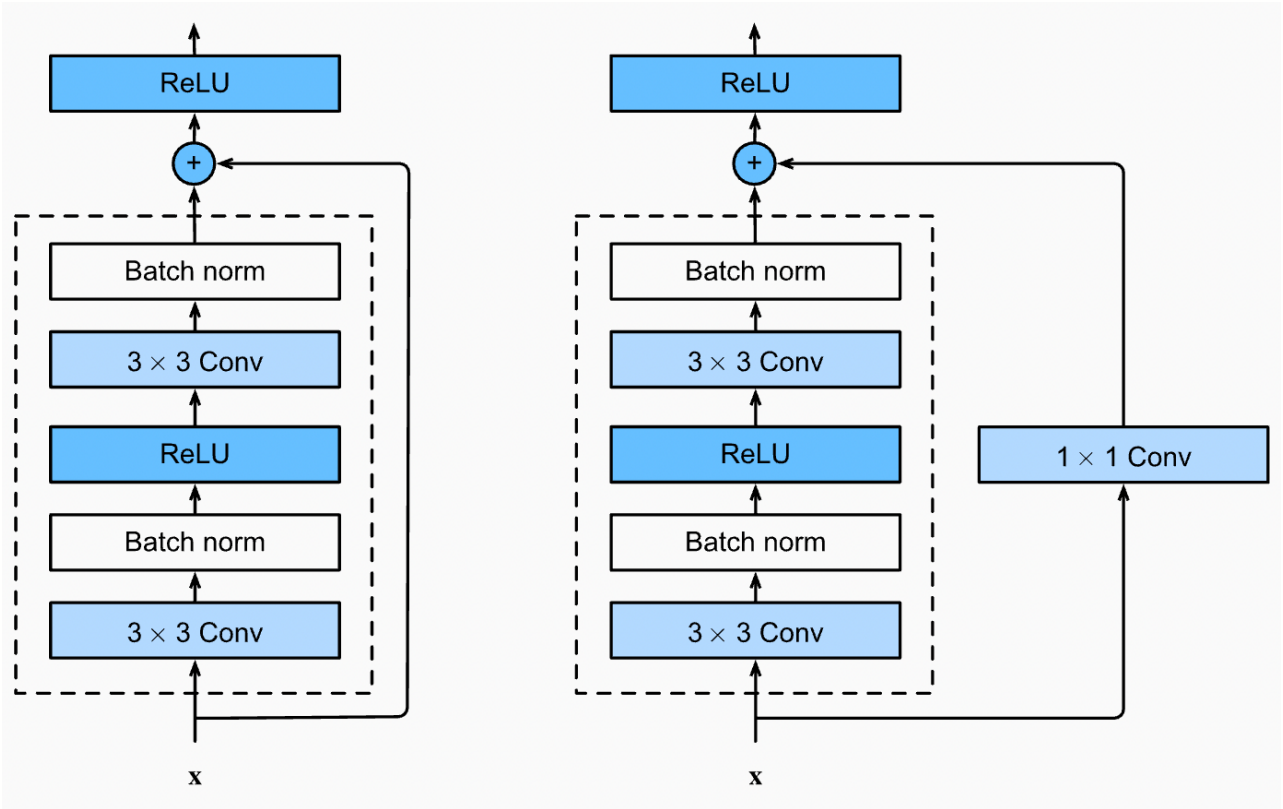
А. И. ЕЛИСЕЕВ, Ю. В. МИНИН, Ю. В. КУЛАКОВ

РЕШЕНИЕ ЗАДАЧ ГЛУБОКОГО ОБУЧЕНИЯ С ИСПОЛЬЗОВАНИЕМ ФРЕЙМВОРКОВ PYTORCH И PYTORCH LIGHTNING



Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2023

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Тамбовский государственный технический университет»

