определяется автоматически. Например, если у нас имеется вложенный список, содержащий четыре значения, то из него можно создать тензор вызовом метода **torch.tensor**(), например

data = [[1, 2], [3, 4]]

data_t = torch.tensor (data).

Таrже тензоры могут быть созданы из массивов NumPy:

array = np.array (data)

x_t = torch.from_numpy(array).

В этом примере исходный список предварительно был преобразован в numpy массив, и потом из него уже создан тензор вызовом метода torch.from_numpy

Тензора можно создавать для различных типов данных: **HalfTensor – float16, Short-Tensor – int16, CharTensor – int8, FloatTensor – float32, IntTensor - int32, ByteTensor – uint8, DoubleTensor – float64, LongTensor – int64, BoolTensor – bool**

Размеры тензоров и атрибуты можно вывести на консоль, используя свойства shape, dtype, device. Приведем пример вывода информации о тензоре с использованием метода print:

print(f"Shape of tensor: {tensor.shape}")

print(f"Datatype of tensor: {tensor.dtype}")

print(f"Device tensor is stored on: {tensor.device}")

При создании тензора определенного размера его можно инициализировать случайными значениями. Например, для инициализации тензора случайными значениями из равномерного распределения в интервале [от 0 до 1] необходимо использовать метод **torch.rand()**, с указанием размера необходимого тензора: **w = torch.rand (4,5).**

Инициализация тензора случайными значениями из стандартного нормального распределения с нулевой средней и единичной дисперсией осуществляется методом **torch.rand**, например: **w = torch.randn (4,5)**

Если необходимо создать тензоры, состоящие из нулевых значений и единичек, то необходимо использовать методы **torch.zeros** и **torch.ones**, например

w = torch.zeros(3,3,4)

w = torch.ones(2,5,7)

Диагональный тензор создается использованием метода **torch.diag**, например **x = torch.diag(torch.ones(3)).**

Рассмотрим некоторые наиболее важные операции с тензорами. Необходимо отметить, что большинство операций с тензорами очень похожи на аналогичные операции, имеющиеся в библиотеке **numpy**. Поэтому если вы хорошо знакомы с **numpy**, то вам не составит особого труда понять, как выполняются различные операции с тензорами в PyTorch.

Сначала рассмотрим простые операции поэлементного сложения, вычитания, умножения тензоров.

Например, если у нас имеются 2 тензора x1 и x2:

x1= torch.ones(2,3)

x2= torch.ones(2,3),

то сложение этих тензоров осуществляется с помощью оператора плюс **y = x1 + x2,** вычитание оператором минус **y = x1 - x2**, умножение оператором звездочка **y = x1*2,** деление оператором слеш. Так как это поэлементные операции, то размеры тензоров должны быть одинаковыми.

При помощи индексов можно делать выборки элементов. Приведем пример того, как можно выбрать определенную строку, столбец или же какой-либо элемент тензора.

y = x [:, 2]

z = x [3,:]

t = x [2, 4]

Можно осуществлять выбор элементов тензора по какому-либо условию следующим образом **b = a [a > 0.5]** . Здесь приведен пример, в котором мы получим тензор, содержащий значения элементов больше 0.5 из исходного тензора.

Конкатенацию тензоров можно делать вдоль определенной оси, используя метод **torch.cat**, например **y = torch.cat([x1, x2], dim = 0)**. Аргумент **dim** этого метода задает ось, вдоль которой будет производиться конкатенацию. Также имеется метод **stack**, который позволяет делать аналогичные операции, **y = torch.stack([x1, x2])** .

Транспонирование – это обычная операция при работе с матрицами и тензорами. В Pytorch сделать это можно двумя способами, через свойство, обозначенное заглавной буквой T, или же методом t(), например **y = tens1.T** или **y = tens1.t()** .

Функция **view()** позволяет изменять форму тензора. Новые размеры должны быть совместимы с оригиналом. Например, если имеется тензор 3x3, то возможны только изменения размеров этого тензора в 9x1 и 1x9, например **y = tens.view(9, 1).**

В PyTorch имеется большой набор встроенных математических функций для обработки тензоров. Рассмотрим примеры применения некоторых их них. Функция mean вычисляет среднее значение в тензоре (например **y = torch.mean(a)),** min – вычисляет минимальное значение в тензоре (например **y = torch.min(a)),** max – вычисляет максимальное значение в тензоре (например **y = torch.max(a)).** Функции argmin, argmax возвращают индексы максимального или минимального значения в тензоре, например **y = torch.argmax(a).**

Необходимо учесть, что большинство функции принимает только типы с плавающей запятой, поэтому мы должны сначала преобразовать в тип float (**flt_tens = tens.to(torch.float32)**).

При обучении больших и тяжелых моделей в PyTorch требуется использовать графические процессоры. Графический процессор ускоряет процесс обучения во много раз по сравне-

нию с CPU. Сначала важно убедиться, что во время обучения используется графический процессор. Модуль **torch.cuda** используется для настройки и запуска операций на CUDA. Он отслеживает текущий выбранный графический процессор, и все новые тензоры CUDA будут по умолчанию созданы на этом устройстве. Проверить доступность CUDA устройства можно с помощью метода **torch.cuda.is_available()**. Выбранное устройство можно изменить с помощью диспетчера контекста **torch.cuda.device.**

На рисунке 3 показан пример программного кода с использованием CUDA. Программный код примера расположен по адресу:
https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/cuda_example.ipynb).

Сначала мы создаем тензор x на CPU. Потом проверяем доступность устройства CUDA, и если он доступен, то создаем ссылку на объект устройства CUDA. После этого перемещаем тензор **x** на устройство CUDA. Второй тензор **y** изначально создаем на устройстве CUDA, указав в параметре **device** конструктора, создающего новый тензор, ссылку на это устройство. Далее выполняем сложение двух тензоров, эта операция будет проводиться уже на графическом CUDA устройстве. Вывод значений тензоров на консоль показывает, что они находятся на этом устройстве (см. рис. 4).

```
import torch
#По умолчанию тензор создается в CPU
x=torch.ones(3,4)
print('x=',x)
#Если доступно устройство CUDA
if torch.cuda.is_available():
    #то перемещаем его на устройство “cuda”
    device = torch.device("cuda")
    x = x.to(device)
    # создаем тензор y на устройстве cuda
    y = torch.ones_like(x, device=device)
    # выполняем сложение тонзоров x и y
    z = x + y
    #вывод информации о тензорах
    print('x=',x)
    print('y=',y)
    print('z=',z)
```

Рис. 3. Программа с использованием CUDA

```
x= tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])
x= tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]], device='cuda:0')
y= tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]], device='cuda:0')
z= tensor([[2., 2., 2., 2.],
        [2., 2., 2., 2.],
        [2., 2., 2., 2.]], device='cuda:0')
```

Рис. 4. Результат выполнения программы

Рассмотрим, насколько эффективно PyTorch выполняет операции с тензорами, а также эффективность использования CUDA устройств. Для этого мы вычислим матричное произведение двух тензоров различными способами (см. рис. 5). В первом способе для этого используется программа, содержащая два вложенных цикла, первый цикл по строкам, второй по столбцам. Внутри цикла выбираются столбцы и строки тензоров *x* и *y* и перемножаются.

```
import torch
import time
a=1000
x= torch.ones(a,a)
y= torch.ones(a,a)
res=torch.zeros(a,a)
t1 = time.perf_counter()
for i in range (a):
  for j in range(a):
    row=x[i,:]
    col=y[:,j]
    res[i,j]=torch.sum(row*col)
t2 = time.perf_counter()
print('время вычисления:',t2-t1)
```

```
время вычисления: 51.803100266
```

Рис. 5. Вычисление матричного произведения без использования встроенной функции и CUDA

Время выполнения этой программы составило почти 52 секунды. Естественно, этот способ неэффективный. В библиотеке PyTorch для этого случая имеется функция **torch.mm**. Эта функция производит матричное произведение двух тензоров. В этом случае время, затрачиваемое на те же самые вычисления, уже составляет 37 сотых секунды (см. рис. 6).

```
[4]  t1 = time.perf_counter()
     z=torch.mm(x,y)
     t2 = time.perf_counter()
     print('время вычисления:',t2-t1)
```

Рис. 6. Вычисление матричного произведения с использованием встроенной функции mm

Но наилучший по быстродействию результат получается, если вычисления проводить на CUDA устройстве. Это продемонстрировано на рисунке 8, время вычислений оказалось в 100 раз выше, чем вычисление с помощью встроенной функции для матричного умножения на обычном процессоре. Поэтому вывод такой: если есть возможность использовать GPU устройства, то вычисления необходимо проводить на них, особенно если операции выполняются с большими тензорами.

```
x_gpu=x.to('cuda')
y_gpu=y.to('cuda')
t1 = time.perf_counter()
torch.mm(x_gpu,y_gpu)
t2 = time.perf_counter()
print('время вычисления:',t2-t1)
```

```
время вычисления: 0.0003205380000963487
```

Рис. 7. Вычисление матричного произведения на устройстве CUDA

Вопросы

1. Перечислите типы данных, которые могут быть представлены с использованием тензоров.
2. Каким образом можно создать тензор с определенными размерами?
3. Каким образом можно узнать размер (shape) тензора?
4. Как можно изменить форму (reshape) тензора?
5. Перечислите основные операции для выполнения над тензорами.
6. Как создать тензор, заполненный нулями или единицами?
7. Как можно создать тензор, заполненный случайными значениями?
8. Как можно проверить, содержит ли тензор элементы с определенным значением?
9. Как можно выполнить арифметические операции с тензорами?
10. Как можно выполнить поэлементное умножение двух тензоров?

11. Как можно изменить тип данных (dtype) тензора?

12. Как можно сконвертировать тензор в массив NumPy и наоборот?

13. С помощью какой функции можно выполнить конкатенацию тензоров вдоль определенной оси?

14. Как умножить каждый элемент тензора на определенное число?

15. Как можно выполнить операции на тензорах на GPU?

16. Как можно создать тензор с линейной последовательностью чисел в PyTorch?

17. Как можно выполнить операции с тензорами, используя условные выражения?

18. Как можно выполнить операцию транспонирования для тензора?

19. С помощью какой встроенной функции Pytorch можно узнать максимальное и минимальное значение в тензоре?

20. Каким образом можно создать тензор с заданными значениями?

21. Как можно выполнить операции с тензорами на разных устройствах (например, CPU и GPU)?

22. С помощью каких функций можно выполнить операцию редукции (например, суммирование, усреднение) для тензора?

23. Как создать тензор, содержащий значения с определенным шагом (step) в заданном диапазоне?

24.Как можно выполнить операции с тензорами, используя математические функции (например, sin, cos)?

25. Как можно создать тензор, заполненный случайными значениями из определенного распределения?

26. Как можно сгенерировать тензор с определенным количеством элементов, равномерно распределенных в заданном диапазоне?

ГЛАВА 2. Вычислительные графы и автоматическое вычисление градиентов

В этой главе мы изучим вычислительные графы и основные особенности модуля **autograd** библиотеки PyTorch. В основном PyTorch используется для реализации нейронных сетей, а в них используются разнообразные и довольно сложные функции. Одна из основных причин использования PyTorch в проектах машинного обучения заключается в том, что с его помощью мы можем автоматически вычислять производные и градиенты любых функций, которые мы определяем.

Выполнения программ с использованием PyTorch немного отличается от базовой логики выполнения программ на Python, эта библиотека записывает выполнение работающей программы. PyTorch запоминает последовательность операций, которые совершаются над данными, т. е. создает динамический вычислительный граф. Такой граф нужен для выполнения операции, которая называется «алгоритм обратного распространения ошибки», поскольку этот алгоритм должен в обратном направлении проходить по цепочке выполненных операций.

Вычислительный граф – это запись последовательности вычислительных операций, состоящая из вершин и ребер. Вершины, иногда их еще называют узлами, – это вычислительные операции, которые необходимо выполнить, а рёбра связывают их в определённую последовательность. Вычислительные графы могут быть динамические или статические. Динамические вычислительные графы не требуют компиляции перед каждым его выполнением. В этом случае можно спокойно изменять входные данные в процессе работы. Статические вычислительные графы, с другой стороны, требуют перекомпиляции при изменении входных параметров. Они позволяют получить высокую производительность, однако закрыты для изменений. В PyTorch используются динамические вычислительные графы.

На рисунке 8 приведен простой пример последовательности вычислений и визуализирован соответствующий вычислительный граф. Сначала вычисляется узел b как произведение w1 на a (**b = w1 * a**), и узел c как произведение w2 на a (**c = w2 * a**). Далее вычисляется узел d как сумма двух произведений w3 на b и w4 на c (**d = (w3 * b) + (w4 * c)**). И в самом конце на основе d вычисляется некоторая функция, а результат получается в узле L. Диаграмма (рис. 9) наглядно показывает весь вычисленный граф для этой последовательности вычислений. Кружочками обозначены узлы вычислительного графа, а стрелками ребра, которые связывают вычисления в определенную последовательность.

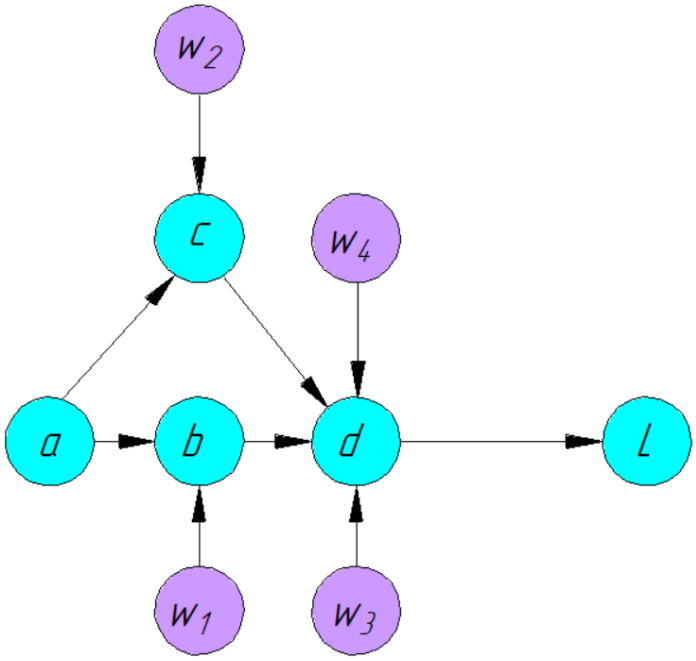


Рис. 8. Визуализация вычислительного графа

В основе большинства современных методов машинного обучения лежит расчет градиентов и производных. Это в особенности касается нейронных сетей, в которых для настройки весовых коэффициентов используется алгоритм обратного распространения ошибки. Именно поэтому в PyTorch есть сильная нативная поддержка вычисления производных функций, которая называется автоматическим дифференцированием. Автоматическое дифференцирование позволяет эффективно проводить операцию обратного распространение ошибки в нейронных сетях. Этот метод рекурсивно использует правило дифференцирования сложной функции для вычисления градиента каждой переменной в вычислительном графе. В PyTorch инструментом для автоматического вычисления производных является модуль Autograd. Этот модуль организует запись последовательности всех операций, выполняемых над тензорами с включенным градиентом в динамическом вычислительный графе.

Базовый процесс построения модели обучения нейронных сетей заключается в построении графа вычислений, вычислении функции потерь, а затем вычислении производной функции потерь по параметрам модели, с использованием таких методов, как градиентный спуск для обновления параметров.

Рассмотрим использование autograd на конкретных примерах. Программный код примеров расположен по адресу:
https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/autograd.ipynb.

По замыслу, в PyTorch градиенты могут быть рассчитаны только для тензоров с плавающей запятой, поэтому создадим массив чисел с плавающей запятой перед тем, как преобразовать его в тензор PyTorch с поддержкой автоматического вычисления градиентов. Для того,

чтобы отслеживать все операции с тензором, необходимо установить свойство **requires_grad** тензора в True. Это можно сделать прямо в конструкторе при создании тензора.

```
import torch
x = torch.randn(4, requires_grad=True)
print('x=',x)
y=torch.randn(4)
print('y=',y)
z = x + 2
print('z=',z)
t=y+2
print('t=',t)
print()

x1 = torch.arange(6.0)
print('x1=',x1)
x1.requires_grad_(True)
print('x1=',x1)
```

Рис. 9. Программа, демонстрирующая использование параметра requires_grad

В этом примере (рис. 9) мы создаем тензор *x*, состоящий из четырех случайных чисел. После вывода на экран информации об этом тензоре помощью функции **print** можно заметить, что, кроме чисел, хранящихся в этом тензоре, выводится также и свойство **requires_grad = True**. Это свойство означает то, что будет храниться история всех операций, в которых будет участвовать этот тензор. Если же мы не указываем свойство **requires_grad** в конструкторе, то будет создан обычный тензор, в котором не будет отслеживаться история операции. Это показано на примере создания тензора y. Вычислим вектор z сложением вектора x и скаляра 2, и выведем значения z на консоль.

```
x= tensor([ 1.8019,  0.5167, -0.6669,  1.2953], requires_grad=True)
y= tensor([ 1.5443,  0.5302, -1.2922, -0.6731])
z= tensor([3.8019, 2.5167, 1.3331, 3.2953], grad_fn=<AddBackward0>)
t= tensor([3.5443, 2.5302, 0.7078, 1.3269])

x1= tensor([0., 1., 2., 3., 4., 5.])
x1= tensor([0., 1., 2., 3., 4., 5.], requires_grad=True)
```

Рис. 10. Результат выполнения программы

У нового тензора z появилось свойство **grad_fn**, в котором записано значение AddBackward0 (рис. 10). Это значит, что переменная z хранит историю операции. Можно также попробовать провести аналогичные вычисления с тензором y, но в этом случае результирующий тензор t не будет содержать свойства **grad_fn**, т. к. в конструкторе тензора y мы не

указали необходимость отслеживания истории вычислений. Есть и другой способ для указания того, что необходимо отслеживать все операции с тензором. Для этого необходимо вызвать метод **requires_grad_** тензора и в качестве аргумента указать True.

В свойстве **grad** тензоров хранится значение градиентов. Для вычисления градиента необходимо вызвать метод **backward()** у результирующего тензора. В следующем примере (рис. 11) мы создаем тензор x и устанавливаем его свойство **requires_grad** в True.

```
import torch
x = torch.arange(6.0)
x.requires_grad_(True)
print('x=',x)
print('x.grad=',x.grad)
y = 2 * torch.dot(x, x)
print('y=',y)
y.backward()
print('x.grad=',x.grad)
print(x.grad == 4 * x)

print('x.grad_fn=',x.grad_fn)
print('y.grad_fn=',y.grad_fn)
```

Рис. 11. Программа, демонстрирующая вычисление градиентов

Если мы выведем на консоль значения свойства **grad** сразу после создания тензора, то есть до проведения с ним каких-либо вычислений, то получим None (рис. 12).

```
x= tensor([0., 1., 2., 3., 4., 5.], requires_grad=True)
x.grad= None
y= tensor(110., grad_fn=<MulBackward0>)
x.grad= tensor([ 0.,  4.,  8., 12., 16., 20.])
tensor([True, True, True, True, True, True])
x.grad_fn= None
y.grad_fn= <MulBackward0 object at 0x7fd46e5dfdd0>
```

Рис. 12. Результат выполнения программы

Это означает, что пока значения градиентов по этому тензору не вычислены. Выполним следующую последовательность операций с этим тензором: вычислим скалярное произведение двух тензоров **x** методом **torch.dot**, и умножим результат на 2. В результате получаем тензор, содержащий только одно значение, равное 110, со свойством **grad_fn** равным **MulBackward**0. Далее для вычисления градиентов вызовем метод **backward()** переменной **y**. Градиенты вычисляются на основе правила вычисления производной сложной функции. Так как исходная функция у нас была $\mathbf{2x^2}$, то производная должна быть разна **4x**. Далее мы выводим значения вычисленных производных и видим, что производные вычислены правильно. Тензор **y** был создан в результате операции, поэтому у него есть атрибут **grad_fn**, который

ссылается на функцию, создавшую этот тензор. Для тензора **x**, созданного пользователем, свойство **grad_fn** равно **None**, т. к. он не является результатом каких-либо вычислений (рис. 12).

Необходимо отметить, что в многомерном случае модуль **autograd** вычисляет произведение Якобиана на какой-либо вектор. Отдельные частные производные вычисляются на основе правил вычисления производной сложной функции. Если у нас имеется модель с не скалярным выходом, то необходимо явно указать аргумент для метода **backward()**, который должен быть тензором соответствующей формы. Рассмотрим это на примере (рис. 13). Создадим тензор, содержащий 5 чисел от нуля до четырех, с включенным режимом отслеживания операций, умножим его на 8 и прибавим 15, результат запишем в тензор **y**. В этом случае размерность результирующего тензора будет уже не единица, а 5, поэтому мы должны подготовить вектор **v**, который мы передадим как аргумент метода **backward()**. В качестве такого вектора мы использовали тензор, состоящий из единичек, размерность которого равна размерности тензоров **x** и **y**. Это вектор может содержать и другие значения в зависимости от задачи.

```
import torch
x= torch.arange(5.0, requires_grad=True)
print('x=',x)
y = x * 8+15
print('y=',y)
print('y.shape=',y.shape)
v = torch.tensor([1.0, 1.0, 1.0,1.0,1.0], dtype=torch.float32)
y.backward(v)
print('x.grad=',x.grad)
```

```
x= tensor([0., 1., 2., 3., 4.], requires_grad=True)
y= tensor([15., 23., 31., 39., 47.], grad_fn=<AddBackward0>)
y.shape= torch.Size([5])
x.grad= tensor([8., 8., 8., 8., 8.])
```

Рис. 13. Программа вычисления градиентов для не скалярного результирующего тензора

Часто бывает необходимо отключить режим отслеживания операций с тензорами, которые были изначально созданы со свойством **requires_grad=True**. Например, такое бывает необходимо во время цикла обучения различных моделей машинного обучения, когда мы хотим обновить наши веса, а операция обновления весов не должна быть частью вычисления градиента. Вычисление градиентов и отслеживание операций требует много вычислительных ресурсов, и если нет обязательной необходимости это делать, то желательно отключить этот процесс.

```
print('.requires_grad_(False)')
a = torch.randn(2, 2,requires_grad=True)
print('a=',a)
a.requires_grad_(False)
print('a=',a)

print()
print(' .detach():')
a = torch.randn(3, 2, requires_grad=True)
print('a=',a)
b = a.detach()
print('b=',b)

print()
print('with torch.no_grad():')
a = torch.randn(5, 5, requires_grad=True)
print(a.requires_grad)
with torch.no_grad():
    print((x ** 2).requires_grad)
print(a.requires_grad)
```

Рис. 14. Программа, демонстрирующая отключение отслеживания операций с тензорами

Для этого есть несколько способов (см. рисунки 14 и 15):

1. Вызов метода тензора **requires_grad_** с аргументом **False**.
2. Использование метода **detach()**.
3. Обертывание операций выполняемых с тензорам в конструкцию **'with torch.no_grad():'**.

```
.requires_grad_(False)
a= tensor([[-0.1619,  0.2761],
        [ 0.1734, -1.1662]], requires_grad=True)
a= tensor([[-0.1619,  0.2761],
        [ 0.1734, -1.1662]])

 .detach():
a= tensor([[ 1.0805, -0.2388],
        [-0.5859,  0.2217],
        [-0.1312,  0.4508]], requires_grad=True)
b= tensor([[ 1.0805, -0.2388],
        [-0.5859,  0.2217],
        [-0.1312,  0.4508]])

with torch.no_grad():
True
False
True
```

Рис. 15. Результат выполнения программы

Необходимо учитывать, что повторные вызовы метода **backward()** приводят к накоплению градиента для этого тензора в атрибуте **grad**. При решении задач оптимизации, например, в обучении нейронных сетей, это может приводить к проблемам. Поэтому необходимо использовать **grad.zero()**, чтобы очистить градиенты перед новым шагом оптимизации. На рисунке 16 показан простой пример с демонстрацией очистки градиентов после завершения итерации обучения.

```
weights = torch.ones(4, requires_grad=True)
for epoch in range(3):
    model_output = (weights*3).sum()
    model_output.backward()
    print(weights.grad)
    with torch.no_grad():
        weights -= 0.1 * weights.grad
    weights.grad.zero_()
print(weights)
print(model_output)
```

```
tensor([3., 3., 3., 3.])
tensor([3., 3., 3., 3.])
tensor([3., 3., 3., 3.])
tensor([0.1000, 0.1000, 0.1000, 0.1000], requires_grad=True)
tensor(4.8000, grad_fn=<SumBackward0>)
```

Рис. 16. Программа, демонстрирующая очистку вычисленных градиентов

Рассмотрим применение модуля **autograd** для решения простой задачи линейной регрессии. Для демонстрации преимуществ использования autograd мы сначала разработаем программу, которая будет построена полностью только на основе массивов **numpy**, без использования PyToch и **autograd** (рис. 17). Программный код этого примера расположен по адресу: https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/autograd.ipynb. Пусть линейная зависимость переменной Y от X задается простой функцией **f = w*x**, то есть значение функции равно произведению аргумента на некоторый вес. Подготовим данные в виде двух numpy массивов, **X_data** со входными значениями [1,2,3,4] и **Y_data** с целевыми значениями [3, 6, 9, 12]. По этим значениям понятно, что вес W здесь должен быть равен 3. В самом начале, до обучения, мы инициализируем этот вес нулем. Модель описываем в виде функции, которая возвращает произведение W на X. В качестве функционала ошибок выберем среднеквадратичную ошибку и опишем ее тоже в виде отдельной функции, в которую будем передавать значения, вычисленные моделью, и целевые значения. Эта функция вычисляет поэлементную разность двух массивов, возводит их в квадрат и возвращает среднее значение. Далее, для использования метода градиентного спуска мы должны реализовать вычисление производной этой

функции по переменной w. Для этого мы в эту программу добавили функцию **calculate_gradient**, которая реализует эту формулу.

```
import numpy as np
# f = w * x
# f = 3 * x
X_data = np.array([1, 2, 3, 4], dtype=np.float32)
Y_data = np.array([3, 6, 9, 12], dtype=np.float32)
w = 0.0
# модель
def calculate_output(x):
    return w * x
# ошибка
#MSE = 1/N * (w*x - y)**2
def calculate_loss(y, y_pred):
    return ((y_pred - y)**2).mean()
#градиент
# dJ/dw = 1/N * 2x(w*x - y)
def calculate_gradient(x, y, y_pred):
    return np.dot(2*x, y_pred - y).mean()
```

Рис. 17. Программа линейной регрессии без использования возможностей autograd

Перед началом обучения выведем выходной результат модели для значения аргумента, равного 5 (рис. 18). Мы задаем переменную **learning_rate**, которая будет обозначать скорость обучения, и количество итераций градиентного спуска **n_iter**. Далее в цикле реализуем алгоритм обучения нашей модели методом градиентного спуска. Сначала вычисляем прогнозируемое значение **y_pred**, затем вычисляем среднеквадратичную ошибку и далее вычисляем значение производной.

```
print(f'Прогноз перед обучением: f(5) = {calculate_output(5):.3f}')
# Обучение модели
learning_rate = 0.02
n_iters = 15
for epoch in range(n_iters):
    # вычисление прогнозируемого значения
    y_pred = calculate_output(X_data)
    # вычисление ошибки
    loss = calculate_loss(Y_data, y_pred)
    #  вычисляем градиенты
    dw = calculate_gradient(X_data, Y_data, y_pred)
    # обновляем значение параметра w
    w -= learning_rate * dw
    if epoch % 2 == 0:
        print(f'Эпоха {epoch+1}: w = {w:.3f}, ошибка = {loss:.8f}')

print(f'Прогноз после обучения: f(5) = {calculate_output(5):.3f}')
```

Рис. 18. Программа линейной регрессии без autograd

Зная значение производной от ошибки по весовому коэффициенту w, выполняем шаг обучения, а на каждом втором шаге печатаем на консоли номер эпохи, значение весового коэффициента w и величину ошибки. Видно, что уже на девятой итерации ошибка стала равна нулю (рис. 19).

```
Прогноз перед обучением: f(5) = 0.000
Эпоха 1: w = 3.600, ошибка = 67.50000000
Эпоха 3: w = 3.024, ошибка = 0.10799979
Эпоха 5: w = 3.001, ошибка = 0.00017279
Эпоха 7: w = 3.000, ошибка = 0.00000028
Эпоха 9: w = 3.000, ошибка = 0.00000000
Эпоха 11: w = 3.000, ошибка = 0.00000000
Эпоха 13: w = 3.000, ошибка = 0.00000000
Эпоха 15: w = 3.000, ошибка = 0.00000000
Прогноз после обучения: f(5) = 15.000
```

Рис. 19. Результат выполнения программы

На следующем примере рассмотрим реализацию этого процесса на основе autograd (рис. 20). Программный код этого примера расположен по адресу: https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/lin_regr_autograd.ipynb.

В PyTorch данные необходимо задать в виде тензоров, весовой коэффициент W также должен быть тензором, с указанием свойства **requires_grad True** в конструкторе. В отличие от предыдущего примера, нам нет необходимости создавать отдельную функцию для вычисления в производных и вообще их как-то вычислять, потому что это будет выполняться автоматически с помощью **autograd**.

```python
import torch
# f = w * x
# f = 3 * x
X_data = torch.tensor([1, 2, 3, 4], dtype=torch.float32)
Y_data = torch.tensor([3, 6, 9, 12], dtype=torch.float32)
w = torch.tensor(0.0, dtype=torch.float32, requires_grad=True)
# модель
def calculate_output(x):
    return w * x
# ошибка
#MSE = 1/N * (w*x - y)**2
def calculate_loss(y, y_pred):
    return ((y_pred - y)**2).mean()
```

Рис. 20. Программа линейной регрессии с autograd

Далее мы осуществим обучение нашей модели, также с использование метода градиентного спуска (рис. 21). В этом случае производная будет вычисляться при вызове метода **loss.backward()**. Модуль autograd библиотеки PyTorch отслеживал все операции, в которых

принимал участие тензор w, и строил вычислительный граф. В результате вызова метода **backward()** в свойстве **grad** тензора w появилось значение производной. Если проанализировать процесс обучения модели, то можно заметить, что в этом случае модель обучается намного медленнее, ошибка становится равной нулю только на 36-м шаге обучения. Это связано с тем, что производная в этом случае не считается по конкретной формуле, а вычисляется численно (рис. 22).

```
print(f'Прогноз перед обучением: f(5) = {calculate_output(5).item():.3f}')
# Обучение модели
learning_rate = 0.02
n_iters = 50
for epoch in range(n_iters):
    # вычисление прогнозируемого значения
    y_pred = calculate_output(X_data)
    # вычисление ошибки
    loss = calculate_loss(Y_data, y_pred)
    #  вычисляем градиенты
    loss.backward()
    # обновляем значение параметра w
    with torch.no_grad():
        w -= learning_rate * w.grad
    # обнуляем градиенты
    w.grad.zero_()
    if epoch % 5 == 0:
        print(f'Эпоха {epoch+1}: w = {w.item():.3f}, ошибка = {loss.item():.8f}')

print(f'Прогноз после обучения: f(5) = {calculate_output(5).item():.3f}')
```

Рис. 21. Программа линейной регрессии с autograd

```
Прогноз перед обучением: f(5) = 0.000
Эпоха 1: w = 0.900, ошибка = 67.50000000
Эпоха 6: w = 2.647, ошибка = 1.90670800
Эпоха 11: w = 2.941, ошибка = 0.05385974
Эпоха 16: w = 2.990, ошибка = 0.00152141
Эпоха 21: w = 2.998, ошибка = 0.00004297
Эпоха 26: w = 3.000, ошибка = 0.00000121
Эпоха 31: w = 3.000, ошибка = 0.00000003
Эпоха 36: w = 3.000, ошибка = 0.00000000
Эпоха 41: w = 3.000, ошибка = 0.00000000
Эпоха 46: w = 3.000, ошибка = 0.00000000
Прогноз после обучения: f(5) = 15.000
```

Рис. 22. Результат выполнения программы

Вопросы

1. Что означает термин «вычислительный граф» в контексте обучения нейронных сетей?
2. Из каких основных компонент состоит вычислительный граф?
3. Какова роль узлов и ребер в вычислительном графе?

4. Как можно создать вычислительный граф в PyTorch?

5. В чем заключается разница между статическим и динамическим вычислительным графом?

6. Как работает механизм вычисления градиентов в PyTorch?

7. Как можно включить и выключить вычисление градиентов для тензора?

8. Как определить, является ли тензор листом (leaf) в вычислительном графе?

9. Как можно использовать метод **backward()** для выполнения операции обратного распространения ошибки?

10. Как отключить градиенты для определенной части вычислительного графа?

11. Как узнать, вычислены ли градиенты для определенного тензора в PyTorch?

12. Как использовать градиенты для обновления параметров модели в процессе обучения?

13. Как использовать контекстный менеджер **torch.no_grad()** для управления вычислением градиентов?

14. Как обрабатываются вычислительные графы в PyTorch при использовании функции **torch.autograd.grad()**?

15. Как использовать функцию **torch.autograd.grad()** для расчета градиентов по нескольким переменным одновременно?

16. Как предотвратить обновление градиентов для определенных параметров в PyTorch?

17. Как можно использовать градиенты для оптимизации параметров модели с использованием оптимизаторов в PyTorch?

18. Как проверить, вычислены ли градиенты для всех переменных в вычислительном графе?

19. Как отключить вычисление градиентов для всех параметров модели в PyTorch?

20. Как можно использовать вычисленные градиенты для выполнения градиентного шага без использования оптимизатора?

ГЛАВА 3. Однослойные и многослойные нейронные сети в PyTorch

Искусственные нейронные сети состоят из узлов, которые называются нейронами. На рисунке 23 изображена схема формального нейрона. У нейрона может быть много входов, но только один выход. Через входы на нейрон поступают входные данные (признаки), а на выходе формируется результат работы. Нейрон вычисляет взвешенную сумму входных сигналов, а затем преобразует полученную сумму с помощью заданной нелинейной функции, которая называется функцией активации нейрона. Множество, состоящее из порогового уровня и всех весов, называют параметрами нейрона. Количество весовых коэффициентов нейрона равно количеству входов нейрона. Данные, подаваемые на вход нейрона, обычно называют входным паттерном, он обычно бывает в виде массива (или тензора) чисел.

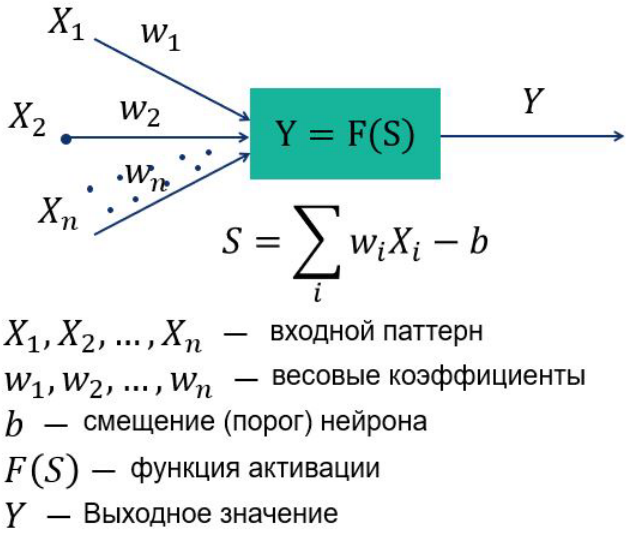


Рис. 23. Формальный нейрон

Функция активации определяет выходное значение нейрона в зависимости от результата взвешенной суммы входов и порогового значения. На рис. 24 в качестве примеров приведены графики классических функции активации. Пороговая функция активации является одной из наиболее простых функций активации. Результат этой функции равен единице, если значение аргумента выше определенного значения, которое называется пороговым, иначе результат функции равен нулю. В этом случае нейрон может выдавать на выходе только два значения: 0 и 1. Линейная функция активация представляет собой прямую. Такой выбор активационной функции позволяет получать спектр значений, а не только бинарный ответ.

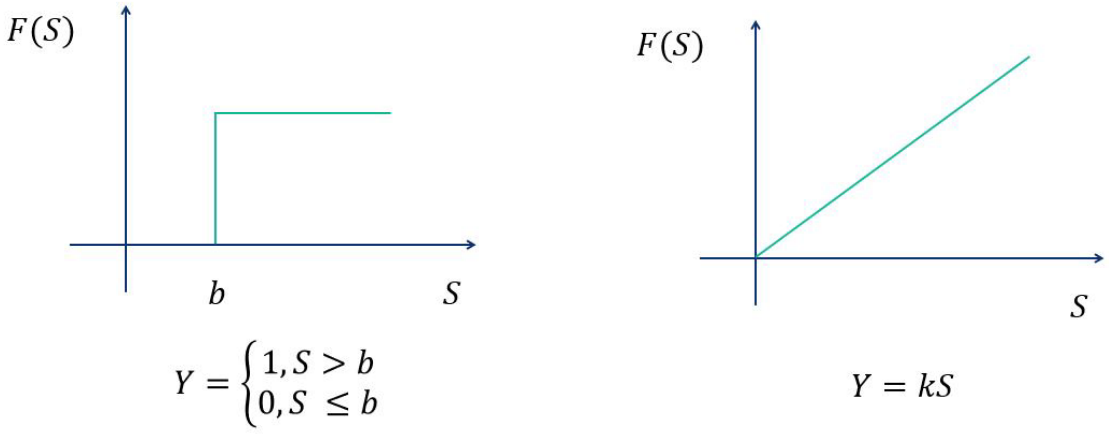


Рис. 24. Пороговая и линейная функции активации

В настоящее время в библиотеке PyTorch реализованы десятки различных функций активации. Однако на практике чаще всего используются только несколько из них. На рисунке 25 приведены графики наиболее часто используемых функций активации, перечисленных ниже.

1. **Rectified Linear Unit (ReLu**). Эта функция возвращает значение аргумента, если его значение положительно, и 0 – в противном случае.

2. **Сигмоида (sigmoid).** Сигмоида выглядит гладкой и подобна ступенчатой функции. Ее преимуществами является то, что сигмоида – это нелинейная функция по своей природе, а комбинация таких функций производит тоже нелинейную функцию. Еще одно преимущество этой функции, это то, что она не бинарна, это делает активацию аналоговой, в отличие от ступенчатой функции.

3. **Гиперболический тангенс (tanh).** Гиперболический тангенс очень похож на сигмоиду. Эта функция тоже нелинейная, она хорошо подходит для комбинации слоев, а диапазон значений функции -(-1, 1).

4. **Softplus**. Эта функция представляет собой плавное приближение к функции активации ReLU поэтому иногда используется в нейронных сетях вместо ReLU.

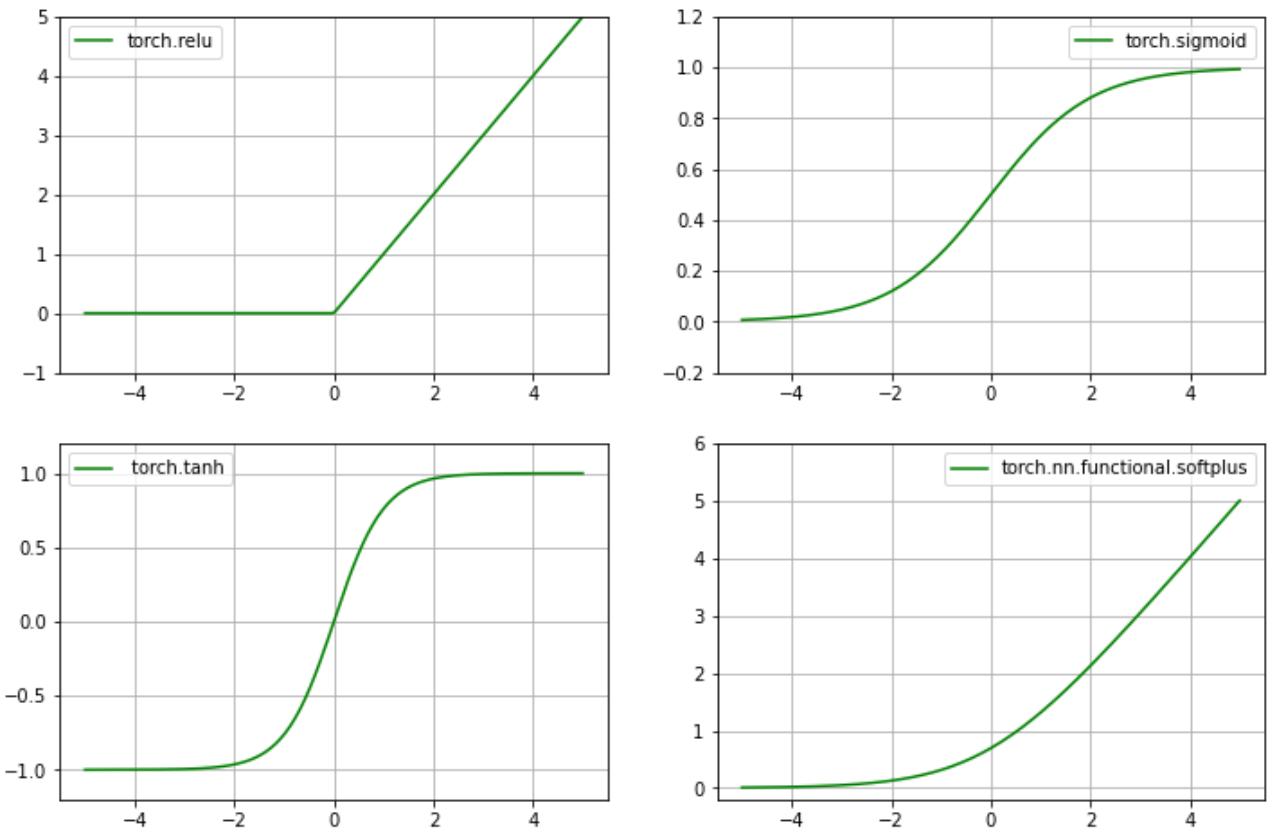


Рис. 25. Графики функций активации (ReLu, sigmoid, tanh, softplus)

На рис. 26 приведен программный код, демонстрирующий вычисление этих функции методами библиотеки PyTorch (программный код расположен по адресу: https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/atcivation_functions.ipynb).

```
x = torch.linspace(-5, 5, 200)
x = Variable(x)
x_np = x.data.numpy()

y_relu = torch.relu(x).data.numpy()
y_sigmoid = torch.sigmoid(x).data.numpy()
y_tanh = torch.tanh(x).data.numpy()
y_softplus = F.softplus(x).data.numpy()
```

Рис. 26. Вычисление значений функций активации: ReLu, sigmoid, tanh, softplus

Один нейрон может выполнять простейшие вычисления, но основные функции нейросети обеспечиваются не отдельными нейронами, а сетями нейронов. Наиболее простой нейронной сетью является однослойный перцептрон, который представляет собой простейшую нейронную сеть, состоящую из группы нейронов, образующих слой (рис. 27). Входные данные нейронной сети кодируются вектором значений, каждый элемент подается на соответ-

ствующий вход каждого нейрона в слое. В свою очередь, нейроны вычисляют выход независимо друг от друга. В однослойных нейронных сетях размерность выхода (то есть количество выходных элементов) равна количеству нейронов, а количество связей у всех нейронов должно быть одинаково и совпадать с размерностью входного сигнала. Здесь X_1, X_2, X_3 – это элементы входного паттерна, а Y_1, Y_2, Y_3 – элементы выходного паттерна, а $w\_(i,j)$- это j-ый весовой коэффициент i-го нейрона (см. рис. 28).

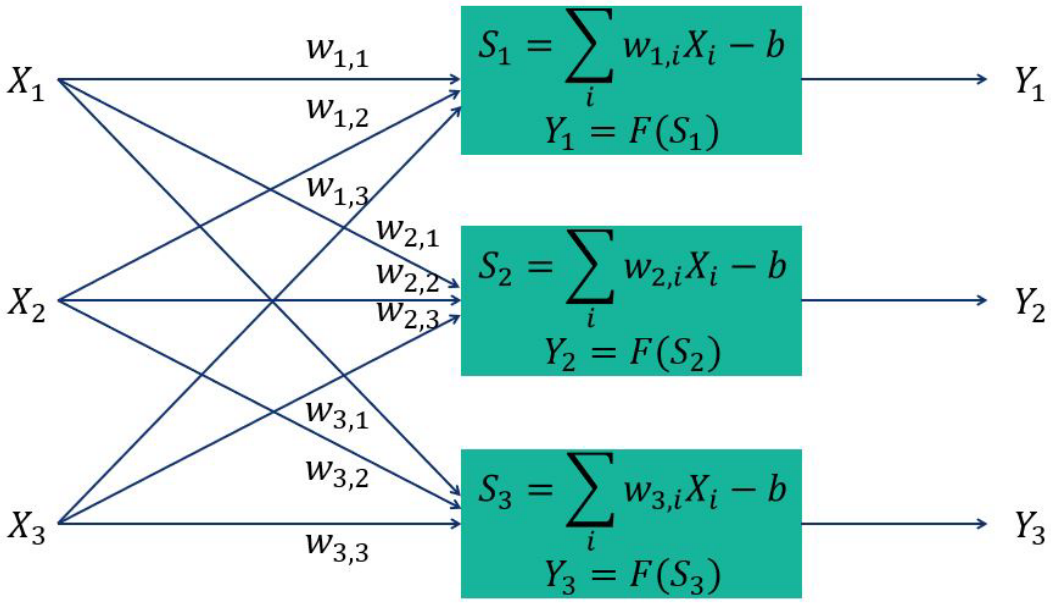


Рис. 27. Однослойная нейронная сеть

Одним из наиболее простых методов обучения нейронных сетей является метод градиентного спуска в пространстве весовых коэффициентов. Градиентный спуск – это итеративный метод нахождения локального экстремума (минимума или максимума) функции с помощью движения вдоль градиента. На рис. 29 приведены формулы и схематическое изображение метода градиентного спуска, показывающие изменение значений весовых коэффициентов и пороговых значений на каждом шаге итерации обучения. Здесь E – это функционал ошибок, который также еще называется функцией потерь, а alpha – это шаг обучения. При использовании этого метода сначала вычисляются градиенты от функции потерь по параметрам модели. Потом параметры модели уменьшаются на величину соответствующих вычисленных градиентов, умноженных на величину шага обучения.

$$w_{i,j}(t+1) = w_{i,j}(t) - \alpha \frac{\partial E}{\partial w_{i,j}}$$

$$b_i(t+1) = b_i(t) - \alpha \frac{\partial E}{\partial b_i}$$

E - функционал ошибки

a - скорость обучения

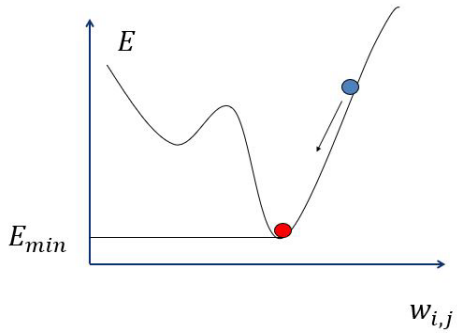
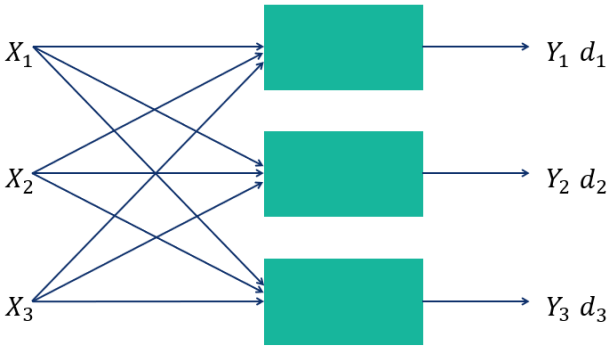


Рис. 28. Метод градиентного спуска в пространстве весовых коэффициентов

Рассмотрим процесс обучения простой однослойной нейронной сети. На рис. 29 приведены формулы, которые описывают изменение параметров однослойной нейронной сети в результате выполнения одного шага обучения на основе алгоритма Уидроу – Хоффа. Это наиболее простой метод обучения, он годится только для однослойной нейронной сети. Здесь векторы X и D образуют обучающую пару, то есть X – это вектор входных значении, а вектор D – это целевые значения.



$$E = \frac{1}{2}\sum_j (Y_j - d_j)^2$$

$$w_{ij}(t+1) = w_{ij}(t) - \alpha(Y_j - d_j)X_i$$

$$b_j(t+1) = b_j(t) - \alpha(Y_j - d_j)$$

$\{X_1, X_2, X_3\}$ и $\{d_1, d_2, d_3\}$ - обучающая пара

Рис. 29. Обучение однослойной нейронной сети (алгоритм Уидроу – Хоффа)

Многослойная нейронная сеть – это нейронная сеть, состоящая из более чем одного слоя нейронов (см. рис. 30). В таких нейронных сетях обработка информации, поступающей на вход, осуществляется последовательно каждым слоем. Выходные данные первого слоя передаются на вход второго слоя, и таким образом информация движется по остальным слоям. Выходом нейронной сети будут являться выходные значения последнего слоя. Многослойные нейронные сети намного сложнее в обучении чем однослойные, для их обучения необходимо

использовать метод обратного распространения ошибки. Библиотека PyTorch дает нам возможность обучать любые нейронных сети благодаря инструменту autograd, который был подробно рассмотрен нами в предыдущей разделе.

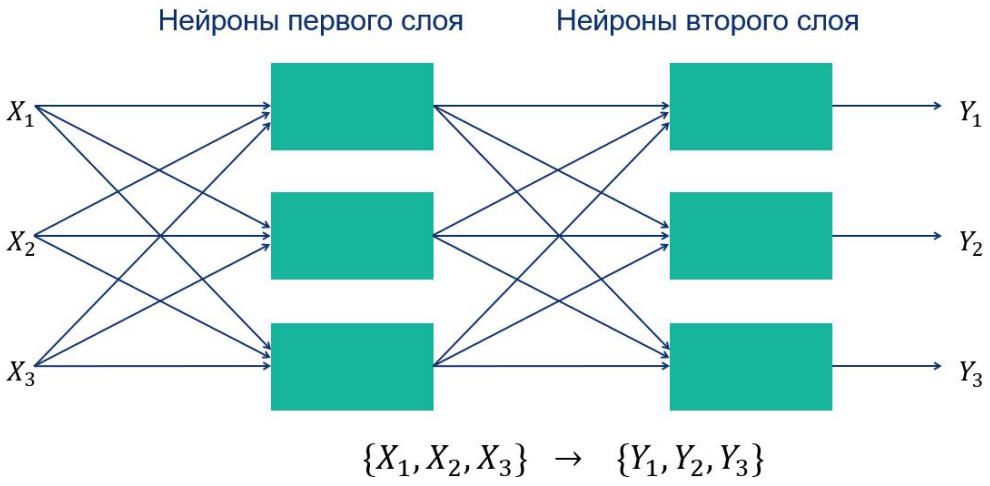


Рис. 30. Многослойные нейронные сети

Рассмотрим подробно создание однослойных и многослойных нейронных сетеq на основе возможностей библиотеки PyTorch. Базовым классом для всех модулей нейронной сети является класс **nn.Module**. Все модели нейронных сетей, создаваемые пользователями, должны быть подклассом этого класса. Модули также могут содержать другие модули, что позволяет размещать их в древовидной структуре. Подмодули можно добавлять, как обычные атрибуты класса. На рисунках 31 и 32 показаны примеры создания однослойной и многослойной нейронной сети на основе класса **nn.Module**. Программные коды примеров расположены по адресу: https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/simple_nn.ipynb.

```
import torch.nn as nn
import torch.nn.functional as F
class NN_sinleLayer(nn.Module):
    def __init__(self):
        super(NN_sinleLayer, self).__init__()
        self.linear1 = nn.Linear(20, 5)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        return x
```

Рис. 31. Создание однослойной нейронной сети в PyTorch

Все модели, создаваемые таким образом, обязательно должны переопределять конструктор базового класса и содержать метод **forward**. В конструкторе класса инициализируются все параметры и создается структура нейронной сети добавлением необходимых слоев

или подмодулей. В методе **forward** подробно описывается обработка данных нейронной сетью. В примере с однослойной нейронной сетью мы добавили только один линейный слой, в котором входная размерность равна 20, а выходная 5 (см. рис. 31). В примере с многослойной нейронной сетью мы добавили три линейных слоя (см. рис. 32).

```
import torch.nn as nn
import torch.nn.functional as F

class NN_MultiLayer(nn.Module):
    def __init__(self):
        super(NN_MultiLayer, self).__init__()
        self.linear1 = nn.Linear(100, 50)
        self.linear2 = nn.Linear(50, 20)
        self.linear2 = nn.Linear(20, 1)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = F.relu(self.linear2(x))
        return x
```

Рис. 32. Создание многослойной нейронной сети в PyTorch

В модуле torch.nn содержится много различных классов для создания нейронных сетей и организации процесса их обучения. Например, это класс **torch.nn.Sequential()**, который представляет собой последовательный контейнер, используемый для объединения различных слоев при создании сети прямого распространения. На рисунке 33 показан пример использования этого контейнера для создания простой многослойной нейронной сети. В этом примере в модель последовательно добавлены следующие слои: линейный слой, сигмоидальный слой, снова линейный слой и слой SoftMax. Структура модели нейронной сети выведена на консоль методом **print()**. Класс **torch.nn.ModuleList()** содержит индексированный список Python, который содержит подмодули. Класс **torch.nn.ModuleDict()** – создает словарь Python, содержащий подмодули. Эти контейнеры удобны при создании сложных нейронных сетей, а также для группировки различных подмодулей в понятную и удобную структуру. С помощью **torch.nn.ParameterList()** можно получить индексированный список Python, который содержит параметры модели, а метод **torch.nn.ParameterDict()** позволяет получить эти параметры в виде словаря Python.

```
model = nn.Sequential(
    nn.Linear(10, 30),
    nn.Sigmoid(),
    nn.Linear(30, 2),
    nn.Softmax()
)
print(model)
```

```
Sequential(
  (0): Linear(in_features=10, out_features=30, bias=True)
  (1): Sigmoid()
  (2): Linear(in_features=30, out_features=2, bias=True)
  (3): Softmax(dim=None)
)
```

Рис. 33. Многослойная нейронная сеть на основе контейнера **nn.Sequential()**

В PyTorch имеется несколько десятков встроенных слоев, которые могут быть использованы в создании нейронных сетей. Рассмотрим наиболее важные и наиболее часто используемые из них.

Линейные слои. Они выполняют линейное преобразование входных данных, т. е. умножают входной тензор на тензор весовых коэффициентов и добавляют смещение. Например, это слой **torch.nn.Linear()**. Это наиболее часто используемый слой при разработке нейронных сетей.

Рекуррентные слои. Они обычно используются при разработке рекуррентных нейронных сетей. Например, это такие часто используемы слои, как RNN(), LSTM() и GRU(). Рекуррентные слои и рекуррентные нейронные сети будут рассмотрены в следующем разделе.

Слои свертки. Они выполняют операции свертки, применяются в основном для обработки изображений. Примеры: **torch.nn.Conv1d(), torch.nn.Conv2d(), torch.nn.Conv3d().**

Слои субдискретизации. Используются в сверточных нейронных сетях для уменьшения пространственного объёма изображения. Примеры: **torch.nn.MaxPool1d(), torch.nn.MaxPool2d().**

Дропаут (Dropout) слои. Позволяют исключать случайным образом часть нейронов из процесса обучения. Эти слои используются для решения проблемы переобучения нейронных сетей. Примеры: **torch.nn.Dropout(), torch.nn.Dropout2d().**

В модуле **torch.optim** фреймворка PyTorch реализованы алгоритмы оптимизации. Этот модуль содержит наиболее часто используемые методы оптимизации и имеющие достаточно общий интерфейс, в который легко интегрировать более сложные методы. Чтобы использовать оптимизатор, необходимо создать объект оптимизатора, который будет содержать текущее состояние параметров модели и обновлять пошагово с помощью него параметры на основе вычисленных градиентов.