А. И. ЕЛИСЕЕВ, Ю. В. МИНИН, Ю. В. КУЛАКОВ

РЕШЕНИЕ ЗАДАЧ ГЛУБОКОГО ОБУЧЕНИЯ С ИСПОЛЬЗОВАНИЕМ ФРЕЙМВОРКОВ PYTORCH И PYTORCH LIGHTNING

Утверждено Ученым советом университета в качестве учебного пособия
для студентов 3 и 4 курсов направления подготовки
09.03.02 «Информационные системы и технологии»,
1 и 2 курсов направления подготовки
09.04.02 «Информационные системы и технологии»

Учебное электронное издание

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2023

УДК 004(075.8)
ББК *з*973-018.2я73
Е51

Рецензенты:

Кандидат технических наук, доцент, доцент Института математики физики и информационных технологий ФГБОУ ВО «ТГУ им. Г. Р. Державина»
И. А. Зауголков

Доктор технических наук, доцент, профессор кафедры «Мехатроника и технологические измерения» ФГБОУ ВО «ТГТУ»
А. П. Савенков

Елисеев, А. И.

Е51 Решение задач глубокого обучения с использованием фреймворков Pytorch и Pytorch Lightning [Электронный ресурс] : учебное пособие / А. И. Елисеев, Ю. В. Минин, Ю. В. Кулаков. – Тамбов : Издательский центр ФГБОУ ВО «ТГТУ», 2023. – 1 электрон. опт. диск (CD-ROM). – Системные требования : ПК не ниже класса Pentium II ; CD-ROM-дисковод ; 3,0 Mb ; RAM ;Windows 95/98/XP ; мышь. – Загл. с экрана.

ISBN 978-5-8265-2659-0

Представляет собой комплексное руководство, предназначенное для изучения и применения фреймворков PyTorch и PyTorch Lightning в контексте задач глубокого обучения с акцентом на область компьютерного зрения.

Предназначено для студентов 3 и 4 курсов направления подготовки 09.03.02 «Информационные системы и технологии», 1 и 2 курсов направления подготовки 09.04.02 «Информационные системы и технологии».

УДК 004(075.8)
ББК *з*973-018.2я73

ISBN 978-5-8265-2659-0

ВВЕДЕНИЕ

Данное учебное пособие представляет собой комплексное руководство, предназначенное для изучения и применения фреймворков PyTorch и PyTorch Lightning в контексте задач глубокого обучения с акцентом на область компьютерного зрения.

В начальной части пособия представлено детальное описание фреймворка PyTorch, который зарекомендовал себя как эффективный и гибкий инструмент решения задач в области искусственного интеллекта. Дополнительно рассматривается фреймворк PyTorch Lightning – расширение фреймворка PyTorch, предназначенное для оптимизации и упрощения процесса обучения моделей. Специфическое применение этих инструментов демонстрируется на примере решения задач компьютерного зрения. В частности, рассматривается использование фреймворка PyTorch Lightning и библиотеки PyTorch Image Models для эффективного решения подобных задач.

Важной составляющей пособия является представление архитектуры Transformer – инновационного подхода к обучению моделей, активно применяемого в области обработки естественного языка и набирающего популярность в области компьютерного зрения. Пособие включает в себя руководство по использованию данной архитектуры для решения задач компьютерного зрения.

Целью данного пособия является предоставление читателям практико-ориентированного ресурса для изучения и применения фреймворков PyTorch и PyTorch Lightning в контексте задач глубокого обучения.

1. ВВЕДЕНИЕ В ФРЕЙМВОРК PYTORCH

1.1. ОСНОВЫ ФРЕЙМВОРКА PYTORCH

PyTorch – это фреймворк машинного обучения с открытым исходным кодом, который позволяет создавать нейронные сети и эффективно их обучать. Однако PyTorch – не единственный фреймворк такого рода. Альтернативами PyTorch являются TensorFlow/Keras (https://www.tensorflow.org/), JAX (https://github.com/google/jax). Фреймворк PyTorch имеет огромное сообщество разработчиков и особенно часто используется в научных исследованиях. Между тем, TensorFlow является библиотекой глубокого обучения для запуска моделей в производственной среде. Тем не менее, если Вы хорошо знаете один фреймворк машинного обучения, Вам будет очень легко изучить другой, поскольку почти все фреймворки используют одни и те же концепции и идеи. Например, реализация фреймворка TensorFlow версии 2 была вдохновлена популярными функциями PyTorch, что делает эти фреймворки еще более похожими друг на друга.