Наиболее популярные оптимизаторы, которые реализованы в PyTorch – это:

1) **torch.optim.SGD.** Реализует стохастический градиентный спуск
2) **torch.optim.Adam.** Реализует алгоритм Адама
3) **torch.optim.Adagrad.** Реализует алгоритм Adagrad.

Функции потерь используются для измерения ошибки между выходными данными прогноза и заданным целевым значением. Она вычисляет, насколько далека модель от реализации ожидаемого результата. Наиболее часто используемые функции потерь – это:

1) **torch.nn.MSELoss.** Эта функция потерь вычисляет среднеквадратичную ошибку.
2) **torch.nn.CrossEntropyLoss.** Измеряет расхождение между двумя вероятностными распределениями. Если значение кросс-энтропии велико, это означает, что разница между двумя распределениями большая, а если кросс-энтропия мала, то распределения похожи друг на друга.

На рис. 34 приведена программа, реализующая алгоритм обучения модели простой нейронной сети. Предполагается что модель нейронной сети уже создана, как было показано выше. В самом начале процесса обучения мы определяем функцию потерь, в нашем случае – это среднеквадратичная ошибка. Далее создаем оптимизатор, в конструкторе оптимизатора указываем параметры нашей модели и значение шага обучения. Далее начинаем цикл обучения в количестве 500 эпох. Внутри этого цикла реализуется выполнение стандартной последовательности операций, связанных с обучением модели, которая обычно используется во фреймворке PyTorch. Это вычисление выходных значений модели, вычисление функции потерь, обнуление градиентов с предыдущего цикла, выполнение процедуры обратного распространения ошибки и выполнение шага оптимизатора.

```
#Задаем функцию потерь
loss_fn = torch.nn.MSELoss(reduction='sum')
#шаг обучения
learning_rate = 1e-4
# создаем оптимизатор
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
#цикл обучения модели
for t in range(500):
    #вычисление выходных значений модели
    y_pred = model(x)
    #вычисление потерь
    loss = loss_fn(y_pred, y)
    #обнуляем градиенты
    optimizer.zero_grad()
    #обратное распространение ошибки
    loss.backward()
    #шаг оптимизатора
    optimizer.step()
```

Рис. 34. Обучение модели нейронной сети в PyTorch

Вопросы

1. Какие компоненты включает в себя один нейрон?
2. Что такое нейронные сети?
3. Что представляет собой однослойная нейронная сеть?
4. Какие функции активации используются в однослойных нейронных сетях?
5. Как обучаются однослойные нейронные сети?
6. Какие ограничения у однослойных нейронных сетей в решении сложных задач?
7. Что такое многослойные нейронные сети (МНС)?
8. Какие преимущества предоставляют многослойные нейронные сети?
9. Какова структура многослойной нейронной сети?
10. Какие слои чаще всего встречаются в многослойных нейронных сетях?
11. Как происходит обучение многослойных нейронных сетей?
12. Что такое функции потерь в контексте обучения нейронных сетей?
13. Что такое градиентный спуск и как он используется при обучении нейронных сетей?
14. Какие техники и методы оптимизации применяются при обучении нейронных сетей в PyTorch?
15. Как оценить производительность многослойной нейронной сети?
16. Как реализовать однослойную нейронную сеть в PyTorch?
17. Как создать многослойную нейронную сеть в PyTorch?
18. Какие бывают типы оптимизаторов в PyTorch и для чего они используются?
19. Какие функции активации доступны в PyTorch?
20. Как происходит обратное распространение ошибки в PyTorch?

ГЛАВА 4. Подготовка данных и организация процесса обучения нейронной сети

В этой главе мы разработаем программу на основе библиотеки PyTorch и продемонстрируем процесс обучения многослойной полносвязной нейронной сети с использованием набора данных о ценах на жилье.

Процесс решения задачи классификации или регрессии с помочью нейронных сетей состоит из следующих этапов:

Сбор данных для обучения. Выбор данных для обучения сети и их обработка является самым сложным этапом решения задачи. Исходные данные преобразуются к виду, в котором их можно подать на входы сети. Каждая запись в файле данных называется обучающей парой или обучающим вектором.

Выбор топологии сети. Выбирать тип сети следует исходя из постановки задачи и имеющихся данных для обучения. Для обучения с учителем требуется наличие для каждого элемента выборки «экспертной» оценки.

Экспериментальный подбор характеристик сети. После выбора общей структуры нужно экспериментально подобрать параметры сети. Для сетей, подобных перцептрону, это будет число слоёв, число нейронов в скрытых слоях, наличие или отсутствие обходных соединений, передаточные функции нейронов.

Экспериментальный подбор параметров обучения. После выбора конкретной топологии необходимо выбрать параметры обучения нейронной сети. От правильного выбора параметров зависит не только то, насколько быстро ответы сети будут сходиться к правильным ответам. Например, выбор низкой скорости обучения увеличит время схождения, однако иногда позволяет избежать паралича сети. Значения параметров нужно выбирать экспериментально, руководствуясь при этом критерием завершения обучения (например, минимизация ошибки или ограничение по времени обучения).

Обучение сети. В процессе обучения сеть в определённом порядке просматривает обучающую выборку. При обучении с учителем набор исходных данных делят на две части – собственно обучающую выборку и тестовые данные; принцип разделения может быть произвольным. Обучающие данные подаются сети для обучения, а тестовые используются для расчета ошибки сети (проверочные данные никогда для обучения сети не применяются). Таким образом, если на тестовых данных ошибка уменьшается, то сеть действительно выполняет обобщение. Если ошибка на обучающих данных продолжает уменьшаться, а ошибка на тестовых данных увеличивается, значит, сеть перестала выполнять обобщение и просто «запоминает» обучающие данные. Это явление называется переобучением сети. В таких случаях обучение обычно прекращают.

Проверка адекватности обучения. Тестирование качества обучения нейросети необходимо проводить на примерах, которые не участвовали в её обучении.

Программный код примера расположен по адресу:
https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/train_ffnn.ipynb.

В самом начале программы мы подключаем необходимые нам модули и функции Python. Библиотека Pandas дает нам возможность быстро и эффективно манипулировать таблицами с данными. Библиотека **numpy** необходима для работы с массивами, а **matplotlib** для визуализации данных. Таже мы импортируем необходимые функции библиотеки **sklearn**, которые будут использованы для загрузки и подготовки данных, а также для оценки качества обученной модели. Это функция **load_boston** для загрузки данных, функция **train_test_split** – для разделения данных на обучающую и тестовую выборки и r2_score – для вычисления коэффициента детерминации. Естественно, мы также подключаем библиотеку PyTorch, и функции этой библиотеки, которые позволят нам построить и обучить нейронную сеть (см. рис. 35).

```
import pandas as pd
import numpy as np
import matplotlib.pylab as plt

from sklearn.datasets import load_boston
from sklearn import preprocessing
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

Рис. 35. Подключение необходимых библиотек

В качестве примера используем Boston Housing Dataset. Этот набор данных состоит из 13 признаков и содержит информацию, собранную Службой переписи населения США о характеристиках недвижимости в районах Бостона. Среди признаков имеются такие, как количество комнат, уровень преступности в районе, ставка налога на недвижимость, возраст людей, которым принадлежит дом, соотношение числа учащихся и преподавателей в районе и другие признаки. Мы будем использовать этот набор данных для прогнозирования средней стоимости дома на основе других признаков имеющихся в этом наборе данных, т. е. мы будем

решать задачу регрессии. Загружаем этот набор данных в переменную Python, используя метод **load_boston**, поэтому нам нет особой необходимости скачивать его откуда-либо. Однако вы можете его скачать самостоятельно с сайта, который указан в дополнительных материалах и загрузить с помощью методов библиотеки **pandas**.

Далее мы конвертируем его в датафрейм **Pandas**, для того, что мы могли с ним удобно работать и выводим его на консоль, чтобы посмотреть значения, имеющиеся в этом наборе данных. Также мы можем визуализировать эти данные в виде различных графиков с использованием возможностей библиотеки **matplotlib**. (см. рис. 36).

```
Загрузка данных

boston_dataset = load_boston()

Конвертация данных в датафрейм Pandas

df_boston = pd.DataFrame(boston_dataset.data)
df_boston.columns = boston_dataset.feature_names
df_boston["PRICES"] = boston_dataset.target
print(df_boston.head())

Построение диаграммы рассеяния

df_boston.plot.scatter(x='RM',y='PRICES', c='DarkBlue')
plt.xlabel("RM")
plt.ylabel("PRICES")
plt.show()
```

Рис. 36. Загрузка и визуализация данных (Boston Housing Dataset)

На рис. 37 показан пример диаграммы рассеяния, которая показывает зависимость стоимости дома от количества комнат. Аналогично вы можете построить зависимости цен домов и от других параметров.

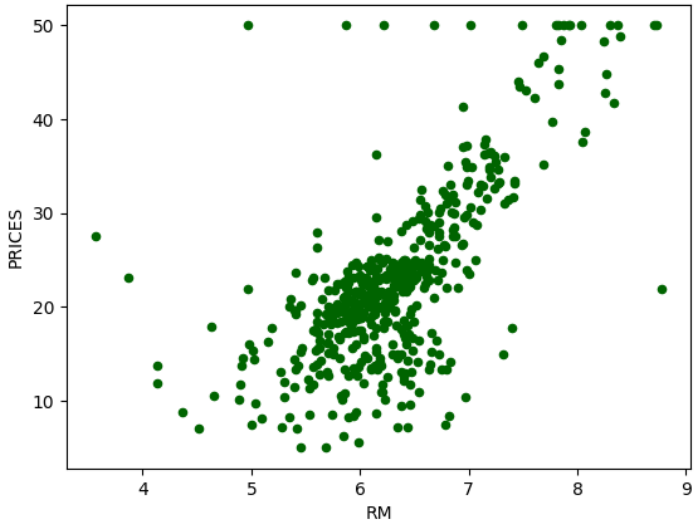


Рис. 37. Зависимость цены дома от количества комнат

На этапе подготовки данных сначала выбираем столбцы таблицы с данными, которые будут использованы в качестве входных данных и целевых значений. При решении задач машинного обучения рекомендуется нормализовать данные, чтобы значения находились в определенном интервале. Для этого здесь мы используем класс **StandardScaler** библиотеки **sklearn** (см. рис. 38).

Далее мы разделяем весь имеющийся у нас набор данных на тестовую и обучающую выборки. Это делается с помощью вызова метода **train_test_split**, в котором в качестве параметров мы также указываем и долю данных, которые будут выделены в тестовую выборку. В нейронную сеть данные необходимо подавать в виде тензоров PyTorch, поэтому мы конвертируем наши данные в тензоры. В результате мы получаем тензор **x_train_tensor** в качестве входных данных, а **y_train_tensor** – это целевые значения.

```
Прогнозируемое значение и свойства, используемые для прогнозирования

TargetColumn = "PRICES“
FeatureColumns = ["CRIM","ZN", 'INDUS','CHAS', 'NOX', 'RM',
'AGE','DIS','RAD','TAX','PTRATIO','B','LSTAT']
X = df_boston[FeatureColumns]
Y = df_boston[TargetColumn]

Нормализация входных данных

scaler = preprocessing.StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

Разделение всех данных на обучающую и тестовую выборку

X_train, X_test, Y_train, Y_test = train_test_split(X_scaled, Y, test_size=0.2,
random_state=99)

Конвертация данных  в тензора Pytorch

x_train_tensor = torch.tensor(np.array(X_train), dtype=torch.float)
y_train_tensor = torch.tensor(np.array(Y_train).reshape(-1, 1), dtype=torch.float)
```

Рис. 38. Подготовка данных

Нейронные сети в PyTorch создаются в виде классов, которые наследуются от класса **nn.Module**. В конструкторе класса мы описываем структуру нейронной сети, добавляем последовательно необходимые слои и указываем функции активации. В нашем случае мы будем использовать трёхслойную нейронную сеть, в которой все слои являются линейными. Количество входов нейронной сети, а значит и размерность входа первого слоя равна количеству

входных признаков. В качестве размерности входов и выходов промежуточных слоев мы можем ставить произвольные значения. Однако необходимо понимать, что если эти значения будут слишком низкими, то нейросеть окажется слабой и будет плохо обучаться. Но не следует ставить и слишком высокие значения, в этом случае нейросеть будет, во-первых, слишком долго обучаться, а также вы можете столкнуться с проблемой переобучения. В качестве функции активации используем функцию **ReLu**. Так как мы прогнозируем значение только одного параметра, цену дома, то количество выходов равно единице (см. рис. 39).

Описание класса НС

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.linear_layer1 = nn.Linear(X.shape[1], 16)
        self.linear_layer2 = nn.Linear(16, 8)
        self.linear_layer3 = nn.Linear(8, 1)

    def forward(self, x):
        x = F.relu( self.linear_layer1(x) )
        x = F.relu( self.linear_layer2(x) )
        x = self.linear_layer3(x)
        return x
```

Создание экземпляра НС

```
model = NeuralNetwork()
```

Вывод информации об архитектуре НС на консоль

```
print(model)
```

```
NeuralNetwork(
  (linear_layer1): Linear(in_features=13, out_features=16, bias=True)
  (linear_layer2): Linear(in_features=16, out_features=8, bias=True)
  (linear_layer3): Linear(in_features=8, out_features=1, bias=True)
)
```

Рис. 39. Создание модели нейронной сети

В методе **forward** мы описываем последовательность вычислений, выполняемых нейронной сетью. Входной тензор передается в метод через переменную x. Сначала вычисления проводятся первым линейным слоем, и вычисляется функция активации. Результат мы также записываем в переменную x. Дальше производится аналогичная обработка также и на втором слое. И наконец, результат вычисления второго слоя, после вычисления функции активации, поступает на выходной слой и возвращается из метода **forward** (см. рис. 39).

Для того чтобы использовать нейронную сеть, мы создаем объект класса нейронной сети, которая будет храниться в переменной **model**. С помощью метода **print** мы выводим информацию об архитектуре нейронной сети на консоль.

Теперь у нас все готово для начала процесса обучения нейронной сети. Так как мы решаем задачу регрессии, то в качестве оптимизатора выберем стохастический градиентный спуск (SGD), а в качестве функции потерь используем среднеквадратичную ошибку.

Далее мы задаем количество эпох обучения и запускаем процесс итеративного обучения нашей модели. Обучение нейронной сети в PyToch обычно состоит из последовательности стандартных шагов. Сначала мы должны обнулить градиенты вызвав метод **zero_grad()** оптимизатора (см. рис. 40). Это очень важный шаг, и его необходимо выполнить обязательно.

Оптимизатор

```
optimizer = optim.SGD(model.parameters(), lr=0.003)
```

Функция потерь

```
loss_function = nn.MSELoss()
```

Обучение НС

```
epochs_count = 300
for i in range(epochs_count):
    optimizer.zero_grad() # инициализируем градиенты
    y_val = model(x_train_tensor) # вычисление выходных значений
НС
    loss = loss_function(y_val, y_train_tensor) # вычисляем ошибку НС
    loss.backward() # выполняем обратное распространение ошибки
    optimizer.step()    # обновляем параметры НС
    if (i % 20) == 0:    # выводим информацию об ошибке
      print('epoch: {},'.format(i) + 'loss: {:.5f}'.format(loss))
```

Рис. 40. Обучение нейронной сети

Далее мы подаем на вход нейронной сети наш входной тензор с обучающими данными, на выходе модели получаем выходные значения. Здесь автоматически вызывается метод **forward()** нейронной сети. Далее производится вычисление значения функции потерь, передачей в эту функцию значений, спрогнозированных нейронной сетью, и целевых значений. Функция потерь вычисляет, насколько сильно отличаются выходные значения нейронной сети от целевых значений. Далее выполняется процедура обратного распространение ошибки вызовом метода **backward()**. Теперь, когда мы знаем значения градиентов ошибок, то мы можем осуществить процедуру настройки параметров нейронной сети. Эта операция выполняется вызовом метода **step()** оптимизатора. Для того чтобы видеть, как обучается нейронная сеть, мы

выводим значения ошибки на каждом 20-м шаге обучения. Мы видим, что в самом начале процесса обучения значения функции потерь очень велики, но постепенно это значение уменьшается. Это означает, что нейронная сеть довольно неплохо обучается на тренировочном наборе данных. Мы обучали нашу нейронную сеть в течение 300 эпох, можно обучать и подольше, в этом случае ошибка была бы еще ниже (см. рис. 41).

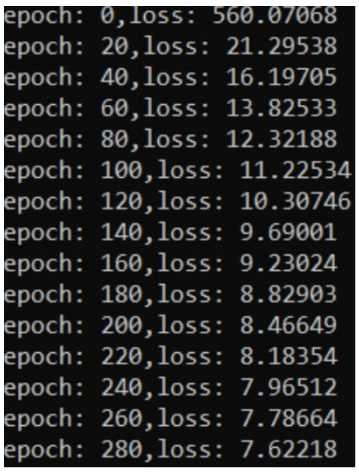


Рис. 41. Уменьшение ошибки НС в процессе обучения

Для того чтобы оценить, насколько хорошо обучена наша модель, мы должны проверить ее работу на тестовом наборе. Потому что может быть так, что нейронная сеть будет показывать очень хороший результат на обучающей выборке, а на тестовой может сильно ошибаться. Поэтому именно качество прогноза на тестовых данных показывает, насколько хорошо обучена модель. Для оценки качества модели можно использовать различные метрики в зависимости от типа задачи. Здесь мы вычисляем коэффициент детерминации на основе выходных значений, которые были спрогнозированы нейронной сетью на основе тестовых данных. Мы получаем значение 0.83, что довольно неплохо (см. рис. 42).

```
# Прогноз на обучающей и тестовой выборке
Y_train_pred = model(torch.tensor(X_train, dtype=torch.float))
Y_test_pred = model(torch.tensor(X_test, dtype=torch.float))

Конвертация тензоров в numpy массивы
Y_train_pred = Y_train_pred.detach().numpy()
Y_test_pred = Y_test_pred.detach().numpy()

Вычисляем коэффициент детерминации R2 для оценки
качества модели

R2 = r2_score(Y_test, Y_test_pred)
print(R2)
```

Рис. 42. Оценка качества модели

Также мы можем визуализировать результаты работы нейронной сети в виде диаграммы рассеяния, по горизонтальной оси – это реальная цена, а по вертикальной оси – цена, спрогнозированная нейронной сетью (см. рис. 43).

```
plt.figure(figsize=(5, 5), dpi=100)
plt.xlabel("PRICES")
plt.ylabel("Predicted PRICES")
plt.scatter(Y_train, Y_train_pred, lw=1, color="r", label="train")
plt.scatter(Y_test, Y_test_pred, lw=1, color="b", label="test")
plt.legend()
plt.show()
```

Рис. 43. Графическое отображение результатов

Красными точками отображены результаты на обучающей выборке, а синими – на тестовой выборке. Чем ближе точки находятся к главной диагонали, тем выше прогнозная мощность обученной нейронной сети. В нашем случае большинство точек очень близки к главной диагонали (см. рис. 44).

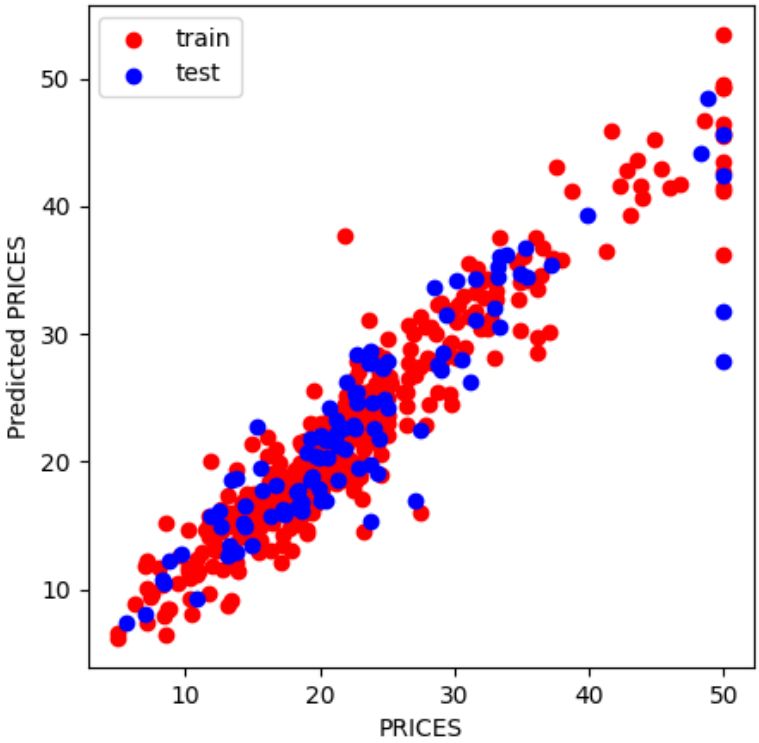


Рис. 44. Диаграмма рассеяния

Вопросы

1. Какие шаги включает в себя процесс подготовки данных для обучения нейронной сети?

2. Как загрузить данные в PyTorch для использования в многослойной нейронной сети?

3. Как разделить данные на обучающий и тестовый наборы?

4. Как обработать категориальные переменные в данных для использования в многослойной нейронной сети?

5. Как провести нормализацию данных перед обучением нейронной сети?

6. Как использовать библиотеку PyTorch для создания собственного датасета?

7. Какие функции потерь выбрать для различных типов задач (классификация, регрессия)?

8. Как провести кодирование меток классов (one-hot encoding) для категориальных переменных?

9. Как оценить качество данных перед обучением многослойной нейронной сети?

10. Как обработать неструктурированные данные для использования в многослойной нейронной сети?

11. Как создать собственный класс Dataset в PyTorch?

12. Что такое батчи и как они используются при обучении нейронных сетей?

ГЛАВА 5. Рекуррентные нейронные сети

В этой главе мы изучим рекуррентные нейронные сети. Вначале мы рассмотрим основные особенности и области применения рекуррентных нейронных сетей, изучим алгоритм работы обычной рекуррентной нейронной сети (RNN). Кратко коснемся основных моментов касательно обучения рекуррентных нейронных сетей, рассмотрим LSTM нейронные сети. И в конце рассмотрим RNN и LSTM в слоях библиотеки PyTorch и режимы работы рекуррентной нейронной сети (RNN).

Ранее мы познакомились с основными принципами построения архитектур полносвязных однослойных и многослойных нейронных сетей. Такие нейронные сети состоят из одного слоя или группы слоёв и принимают на вход тензор фиксированного размера. Каждый слой применяет к этому объекту какие-либо преобразования, и на выходе мы получаем результат. Но в некоторых областях машинного обучения нам необходимо иметь большую гибкость в типах данных, которые мы могли бы обрабатывать. Классические нейронные сети без рекуррентных связей не могут запоминать информацию, и это является их главным недостатком. В полносвязных нейронных сетях с прямыми связями подразумевается, что все входы и выходы независимы. Например, если необходимо классифицировать события, происходящие в каждом кадре фильма. Или, например, если вы хотите предсказать следующее слово в предложении, то необходимо учитывать предшествующие ему слова. Непонятно, как классическая нейронная сеть может использовать свои предыдущие выводы для дальнейших решений. Есть много задач, которые невозможно решить с помощью такого подхода. Именно это позволяют делать рекуррентные нейронные сети. Рекуррентные нейронные сети – это сети с обратными или перекрестными связями между различными слоями нейронов. При этом под обратной связью подразумевается связь от логически более удалённого элемента к менее удалённому. Идея RNN заключается в последовательном использовании информации.

Рекуррентные нейронные сети применяются для решения задач языкового моделирования и генерации текстов, в машинном переводе, для распознавания речи, для генерации описания изображений, прогнозирования следующего кадра на основе предыдущих, в разнообразных задачах, связанных с анализом текстов, для обработки последовательности звуков, при прогнозировании временных рядов, и во многих других задачах.

Рекуррентные нейронные сети выполняют одинаковые вычисления для каждого элемента последовательности, причем выход зависит от предыдущих вычислений. Т. е. рекуррентные нейронные сети, это сети, у которых есть «память», учитывающая предшествующую информацию. Эта память описывается в виде скрытого состояния h. Такие нейронные сети

могут использовать информацию о произвольно длинных последовательностях, хотя на практике они ограничены лишь несколькими шагами (см. рис. 45).

Вычисления в большинстве типовых рекуррентных сетей можно разложить на три преобразования:

1) преобразование входа x в скрытое состояние h;
2) преобразование от предыдущего скрытого состояния в следующее скрытое состояние;
3) преобразование скрытого состояния h в выход y.

На рис. 46 приведены формулы, по которым вычисляется значение скрытого состояния нейронной сети и выходное значение.

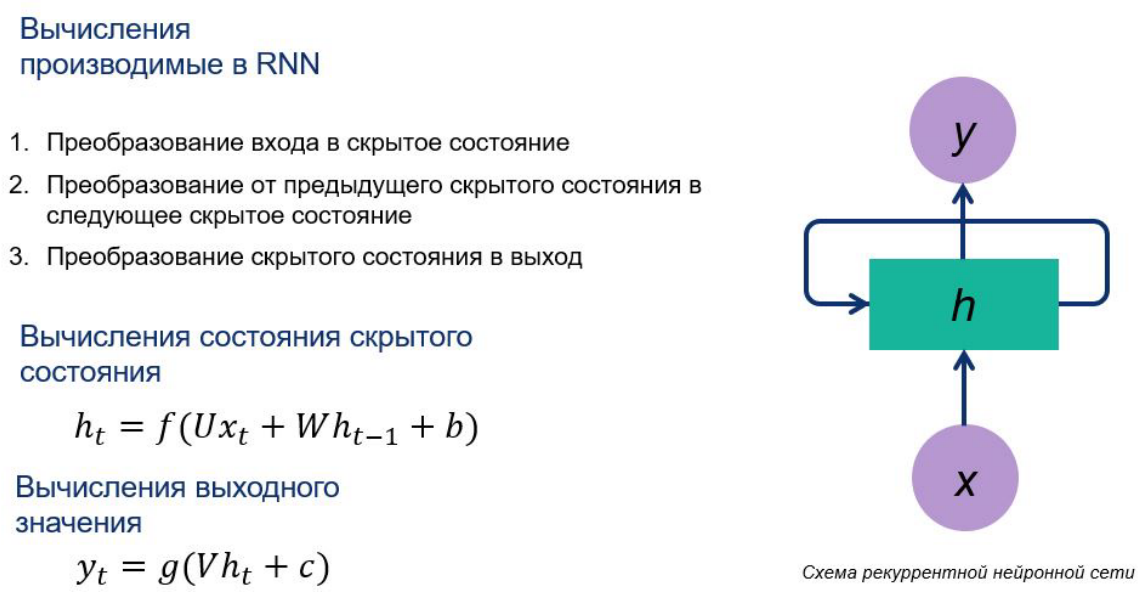


Рис. 45. Алгоритм вычислений обычной рекуррентной нейронной сети

Из-за циклов рекуррентные нейронные сети становятся трудными в понимании и в обучении. Рекуррентную сеть можно развернуть в последовательность одинаковых обыкновенных нейронных сетей, передающих информацию к последующим. На диаграмме (см. рис. 46) показано, как рекуррентная нейронная сеть разворачивается во времени. Разверткой мы просто выписываем сеть до полной последовательности. Например, если последовательность состоит из 3 элементов, то развертка будет состоять из 3 слоев, по слою на каждый элемент данных. Здесь x(t) – входное значение на временном шаге t; x(t-1) – входное значение на предыдущем временном шаге; x(t+1) – входное на последующем временном шаге. h(t) – это скрытое состояние на шаге времени t. h(t) зависит, как функция, от предыдущих состояний h(t-1) и от текущего входного значения x(t). y(t) – выходное значение на шаге t (см. рис. 46).

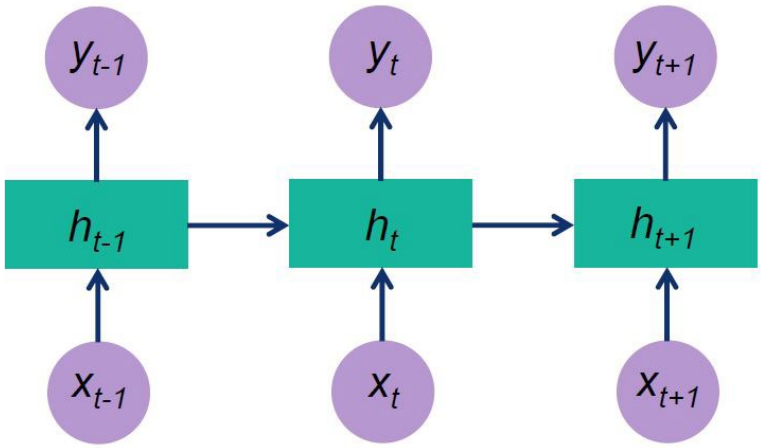


Рис. 46. Схема рекуррентной нейронной сети развернутой во времени

На вход нейронной сети поступают элементы последовательности. На вход первой копии поступает первый элемент последовательности входных данных, следующий элемент поступает на вход второй копии и так далее. Рекуррентная нейронная сеть возвращает два значения. Первое значение – это выходное значение, которое поступает на выход из нейронной сети. Также рекуррентная нейронная сеть выдает второе значение – h, которое поступает на вход следующей копии нейронной сети. Следующая копия нейронной сети на вход получает второй элемент последовательности x, а также скрытое состояние с предыдущего этапа. Вторая копия нейронной сети анализирует одновременно текущий элемент последовательности и данные со скрытого состояния предыдущей копии нейронной сети и, в зависимости от результатов анализа, также выдает два значения: выходное значение и скрытое состояние, которое передается следующей копии нейронной сети. И так продолжается, пока мы не дойдем до последнего элемента данных в последовательности. Для него рекуррентная сеть выдает уже одно значение, которое подается на выход без скрытого состояния.

Рекуррентные нейронные сети могут состоять из множества последовательных рекуррентных слоев. На рис. 47 схематично изброжена многослойная рекуррентная нейронная сеть. Сначала входные данные подаются на первый слой. Выходные значения первого слоя передаются на следующий слой. Таким образом осуществляется последовательная обработка информации всеми слоями нейронной сети, и выход последнего слоя будет выходом всей нейронной сети.

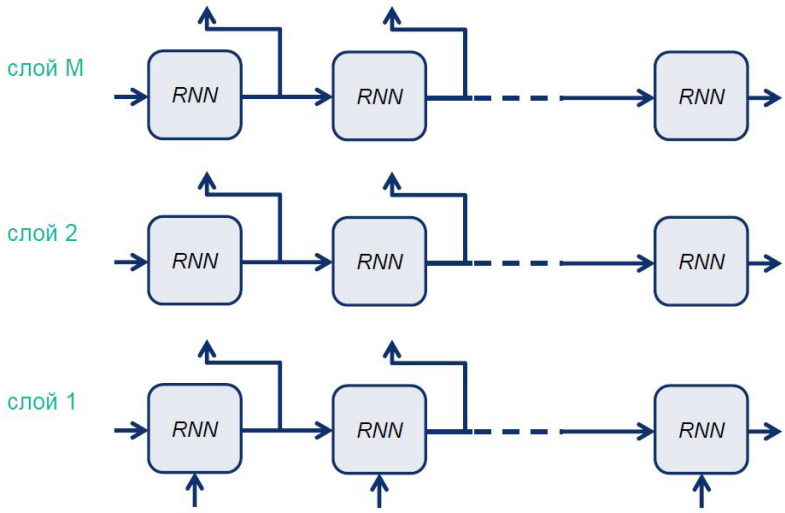


Рис. 47. Схема многослойной рекуррентной нейронной сети развернутой во времени

Обучение RNN аналогично обучению обычной полносвязной нейронной сети. Здесь также используется алгоритм обратного распространения ошибки, однако с небольшим изменением в связи с особенностями рекуррентной нейронной сети. Поскольку одни и те же параметры используются на всех временных этапах в сети, градиент на каждом выходе зависит не только от расчетов текущего шага, но и от предыдущих временных шагов. Например, чтобы вычислить градиент при t = 3, нам нужно было бы провести обратное распространение ошибки на 3 предыдущих шага и суммировать градиенты (см. рис. 48). Такой алгоритм называется «алгоритмом обратного распространения ошибки сквозь время». Необходимо отметить, что рекуррентные нейронные сети испытывают проблемы при обучении из-за затухания или взрывания градиентов. Для того, чтобы обойти эти проблемы, были разработаны специальные архитектуры RNN (например LSTM сети).

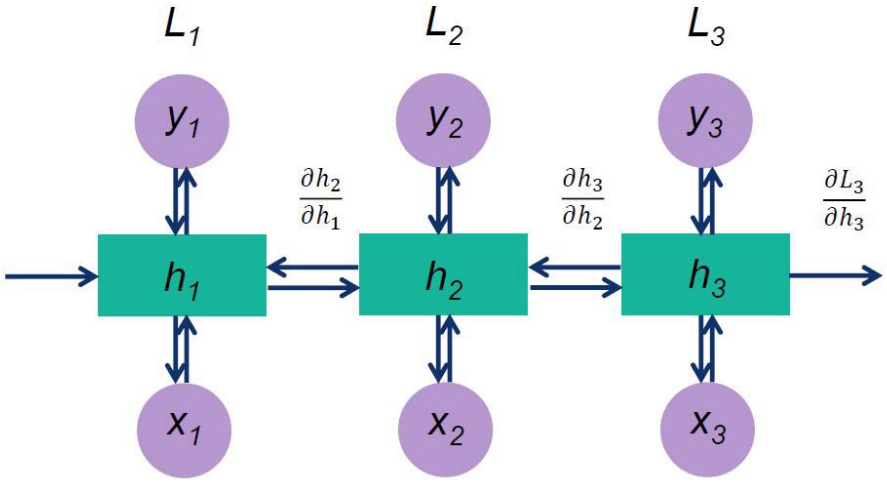


Рис. 48. Схема «алгоритма обратного распространения ошибки сквозь время»

Долгая краткосрочная память (Long short-term memory; LSTM) – особая разновидность архитектуры рекуррентных нейронных сетей, способная к обучению долговременным зависимостям. Они успешно решают целый ряд разнообразных задач и в настоящее время широко используются. Эти нейронные сети разработаны специально, чтобы избежать проблемы запоминания долговременных зависимостей. Структуру LSTM также можно изобразить в виде цепочки, но модули выглядят иначе (см. рис. 49). Вместо одного слоя нейронной сети они содержат целых четыре, которые взаимодействуют особенным образом.

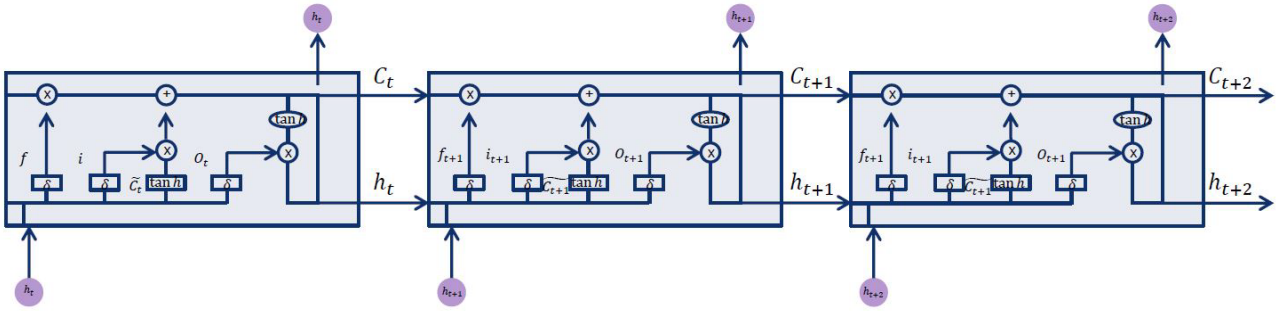


Рис. 49. Схема LSTM нейронной сети

На рис. 50 подробно изображена внутренняя структура LSTM ячейки, а также приведены формулы, описывающие обработку информации внутри нее. Ключевым понятием здесь является состояние ячейки – это горизонтальная линия, проходящая через верхнюю часть диаграммы. В LSTM ячейках уменьшается или увеличивается количество информации в состоянии ячейки, в зависимости от потребностей. Для этого используются настраиваемые структуры, пропускающие или не пропускающие информацию, которые называются гейтами или воротами. Гейты состоят из сигмовидного слоя и операции поточечного умножения.

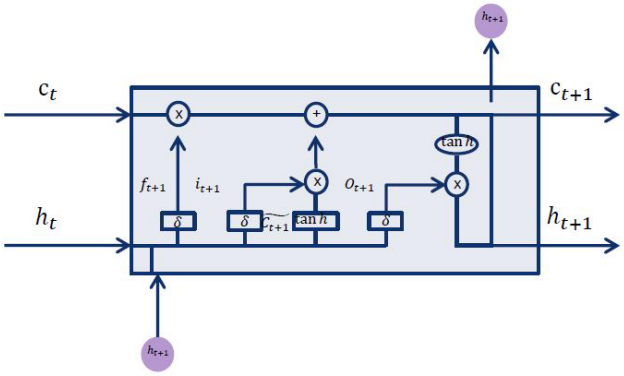
На первом шаге необходимо определить, какую информацию можно удалить из состояния ячейки. Это выполняется сигмоидальным слоем, называемый «слоем забывания» (forget gate layer). Этот гейт на основе предыдущего скрытого состояния h_{t-1} и текущего входа x_t и возвращает число от 0 до 1, 1 означает «полностью сохранить», а 0 – «полностью забыть».

На следующем шаге принимается решение о том, какая новая информация будет храниться в состоянии ячейки. На этом шаге сначала сигмоидальный слой под названием «слой входного фильтра» (input layer gate) определяет, какие значения следует обновить. Затем слой, в котором используется тангенциальная функция активации, строит вектор новых значений-кандидатов, которые можно добавить в состояние ячейки.

Затем старое состояние ячейки C_{t-1} заменяется на новое состояние C_t. Для этого старое состояние умножается на f_t, и прибавляется i_t, умноженное на g_t.

Выходное значение будет вычисляться основе состоянии ячейки, с применением некоторых фильтров. Сначала применятся сигмоидальный слой, затем значения состояния ячейки

проходят через слой с активационной функцией гиперболический тангенс, чтобы получить на выходе результат в диапазоне от -1 до 1, и далее перемножаются с выходными значениями сигмоидального слоя, что позволяет выводить только требуемую информацию.



$$f_t = \sigma\left(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}\right)$$
$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi})$$
$$g_t = \tan h\left(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}\right)$$
$$c_t = f_t * c_{(t-1)} + i_t * g_t$$
$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho})$$
$$h_t = o_t * \tan h\,(c_t)$$

Рис. 50. Схематическое изображение LSTM ячейки

На рис. 51 показаны примеры использования RNN и LSTM слоев библиотеки PyTorch Классический рекуррентный RNN слой добавляется как функция **torch.nn.RNN()**, LSTM слой добавляется через функцию **torch.nn.LSTM()**. Основные параметры, которые необходимо указывать при добавлении этих слоев в нейронную сеть, это input_size – размерность входа, hidden_size – размерность скрытого состоянии и num_layers – количество повторяющихся слоев. Например, если укажем количество слоев num_layers = 2, это будет означать двухслойную нейронную сеть, в котором второй слой будет принимать выходные значения первого слоя в качестве входных значений.

Пример использования RNN слоя

```
rnn = nn.RNN(10, 20, 2)
input = torch.randn(5, 3, 10)
h0 = torch.randn(2, 3, 20)
output, hn = rnn(input, h0)
```

Пример использования LSTM слоя

```
rnn = nn.LSTM(10, 20, 2)
input = torch.randn(5, 3, 10)
h0 = torch.randn(2, 3, 20)
c0 = torch.randn(2, 3, 20)
output, (hn,cn) = rnn(input, (h0, c0))
```

Рис. 51. RNN и LSTM слои в PyTorch

Существует четыре основных режима работы и использования рекуррентных нейронных сетей (см. рис. 52).

Первый режим называется «многие ко многим». В этом режиме на вход сети последовательно подаются входные данные, и параллельно на выходе тоже получаем последовательность выходных значений. Обучение последовательностям «многие ко многим» можно использовать для машинного перевода, когда входная последовательность кодирует слова на одном языке, а выходная последовательность – на каком-то другом.

В режиме «многие к одному» у нас есть последовательность в качестве входных данных, и мы должны предсказать единственный выход. Одним из вариантов использования такого режима является анализ тональности или классификация текста.

В режиме «один ко многим» имеется только один входной образ, или входные данные имеются только для одного временного шага, а выходные данные содержат вектор из нескольких значений или для нескольких временных шагов. Такой режим может применяться, например, для генерации аудиозаписи. На вход подаем жанр музыки, который хотим получить, на выходе получаем аудиозапись. Также такой режим может применяться для создания заголовков или текстовых описаний изображений.

В режиме «один к одному» имеется только одно входное и одно выходное значение, такой режим применяет очень редко.

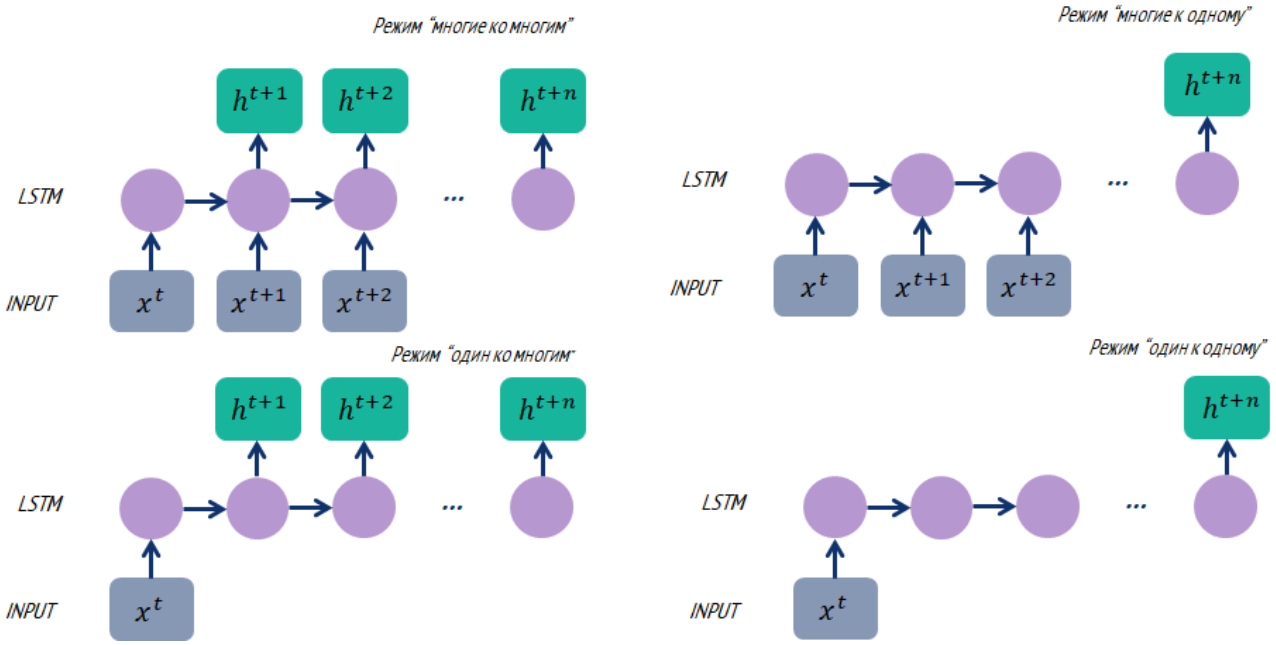


Рис. 52. Режимы работы рекуррентных нейронных сетей

Вопросы

1. Что такое рекуррентные нейронные сети (RNN)?

2. В чем заключается основная идея использования рекуррентных связей в нейронных сетях?

3. Какие задачи можно решать с помощью рекуррентных нейронных сетей?

4. Какова структура базового рекуррентного блока в RNN?

5. Как происходит передача информации внутри рекуррентной нейронной сети на каждом временном шаге?

6. Как обучаются рекуррентные нейронные сети в PyTorch?

7. Как преодолеть проблему затухающих и взрывающихся градиентов в рекуррентных сетях?

8. Как выбрать функцию активации для рекуррентных нейронных сетей?

9. Как происходит обучение рекуррентных нейронных сетей на последовательных данных?

10. Что такое LSTM (Long Short-Term Memory) в контексте рекуррентных нейронных сетей?

11. Какие проблемы решает механизм LSTM в сравнении с обычными рекуррентными нейронными сетями?

12. Какова структура основного LSTM блока в PyTorch?

13. Как LSTM сохраняет и обрабатывает долгосрочные зависимости в данных?

14. Как происходит передача информации через ворота (gates) в LSTM?

15. Какие компоненты включают в себя ворота LSTM: вход, забывания и вывода?

16. Как реализовать однонаправленные LSTM в PyTorch?

ГЛАВА 6. Прогнозирование временных рядов с помощью рекуррентных нейронных сетей

В этой главе мы построим рекуррентные нейронные сети на основе слоев RNN и LSTM, и применим их для решения задачи прогнозирования временных рядов. Временной ряд – это собранный в разные моменты времени статистический материал о значении каких-либо параметров исследуемого процесса. В простейшем случает временной ряд может содержать динамику только одного параметра. В настоящее время прогнозирование временных рядов является одной из наиболее популярных задач, решаемых с использованием методов машинного обучения. Прогноз будущих значений временного ряда используется для эффективного принятия решений. В качестве примеров временных рядов можно привести динамику температуру воздуха, атмосферного давления, уровня воды в каком-либо водоеме, динамику цен акций, обменные курсы валют, динамику численность населения в каком-либо городе. Программный код примера, рассмотренного в этом разделе, расположен по адресу: https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/train_rnn.ipynb.

В самом начале программы мы подключим необходимые библиотеки Python и инициализируем значения основных параметров, связанных с временным рядом и процессом обучения нейронных сетей. Естественно, подключаем библиотеку PyTorch, модуль nn, который позволит нам строить нейронные сети, добавлять необходимые слои в нейронные сети и организовать цикл обучения модели. Далее импортируем классы **Dataset, DataLoader**, которые позволят нам организовать процесс обучения нейронной сети на основе сгенерированного временного ряда (см. рис. 53).

Далее мы инициализируем параметры, которые будут использованы как характеристики генерируемого временного ряда и процесса обучения модели. В этом примере временной ряд мы будем генерировать самостоятельно. Для простоты это будет синусоида с добавлением случайного шума. Используя переменную data_count мы указываем количество элементов временного ряда. Уровень шума, который мы добавляем к синусоиде, будем задавать с помощью переменной noise_level. Для организации обучения нам будет необходима длина скользящего окна, которая будет храниться в переменной sequenceLength. При обучении нейронной сети входные данные будем подавать пачками (батчами), поэтому нам необходимо задать размеры этих пачек. В нашем случае мы вычисляем размер пачек делением количества элементов временного ряда на длину вектора обучения. Далее мы вводим переменные, описывающие характеристики нейронных сетей. Это количество рекуррентных слоев, размерность скрытого слоя, шаг обучения и количество эпох обучения (см. рис. 53).

Подключение библиотек

```
import torch
import torch.nn as nn

from torch.utils.data import Dataset
from torch.utils.data import DataLoader

import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
```

Инициализация параметров

```
data_count=2000
noise_level=0.1

sequenceLength = 100
batchSize = int(data_count/sequenceLength)
num_layers = 1
hiddenSize = 4
learningRate = 0.02
epochs = 500
```

Рис. 53. Подключение необходимых библиотек и инициализация параметров

В качестве временного ряда будет выступать простая синусоида с добавлением случайного шума (см. рис. 54). Уровень шума задается переменной noise_level, которая была инициализирована в самом начале программы. Временные ряды зависят от времени, поэтому сначала создаем переменную t, в которой будут равномерно записаны числа от 0 до 100. Количество временных шагов будет равно значению переменной data_count. Мы будем создавать две выборки: обучающую и тестовую. На обучающей выборке мы будет обучать нейросеть, а на тестовой выборке мы оценим качество нейронной сети. Эти выборки создаются с помощь суммирования результата функции **torch.sin** – которая генерирует синусоиду, и функции **torch.randn** – которая генерирует тензор со случайными значениям. Предварительно, результат, выдаваемый этим методом, мы умножаем на значение переменной noise_level, которая описывает уровень шума. Таким образом, изменяя значение этой переменной, мы может делать наш временной ряд более зашумленным или менее зашумленным. Для того, чтобы посмотреть, как выглядит наш временной ряд, мы можем построить его в виде графика. Вы можете попробовать поменять значение переменной noise_level, для того чтобы посмотреть, как выглядит временной ряд для различного уровня добавляемого шума.

```
t=torch.linspace(0,100,data_count)
#обучающая выборка
sin_data_train = torch.sin(t) +noise_level*torch.randn(data_count)

#построение графика обучающих данных
plt.plot(t[0:500],sin_data_train[0:500])
plt.ylabel('Sin(t)')
plt.xlabel('t')
plt.show()

#тестовая выборка
sin_data_test = torch.sin(t) + noise_level*torch.randn(data_count)
```

Рис. 54. Генерация данных для обучения нейронных сетей

Тестовая выборка, которая не будет участвовать в процессе обучения моделей, а будет применена на этапе оценки качества модели, генерируется аналогично (см. рис. 55).

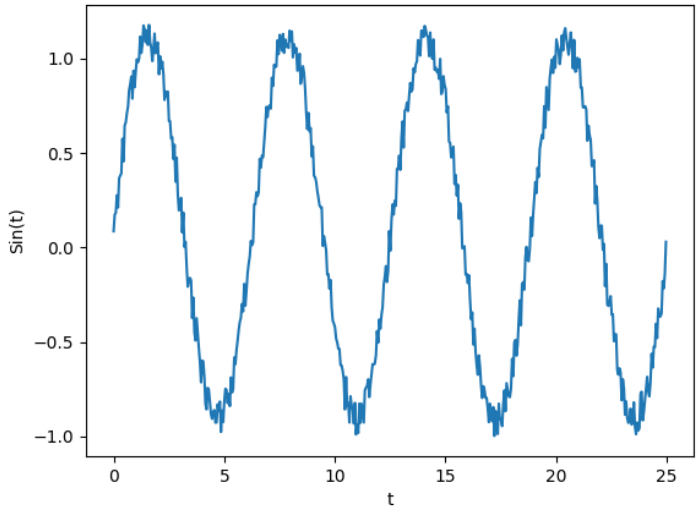


Рис. 55. График обучающей выборки

Используем метод скользящего окна при формировании набора данных для обучения модели. Для построения скользящего окна берется некоторый отрезок временного ряда длиной sequenceLength, который и будет представлять собой входной вектор. Значением желаемого выхода в примере будет следующее по порядку значение. Затем «скользящее окно» сдвигается на одну позицию в направлении возрастания времени, и процесс формирования следующей пары обучающей выборки повторяется.

Для организации процесса обучения моделей нейронных сетей удобно использовать классы, наследуемые от класса **DataSet** библиотеки PyTorch. Для этого создадим класс, который назовем **DatasetSin**, и в качестве родительского класса укажем класс **DataSet**. В этом классе нам необходимо создать три метода. Первый метод – это конструктор, в него мы передаем весь временной ряд и также переменную, sequenceLength, которая задает длину скользящего окна. Эти данные будут хранится в виде внутренних переменных класса. Следующая функция, которую мы должны также обязательно добавить в этот класс – это функция len, возвращающая общее количество обучающих векторов (см. рис. 56).

Получение отдельной обучающей пары реализовано в методе **getitem**. Здесь в переменную data_sequence записывается отрезок данных длиной sequenceLength, а в переменную next_value записываем следующее значение временного ряда. Значения этих переменных образуют обучающую пару и возвращаются из этого метода. (см. рис. 56).

Описание класса DatasetSin

```
class DatasetSin(Dataset):
    def __init__(self, data, sequenceLength):
        self.data = data
        self.sequenceLength = sequenceLength
    def __len__(self):
        return
int(torch.floor(torch.tensor(len(self.data)/self.sequenceLength)
))

    def __getitem__(self, index):
        data_sequence =
self.data[index:index+self.sequenceLength]
        next_value = self.data[index+self.sequenceLength+1]
        return data_sequence, next_value
```

Рис. 56. Описание класса DatasetSin

Далее мы создаем объект класса **DatasetSin** и сохраняем его в переменную **datasetTrain**. В конструкторе передаем наш PyTorch тензор, содержащий данные обучающей выборки. Далее, на основе этого датасета мы создаем объект класса **DataLoader**, который позволит нам организовать процесс итеративного обучения модели нейронной сети. Мы можем проверить, как работает этот объект, организовав циклическую итерацию через конструкцию for. Здесь входной вектор будет возвращаться в виде тензора x_data, а соответствующие им целевые значения в виде вектора y_data. Мы можем вывести эти значения на консоль, используя стандартный метод **print** (см. рис. 57).

Создание объектов Dataset и DataLoader для обучающей выборки

```
datasetTrain = DatasetSin(sin_data_train, sequenceLength)

dataLoaderTrain = DataLoader(datasetTrain,
batch_size=batchSize, shuffle=True)
#вывод на консоль
for x_data,y_data in dataLoaderTrain:
    print(x_data)
    print(y_data)
    break
```

Создание объектов Dataset и DataLoader для тестовой выборки

```
datasetTest = DatasetSin(sin_data_test, sequenceLength)
dataLoaderTest= DataLoader(datasetTest,
batch_size=batchSize, shuffle=True)
```

Рис. 57. Подготовка данных

Здесь мы также создаем аналогичные объекты и для тестового набора данных. Результаты вывода на консоль значений тензоров x_data и y_data показаны на рисунке 58.

```
x_data=tensor([[ 0.4378,  0.4445,  0.5210,  ..., -0.8157, -0.7854, -0.7250],
        [ 0.4445,  0.5210,  0.6590,  ..., -0.7854, -0.7250, -0.5751],
        [ 0.6590,  0.7198,  0.7701,  ..., -0.5751, -0.7153, -0.6376],
        ...,
        [ 0.3791,  0.3045,  0.4919,  ..., -0.8983, -0.7477, -0.8781],
        [ 0.4919,  0.4378,  0.4445,  ..., -0.8781, -0.8157, -0.7854],
        [ 0.1618,  0.1556,  0.2725,  ..., -0.9713, -0.8380, -0.7906]])
y_data=tensor([-0.7153, -0.6376, -0.4361, -0.4186, -0.4418, -0.8781, -0.8157, -0.6542,
        -0.7250, -0.9172, -0.3447, -0.1280, -0.3138, -0.7477, -0.3259, -0.1899,
        -0.4090, -0.7854, -0.5751, -0.8983])
```

Рис. 58. Вывод на консоль

Далее необходимо создать класс, описывающие нейронные сети. Создадим два вида рекуррентных нейронных сетей. Первая сеть будет основана на классических рекуррентных нейронах, слои которых будет добавляться как RNN слои из модуля nn Pytorch, а вторая нейронная сеть будет LSTM нейронной сетью, и для ее создания используем LSTM слой библиотеки PyTorch.