В пособии используется набор стандартных библиотек, которые часто применяются в проектах машинного обучения. Если Вы запускаете примеры кода в среде Google Colab, все библиотеки уже будут предустановлены. Если Вы запускаете примеры кода локально, убедитесь, что в вашем окружении установлены необходимые библиотеки.

Начнем с импорта PyTorch. Оригинальный пакет называется torch, так как он основан на оригинальном фреймворке Torch (http://torch.ch/). В качестве первого шага можно проверить версию пакета:

```
import torch
print("Torch version", torch.__version__)

Torch version 2.0.1+cu118
```

На момент написания пособия текущей стабильной версией является версия 2.0.1. В целом, рекомендуется обновлять версию PyTorch до самой последней. Программные интерфейсы в новых версиях PyTorch меняется не слишком сильно, и, следовательно, весь приведенный код будет работать и в более новых версиях.

Как и в любом фреймворке машинного обучения, в PyTorch используются стохастические функции, например, функция генерации случайных чисел. Очень хорошей практикой является предварительная настройка воспроизводимости кода за счет использования одних и тех же случайных чисел. По этой причине необходимо задать начальное значение счетчика случайных чисел (значения случайного посева):

```
torch.manual_seed(42)
```

Основная структура данных в PyTorch, которая используется для представления и обработки информации во время обучения нейронных сетей и выполнения операций над данными, – тензоры. Тензоры – это эквивалент массивов NumPy в PyTorch, в который добавлена поддержка ускорения за счет использования графического процессора GPU. Термин «тензор» – это обобщение понятий из линейной алгебры. Например, вектор – это одномерный тензор, а матрица – двумерный тензор. При работе с нейронными сетями используются тензоры различной формы и с разным количеством измерений.

Большинство функций, которые присутствуют в библиотеке NumPy, доступны для использования при работе с тензорами в PyTorch. Поскольку массивы NumPy похожи на тензоры, можно преобразовать бóльшую часть тензоров в массивы NumPy (и обратно), но такая необходимость возникает достаточно редко.

Для начала рассмотрим несколько способов создания тензоров. Существует множество возможных вариантов, но самый простой – вызвать функцию `torch.tensor()`, передав данные, которые будут использованы для создания тензора. Данные могу быть представлены в виде списка, кортежа, массива NumPy или других поддерживаемых структур данных:

```
data = [1, 2, 3, 4, 5]
x = torch.tensor(data)
```

Для присвоения значений тензору во время инициализации существует множество альтернатив, включая:

- `torch.zeros(size)`: создает тензор, заполненный нулями;
- `torch.ones(size)`: создает тензор, заполненный единицами;

– `torch.rand(size)`: тензор со случайными значениями, равномерно распределенными в интервале от 0 до 1;

– `torch.randn(size)`: создает тензор со случайными величинами, выбранными из нормального распределения со средним 0 и дисперсией 1;

– `torch.arange (start=0, end, step=1)`: создает тензор, содержащий значения в интервале от start до end с заданным шагом.

Примеры использования функций:

```
# Создание тензора из вложенного списка
x = torch.tensor([[1, 2], [3, 4]])
print(x)

 tensor([[1, 2],
        [3, 4]])

# Создание тензора со случайными значениями от 0 до 1
# с формой [2, 3, 4]
x = torch.rand(2, 3, 4)
print(x)

 tensor([[[0.8823, 0.9150, 0.3829, 0.9593],
          [0.3904, 0.6009, 0.2566, 0.7936],
          [0.9408, 0.1332, 0.9346, 0.5936]],

         [[0.8694, 0.5677, 0.7411, 0.4294],
          [0.8854, 0.5739, 0.2666, 0.6274],
          [0.2696, 0.4414, 0.2969, 0.8317]]])
```

Форму тензора можно получить так же, как и в NumPy (метод `shape`), или с помощью метода `size`:

```
shape = x.shape
print("Shape:", shape)

size = x.size()
print("Size:", size)

dim1, dim2, dim3 = x.size()
print("Size:", dim1, dim2, dim3)
```

```
Shape: torch.Size([2, 3, 4])
Size: torch.Size([2, 3, 4])
Size: 2 3 4
```

Тензоры можно преобразовывать в массивы NumPy, а массивы NumPy – обратно в тензоры. Чтобы преобразовать массив NumPy в тензор, необходимо использовать функцию `torch.from_numpy()`:

```
np_arr = np.array([[1, 2], [3, 4]])
tensor = torch.from_numpy(np_arr)

print("Numpy array:", np_arr)
print("PyTorch tensor:", tensor)
```

```
Numpy array: [[1 2]
 [3 4]]
PyTorch tensor: tensor([[1, 2],
        [3, 4]])
```

Чтобы преобразовать тензор PyTorch обратно в массив NumPy, можно использовать метод `numpy()`:

```
tensor = torch.arange(4)
np_arr = tensor.numpy()

print("PyTorch tensor:", tensor)
print("Numpy array:", np_arr)
```

```
PyTorch tensor: tensor([0, 1, 2, 3])
Numpy array: [0 1 2 3]
```

Для преобразования тензоров в массив NumPy требуется, чтобы тензор находился на CPU, а не на GPU. Если тензор находится на GPU, необходимо предварительно вызвать метод `cpu()` у тензора. Следовательно, код будет иметь следующий вид: `np_arr = tensor.cpu().numpy()`.

Большинство операций, которые существуют в NumPy, поддерживаются и в PyTorch. Полный список операций можно найти в документации PyTorch. Рассмотрим наиболее важные из них.

Самая простая операция – сложение двух тензоров:

```
x1 = torch.rand(2, 3)
x2 = torch.rand(2, 3)
y = x1 + x2

print("X1", x1)
print("X2", x2)
print("Y", y)

X1 tensor([[0.1053, 0.2695, 0.3588],
        [0.1994, 0.5472, 0.0062]])
X2 tensor([[0.9516, 0.0753, 0.8860],
        [0.5832, 0.3376, 0.8090]])
Y tensor([[1.0569, 0.3448, 1.2448],
        [0.7826, 0.8848, 0.8151]])
```

Выполнение выражения `x1 + x2` приводит к созданию нового тензора, содержащего сумму двух тензоров. Однако также можно использовать операции «на месте» (in-place), которые применяются непосредственно к текущему тензору. Данная операция имеет смысл, если больше не будет необходимости в повторном доступе к предыдущим значениям тензора. Пример показан ниже:

```
x1 = torch.rand(2, 3)
x2 = torch.rand(2, 3)
print("X1 (before)", x1)
print("X2 (before)", x2)

x2.add_(x1)
print("X1 (after)", x1)
print("X2 (after)", x2)

X1 (before) tensor([[0.5779, 0.9040, 0.5547],
        [0.3423, 0.6343, 0.3644]])
X2 (before) tensor([[0.7104, 0.9464, 0.7890],
        [0.2814, 0.7886, 0.5895]])
X1 (after) tensor([[0.5779, 0.9040, 0.5547],
        [0.3423, 0.6343, 0.3644]])
X2 (after) tensor([[1.2884, 1.8504, 1.3437],
        [0.6237, 1.4230, 0.9539]])
```

Операции, выполняемые «на месте», обычно обозначаются постфиксом со знаком подчеркивания (например, add_ вместо add).