Описание класса сети

```
class RNN(nn.Module):
    def __init__(self, inputSize, hiddenSize, numLayers):
        super(RNN,self).__init__()
        self.RNN = nn.RNN(input_size=inputSize,
                          hidden_size=hiddenSize,
                          num_layers=numLayers,
                          nonlinearity='tanh',
                          batch_first=True)
        self.linear = nn.Linear(hiddenSize,1)
    def forward(self,x,hState):
        # прямой проход через RNN слой
        x, h = self.RNN(x,hState)
        #прямой проход через выходной слой
        y_pred = self.linear(x[:,-1,:])
        return y_pred
```

Создание экземпляра нейронной сети

```
modelRNN = RNN(1,hiddenSize,num_layers)
```

Вывод информации об архитектуре НС на консоль

```
print(modelRNN)
```

Рис. 59. Создание рекуррентной нейронной сети на основе слоя RNN

На рис. 59 показано создание нейронной сети на основе слоя RNN. Эту нейронную сеть мы создаем в виде класса, который наследуются от класса **nn.Module**. В конструкторе класса мы описываем структуру нейронной сети. Мы создадим нейронную сеть, состоящую из рекуррентного RNN слоя и выходного линейного слоя. При добавлении рекуррентного RNN слоя мы указываем размерность входного вектора, количество рекуррентных слоев, функцию активации. Данные параметры передаются как аргументы конструктора при создании конкретной модели нейронной сети. На выходе нейронной сети добавляем линейный слой со входной размерностью, равной скрытой размерности (hiddenSize) рекуррентного слоя, с выходной размерностью равной единице, т. к. мы прогнозируем только одно значение.

В методе **forward()** мы описываем последовательность вычислений, выполняемых нейронной сетью. Входной тензор передается в метод через переменную x, также в этот метод мы передаем тензор с данными для инициализации скрытого состояния нейронов рекуррентного слоя. Сначала входной вектор обрабатывается рекуррентным слоем. Результатом обработки рекуррентного слоя является выходное значение, которое мы записываем в переменную x, а также тензор со значениями скрытого состояния. И наконец выходное значение поступает на последний слой, который является линейным слоем и возвращается из метода **forward**().

Для того, чтобы использовать нейронную сеть, мы создаем объект класса нейронной сети, которая будет храниться в переменной **modelRNN**. В конструктор передаем следующие параметры: размерность входа, размерность скрытого слоя, и количество слоев. С помощью метода print мы выводим информацию об архитектуре нейронной сети на консоль (см. рис. 59).

Рекуррентная нейронная сеть на основе LSTM слоя создается аналогично (см. рис. 60). В конструкторе класса кроме описания слоев нейронной сети мы также записали некоторые входные параметры в качестве внутренних переменных. Необходимо отметить, что в отличие от слоя RNN, в слое LSTM для хранения скрытого состояния необходимо два тензора. Здесь мы добавили отдельный метод для инициализации скрытых состояний. Этот метод будет вызываться во время обучения нейронной сети. Метод **forward()** устроен аналогично методу **forward()** для классической рекуррентной нейронной сети, созданной на основе слоя RNN. Аналогично мы создаем экземпляр модели этой нейронной сети, записав ее в переменную **modelLSTM**, и выводим информацию об архитектуре нейронной сети на консоль.

Описание класса LSTM сети

```
class LSTM(nn.Module):
    def __init__(self, inputSize, hiddenSize, batch_size,num_layers):
        super(LSTM, self).__init__()
        self.hidden_dim = hiddenSize
        self.batch_size = batch_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size=inputSize, hidden_size=hiddenSize,
                            num_layers=num_layers, batch_first=True)
        self.linear = nn.Linear(self.hidden_dim, 1)
    def init_hidden(self):
        return (torch.zeros(self.num_layers, self.batch_size, self.hidden_dim),
                torch.zeros(self.num_layers, self.batch_size, self.hidden_dim))
    def forward(self, input):
        lstm_out, self.hidden = self.lstm(input)
        y_pred = self.linear(lstm_out[:,-1,:])
        return y_pred
```

Создание экземпляра RNN нейронной сети

```
modelLSTM = LSTM(1, hiddenSize, batch_size=batchSize, num_layers=num_layers)
```

Вывод информации об архитектуре НС на консоль

```
print(modelLSTM)
```

Рис. 60. Создание LSTM нейронной сети

Далее рассмотрим процесс обучения рекуррентных нейронной сетей, созданных выше. Необходимо сказать, что процесс обучения таких сетей аналогичен процессу обучения полно-связных нейронный сетей, поэтому мы не будем подробно описывать этот процесс. В качестве оптимизатора выберем оптимизатор Адама, а в качестве функции потерь используем средне-квадратичную ошибку. Далее переводим модель в режим обучения вызвав его метод **train**(). Будем записывать динамику значения среднеквадратичной ошибки в переменной lossHistory (см. рис. 61).

Процесс обучения организован в виде двух вложенных циклов. Первый, внешний цикл – это цикл эпох обучения. Количество эпох обучения мы задали в самом начале программы. Внутренний цикл – это цикл обучения, организованный через объект класса **DataLoader**. Внутри этого цикла **DataLoader** получает набор обучающий пар, количество которых равно размеру пачки, и подает входной тензор из этой пары на вход нейронной сети.

Далее вычисляется функция потерь на основе сравнения результатов, спрогнозированных нейронной сетью и целевых значений. Затем выполняется процедура обратного распространения ошибки вызовом метода **backward()**. Вызов метода **step()** оптимизатора осуществляет настройку параметров нейронной сети. Для каждой эпохи мы вычисляем среднеквадратичную ошибку и добавляем это значение в список ошибок, для дальнейшей визуализации (см. рис. 61).

Обучение НС

```
loss_function = nn.MSELoss()
optimizerLSTM = torch.optim.Adam(modelRNN.parameters(), lr=learningRate)
modelRNN.train()
lossHistory = []
for epoch in range(epochs):
    lossTotal = 0
    for x_data,y_data in dataLoaderTrain:
        hddenState = torch.zeros([num_layers, batchSize, hiddenSize])
        y_pred= modelRNN(x_data.reshape([batchSize, sequenceLength, 1]),hddenState)
        loss = loss_function(y_pred.view(-1),y_data)
        modelRNN.zero_grad()
        loss.backward()
        optimizerLSTM.step()
        lossTotal +=loss
    lossHistory.append(lossTotal.item())
```

Рис. 61. Обучение нейронной сети RNN

Для визуализации динамики среднеквадратичной ошибки мы можем построить ее в виде графика. По графику видно, что в самом начале процесса обучения, значения функции потерь очень велики, но постепенно это значение уменьшается. Во время обучения среднеквадратичная ошибка не всегда будет спадать монотонно, могут наблюдаться и всплески (см. рис. 62).

```
plt.plot(lossHistory)
plt.title('Функция потерь')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```

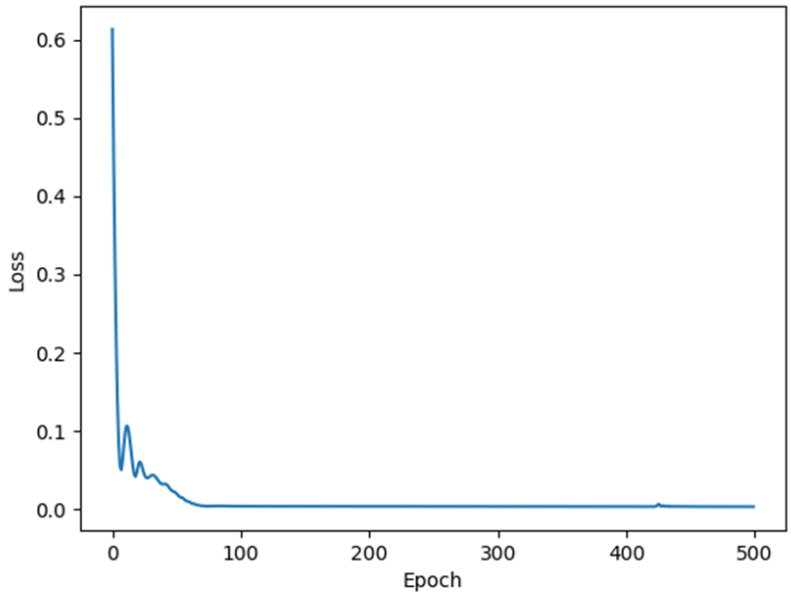


Рис. 62. Визуализация процесса обучение нейронной сети на основе RNN

Процесс обучения рекуррентной сети на основе LSTM слоев выглядит аналогично. Здесь отличается только название модели, а также добавляется процесс инициализации скрытого слоя (см. рис. 63).

```
loss_function = nn.MSELoss()
optimizerLSTM = torch.optim.Adam(modelLSTM.parameters(), lr=learningRate)
modelLSTM.train()
lossHistory = []
for epoch in range(epochs):
    lossTotal = 0
    for x_data,y_data in dataLoaderTrain:
        modelLSTM.hidden = modelLSTM.init_hidden()
        y_pred= modelLSTM(x_data.reshape([batchSize, sequenceLength, 1]))
        loss = loss_function(y_pred.view(-1),y_data)
        modelLSTM.zero_grad()
        loss.backward()
        optimizerLSTM.step()
        lossTotal +=loss
    lossHistory.append(lossTotal.item())
```

Рис. 63. Обучение нейронной сети LSTM

Качество модели оценим на тестовых данных, используя **DataLoader** для тестовых данных **dataLoaderTest**. Здесь мы оценим точность модели основанной на LSTM, для RNN все делается аналогично. Вначале переведем модель в режим оценки вызовом метода **eval()**. Прогноз выполняется с использованием обученной модели. По горизонтальной оси отображается целевое значение, по вертикальной – прогнозируемое значение (см. рис. 64).

Прогноз на тестовой выборке

```
modelLSTM.eval()
for x_data,y_data in dataLoaderTest:
    modelLSTM.hidden = modelLSTM.init_hidden()
    y_pred= modelLSTM(x_data.reshape([batchSize,
sequenceLength, 1]))
    print(y_pred)
    plt.figure(figsize=(5, 5), dpi=100)
    plt.xlabel("Y")
    plt.ylabel("Predicted Y")
    # Конвертация тензоров в numpy массивы
    y_data = y_data.detach().numpy()
    y_pred = y_pred.detach().numpy()
    plt.scatter(y_data, y_pred, lw=1, color="b", label="test")
    plt.legend()
    plt.show()
```

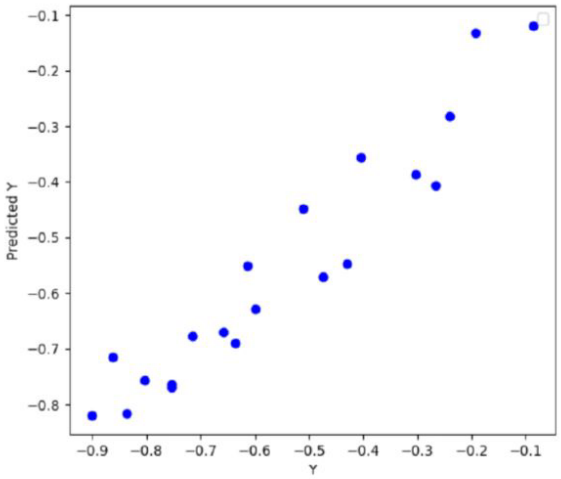


Диаграмма рассеяния

Рис. 64. Графическое отображение результатов обучения рекуррентной нейронной сети

Можно самостоятельно экспериментировать с этой программой, задавая различные параметры касательно генерируемого временного ряда, а также параметры обучения. Уменьшение уровня шума должно улучшать качество прогноза модели, в увеличение шума должно ухудшать его. Для того чтобы продемонстрировать это, проведем обучение модели и построим диаграммы рассеяния для двух случаев: первый – при полном отсутствии шума, и второй для уровня шума, равного 0.2. На рис. 65 отображены эти результаты. Результаты показывает, что если мы прогнозируем на чистой синусоиде без шума, то все точки лежат на главной диагонали, это означает, что прогнозируемые данные очень близки к реальным. При уровне шума больше нуля разброс точек относительно главной диагонали усиливается.

В заключение необходимо сказать, что программу, разработанную в этой главе, вы можете использовать для прогнозирования различных временных рядов, включая также и финансовые временные ряды.

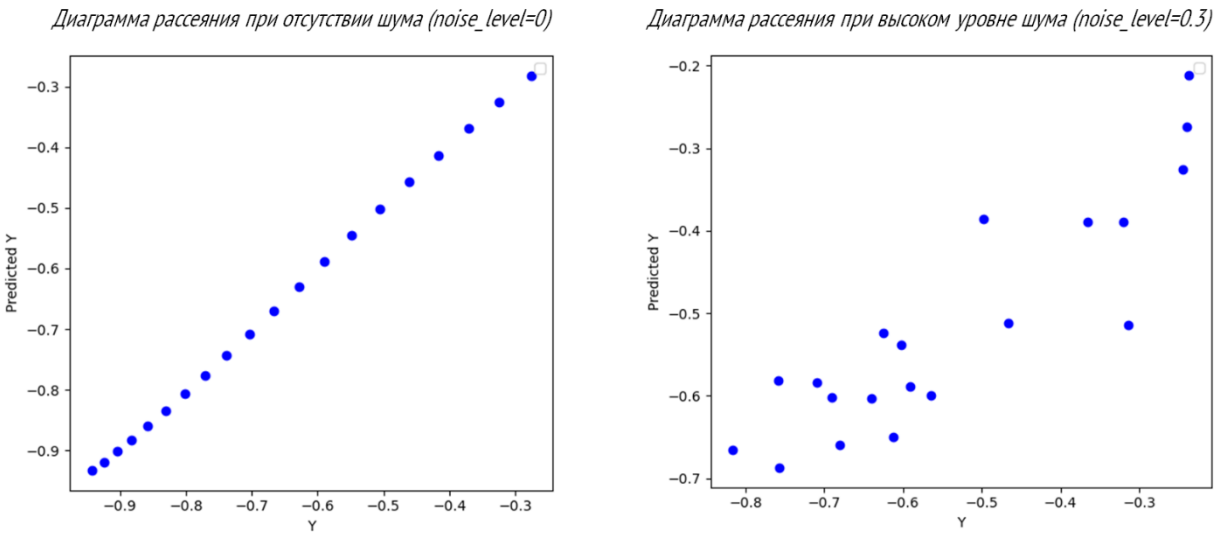


Рис. 65. Графическое отображение результатов

Вопросы

1. Что такое временные ряды и почему они важны в анализе данных?

2. Какие задачи можно решать с использованием рекуррентных нейронных сетей в прогнозировании временных рядов?

3. Какие характеристики временных рядов влияют на выбор методов и моделей для прогнозирования?

4. Как организовать временные ряды для использования в обучении рекуррентных нейронных сетей?

5. Как реализовать разделение временных рядов на обучающий и тестовый наборы данных?

6. Как использовать окна (windows) для подготовки данных временных рядов для обучения рекуррентных нейронных сетей?

7. Как использовать LSTM (Long Short-Term Memory) для прогнозирования временных рядов?

8. Какие метрики оценки эффективности модели прогнозирования временных рядов можно использовать?

9. Как провести обработку и анализ данных временных рядов перед использованием их в рекуррентной нейронной сети?

ГЛАВА 7. Сверточные нейронные сети и их приложения

Свёрточная нейронная сеть (convolutional neural network, CNN) – специальная архитектура искусственных нейронных сетей, нацеленная на эффективное распознавание изображений. Основная архитектурная особенность сверточных нейронных сетей заключается в чередовании двух типов слоев: сверточных слоев (convolution layers) и субдискретизирующих слоев (subsampling layers или pooling layers). Такая архитектура нейросети хорошо улавливает локальный контекст, когда информация в пространстве непрерывна, то есть её носители находятся рядом. Например, пиксели – части изображения, которые расположены близко друг к другу и содержат визуальные данные: яркость и цвет. Это принципиально многослойные и однонаправленные (без обратных связей) нейронные сети, для обучения которых используются стандартные методы, чаще всего метод обратного распространения ошибки. Необходимо отметить, что идея сверточных нейронных сетей основана на некоторых особенностях зрительной коры, в которой были открыты так называемые простые клетки, реагирующие на прямые линии под разными углами, и сложные клетки, реакция которых связана с активацией определенного набора простых клеток.

Название эта архитектура сети получила из-за наличия операции свертки, суть которой в том, что каждый фрагмент изображения умножается на матрицу (ядро) свертки поэлементно, а результат суммируется и записывается в аналогичную позицию выходного изображения (см. рис. 66).

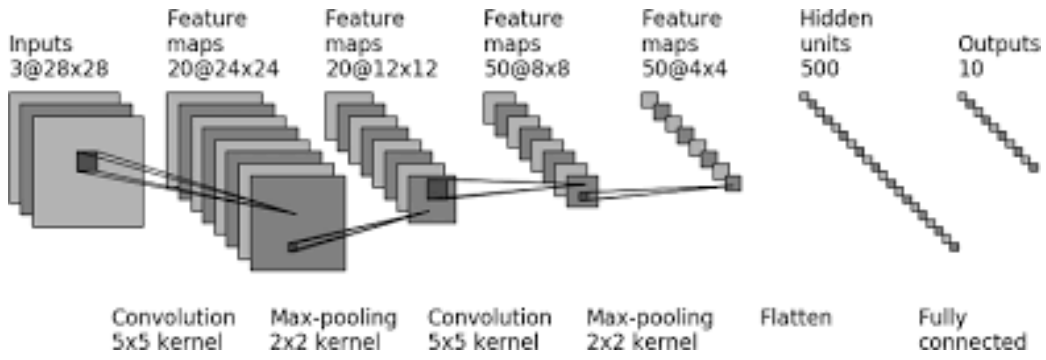


Рис. 66. Сверточная нейронная сеть

В операции свёртки используется ограниченная матрица весов небольшого размера, которую «двигают» по всему обрабатываемому слою (в самом начале – непосредственно по входному изображению), формируя после каждого сдвига сигнал активации для нейрона следующего слоя с аналогичной позицией. Для различных нейронов выходного слоя используется одна и та же матрица весов, которую также называют ядром свёртки. Ядро свёртки интерпретируют как графическое кодирование какого-либо признака, например, наличие наклонной линии под определенным углом.

Следующий слой, получившийся в результате операции свертки такой матрицей весов, показывает наличие данного признака в обрабатываемом слое и его координаты, формируя так называемую карту признаков (feature map). При этом такие ядра свертки не закладываются исследователем заранее, а формируются самостоятельно путем обучения сети классическим методом обратного распространения ошибки. Проход каждым набором весов формирует свой собственный экземпляр карты признаков, делая нейронную сеть многоканальной (много независимых карт признаков на одном слое).

Таким образом слой свертки – это основной блок сверточной нейронной сети. Слой свёртки включает в себя для каждого канала свой фильтр, ядро свертки которого обрабатывает предыдущий слой по фрагментам (суммируя результаты матричного произведения для каждого фрагмента). Весовые коэффициенты ядра свертки (небольшой матрицы) неизвестны и устанавливаются в процессе обучения. Особенностью сверточного слоя является сравнительно небольшое количество параметров, устанавливаемое при обучении. Например, если исходное изображение имеет размерность 100×100 пикселей по трём каналам (это значит – 30000 входных нейронов), а сверточный слой использует фильтры с ядром 3x3 пикселя с выходом на 6 каналов, тогда в процессе обучения определяется только 9 весов ядра, однако по всем сочетаниям каналов, то есть 9×3×6 =162, в таком случае данный слой требует нахождения только 162 параметров, что существенно меньше количества искомых параметров полносвязной нейронной сети (см. рис. 67).

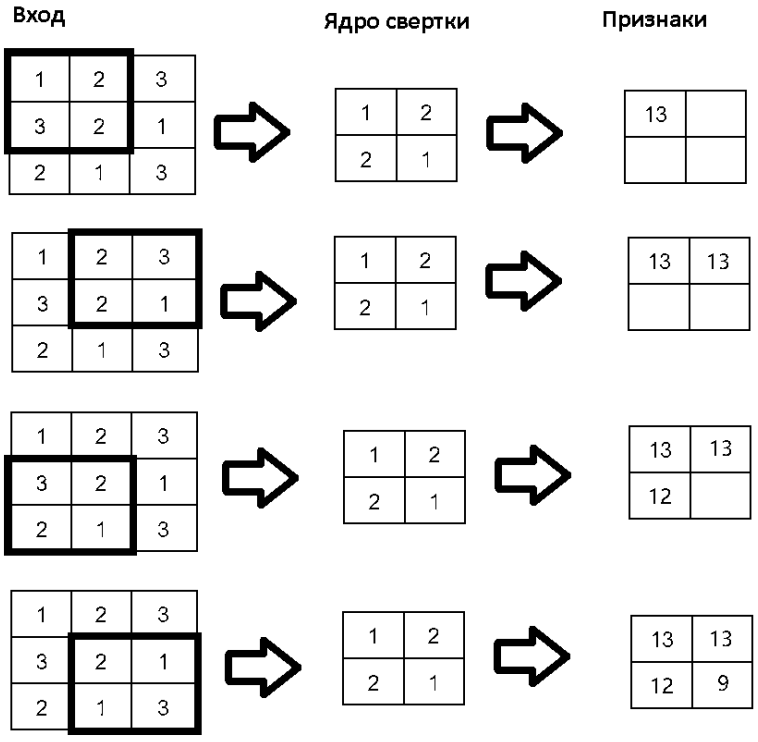


Рис. 67. Графическая изображения операций сверточного слоя

После сверточных слоев обычно используется слой ReLU (rectified linear unit), который графически выглядит как линейная функция с нулевым отсечением на оси абсцисс в точке 0 (см. рис. 68). Это значит, что функция имеет постоянный наклон во всех точках, кроме точки 0, где происходит отсечение. Такая функция показывает хорошие результаты при обучении нейронных сетей и отвечает за отсечение ненужных деталей в канале (например, при отрицательном выходе).

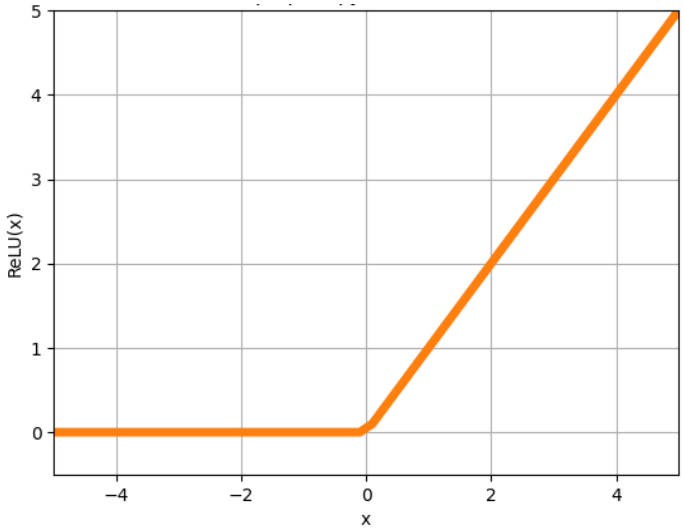


Рис. 68. График функции ReLU

Сверточные нейронные сети также содержат слои пулинга (подвыборки, субдискретизации), которые выполняют нелинейное уплотнение карты признаков, при этом группа пикселей (обычно размера 2×2) уплотняется до одного пикселя, проходя нелинейное преобразование (см. рис. 69). Преобразования затрагивают непересекающиеся прямоугольники или квадраты, каждый из которых ужимается в один пиксель, при этом выбирается пиксель, имеющий максимальное значение. Операция пулинга позволяет существенно уменьшить пространственный объём изображения. Пулинг интерпретируется следующим образом. Если на предыдущей операции свёртки уже были выявлены некоторые признаки, то для дальнейшей обработки настолько подробное изображение уже не нужно, и оно уплотняется до менее подробного. Слой пулинга, как правило, вставляется после слоя свёртки перед слоем следующей свёртки. В целях более агрессивного уменьшения размера получаемых представлений всё чаще находят распространение идеи использования меньших фильтров или полный отказ от слоёв пулинга.

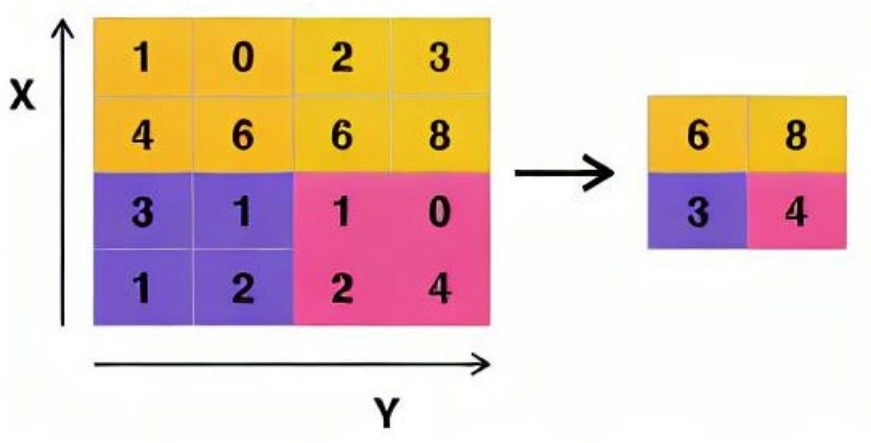


Рис. 69. Пулинг с функцией максимума и фильтром 2×2 с шагом 2

После нескольких прохождений свёртки изображения и уплотнения с помощью пулинга система перестраивается от конкретной сетки пикселей с высоким разрешением к более абстрактным картам признаков; как правило, на каждом следующем слое увеличивается число каналов и уменьшается размерность изображения в каждом канале. В конце концов остаётся большой набор каналов, хранящих небольшое число данных, которые интерпретируются как самые абстрактные понятия, выявленные из исходного изображения. Эти данные объединяются и передаются на обычную полносвязную нейронную сеть, которая тоже может состоять из нескольких слоёв (см. рис. 70). При этом полносвязные слои уже утрачивают пространственную структуру пикселей и обладают сравнительно небольшой размерностью (по отношению к количеству пикселей исходного изображения).

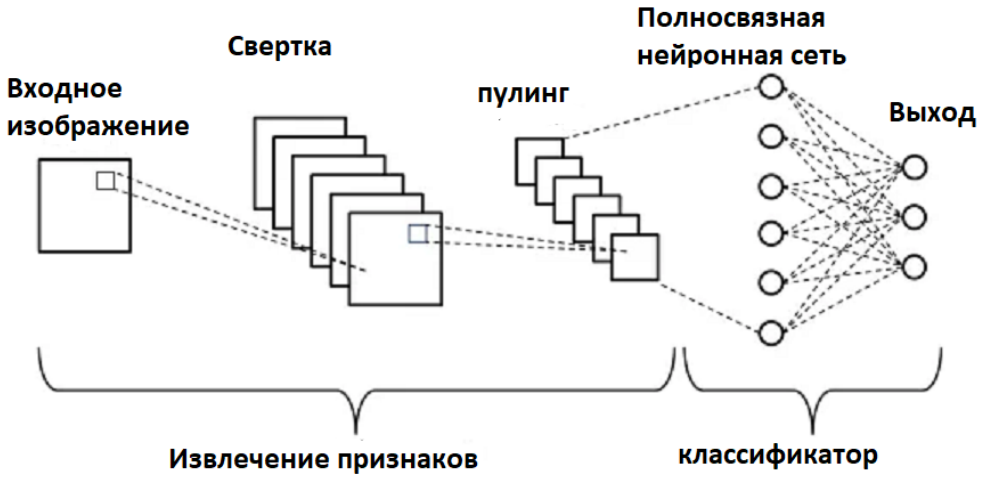


Рис. 70. Схема сверточной нейронной сети

Работа сверточной нейронной сети обычно интерпретируется как переход от конкретных особенностей изображения к более абстрактным деталям и, далее, к ещё более абстрактным деталям вплоть до выделения понятий высокого уровня. При этом сеть самонастраивается и вырабатывает сама необходимую иерархию абстрактных признаков (последовательности

карт признаков), фильтруя маловажные детали и выделяя существенное. Фактически «признаки», вырабатываемые сложной сетью, малопонятны и трудны для интерпретации (см. рис. 71). В практических системах не особенно рекомендуется пытаться понять содержание этих признаков или пытаться их «подправить», вместо этого рекомендуется усовершенствовать саму структуру и архитектуру сети чтобы получить лучшие результаты.

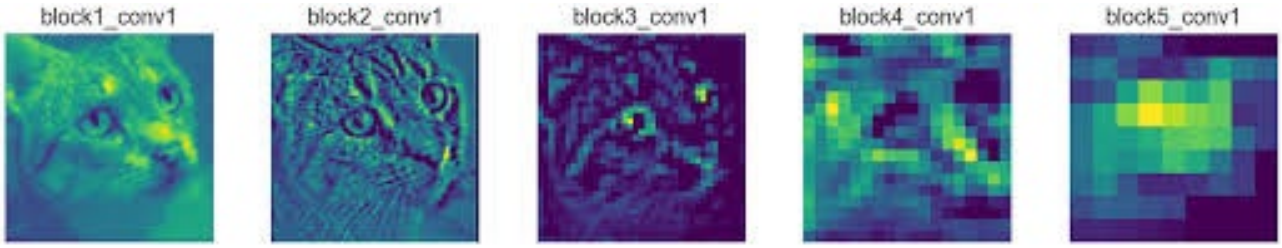


Рис. 71. Интерпретация работы свёрточной нейронной сети

Наиболее простым и популярным способом обучения является метод обучения с учителем (на размеченных данных) – метод обратного распространения ошибки и его модификации. Также можно скомбинировать сверточную нейросеть с другими технологиями глубинного обучения. Для улучшения работы сети, повышения её устойчивости и предотвращения переобучения применяется также исключение (дропаут) – метод тренировки подсети с выбрасыванием случайных одиночных нейронов.

В PyTorch для создания сверточных нейронных сетей (CNN) обычно используются различные типы слоев, предназначенных для обработки изображений и извлечения признаков. Основной слой сверточной нейронной сети, который выполняет операцию свертки – это слой **torch.nn.Conv2d**. Добавляется в нейронную сеть следующим образом: **conv_layer = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding).** Основные параметры этого слоя это:

in_channels – количество входных каналов. Например, для цветных изображений RGB это будет 3.

out_channels – количество выходных каналов (или фильтров). Каждый из этих фильтров изучает различные признаки входных данных.

kernel_size – размер ядра свертки, он определяет размер фильтра, который скользит по входным данным. Может быть задано как одно число (для квадратного ядра) или кортеж из двух чисел (ширина и высота ядра).

Stride – шаг (или сдвиг) ядра при сканировании входных данных, который определяет, насколько далеко смещается ядро при каждом шаге. Может быть задано как одно число или кортеж из двух чисел.

Padding – добавляет нулевые значения вокруг краев входных данных, что может быть полезно для избегания потери информации по краям. Может быть задано как одно число или кортеж из двух чисел.

В качестве примера создадим и обучим сверточную нейронную сеть решению задачи классификации изображений. В качестве набора данных для обучения будет использован известный датасет MNIST. Этот набор данных представляет собой коллекцию изображений рукописных цифр от 0 до 9, написанных различными людьми, и является своего рода стандартным бенчмарком для оценки производительности алгоритмов распознавания изображений. Все изображения представляют собой черно-белые изображения размером 28x28 пикселей, разделенные на 10 классов, соответствующих цифрам от 0 до 9. Всего имеется 60 000 изображений для обучения и 10 000 для тестирования.

На рис. 72 и рис. 73 показана реализация сверточной нейронной сети в PyTorch (пример расположен по адресу:
https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/cnn_mnist.ipynb).

Аналогично полносвязным и рекуррентным нейронным сетям, сверточная нейронная сеть создается виде класса, который наследуется от **nn.Module**. В конструкторе класса создается ее архитектура в виде последовательности необходимых слоев.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

Рис. 72. Реализация сверточной нейронной сети в PyTorch

В нашем примере нейронная сеть содержит два слоя дропаут, и два линейных слоя для решения задачи классификации изображений. В методе forward реализуется выполнение различных вычислительных операций над входным признаком: прохождение через слои свертки, вычисления функции активации **(F.relu(x)),** выполнение операции пуллинга **(F.max_pool2d(x,2)),** прохождение через дропоут слои и вычисления функции **softmax (F.softmax(x,dim=1))**. Размерность выходного тензора последнего слоя равно 10, что соответствует количеству классов рукописных цифр.

Создадим отдельный метод **train**, в котором организуем процесс обучения модели (рис. 73).

```
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Рис. 73. Реализация метода **train**

Также создаем отдельный метод test для вычисления точности классификации модели (рис. 74). В этом методе на вход модели подаются изображения из тестового набора данных, через **test_loader,** результаты прогноза модели сравниваются с метками классов. Точность модели выводится на экран в виде процента правильно угаданных меток классов методом **print()**.

```
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

Рис. 74. Реализация метода **test**

Далее создадим основную часть программы, которая собирает в эти методы в единую систему (рис. 75). В самом начале выбираем тип вычислительного устройства CUDA или CPU. Если есть возможность, то рекомендуется использовать CUDA для ускорения процесса обучения модели. Загружаем данные MNIST, используя для этого стандартные методы Pytorch, и на их основе создаем соответствующие объекты классов **DataLoader** для тренировочного (**train_loader**) и тестового (**test_loader**) набора данных. Далее создаем объект класса **Net** -нейронной сети, который был создан выше, и организуем стандартный процесс обучения модели в количестве 9-ти эпох.

```
use_cuda =True
if use_cuda:
  device = torch.device("cuda")
else:
  device = torch.device("cpu")
batch_size=128
transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
        ])
dataset1 = datasets.MNIST('../data', train=True, download=True, transform=transform)
dataset2 = datasets.MNIST('../data', train=False, transform=transform)
train_loader = torch.utils.data.DataLoader(dataset1,batch_size=batch_size)
test_loader = torch.utils.data.DataLoader(dataset2,batch_size=batch_size)

model = Net().to(device)

optimizer = optim.Adadelta(model.parameters(), lr=0.01)

for epoch in range(1, 10):
    print('Epoch:'+str(epoch))
    train(model, device, train_loader, optimizer, epoch)
    test(model, device, test_loader)
```

Рис. 75. Обучение сверточной нейронной сети в PyTorch

Необходимо отметить, что сверточные нейронные сети применяются в различных областях, но наибольшую популярность они получили в обработке и анализе изображений. Наиболее часто встречаемые задачи: классификация изображений, выделение объектов на изображениях и сегментация сцены.

Сверточные нейронные сети обладают следующими преимуществами. Сверточные нейронные сети – это один из лучших алгоритмов по распознаванию и классификации изображений. По сравнению с полносвязной нейронной сетью (типа перцептрона) – гораздо меньшее количество настраиваемых весов, так как одно ядро весов используется целиком для всего

изображения вместо того, чтобы делать для каждого пикселя входного изображения свои персональные весовые коэффициенты. Удобное распараллеливание вычислений, а, следовательно, возможность реализации алгоритмов работы и обучения сети на графических процессорах. Относительная устойчивость к повороту и сдвигу распознаваемого изображения. Обучение при помощи классического метода обратного распространения ошибки.

Существуют также и недостатки сверточных нейронных сетей. Слишком много варьируемых параметров сети, непонятно, для какой задачи и вычислительной мощности какие нужны настройки. К параметрам можно отнести: количество слоев, размерность ядра свёртки для каждого из слоёв, количество ядер для каждого из слоёв, шаг сдвига ядра при обработке слоя, необходимость слоев субдискретизации, степень уменьшения ими размерности, функция по уменьшению размерности (выбор максимума, среднего и т. п.), передаточная функция нейронов, наличие и параметры выходной полносвязной нейросети на выходе сверточной. Все эти параметры существенно влияют на результат, но выбираются исследователями эмпирически.

Вопросы

1. Что такое сверточные нейронные сети (CNN) и какова их основная идея?
2. Какие слои часто включаются в архитектуру сверточных нейронных сетей?
3. Как работают сверточные слои в сверточных нейронных сетях?
4. Как используются ядра свертки для извлечения признаков из входных данных?
5. Как происходит уменьшение размерности данных при использовании сверточных слоев?
6. Какие функции активации часто применяются в сверточных нейронных сетях?
7. Как реализовать пулинг слои в сверточных нейронных сетях и как они влияют на размерность данных?
8. Каковы преимущества сверточных нейронных сетей по сравнению с полносвязными сетями для обработки изображений?
9. Как провести обучение сверточных нейронных сетей для задачи классификации изображений?
10. Как решить проблему переобучения в сверточных нейронных сетях?
11. Как создать сверточный слой в PyTorch?
12. Как определить архитектуру сверточной нейронной сети в PyTorch?

13. Как задать параметры ядра свертки в сверточном слое PyTorch?

14. Как добавить несколько сверточных слоев в модель в PyTorch?

15. Как использовать пулинг слои в сверточной нейронной сети с помощью PyTorch?

16. Как использовать сверточные нейронные сети для задачи классификации изображений в PyTorch?

17. Как добавить dropout слой в сверточную нейронную сеть в PyTorch?

ГЛАВА 8. Графовые нейронные сети и их приложения

Графовая нейронная сеть (Graph Neural Network, GNN) – тип нейронной сети, которая напрямую работает со структурой графа. Использование GNN позволяет работать с данными графов, без предварительной обработки. Такой подход позволяет сохранить топологические отношения между узлами графа.

Самая фундаментальная часть графовых нейронных сетей – это граф. Граф – это структура данных, состоящая из двух компонентов: узлов (вершин) и ребер (см. рис. 76). То есть граф G можно определить как G = (V, E), где V – множество узлов, а E – ребра между ними. Если между узлами есть направленные зависимости, то ребра являются направленными. Если нет, то ребра ненаправленные. Граф может представлять собой такие вещи, как социальные сети или молекулы. В примере социальных сетей узлы – это пользователи, а ребра – связи между пользователями.

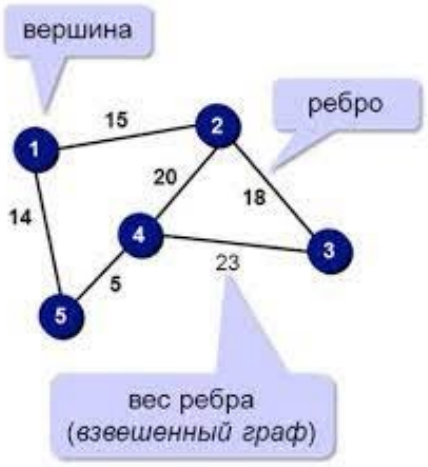


Рис. 76. Основные элементы графов

Графы часто представляется с помощи матрицы смежности. Если граф имеет n узлов, то матрица смежности имеет размерность (n × n) (см. рис. 77).

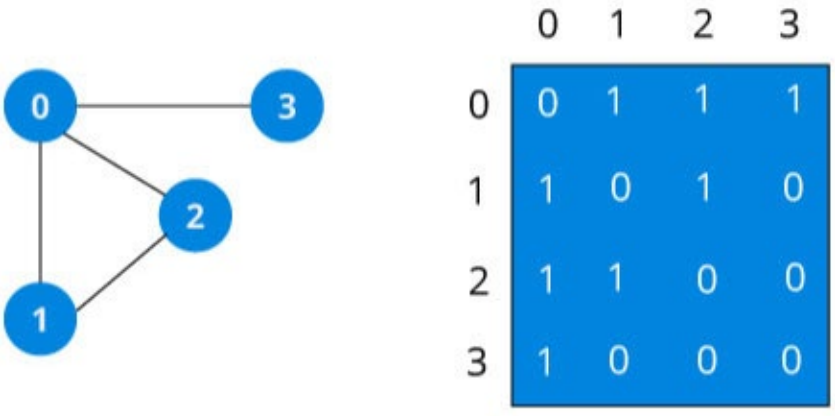


Рис. 77. Матрица смежности

Иногда с узлами ассоциируются некоторые наборы данных (например, информация о профиле пользователя социальной сети). Если в узле есть f признаков, то матрица признаков узла X имеет размерность (n × f) (рис. 78).

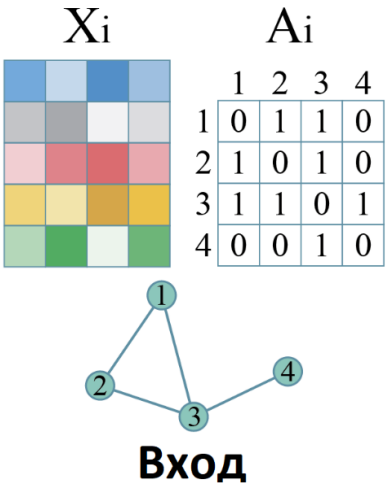


Рис. 78. Матрица смежности

Графически структурированные данные присутствуют повсюду. Задачи, которые решают графовые нейронные сети, можно разделить на следующие категории:

Классификация графов. Задача состоит в том, чтобы классифицировать весь граф по разным категориям. Это похоже на классификацию изображений. Приложения классификации графов многочисленны, например, от задачи определить, является ли белок ферментом или нет в биоинформатике, до классификации документов в НЛП или анализа социальных сетей.

Классификация узлов. Задача состоит в том, чтобы определить характеристики узлов на основе их соседей. Обычно нейронные сети при решении этого типа задач обучаются полуконтролируемым образом, при этом маркируется только часть графа.

Прогнозирование связей. При решении этой задачи алгоритм должен понимать взаимосвязь между объектами на графах, а также пытается предсказать, есть ли связь между двумя объектами. Например, в анализе социальных сетей важно делать выводы о социальных взаимодействиях или предлагать пользователям возможных друзей. Этот алгоритм также используется в задачах рекомендательных систем, и т. д.

Кластеризация графов. Эта задача кластеризации данных, представленных в виде графов. Существуют две различные формы кластеризации данных графа. Кластеризация вершин стремится сгруппировать узлы графа в группы плотно связанных областей на основе либо

весов ребер, либо расстояний между ребрами. Вторая форма кластеризации графов рассматривает графы как объекты, подлежащие кластеризации, и группирует эти объекты на основе сходства.

Вложение (embedding) графов. Это подход, который используется для преобразования узлов, ребер и их элементов в векторное пространство, т. е. в более низкое измерение, с максимальным сохранением таких свойств, как структура и информация о графе.

Генерация графов. Генерация графов полезна для понимания процесса формирования графа, выявления аномальных частей графа, прогнозирования новых структур, моделирования новых структур графа, в некоторых случаях – построения полного графа в случае частично доступных данных графа. Генерация синтетических графов имеет реальные примеры использования, например, создание нового типа молекулы, которая излечивает определенное заболевание, учитывая класс молекул, таких, как токсичные/нетоксичные, и т. д.

Графовые данные настолько сложны, что изначально создали множество проблем для существующих алгоритмов машинного обучения. Причина в том, что обычные инструменты машинного обучения и глубокого обучения специализируются на простых типах данных. Например, изображения с одинаковой структурой и размером мы можем рассматривать как сетки фиксированного размера. Текст и речь – это последовательности, поэтому мы можем представить их как линейный граф. Но есть и более сложные графы, без фиксированной формы, с переменным размером неупорядоченных узлов, где узлы могут иметь разное количество соседей (см. рис. 79). Также не помогает то, что существующие алгоритмы машинного обучения имеют основное предположение о том, что экземпляры данных независимы друг от друга. Это неверно для данных в виде графа, потому что каждый узел, может быть, связан с другими связями различных типов.

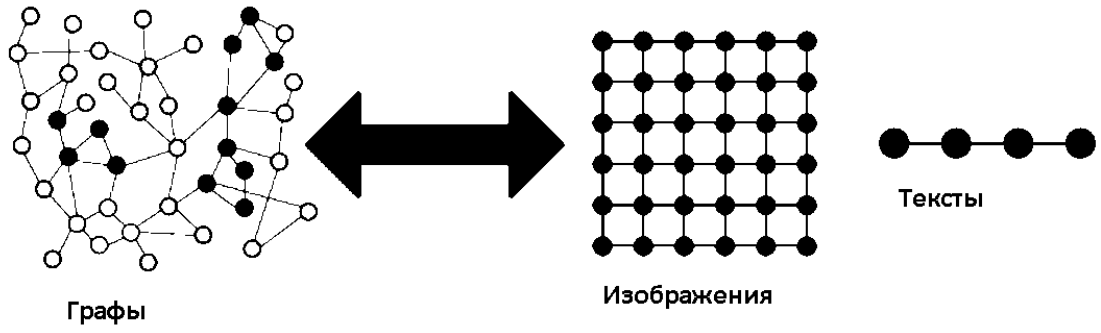


Рис. 79. Представление различных видов данных в виде графов

Рассмотрим основы глубокого обучения для графов на основе концепции векторного представления узлов. Это означает сопоставление узлов с d-мерным пространством (т. е. с малоразмерным пространством) таким образом, что похожие узлы в графе встраиваются близко

друг к другу в этом пространстве. Наша цель – отобразить узлы таким образом, чтобы сходство в векторном пространстве приближалось к сходству в сети. Функцией сходства может быть обычное евклидовое расстоянием. Определим **u** и **v** как два узла в графе. $\mathbf{x_u}$ и $\mathbf{x_v}$ – два вектора признаков, соответственно для узлов **u** и **v**. Теперь мы определим функцию кодировщика **Enc(u)** и **Enc(v)**, которые преобразуют векторы признаков в векторы $\mathbf{z_u}$ и $\mathbf{z_v}$ (см. рис. 80)

Необходимо отметить, что функция кодировщика должна иметь возможность выполнять:

1) локальные вычисление, т. е. выполнять вычисления для окрестности узла;
2) агрегировать информацию;
3) производить наложение и вычисление для нескольких слоев.

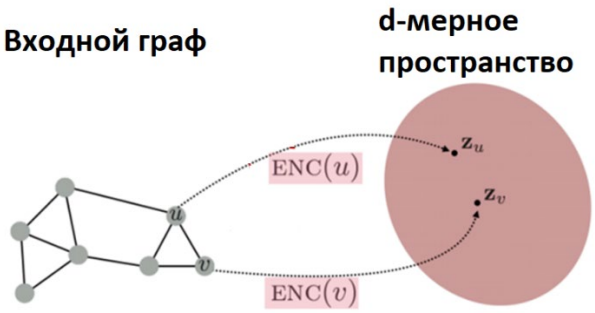


Рис. 80. Вычислительный граф

Информация о местоположении конкретного узла может быть получена с помощью вычислительного графа, так как он связан со своими соседями и соседями этих соседей. Мы знаем все возможные соединения и поэтому можем сформировать граф вычислений. Как только информация о местонахождении узла преобразована в вычислительный граф, мы можем производить агрегирование. В основном это делается с помощью обычных нейронных сетей. Эти нейронные сети представлены на рисунке 81 в виде серых прямоугольников. Операции агрегации должны быть инвариантны к порядку входных значений, например, это может быть сумма, среднее значение, максимум, потому что они являются инвариантными к перестановке функциями. Это свойство позволяет выполнять агрегирование.

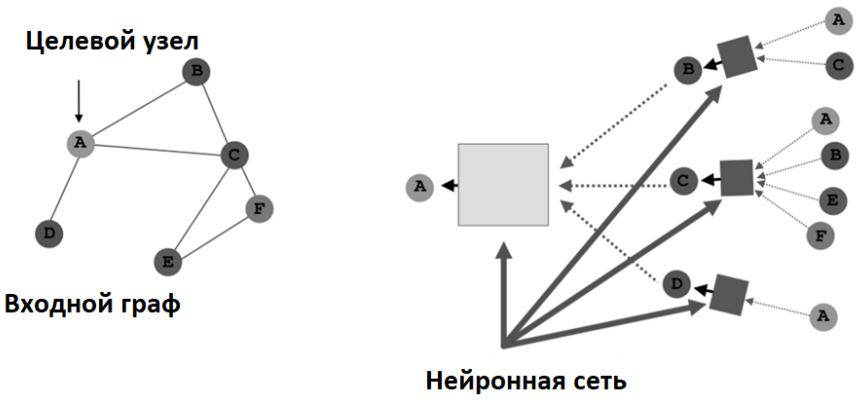


Рис. 81. Вычислительный граф

Рассмотрим алгоритм прямого распространения информации в многослойной графовой нейронной сети. Эта процедура вычислений определяет, как информация со входа будет поступать на выход нейронной сети. Каждый узел имеет вектор признаков. Например, $\mathbf{X_A}$ – это вектор признаков узла **A**. Входными данными являются эти векторы признаков, в этом случае блок вычислений на основе двух векторов признаков (X_A и X_c) объединит информацию о них и затем перейдет к следующему слою (см. рис. 82).

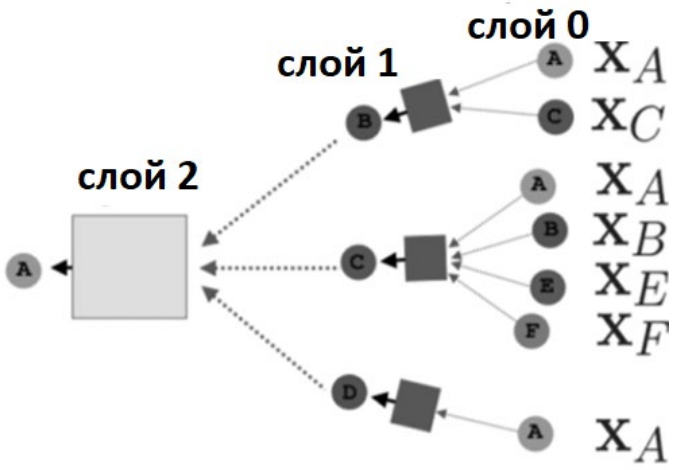


Рис. 82. Многослойная графовая нейронная сеть

Чтобы выполнить прямое распространение в этом вычислительном графе, нужно выполнить 3 шага:

1. Инициализация блоков активации:

$$h_v^0 = X_v\ (feature\ vector)$$

2. На втором шаге для каждого уровня сети вычисление проводятся по формуле:

$$h_v^k = \sigma(W_k \sum \frac{h_u^{k-1}}{\mid N(v)} + B_k h_v^{k-1})\ \text{where}\ k = 1, ..., k-1$$

Мы можем заметить, что это уравнение состоит из двух частей. Первая часть в основном усредняет по всем соседям узла v, здесь W_k – это матрица весовых коэффициентов k-го узла. Вторая часть вычисляет вложение предыдущего слоя узла **v**, умноженное на смещение B_k, которое представляет собой обучаемую весовую матрицу σ: нелинейная функция активации, которая выполняется для обоих частей.

3. Выходными значениями графовой нейронной сети являются значения h, вычисленные на последнем слое для каждого узла.

$$z_v = h_v^K$$

Это вложение после K слоев агрегации окрестности узлов. Теперь, чтобы обучить модель, нам нужно определить функцию потерь для векторных представлений. Далее мы можем передать их в любую функцию потерь и запустить стохастический градиентный спуск для

обучения весовых параметров. Таким образом, в основе GNN заложен механизм распространения информации. Граф обрабатывается набором модулей, которые связаны между собой в соответствии со связями графа. В процессе обучения модули обновляют свои состояния и обмениваются информацией. Это продолжается до тех пор, пока модули не достигнут устойчивого равновесия. Выходные данные GNN вычисляются на основе состояния модуля на каждом узле.

Построим модель простой графовой нейронной сети на основе PyTorch. Самая простая графовая нейронная сеть может состоять всего из трех разных последовательных операций: операции свертки графа, линейного слоя и нелинейной функции активации. Вместе они составляют один слой нейронной сети. Мы можем объединить эти слои в несколько слоев, чтобы сформировать многослойную графовую нейронную сеть. На рис. 83 представлена реализация простой графовой нейронной сети на языке Python на основе библиотеки PyTorch (https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/gnn_simple.ipynb).

```
import torch
from torch import nn
class GCN(nn.Module):
    def __init__(self, *sizes):
        super().__init__()
        self.layers = nn.ModuleList([nn.Linear(x, y) for x, y in zip(sizes[:-1], sizes[1:])])
    def forward(self, vertices, edges):
        # ----- Построение матрицы смежности -----
        # Диагональная матрица (соединение узла с собой)
        adj = torch.eye(len(vertices))
        # ребра содержат соединенные вершины: [узел_0, узел_1]
        adj[edges[:, 0], edges[:, 1]] = 1
        adj[edges[:, 1], edges[:, 0]] = 1
        # ----- Приямое распространие информации по слоям -----
        for layer in self.layers:
            vertices = torch.sigmoid(layer(adj @ vertices))
        return vertices
```

Рис. 83. Простая графовая нейронная сеть на языке Python

Для работы с графовыми нейронными сетями на основе фреймворка PyTorch разработана библиотека PyTorch Geometric. Эта библиотека предоставляет удобные средства для представления и обработки графов в задачах машинного обучения. Библиотеку PyTorch Geometric необходимо предварительно установить, выполнив команду «pip install torch_geometric». На рис. 84 продемонстрировано создание и использования графовой нейронной сети на основе библиотеки Torch geometric для классификации статей в графе цитирования (данный пример расположен по адресу:
https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/gnn_cora.ipynb).

Для решения этой задачи мы загружаем набор данных Cora и создаем простую двухслойную модель GCN, используя предопределенный слой GCNConv из библиотеки Torch geometric.

```python
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='/tmp/Cora', name='Cora')

class GCN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = GCNConv(dataset.num_node_features, 16)
        self.conv2 = GCNConv(16, dataset.num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)

        return F.log_softmax(x, dim=1)
```

Рис. 84. Создание архитектуры графовой нейронной сети с помощью PyTorch Geometric

В библиотеке PyTorch Geometric слой GCNConv представляет собой реализацию Graph Convolutional Network (GCN), который является одним из популярных слоев для графовых нейронных сетей. Основные параметры слоя **GCNConv** это:

in_channels – количество входных каналов, то есть количество признаков для каждого узла во входном графе.

out_channels – количество выходных каналов, то есть количество признаков, которые производит каждый узел после прохождения через слой. Этот узел задает размерность выходных признаков.

Так как мы решаем задачу классификации, то выходной вектор пропускается через **softmax** функцию перед тем, как будет возвращен из функции **forward()**.

Организация процесса обучения графовых нейронных сетей аналогична процессу обучения полносвязных нейронных сетей (рис. 85).

```
#обучение
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = GCN().to(device)
data = dataset[0].to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)

model.train()
for epoch in range(2000):
    optimizer.zero_grad()
    out = model(data)
    loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
#тестирование
model.eval()
pred = model(data).argmax(dim=1)
correct = (pred[data.test_mask] == data.y[data.test_mask]).sum()
acc = int(correct) / int(data.test_mask.sum())
print(f'Accuracy: {acc:.4f}')
```

Рис. 85. Обучение графовой нейронной сети

Рассмотрим несколько типов слоев, часто применяющихся в графовых нейронных сетях, которые имеются в библиотеке PyTorch Geometric.

Graph Convolutional Layer (GCN). GCN является одним из основных типов слоев для графовых нейронных сетей. Он обрабатывает информацию о вершинах графа и их соседях, выполняя операцию свертки, аналогичную свертке в сверточных нейронных сетях (CNN). Это позволяет моделям учитывать структуру графа при извлечении признаков.

GraphSAGE Layer. GraphSAGE (Graph Sample and Aggregation) является слоем, который агрегирует информацию от соседних вершин, используя выборку (семплирование). Он позволяет учесть контекст внутри локальных областей графа.

Graph Attention Layer (GAT). GAT использует механизм внимания для определения весов взаимодействия между вершинами в графе. Это позволяет уделять больше внимания ближайшим соседям при обработке каждой вершины.

Graph Isomorphism Network Layer (GIN). GIN применяет операции агрегации и сжатия, которые не зависят от порядка вершин в графе, делая слой инвариантным к изоморфизму. Это означает, что графы, структурно равные, будут иметь одинаковые представления.

ChebNet Layer. ChebNet использует полиномиальные фильтры Чебышева для аппроксимации операции свертки на графах. Этот метод может быть эффективен для обработки сигналов на графах.

Graph Pooling Layer. Графы могут быть динамически изменяемыми в размерах, и слои пулинга графов используются для агрегации информации на более высоком уровне, например, при уменьшении размера графа путем объединения вершин.

Graph Attention Pooling Layer. Аналогично обычному пулингу, но с использованием механизма внимания для определения важности каждой вершины при объединении.

Эти слои обеспечивают специализированные механизмы для обработки графов и учитывают их структуру и взаимосвязи между элементами. В зависимости от конкретной задачи и характеристик графов, выбираются соответствующие слои для построения архитектуры графовой нейронной сети.

Рассмотрим основные области применения графовых нейронных сетей.

В обработке естественного языка. Текст – это тип последовательных данных, которые можно описать с помощью RNN или LSTM. Однако графы широко используются в различных задачах обработки естественного языка из-за их естественности и простоты представления. В последнее время наблюдается всплеск интереса к применению GNN для большого количества задач NLP, таких как классификация текста, использование семантики в машинном переводе, определение геолокации пользователя, извлечение отношений или ответы на вопросы.

В компьютерном зрении. Используя обычные сверточные нейронные сети, компьютеры могут различать и идентифицировать объекты на изображениях и видео. Архитектуры GNN также могут применяться для решения задач классификации изображений. Одной из таких проблем является генерация графа сцены, в которой модель нацелена на преобразование изображения в семантический граф, состоящий из объектов и их семантических отношений. Учитывая изображение, модели обнаруживают и распознают объекты и предсказывают семантические отношения между парами объектов. В настоящее время число приложений GNN в компьютерном зрении растет.

В прогнозировании трафика. Прогнозирование скорости движения, объема или плотности дорог в транспортных сетях принципиально важно в интеллектуальной транспортной системе. Графовая нейронная сеть рассматривает транспортную сеть как пространственно-временной граф, узлы которого представляют собой датчики, установленные на дорогах, края измеряются расстоянием между парами узлов, а каждый узел имеет среднюю скорость движения в окне в качестве динамических входных признаков.

В химии. Химики могут использовать GNN для исследования графовой структуры молекул или соединений. В этих графах узлы – это атомы, а ребра – химические связи.

Применение GNN не ограничивается вышеуказанными областями и задачами. GNN применяются для решения множества проблем, таких как предсказание поведения пользователей социальных сетей, анализ социального влияния, построение рекомендательных систем, моделирование и анализ электронных записей о состоянии здоровья, предотвращение атак в компьютерных сетях и т. д.

Вопросы

1. Что такое графовые нейронные сети (GNN) и в чем заключается их основная концепция?

2. Какие типы задач могут быть решены с использованием графовых нейронных сетей?

3. Каковы основные компоненты графовой нейронной сети?

4. Как графовые нейронные сети отличаются от традиционных нейронных сетей?

5. Какие преимущества предоставляют графовые нейронные сети при анализе графовых данных?

6. Как организовать данные в виде графа для использования в графовых нейронных сетях?

7. Какие методы представления узлов и ребер в графовых нейронных сетях используются для обучения?

8. Какие алгоритмы обработки графовых данных часто применяются в графовых нейронных сетях?

9. Какова роль узловых эмбеддингов в графовых нейронных сетях?

10. Как графовые нейронные сети могут решать задачи классификации узлов и ребер в графах?

11. Как создать графовую нейронную сеть в PyTorch?

12. Как задать графовую структуру в PyTorch для использования в графовой нейронной сети?

13. Как добавить атрибуты (features) к узлам и ребрам графа в PyTorch?

14. Как использовать графовую нейронную сеть для задачи классификации узлов в графе в PyTorch?

15. Как реализовать графовую нейронную сеть для задачи предсказания связей между узлами в графе в PyTorch?

16. Как добавить слои агрегации в графовую нейронную сеть в PyTorch?

17. Как оценить качество работы графовой нейронной сети на задаче классификации ребер в графе?

ГЛАВА 9. Нейросетевые архитектуры на основе Transformer

Трансформеры – это относительно новый тип нейросетей, основанный на использование механизма внимания. На сегодня это самая продвинутая техника в области обработки естественной речи (NLP). Задача машинного перевода (как и многие другие задачи) в deep learning сводится к работе с последовательностями: мы тренируем модель, которая может получить на вход предложение как последовательность слов на одном языке и выдать последовательность слов на другом языке. Для решения этой задачи в текущих подходах модель обычно состоит из энкодера и декодера (рис. 86). Энкодер преобразует слова входного предложения в один или больше векторов в некоем векторном пространстве, декодер – генерирует из этих векторов последовательность слов на другом языке. До 2017 года в задачах глубокого обучения, связанного с пониманием текста, обычно использовали рекуррентные нейронные сети. Однако использование такого подхода имеет ряд проблем.

Во-первых, RNN плохо обрабатывают большие последовательности текста. Например, к моменту продвижения к концу абзаца они «забывают» содержание начала.

Во-вторых, RNN сложно и долго обучаются, также они подвержены так называемой проблеме исчезающего или взрывающегося градиента.

В-третьих, они обрабатывают данные последовательно, поэтому рекуррентную нейросеть трудно распараллелить. Ускорить обучение RNN, используя больше графических процессоров нельзя, поэтому ее практически невозможно обучить на большом количестве данных.

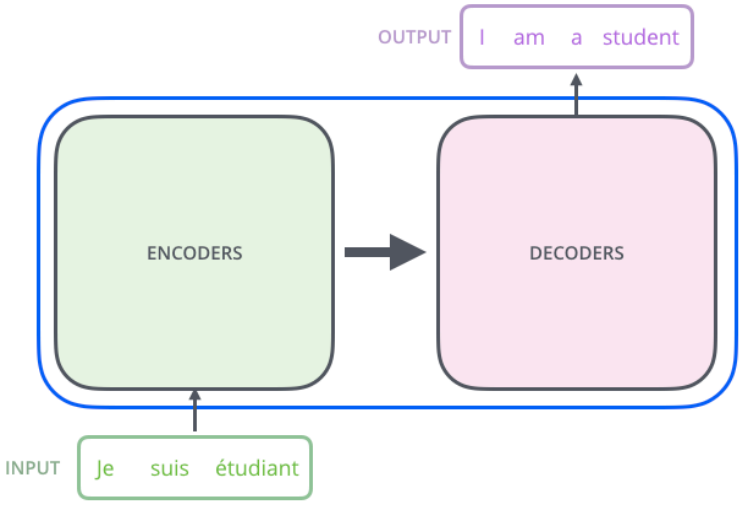


Рис. 86. Архитектура Encoder -decoder

В отличие от рекуррентных нейронных сетей, трансформеры не обрабатывают последовательности по порядку, используя механизм внимания (attention). Благодаря использова-

нию внимания трансформеры можно распараллелить и обучить значительно быстрее. Механизм внимания в нейронных сетях является техникой, которая позволяет сети фокусироваться на определенных частях входных данных в процессе обработки информации. Этот механизм приобрел большую популярность в области обработки естественного языка (Natural Language Processing, NLP), особенно в машинном переводе, где сети должны обрабатывать входные последовательности переменной длины. Основная идея механизма внимания заключается в том, чтобы давать модели возможность сосредотачиваться на различных частях входных данных в зависимости от их важности для текущей задачи. Это достигается путем присвоения весов различным элементам входных данных на основе их влияния на результат.

Рассмотрим механизм внимания более подробно. Предположим, у нас есть три входные последовательности $\mathbf{X_Q}$, $\mathbf{X_K}$, $\mathbf{X_V}$, каждая из которых имеет длину T элементов. Нам необходимо получить вектор контекста **C**, учитывая веса внимания для каждого элемента последовательности. Алгоритм состоит из следующих шагов:

1. Вычисление тензоров трех преобразованных представлений входных данных для Query (запрос), Key (ключ) и Value (значение) – **Q, K V**. Вычисления проводятся по формулам: $\mathbf{Q=X_Q*W_Q, K=X_K*W_k, V=X_V*W_V}$, где $\mathbf{W_Q, W_k, W_V}$ – это матрицы весов для Query, Key и Value соответственно.

2. Расчет весов внимания. Сначала вычисляем «сырые» веса внимания $\mathbf{e_{ij}}$ между $\mathbf{Q_i}$ и $\mathbf{K_k}$ по формуле $\mathbf{e_{ij}=Score(Q_i, K_j) = Q_i*K_j^T}$. Далее применяем функцию softmax для получения нормализованных весов внимания: $\mathbf{a_{ij}}=\frac{\mathbf{exp}(e_{ij})}{\sum_{t=1}^{T}\mathbf{exp}(e_{it})}$.

3. Вычисления контекстного вектора – взвешенную сумму тензора value **V** с использованием весов внимания: $\mathrm{C_i} = \sum_{j=1}^{T}\mathbf{a_{ij}v_j}$

Механизм самовнимания (Self-Attention) представляет собой частный случай механизма внимания, где значения Query, Key и Value получаются из одной и той же последовательности. Self-Attention особенно полезен для обработки последовательностей, где важные элементы могут быть в разных частях последовательности.

Трансформеры используют особый вид слоя, известный как многоголовое внимание (Multi-head attention) (рис. 87). В это случает несколько механизмов внимания тренируются параллельно, что означает несколько линейных преобразований и параллельных операций скалярного произведения и взвешенной суммы.

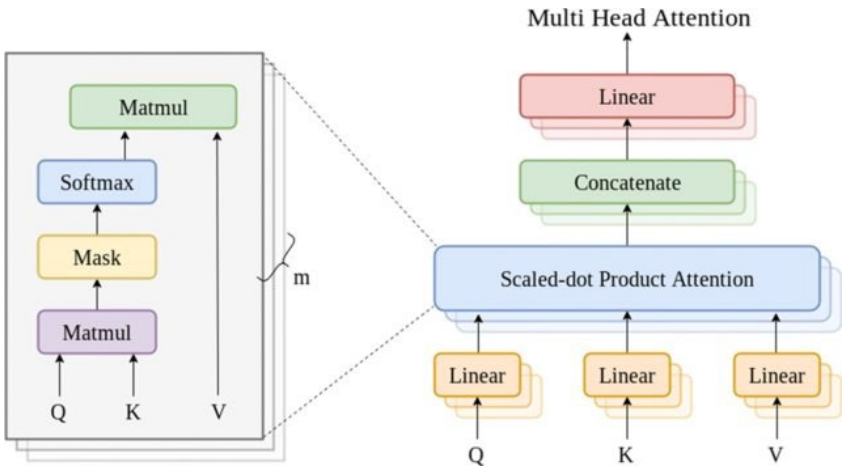


Рис. 87. Схема многоголового внимания

Результаты всех параллельных механизмов внимания конкатенируются, затем проходят через еще одно обучаемое линейное преобразование и поступают на выход. В целом каждый такой модуль получает вектор запроса и набор векторов ключей и значений, выводя при этом один вектор того же размера, что и каждый из входов. Многоголовое внимание позволяет модели совместно воспринимать информацию из разных подпространств представлений на разных позициях. При одной голове внимания усреднение препятствует этому.

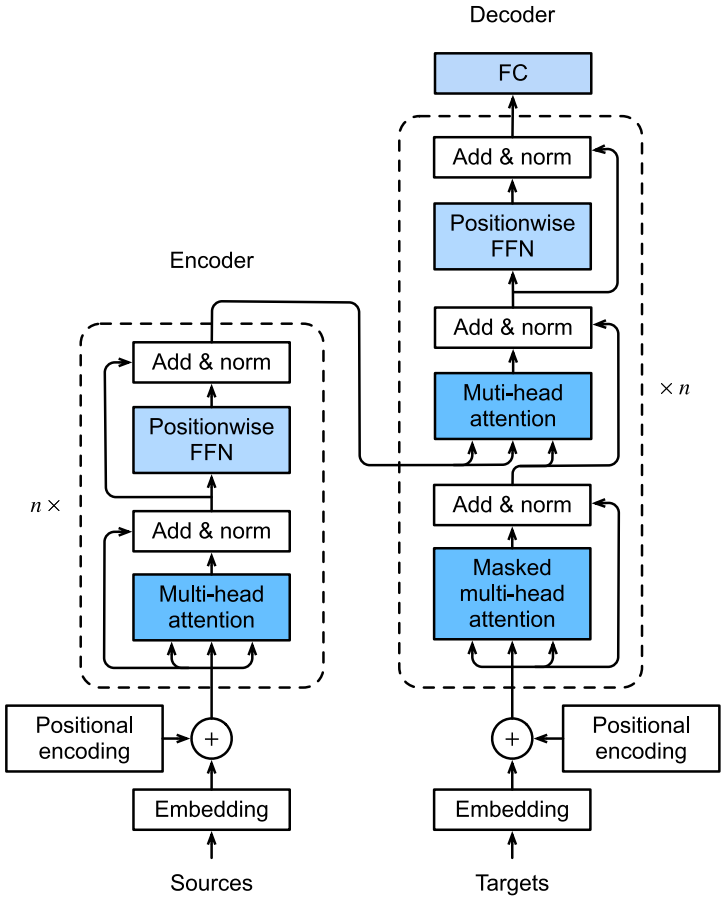


Рис. 88. Архитектура трансформера

Изучим подробно строение и функционирование трансформера на примере задачи перевода текста с одно языка на другой. Трансформер состоит из следующих основных блоков (рис. 88):

1. **Механизм внимания (Attention Mechanism).** Использование этого механизма позволяет модели фокусироваться на различных частях входных данных при выполнении задачи. Трансформер использует механизм многоголового внимания, который позволяет модели параллельно обрабатывать информацию из разных представлений входных данных, повышая её эффективность и глубину понимания.

2. **Позиционное кодирование (Positional Encoding).** Поскольку трансформер не использует рекуррентность или свёртки для обработки последовательностей, ему необходим способ учёта порядка элементов в последовательности. Для этого вводные данные дополняются позиционным кодированием, которое содержит информацию о позиции элемента в последовательности.

3. **Блоки энекодера и декодера (Encoder and Decoder Blocks):** Архитектура трансформера состоит из стеков блоков энкодеров и декодеров. Каждый блок энкодера содержит две основные части: слой многоголового внимания и позиционно-зависимую полносвязную сеть (pointwise feed-forward neural network). Между этими частями и после них применяется нормализация по слоям (layer normalization). Блоки декодера аналогично содержат три основные части: механизм многоголового внимания, многоголовое внимание, направленное на выходы кодировщика, и позиционно-зависимую полносвязную сеть. На вход декодера подается выход энкодера. Главное отличие архитектуры декодера от энкодера заключается в том, что дополнительно имеется внимание к вектору, который получен из последнего блока кодирующего компонента. Компонент декодера тоже многослойный и каждому блоку компонента на вход подается вектор именно с последнего блока кодирующего компонента.

4. **Сквозные (residual) связи и нормализация по слою (Add & Norm).** Каждый подслой (например, многоголовое внимание или полносвязный слой) в блоках кодировщика и декодировщика снабжается сквозными связями (Residual Connection) и последующей нормализацией по слою (Layer Normalization). Это улучшает обучение, предотвращая проблему исчезающего градиента и улучшая поток градиентов через сеть.

5. **Выходной слой.** В декодере последний слой перед выходом обычно представляет собой полносвязный слой **(FC)**, который преобразует векторы в вероятности символов или других элементов выходных данных. Наконец, в конце сети присутствует стандартный softmax слой, который вычисляет вероятности для каждого слова.

Рассмотрим пример реализации трансформера на основе библиотеки PyTorch. Полный пример вы можете найти в репозитории по адресу: https://github.com/fgafarov1977/pytorch_nn_tutorial/blob/main/transformer_translator.ipynb.

Мы построим модель, а также напишем код ее обучения для решения задачи перевода текста с английского языка на русский язык. Для обучения трасформера решению задачи перевода текста с одного языка на другой необходим советующий набор данных, который должен содержать предложения на английском и русском языках. Для обучения и тестирования модели используем известный набор данных **opus_books**, который расположен по адресу: https://huggingface.co/datasets/opus_books.

В самом начале программы добавим классы для реализации класса для нормализации по слоям **NormalizationLayer** и классы **FeedForwardBlock и ProjectionLayer**, которые будут служить строительными блоками в разных частях трансформера. Нормализация по слоям (**NormalizationLayer)** приводит активации каждого слоя к нулевому среднему и единичному стандартному отклонению, что способствует более стабильному распределению данных и предотвращает проблемы, связанные с внутренними сдвигами и масштабами. Модуль **FeedForwardBlock** состоит из двух линейных слоев, разделенных функцией активации ReLU и слоем дропаута. Каждый слой трансформера включает этот модуль после применения механизма внимания. Задача модуля **FeedForwardBlock** – обрабатывать контекстные векторы, полученные из механизма внимания, и создавать новые, более сложные представления для каждого токена. Этот блок позволяет трансформеру улавливать нелинейные зависимости между токенами и вносить дополнительную гибкость в модель. **ProjectionLayer** – это слой, который будет применяться к выходным представлениям декодера. Задача этого слоя – преобразовать выходные представления в пространство, соответствующее размеру словаря, он включает в себя линейное преобразование для последующего применения функции **softmax** для получения вероятностного распределения по выходному словарю или классам.

```
class NormalizationLayer(nn.Module):
    def __init__(self, features: int, eps:float=10**-6) -> None:
        super().__init__()
        self.eps = eps
        self.alpha = nn.Parameter(torch.ones(features))
        self.bias = nn.Parameter(torch.zeros(features))
    def forward(self, x):
        mean = x.mean(dim = -1, keepdim = True)
        std = x.std(dim = -1, keepdim = True)
        return self.alpha * (x - mean) / (std + self.eps) + self.bias

class FeedForwardBlock(nn.Module):
    def __init__(self, d_model: int, d_hidden: int, dropout: float) -> None:
        super().__init__()
        self.linear1 = nn.Linear(d_model, d_hidden)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(d_hidden, d_model)
    def forward(self, x):
```

```
        return self.linear2(self.dropout(torch.relu(self.linear1(x))))
class ProjectionLayer(nn.Module):
    def __init__(self, d_model, vocab_size) -> None:
        super().__init__()
        self.proj = nn.Linear(d_model, vocab_size)
    def forward(self, x) -> None:
        return self.proj(x)
```

Далее создадим модуль, реализующий класс **InputEmbBlock**, который принимают входные данные и преобразуют их в векторное представления, называемые входными эмбедингами. С помощью этого модуля сначала мы преобразуем предложение на английском языке в список входных идентификаторов слов, то есть в список чисел, соответствующих положению каждого слова в словаре, а затем переводим каждое из этих чисел в вектора размера 512. В конструктор этого класса передаем размер модели (**d_model)**, а также размер словаря (**vocab_size)**. В PyTorch уже имеется слой, который позволяет вычислять эмбединги, этот слой называется **Embedding,** поэтому мы его включим в этот класс. В методе **forward** модели мы пропускаем входной тензор через слой **Embedding** и умножаем на корень квадратный из размера модели.

```
class InputEmbBlock(nn.Module):
    def __init__(self, d_model: int, vocab_size: int) -> None:
        super().__init__()
        self.d_model = d_model
        self.vocab_size = vocab_size
        self.embedding = nn.Embedding(vocab_size, d_model)

    def forward(self, x):
        return self.embedding(x) * math.sqrt(self.d_model)
```

В трансформерах отсутствует внутреннее представление о порядке слов в последовательности, так как они обрабатывают входные данные параллельно. Однако в задачах обработки естественного языка порядок слов имеет большое значение. Поэтому используется метод позиционного кодирования, который добавляет векторы, представляющие позиции слов в предложении в исходные эмбеддинги слов. Путем добавления этих позиционных кодов к входным эмбеддингам слов, модель трансформера становится способной учитывать информацию о порядке слов в последовательности (предложении), позволяет эффективно обрабатывать текст, сохраняя контекст и структуру. Блок для добавления позиционного кодирования во входные эмбединги реализуем в отдельном классе – **PositionalEncodingBlock**. Реализуем наиболее популярный метод, основанный на использовании синусоидальных (тригонометрических) векторов, по формулам $\mathbf{PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)}$ и $\mathbf{PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)}$, где pos – позиция токена в последовательности, i – индекс внутри эмбеддинга, d – размерность эмбеддинга. Эти значения добавляются к эмбеддингам токенов, обеспечивая модели информацию о позиции в последовательности.

```
class PositionalEncodingBlock(nn.Module):
    def __init__(self, d_model: int, seq_len: int, dropout: float) -> None:
        super().__init__()
        self.d_model = d_model
        self.seq_len = seq_len
        self.dropout = nn.Dropout(dropout)
        pe = torch.zeros(seq_len, d_model)
        pos = torch.arange(0, seq_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float()
                             * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(pos * div_term)
        pe[:, 1::2] = torch.cos(pos * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)
    def forward(self, x):
        x = x + (self.pe[:, :x.shape[1], :]).requires_grad_(False)
        return self.dropout(x)
```

В трансформерах блоки, такие как self-attention и полносвязные слои, могут быть очень глубокими, и простое добавление большого количества слоев может сделать обучение сложным из-за проблемы затухания или исчезающих градиентов. Сквозные (residual) соединения предоставляют эффективное решение этой проблемы, представляя собой прямое (поэлементное) сложение входного блока с его выходом. Это соединение служит для облегчения обучения глубоких нейронных сетей, предотвращая проблему затухания градиентов при обучении трансформера. Сквозные соединения реализуем в виде отдельного класса **ResConnection**, который будет использоваться и в энкодере и декодере.

```
class ResConnection(nn.Module):
        def __init__(self, features: int, dropout: float) -> None:
            super().__init__()
            self.dropout = nn.Dropout(dropout)
            self.norm = NormalizationLayer(features)
        def forward(self, x, sublayer):
           return x + self.dropout(sublayer(self.norm(x)))
```

Далее реализуем один из ключевых компонент трансформеров **MultiHeadAttention** (многоголовое внимание), который позволяет модели фокусироваться на различных частях входных данных параллельно. Как уже было сказано ранее, многоголовое внимание представляет собой комбинацию нескольких «голов» (heads), каждая из которых вычисляет внимание над входными данными. Результаты каждой головы объединяются и проходят через линейное преобразование для получения окончательного выхода. В конструктор класса **MultiHeadAttention** передаем размерность модели **d_model**, количество голов **heads_count** и значение **dropout**, а также создаем линейные слои для умножения на соответствующие матрицы **(w_query, w_key, w_value, w_o**). В классе имеется статический метод **attention** для реализации механизма внимания. Метод **forward** получает на вход тензора **q, k, v,** на основе которых будет вычисляться внимание, а также тензор маски **mask**. Маска представляет собой матрицу, где элементы, соответствующие позициям, которые должны быть «замаскированы»

или скрыты, устанавливаются в очень большое отрицательное значение (-inf) перед применением функции **softmax**. Это приводит к тому, что веса для этих позиций становятся близкими к нулю после применения **softmax**, и соответствующие значения в выходной последовательности становятся пренебрежимо малыми. Маска используется для того, чтобы блокировать доступ к будущим токенам в последовательности в процессе вычисления внимания для текущего токена. Такой механизм позволяет учитывать только информацию из предшествующих токенов и предотвращает модель от «заглядывания в будущее» при генерации последовательности. В самом начале этого метода тензора **q, k, v** пропускаются через соответствующие слои для вычисления тензоров **query, key, value,** а далее вызывается метод статический метод **attention** для непосредственного вычисления внимания. Далее собираем тензора от каждой головы в единый тензор, пропускаем через линейные слой **w_o** и возвращаем в качестве выходного знания функции **forward.**

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model: int, heads_count: int, dropout: float) -> None:
        super().__init__()
        self.d_model = d_model
        self.heads_count = heads_count
        assert d_model % heads_count == 0, "d_model не делится на heads_count"
        self.d_k = d_model // heads_count
        self.w_query = nn.Linear(d_model, d_model, bias=False)
        self.w_key = nn.Linear(d_model, d_model, bias=False)
        self.w_value = nn.Linear(d_model, d_model, bias=False
        self.w_o = nn.Linear(d_model, d_model, bias=False)
        self.dropout = nn.Dropout(dropout)

    @staticmethod
    def attention(query, key, value, mask, dropout: nn.Dropout):
        d_query = query.shape[-1]
        attention_scores = (query @ key.transpose(-2, -1)) / math.sqrt(d_query)
        if mask is not None:
            attention_scores.masked_fill_(mask == 0, -1e9)
        attention_scores = attention_scores.softmax(dim=-1)
        if dropout is not None:
            attention_scores = dropout(attention_scores)
        return (attention_scores @ value), attention_scores

    def forward(self, q, k, v, mask):
        query = self.w_query(q)
        key = self.w_key(k)
        value = self.w_value(v)
        query = query.view(query.shape[0], query.shape[1], self.heads_count,
                           self.d_k).transpose(1, 2)
        key = key.view(key.shape[0], key.shape[1], self.heads_count,
                       self.d_k).transpose(1, 2)
        value = value.view(value.shape[0], value.shape[1], self.heads_count,
                           self.d_k).transpose(1, 2)
        # вычисляем внимание
        x, self.attention_scores = MultiHeadAttention.attention(query, key, value,
                                                                mask, self.dropout)
        # собираем все "головы" вместе
        x = x.transpose(1, 2).contiguous().view(x.shape[0], -1, self.heads_count * self.d_k)
        # Умножаем на Wo
       return self.w_o(x)
```

Таким образом, мы создали все необходимые модули, и теперь можем на их основе создать класс энкодера. Так как энкодер трансформера состоит из шести одинаковых последовательно связанных слоев, то заранее создадим отдельный класс EncoderBlock для реализации отдельного слоя энкодера. Далее создаем класс Encoder, который будет содержать в себе эти отдельные слои энкодера. В методе **forward** этого класса реализуем последовательную обработку входного тензора **x** каждым слоем энкодера.

```
class EncoderBlock(nn.Module):
    def __init__(self, features: int, self_attention_block: MultiHeadAttention,
                 feed_forward_block: FeedForwardBlock, dropout: float) -> None:
        super().__init__()
        self.self_attention_block = self_attention_block
        self.feed_forward_block = feed_forward_block
        self.residual_connections = nn.ModuleList(
            [ResConnection(features, dropout) for _ in range(2)])
    def forward(self, x, src_mask):
        x = self.residual_connections[0]
        (x, lambda x: self.self_attention_block(x, x, x, src_mask))
        x = self.residual_connections[1](x, self.feed_forward_block)
        return x

class Encoder(nn.Module):
    def __init__(self, features: int, layers: nn.ModuleList) -> None:
        super().__init__()
        self.layers = layers
        self.norm = NormalizationLayer(features)
    def forward(self, x, mask):
        for layer in self.layers:
            x = layer(x, mask)
    return self.norm(x)
```

Далее, аналогично, реализуем блок декодера в классе **Decoder**. Предварительно создадим отдельные класс **DecoderBlock** для реализации отдельных слоев декодера. Необходимо отметить, что в отличие от слоя энкодера, слой декодера содержит также и блок для вычисления перекрёстного внимания **cross_attention_block.** На этот блок поступают эмбединги, которые получаются на выходе энкодера (тензор **encoder_output**), а также эмбединги, вычисленные декодером (тензор **x**).

```
class DecoderBlock(nn.Module):
    def __init__(self, features: int, self_attention_block: MultiHeadAttention,
                 cross_attention_block: MultiHeadAttention,
                 feed_forward_block: FeedForwardBlock, dropout: float) -> None:
        super().__init__()
        self.self_attention_block = self_attention_block
        self.cross_attention_block = cross_attention_block
        self.feed_forward_block = feed_forward_block
        self.residual_connections = nn.ModuleList([ResConnection(features, dropout)
                                                   for _ in range(3)])
    def forward(self, x, encoder_output, src_mask, tgt_mask):
        x = self.residual_connections[0]
        (x, lambda x: self.self_attention_block(x, x, x, tgt_mask))
        x = self.residual_connections[1]
        (x, lambda x: self.cross_attention_block(x, encoder_output,
                                                 encoder_output, src_mask))
        x = self.residual_connections[2](x, self.feed_forward_block)
```

```
        return x

class Decoder(nn.Module):
    def __init__(self, features: int, layers: nn.ModuleList) -> None:
        super().__init__()
        self.layers = layers
        self.norm = NormalizationLayer(features)
    def forward(self, x, encoder_output, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, encoder_output, src_mask, tgt_mask)
         return self.norm(x)
```

Далее, добавим класс **Transformer** для реализации трансформера, который будет содержать в себе объекты классов, реализующих энкодер и декодер. В конструктор этого класса мы подаем также модули **InputEmbBlock**, которые будут вычислять эбединги для текстов на английском (**src_embed**) и на русском языках **(tgt_embed**), и модули **PositionalEncodingBlock** для добавления позиционного кодирования во входные эмбединги. Выполнение функций эндодера выполняется в методе **encode,** а функциональность декодера реализована в методе **decode.** В методе **project** используется слой **ProjectionLayer**, который будет применяться к выходным представлениям декодера, для преобразования выходных представлений в пространство, соответствующее размеру словаря.

```
class Transformer(nn.Module):
    def __init__(self, encoder: Encoder, decoder: Decoder,
                 src_embed: InputEmbBlock, tgt_embed: InputEmbBlock,
                 src_pos: PositionalEncodingBlock, tgt_pos: PositionalEncodingBlock,
                 projection_layer: ProjectionLayer) -> None:
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.src_pos = src_pos
        self.tgt_pos = tgt_pos
        self.projection_layer = projection_layer
    def encode(self, src, src_mask):
        src = self.src_embed(src)
        src = self.src_pos(src)
        return self.encoder(src, src_mask)
    def decode(self, encoder_output: torch.Tensor, src_mask: torch.Tensor,
               tgt: torch.Tensor, tgt_mask: torch.Tensor):
        tgt = self.tgt_embed(tgt)
        tgt = self.tgt_pos(tgt)
        return self.decoder(tgt, encoder_output, src_mask, tgt_mask)
    def project(self, x):
         return self.projection_layer(x)
```

Для подачи текста в трансформер нам необходимо предварительно преобразовать его в последовательность токенов таким образом, чтобы модель могла эффективно обрабатывать их. В обработке естественного языка используется инструмент или алгоритм, который называется токенизатор, который разбивает текст на отдельные единицы, называемые токенами.

Токены могут быть словами, подсловами, символами или другими элементами текста, в зависимости от конкретной задачи и используемого токенизатора. Создадим отдельный метод, в котором будем использовать класс **Tokenizer** библиотеки **tokenizers**:

```
def build_tokenizer(config, ds, lang):
    tokenizer_path = Path(config['tokenizer_file'].format(lang))
    tokenizer = Tokenizer(WordLevel(unk_token="[UNK]"))
    tokenizer.pre_tokenizer = Whitespace()
    trainer = WordLevelTrainer(special_tokens=["[UNK]", "[PAD]",
                                               "[SOS]", "[EOS]"], min_frequency=2)
    tokenizer.train_from_iterator(get_all_sentences(ds, lang),
                                  trainer=trainer)
    tokenizer.save(str(tokenizer_path))
    return tokenizer
```

Для того, чтобы обучать и использовать трансформер создадим датасета **TranslatorDataset**. В конструктор этого класса мы подаем сырой набор данных, полученных с сайта https://huggingface.co, а также токенизаторы, обученные для английского и русского языков. В конструкторе этого класса мы задаем токены для специальных символов, таких как начало предложения SOS, конец предложения EOS и токен PAD. В методе **__getitem__** датасета реализовано получения набора данных, необходимого для обучения трансформера (**encoder_input** – входной тензор энкодера, **decoder_input** – входной тензор декодера, **encoder_mask** – маска энкодера, **decoder_mask** – маска декодера, а также исходные **src_text** – английский и **tgt_text** - русский текст).

```
class TranslatorDataset(Dataset):
    def __init__(self, ds, tok_src, tok_tgt, seq_len):
        super().__init__()
        self.seq_len = seq_len
        self.ds = ds
        self.tok_src = tok_src
        self.tok_tgt = tok_tgt
        self.sos = torch.tensor([tok_tgt.token_to_id("[SOS]")], dtype=torch.int64)
        self.eos = torch.tensor([tok_tgt.token_to_id("[EOS]")], dtype=torch.int64)
        self.pad = torch.tensor([tok_tgt.token_to_id("[PAD]")], dtype=torch.int64)
    def __len__(self):
        return len(self.ds)
    def __getitem__(self, idx):
        src_text = self.ds[idx]['translation']['en']
        tgt_text = self.ds[idx]['translation']['ru']
        enc_inp = self.tok_src.encode(src_text).ids
        dec_inp = self.tok_tgt.encode(tgt_text).ids
        enc_num_padd = self.seq_len - len(enc_inp) - 2
        dec_num_padd = self.seq_len - len(dec_inp) - 1
        if enc_num_padd < 0 or dec_num_padd < 0:
            raise ValueError("Предложение слишком длинное")
        encoder_input = torch.cat([self.sos,torch.tensor(enc_inp, dtype=torch.int64),
                        self.eos,torch.tensor([self.pad] * enc_num_padd,
      dtype=torch.int64),], dim=0,)
        decoder_input = torch.cat([ self.sos, torch.tensor(dec_inp, dtype=torch.int64),
                torch.tensor([self.pad] * dec_num_padd, dtype=torch.int64),], dim=0,)
        label = torch.cat([torch.tensor(dec_inp, dtype=torch.int64), self.eos,
                torch.tensor([self.pad] * dec_num_padd, dtype=torch.int64),], dim=0,)
        return {
            "encoder_input": encoder_input,
            "decoder_input": decoder_input,
```

```
            "encoder_mask": (encoder_input != self.pad).unsqueeze(0).unsqueeze(0).int(),
            "decoder_mask": (decoder_input != self.pad).unsqueeze(0).int()
            & causal_mask(decoder_input.size(0)),
            "label": label,
            "src_text": src_text,
            "tgt_text": tgt_text,
        }
```

Каждая запись набора данных **opus_books** имеет следующую структуру: номер записи-id, тест на английском языке-‘en’, текст на русском языке ‘ru’, например 18 { "en": *"His eyes sparkled merrily and he smiled as he sat thinking."*, "ru": *"Глаза Степана Аркадьича весело заблестели, и он задумался, улыбаясь."* }.

Создадим отдельный метод **get_dataset** для загрузки и подготовки датасета, который будет подаваться на вход трансформера. Для загрузки датасета с сайта https://huggingface.co/ используем встроенный метод **load_dataset** из библиотеки **datasets**, указав в качестве параметра название датасета и необходимую языковую пару для обучения трансформера (‘en-ru’). Так как этот набор данных довольно большой, обучение на всех имеющихся данных может занимать долгое время, поэтому мы оставим только некоторую часть данных, количество которых задано параметром **data_count**. Далее, на основе этих данных создаем токенизаторы, отдельно для английского и отдельно для русского языков, разделяем набор данных на тренировочную (90 %) и тестовую выборки (10 %). Датасеты в формате, необходимом для подачи на вход трансформера, создаем используя класс **TranslatorDataset**, который был создан ранее. На его основе создаем объекты DataLoader отдельно для тренировочного и отдельно для тестового набора данных.

```
def get_dataset(config):
    ds_inintial = load_dataset("opus_books", "en-ru", split='train')
    ds_inintial=ds_inintial.select(range(config['data_count']))
    ds_inintial.to_json("data.json")
    tok_src = build_tokenizer(config, ds_inintial, "en")
    tok_tgt = build_tokenizer(config, ds_inintial, "ru")
    train_size = int(0.9 * len(ds_inintial))
    val_size = len(ds_inintial) - train_size
    train_ds_initial, val_ds_initial = random_split(ds_inintial, [train_size,
val_size])
    train_ds = TranslatorDataset(train_ds_initial, tok_src, tok_tgt, config['seq_len'])
    val_ds = TranslatorDataset(val_ds_initial, tok_src, tok_tgt,  config['seq_len'])
    train_dataloader = DataLoader(train_ds, batch_size=config['batch_size'],
                                  shuffle=True)
    val_dataloader = DataLoader(val_ds, batch_size=1, shuffle=False)
    return train_dataloader, val_dataloader, tok_src, tok_tgt
```

Для оценки качества нейронной сети создадим отельную функцию, в которой реализуем вычисление следующих метрик: Character Error Rate (CER), WordErrorRate, Bleu score. Метрика Character Error Rate показывает процент символов, которые были предсказаны неправильно. Чем ниже значение, тем выше качество системы, при этом значение CER, равное 0, является высшим показателем. WordErrorRate – это процент слов, которые были предсказаны

неправильно. Метрика Bleu score измеряет схожесть между предложением, сгенерированным системой машинного перевода, и эталонными предложением, созданными человеком. Эта метрика учитывает совпадение слов и фраз, используя n-граммы (последовательности из n слов).

```
def calculate_metrics(predicted, expected):
    metric = torchmetrics.CharErrorRate()
    cer = metric(predicted, expected)
    print('validation cer', cer)
    metric = torchmetrics.WordErrorRate()
    wer = metric(predicted, expected)
    print('validation wer', wer)
    metric = torchmetrics.BLEUScore()
    bleu = metric(predicted, expected)
    print('validation BLEU', bleu)
```

Для организации процесса обучения трансформера создадим отдельный метод **train_model.** На вход этого метода передается информация о параметрах обучения нейронной сети в виде пар ключ-значение: "batch_size": 8, "num_epochs": 10, "lr": 10**-4, "seq_len": 350, "d_model": 512, "data_count":16000. В начале метода устанавливаем устройство CPU или CUDA, в зависимости от наличия CUDA устройства. Модель трансформера для обучения создаем вызовом метода **build_transformer**, который мы создали ранее. Далее создаем папку, в которую будут записываться веса обученной модели, для того чтобы могли их использовать для перевода новых текстов. Необходимые для обучения трансформера объекты **train_dataloader, val_dataloader, tokenizer_src, tokenizer_tgt** получаем вызовом метода **get_dataset**. Используем оптимизатор Адама, с шагом обучения, заданном в конфигурации, а в качестве функции потерь – кросс-энтропию. Обучение модели будем проводить в цикле с количеством итераций, заданным параметром **num_epochs** в настройках конфигурации. В самом начале цикла переводим модель в режим обучения, далее получаем батчи обучающих данных из **train_dataloader** с помощью функции **tqdm.** По всем этим пачкам тензоров проводим следующие действия: отправляем данные на соответствующее устройство, пропускаем тензоры через энккодер, декодер и слой проекции, вычисляем функцию потерь и выполняем шаг оптимизации параметров. В конце каждой эпохи выполняем валидацию (метод **make_test**) модели на основе тестовых данных и записываем на диск ее текущие веса.

```
def train_model(config):
    # Define the device
    device = "cuda" if torch.cuda.is_available() else "cpu"
    print("Using device:", device)
    device = torch.device(device)
    Path('opus_books_weights').mkdir(parents=True, exist_ok=True)
    train_dataloader, val_dataloader, tokenizer_src, tokenizer_tgt = get_dataset(con-
fig)
    model = build_transformer(tokenizer_src.get_vocab_size(),
```

```
                             tokenizer_tgt.get_vocab_size(), config["seq_len"],
                             config['seq_len'], d_model=config['d_model'])
    model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=config['lr'], eps=1e-9)
    loss_fn = nn.CrossEntropyLoss(ignore_index=tokenizer_src.token_to_id('[PAD]'),
                                  label_smoothing=0.1).to(device)

    for epoch in range(0, config['num_epochs']):
        torch.cuda.empty_cache()
        model.train()
        batch_iterator = tqdm(train_dataloader, desc=f"Processing Epoch {epoch:02d}")
        for batch in batch_iterator:
            encoder_input = batch['encoder_input'].to(device)
            decoder_input = batch['decoder_input'].to(device)
            encoder_mask = batch['encoder_mask'].to(device)
            decoder_mask = batch['decoder_mask'].to(device)
            encoder_output = model.encode(encoder_input, encoder_mask)
            decoder_output = model.decode(encoder_output, encoder_mask,
                                          decoder_input, decoder_mask)
            proj_output = model.project(decoder_output)
            label = batch['label'].to(device)
            loss = loss_fn(proj_output.view(-1, tokenizer_tgt.get_vocab_size()),
                           label.view(-1))
            batch_iterator.set_postfix({"loss": f"{loss.item():6.3f}"})

            loss.backward()
            optimizer.step()
            optimizer.zero_grad(set_to_none=True)
        make_test(model, val_dataloader, tokenizer_src, tokenizer_tgt,
                  config['seq_len'], device)
        model_filename =str(Path('.') /'opus_books_weights'/f"{'model'}{epoch}.pt" )
        torch.save({'model_state_dict': model.state_dict()},model_filename)
```

Программный код желательно запустить на машине с возможностью вычислений на устройстве CUDA. В самом начале обучения программа выдает предложения, состоящие из случайных слов, но постепенно в ходе обучения видно, что трансформер начинает выводить более или менее осмысленный перевод. Естественно, в нашем примере мы не сможем добиться качественного перевода, потому что у нас очень маленький датасет.

Текст English: He returned looking thinner, with the skin on his cheeks hanging loose, but in the brightest of spirits.
Текст Russian: Князь вернулся похудевший, с обвислыми мешками кожи на щеках, но в самом веселом расположении духа.
Прогноз трансформера: Он не , что он не , и , и .
Текст English : I shan't go anywhere, especially not to Moscow.'
Текст Russian: Никуда не поеду, особенно в Москву.
Прогноз трансформера: Я не .
Текст English: All this cheered Kitty up, but she could not help being troubled.
Текст Russian: Все это веселило Кити, но она не могла не быть озабоченною.
Прогноз трансформера: Он не не мог не мог не мог не мог не мог .
Текст English: 'But I can't go,' thought she. 'Where should I go to?
Текст Russian: "Но я не могу идти, -- думала Ласка. -- Куда я пойду?
Прогноз трансформера: -- Я не не не не не не не не знаю , -- сказал он .

Processing Epoch 03: 100%|██████████| 1688/1688 [10:48<00:00, 2.60it/s, loss=5.111]

Текст English : He returned looking thinner, with the skin on his cheeks hanging loose, but in the brightest of spirits.
Текст Russian: Князь вернулся похудевший, с обвислыми мешками кожи на щеках, но в самом веселом расположении духа.
Прогноз трансформера: Он , , , , но , , но , .
Текст English : I shan't go anywhere, especially not to Moscow.'
Текст Russian: Никуда не поеду, особенно в Москву.
Прогноз трансформера: Я не могу быть , не .
Текст English : All this cheered Kitty up, but she could not help being troubled.
Текст Russian: Все это веселило Кити, но она не могла не быть озабоченною.
Прогноз трансформера: Но Кити была Кити , но не могла не могла не .
Текст English: 'But I can't go,' thought she. 'Where should I go to?
Текст Russian: "Но я не могу идти, -- думала Ласка. -- Куда я пойду?
Прогноз трансформера: -- Да я не могу быть , -- сказала она . -- Что я могу быть ?

Нейронные сети на основе технологии трансформер в настоящее время широко используются для решения различных задач в области искусственного интеллекта и обработки естественного языка. Ниже перечислено несколько типичных задач, где трансформеры проявляют выдающуюся эффективность.

1. **Машинный перевод.** Трансформеры были впервые успешно применены для задачи машинного перевода, позволяя моделям эффективно обрабатывать последовательности слов и генерировать переводы между различными языками.

2. **Обработка текста и классификация.** Трансформеры применяются для обработки текста и решения задач классификации, таких как анализ тональности, распознавание эмоций, определение темы и другие. Модели, обученные на больших текстовых корпусах, демонстрируют высокую точность в этих задачах.

3. **Генерация текста и языковое моделирование.** Трансформеры успешно используются для генерации текста, создания описаний, ответов на вопросы и других задач языкового моделирования. Примером являются модели GPT (Generative Pre-trained Transformer).

4. **Вопросно-ответные системы.** Трансформеры эффективно применяются в системах вопросно-ответных задач, где модели обучаются отвечать на вопросы, опираясь на контекст и предыдущий опыт обучения.

5. **Распознавание речи.** Трансформеры также применяются в системах распознавания речи, где модели могут анализировать аудио-сигналы и конвертировать их в текстовую форму.

6. **Извлечение информации и суммаризация(сжатие) текстов.** Трансформеры также успешно используются для задач извлечения информации из текстов и создания кратких суммаризаций длинных документов.

7. Генерация изображений и обработка видео. Вариации трансформеров также применяются для задач генерации изображений и обработки видео, хотя в этих областях свою роль также играют другие архитектуры нейронных сетей.

В целом, трансформеры показывают впечатляющую универсальность и успешно применяются в широком спектре задач, связанных с обработкой естественного языка и анализом последовательностей данных. На основе трансформера было создано несколько ключевых моделей языкового моделирования (LLM), предназначенных для обработки естественного языка. Вот несколько из них:

GPT (Generative Pre-trained Transformer): Архитектура GPT представляет собой архитектуру трансформера с множеством слоев кодировщика. Основная идея GPT заключается в предварительном обучении языковой модели на большом объеме текстовых данных и дальнейшем использовании этой модели для различных задач NLP. В этом случае модель обучается предсказывать следующее слово в предложении на основе контекста, что позволяет ей захватывать глубокие зависимости в языке. Далее, эта предобученная модель GPT может быть дообучена для решения конкретных задач, таких как перевод текстов, вопросно-ответные системы и др.

BERT (Bidirectional Encoder Representations from Transformers): Архитектура BERT также основана на трансформере, но в отличие от GPT, BERT использует двунаправленный подход. Модель обучается предсказывать слова в контексте обоих направлений (слева направо и справа налево). Двунаправленный подход позволяет BERT лучше улавливать семантические зависимости и контекст в предложениях. BERT успешно применяется для различных задач, таких как классификация, извлечение именованных сущностей (NER), вопросно-ответные системы и другие.

Архитектура XLNet: XLNet сочетает идеи GPT и BERT. Он использует авторегрессивный подход, подобный GPT, для предсказания следующего слова в предложении, но с учетом контекста в обоих направлениях, аналогично BERT. Такой подход позволяет модели более эффективно захватывать долгосрочные зависимости в тексте. XLNet обычно превосходит по качеству представления другие модели на некоторых задачах.

Эти модели представляют собой лишь несколько примеров языковых моделей на основе трансформера. Каждая из них имеет свои особенности и может быть применена в различных контекстах в зависимости от конкретной задачи NLP.

Вопросы:

1. Что представляет собой архитектура нейронных сетей трансформер?
2. Какие основные компоненты включает в себя модель трансформера?
3. Как реализован механизм внимания в архитектуре трансформера?
4. Какие типы входных данных могут быть обработаны с использованием трансформера?
5. Как устроен блок внимания (attention block) в трансформере?
6. Что означает многоголовое внимание (multi-head attention) в трансформере?
7. Какова роль positional encoding в архитектуре трансформера?
8. Как реализован механизм positional encoding в трансформере?
9. Каким образом трансформер обрабатывает переменную длину входных последовательностей?
10. Как трансформеры применяются в задачах обработки естественного языка?
11. Какие преимущества предоставляют трансформеры по сравнению с рекуррентными нейронными сетями?
12. Какие архитектурные изменения были предложены после базовой модели трансформера?
13. Как создать модель нейронной сети трансформера в PyTorch?
14. Как определить архитектуру трансформера в PyTorch?
15. Как добавить блок внимания (attention block) в модель трансформера с использованием PyTorch?
16. Как реализовать механизм внимания (self-attention) в трансформере с помощью PyTorch?
17. Как использовать многоголовое внимание (multi-head attention) в трансформере с использованием PyTorch?
18. Как создать positional encoding для входных данных трансформера в PyTorch?
19. Как обрабатывать последовательности переменной длины в трансформере с использованием PyTorch?
20. Как обучить модель трансформера на многоязычных данных в PyTorch?

ЛИТЕРАТУРА

1. Алексейчук А. С. Введение в нейронные сети: модели, методы и программные средства : учебное пособие / А. С. Алексейчук. – М.: МАИ, 2023. – 105 с.

2. Барский А. Б. Введение в нейронные сети: краткий учебный курс / А. Б. Барский. – М.: ИНТУИТ, 2016. – 260 с.

3. Басараб М. А. Интеллектуальные технологии на основе искусственных нейронных сетей: учебное пособие / М. А. Басараб, Н. С. Коннова. – М.: МГТУ им. Н.Э. Баумана, 2017. – 56 с.

4. Баюк Д. А. Практическое применение методов кластеризации, классификации и аппроксимации на основе нейронных сетей: монография / Д. А. Баюк, О. А. Баюк, Д. В. Берзин. – М.: Прометей, 2020. – 448 с.

5. Бруссард М. Искусственный интеллект: пределы возможного / Мередит Бруссард; пер. с англ. – М.: Альпина нон-фикшн, 2020. – 362 с.

6. Вейдман С. Глубокое обучение: легкая разработка проектов на Python. – СПб.: Питер, 2021. – 272 с.

7. Галушкин А.И. Нейронные сети: основы теории. / А.И. Галушкин. – М.: РиС, 2015. – 496 с.

8. Гудфеллоу Я. Глубокое обучение / Я. Гудфеллоу, И. Бенджио, А. Курвилль; перевод с английского А. А. Слинкина. – 2-е изд. – М.: ДМК Пресс, 2018. – 652 с.

9. Данилов В. В. Нейронные сети : учебное пособие / В. В. Данилов. – Донецк: ДонНУ, 2020. – 158 с.

10. Зольникова Н. Н. Многослойные нейронные сети прямого распространения: учебно-методическое пособие / Н. Н. Зольникова, Т. А. Филонец. – М.: РУТ (МИИТ), 2018. – 57 с.

11. Косарев В. С. Нейронные сети в экономике и финансах: доклад / В. С. Косарев. – М: Дело (РАНХиГС), 2021. – 118 с.

12. Кревецкий А. В. Основы технологий искусственного интеллекта: учебное пособие / А. В. Кревецкий, Н. И. Роженцова, Ю. А. Ипатов; под общ. ред. А. В. Кревецкого. – Йошкар-Ола: Поволжский государственный технологический университет, 2023. – 272 с.

13. Лекун Я. Как учится машина: Революция в области нейронных сетей и глубокого обучения: научно-популярное издание / Я. Лекун. – М.: Альпина ПРО, 2021. – 335 с.

14. Лю Ю. Обучение с подкреплением на PyTorch. Сборник рецептов: руководство / Ю. Лю; перевод с английского А. А. Слинкина. – М.: ДМК Пресс, 2020. – 282 с.

15. Маккинни У. Python и анализ данных: практическое пособие / У. Маккинни; пер. с англ. А. А. Слинкина. – 2-е изд. – М.: ДМК Пресс, 2020. – 540 с.

16. Малов Д.А. Глубокое обучение и анализ данных. Практическое руководство. – СПб.: БХВ-Петербург, 2023, – 272 с.

17. Манусов В. З. Нейронные сети: прогнозирование электрической нагрузки и потерь мощности в электрических сетях. От романтики к прагматике: монография / В. З. Манусов, С. В. Родыгина. – Новосибирск: Изд-во НГТУ, 2018. – 303 с.

18. Маркус Г. Искусственный интеллект: Перезагрузка. Как создать машинный разум, которому действительно можно доверять: практическое руководство / Г. Маркус, Э. Дэвис. – М.: Альпина ПРО, 2021. – 300 с.

19. Мишра П. Объяснимые модели искусственного интеллекта на Python. Модель искусственного интеллекта. Объяснения с использованием библиотек, расширений и фреймворков на основе языка Python: практическое руководство / П. Мишра; пер. с англ. С. В. Минца. – М.: ДМК Пресс, 2022. – 298 с.

20. Николенко С. Глубокое обучение / С. Николенко, А. Кадурин, Е. Архангельская. – СПб.: Питер, 2018. – 480 с.: ил. – (Серия «Библиотека программиста»).

21. Основы искусственного интеллекта: практические работы по созданию и обучению искусственных нейронных сетей на языке Python: учебно-методическое пособие / Н. В. Маркина, Э. И. Беленкова, Г. А. Диденко [и др.]. – Челябинск: ЮУГМУ, 2023. – 72 с.

22. Плас Дж. В. Python для сложных задач: наука о данных и машинное обучение: практическое руководство / Дж. В. Плас. – СПб.: Питер, 2021. – 576 с.

23. Пойнтер Я. Программируем с PyTorch: Создание приложений глубокого обучения. 2020. – 256 с.

24. Прогнозирование ресурса электроизоляционных материалов силовых кабелей с использованием метода искусственных нейронных сетей: монография / Н. К. Полуянович, М. Н. Дубяго, Н. В. Азаров, А. В. Огреничев; под редакцией Н. К. Полуянович. – Ростов-на-Дону: ЮФУ, 2022. – 116 с.

25. Рагимханова Г. С. Программирование на Python: учебное пособие / Г. С. Рагимханова. – Махачкала: ДГПУ, 2022. – 126 с.

26. Рамальо Л. Python – к вершинам мастерства: Лаконичное и эффективное программирование: практическое руководство / Л. Рамальо; пер. с англ. А. А. Слинкина. – 2-е изд. – М.: ДМК Пресс, 2022. – 898 с.

27. Рашка С. Python и машинное обучение: крайне необходимое пособие по новейшей предсказательной аналитике, обязательное для более глубокого понимания методологии машинного обучения / С. Рашка; пер. с англ. А.В. Логунова. – М.: ДМК Пресс, 2017. – 418 с.

28. Редько В.Г. Эволюция, нейронные сети, интеллект: Модели и концепции эволюционной кибернетики / В.Г. Редько. – М.: Ленанд, 2019. – 224 с.

29. Ростовцев В. С. Искусственные нейронные сети / В. С. Ростовцев. – 4-е изд., стер. – СПб.: Лань, 2024. – 216 с.

30. Рутковская Д. Нейронные сети, генетические алгоритмы и нечеткие системы / Д. Рутковская, М. Пилиньский, Л. Рутковский; пер. с польск. И.Д. Рудинского. – 2-е изд., стереотип. – М.: Гор. линия-Телеком, 2013. – 384 с.

31. Саммерфильд М. Python на практике. Создание качественных программ с использованием параллелизма, библиотек и паттернов: практическое пособие / М. Саммерфильд; пер. с англ. А. А. Слинкина. – 2-е изд. – М.: ДМК Пресс, 2023. – 340 с.

32. Семериков А. В. Классификация объектов на основе нейронной сети и методами дерева решения и ближайших соседей: учебное пособие / А. В. Семериков, М. А. Глазырин. – Ухта: УГТУ, 2022. – 68 с.

33. Соробин А. Б. Сверточные нейронные сети: примеры реализаций: учебно-методическое пособие / А. Б. Соробин. – М.: РТУ МИРЭА, 2020. – 159 с.

34. Стивенс Эли, Антига Лука, Виман Томас. PyTorch. Освещая глубокое обучение. – СПб.: Питер, 2022. – 576 с.

35. Трофимова Е. А. Нейронные сети в прикладной экономике: учебное пособие / Е. А. Трофимова, В. Д. Мазуров, Д. В. Гилев; под общ. ред. Е. А. Трофимовой. – Екатеринбург: Изд-во Уральского ун-та, 2017. – 96 с.

36. Филиппов Ф. В. Моделирование нейронных сетей глубокого обучения : учебное пособие / Ф. В. Филиппов. – СПб.: СПбГУТ им. М.А. Бонч-Бруевича, 2019. – 79 с.

37. Ховард Д., Гуггер С. Глубокое обучение с fastai и PyTorch: минимум формул, минимум кода, максимум эффективности. – СПб.: Питер, 2022. – 624 с.

38. Шматов Г. П. Нейронные сети и генетический алгоритм: учебное пособие / Г. П. Шматов. – Тверь: ТвГТУ, 2019. – 200 с.

39. Шумский С. А. Машинный интеллект. Очерки по теории машинного обучения и искусственного интеллекта: монография / С. А. Шумский. – М.: РИОР, 2019. – 340 с.

40. Ярышев С. Н. Технологии глубокого обучения и нейронных сетей в задачах видеоанализа: учебное пособие / С. Н. Ярышев, В. А. Рыжова. – СПб.: НИУ ИТМО, 2022. – 82 с.

Учебное издание

Гафаров Ф.М., Гилемзянов А.Ф.

НЕЙРОННЫЕ СЕТИ В PYTORCH

Учебное пособие

Подписано к использованию 06.05.2024.
Формат 60x84 1/16. Гарнитура «Times New Roman».
Усл. печ. л. 6,21. Уч.-изд. л. 4,27.