Другая распространенная операция – изменение формы тензора. Тензор размера (2,3) может быть преобразован в тензор любой другой формы с тем же количеством элементов. Например, тензор размера (6) может быть преобразован в тензор размера (3,2). В PyTorch для этого используется операция `view`:

```
x = torch.arange(6)
print("X", x)

X tensor([0, 1, 2, 3, 4, 5])
x = x.view(2, 3)
print("X", x)

X tensor([[0, 1, 2],
        [3, 4, 5]])

x = x.permute(1, 0) # Поменять местами размерность 0 и 1
print("X", x)

X tensor([[0, 3],
        [1, 4],
        [2, 5]])
```

Другие часто используемые операции в PyTorch включают матричные умножения, которые необходимы для обучения нейронных сетей. Довольно часто мы имеем входной вектор **x**, который преобразуется с помощью выученной весовой матрицы **W** в некоторые выходные значения. Существует несколько функций для выполнения умножения матриц. Некоторые из них перечислены ниже:

– `torch.matmul`: выполняет матричное произведение над двумя тензорами. Конкретное поведение функции зависит от размерности. Если оба входных значения являются матрицами (двумерными тензорами), то выполняется стандартное матричное произведение. Для операндов большей размерности функция поддерживает трансляцию (подробности смотрите в документации). Также функция может быть записана в виде `a @ b`, как в NumPy;

– `torch.mm`: выполняет матричное произведение над двумя матрицами, но не поддерживает трансляцию (подробности сморите в документации);

– `torch.bmm`: выполняет матричное произведение с поддержкой размерности пакета (батча). Если первый тензор T имеет форму ($b \times n \times m$), а второй тензор R ($b \times m \times p$), результат O будет иметь форму ($b \times n \times p$), и будет вычислен путем выполнения b матричных умножений подматриц T и R: $O_i = T_i @ R_i$;

– `torch.einsum`: выполняет суммирование произведений элементов операндов по заданным размерам с использованием нотации, основанной на правиле суммировании Эйнштейна.

Из всех перечисленных выше функций часто используются функции `torch.matmul()` или `torch.bmm()`. Ниже показаны примеры выполнения умножения матрицы с помощью функции `torch.matmul()`.

```
x = torch.arange(6)
x = x.view(2, 3)
print("X", x)

X tensor([[0, 1, 2],
        [3, 4, 5]])

# Мы также можем объединить несколько операций в одну строку
W = torch.arange(9).view(3, 3)
print("W", W)

W tensor([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])

h = torch.matmul(x, W)
print("h", h)

h tensor([[15, 18, 21],
        [42, 54, 66]])
```

При работе с тензорами часто приходится сталкиваться с ситуацией, когда необходимо выделить часть тензора. Индексирование тензоров осуществляется таким же образом, как и в NumPy:

```
x = torch.arange(12).view(3, 4)
print("X", x)
X tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])

print(x[:, 1])   # Второй столбец

tensor([1, 5, 9])

print(x[0])      # Первая строка

tensor([0, 1, 2, 3])

print(x[:2, -1]) # Первые две строки, последний столбец

tensor([3, 7])

print(x[1:3, :]) # Две нижние строки

tensor([[ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

Одна из основных причин использования фреймворка PyTorch в проектах глубокого обучения заключается в том, что он позволяет автоматически вычислять градиенты/производные функций.

Если нейронная сеть выводит одно скалярное значение, можно говорить о взятии производной. Но довольно часто мы имеем множество выходных переменных (значений). В этом случае можно говорить о градиентах.

Учитывая вход **x**, мы определяем функцию, которая выполняет действия над этим входом, обычно путем матричных умножений на матрицы весов и сложений с векторами смещений. По мере того, как выполняются операции над входными данными, автоматически создается вычислительный граф. Этот граф показывает, как из входных данных получить выходные. PyTorch позволяет определять граф при выполнении операций.

Динамический вычислительный граф (Dynamic Computational Graph) – это одна из ключевых особенностей фреймворка PyTorch. DCG представляет

собой граф, в котором узлы являются операциями, а ребра – тензорами, использующимися в этих операциях. В отличие от статического графа вычислений, который используется в некоторых других фреймворках глубокого обучения, динамический граф PyTorch строится и оптимизируется во время выполнения кода. Это означает, что граф формируется по мере выполнения операций, и мы можем изменять его структуру и поведение на ходу. В PyTorch каждая операция, применяемая к тензору, записывается в граф вычислений, и для каждой операции известно, какие тензоры являются ее входами, а какие являются выходами. Эта особенность позволяет автоматически вычислять градиенты методом обратного распространения ошибки (backpropagation) для обучения нейронных сетей.

Как было сказано выше, в процессе выполнения операций в нейронной сети автоматически создается вычислительный граф, который показывает, как получить выходные данные из тех, что были поданы на вход. Но для начала необходимо указать, какие тензоры потребуют вычисления градиентов.

Стоит отметить, что для тензора, созданного с параметрами по умолчанию, градиенты не вычисляются:

```
x = torch.ones((3,))
print(x.requires_grad)
```

```
False
```

Мы можем изменить это поведение для существующего тензора с помощью функции `requires_grad_()` (подчеркивание указывает на то, что это операция выполняется «на месте»). В качестве альтернативы при создании тензора можно передать аргумент `requires_grad=True` в большинство функций инициализации, которые рассматривались выше.

```
x.requires_grad_(True)
print(x.requires_grad)
```

Создадим вычислительный граф для следующей функции:

$$y = \sum_i 3x_i^3 - 2x_i^2 + 5x_i - 1.$$

Можно представить, что x – это параметры нейронной сети, а мы хотим оптимизировать (максимизировать или минимизировать) выход. Для этого необходимо получить значения градиентов $\partial y/\partial x$. Для примера используем значение x = [0, 1, 2] в качестве входных данных.

```
x = torch.arange(3, dtype=torch.float32, requires_grad=True)
# Только тензоры с плавающей точкой могут иметь градиенты
print("X", x)
```

```
X tensor([0., 1., 2.], requires_grad=True)
```

Выполним пошаговое построение графа вычислений. Можно объединить несколько операций в одной строке, но мы разделим их для лучшего понимания того, как происходит добавление операций в вычислительный граф:

```
x_pow3 = torch.pow(x, 3)
x_pow2 = torch.pow(x, 2)
y = torch.sum(3 * x_pow3 - 2 * x_pow2 + 5 * x - 1)
print("Y", y)
```

```
Y tensor(29., grad_fn=<SumBackward0>)
```

Используя приведенные выше выражения, мы создали вычислительный граф, который выглядит так, как показано на рис. 1.

Визуализация представляет собой абстракцию зависимостей между входами и выходами операций. Каждый узел графа вычислений автоматически определяет функцию для вычисления градиентов относительно своих входов – `grad_fn()`. Ее наличие можно увидеть при выводе на экран значения тензора y. Можно выполнить обратное распространение на графе вычислений путем вызова функции `backward()` у последнего выхода, которая выполняет вычисление градиентов для каждого тензора, имеющего свойство `requires_grad=True`:

```
y.backward()
```

Величина `x.grad` будет содержать значение градиента $\partial y/\partial x$, которое показывает, как изменение x повлияет на выход y при текущем значении входа x = [0, 1, 2]:

```
print(x.grad)
```

```
    tensor([ 5., 10., 33.])
```

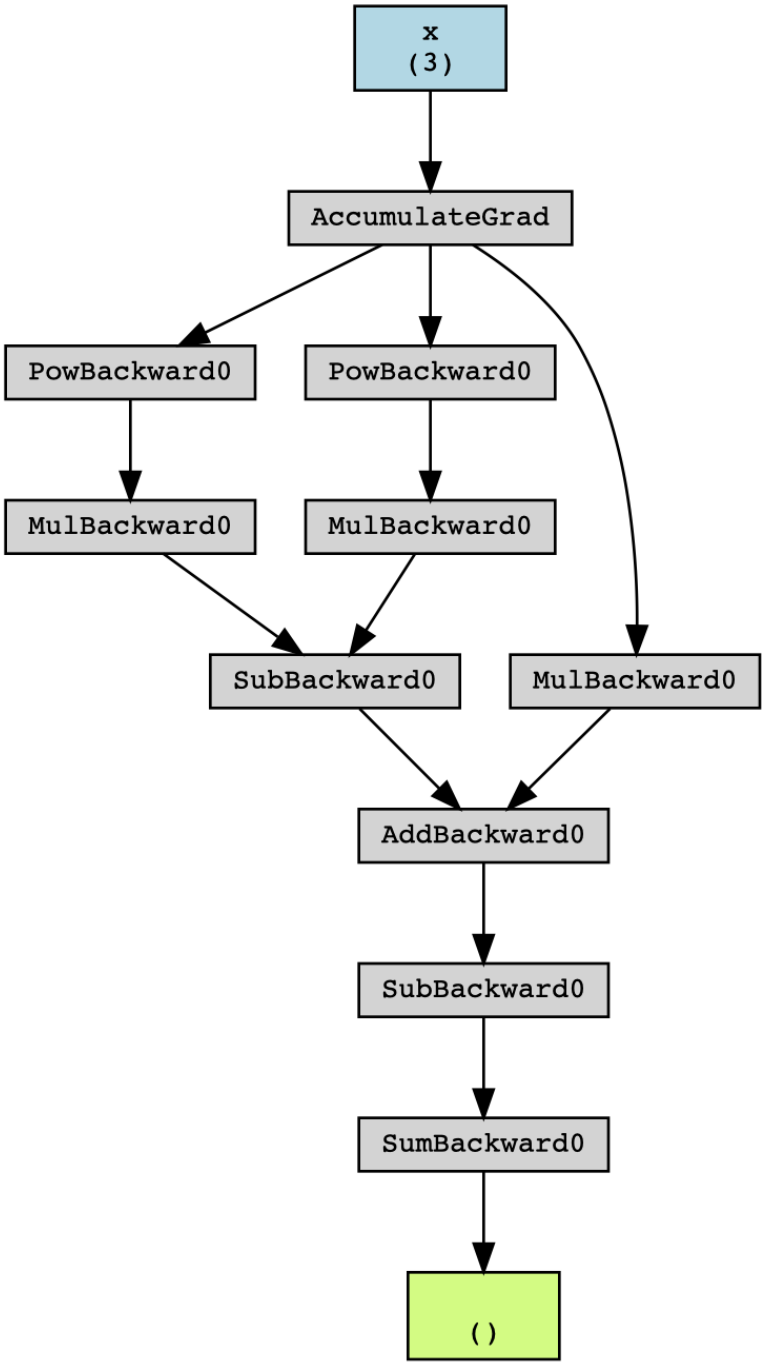


Рис. 1. Пример графа вычислений

Другой важнейшей особенностью PyTorch является поддержка графических процессоров – GPU (Graphics Processing Unit). Графический процессор позволяет параллельно выполнять тысячи простых операций, что делает его очень подходящим для выполнения сложных матричных вычислений в нейронных сетях.

Процессоры CPU и GPU имеют как преимущества, так и недостатки. Поэтому в персональных компьютерах присутствуют оба процессора, которые используются для разных задач.

Процессоры GPU позволяют ускорить обучение нейронных сетей в сотни раз, что очень важно для больших нейронных сетей. В PyTorch реализованы функции для поддержки GPU (в первую очередь компании NVIDIA, благодаря библиотекам CUDA и cuDNN).

Перед работой с GPU необходимо проверить, доступен ли графический процессор:

```
gpu_avail = torch.cuda.is_available()
print(f"Is the GPU available? {gpu_avail}")
```

```
Is the GPU available? False
```

Если на компьютере есть графический процессор, но в результате выполнения кода выше возвращается значение False, необходимо убедиться, что установлена правильная версия библиотеки CUDA. Если Вы работаете в облачной среде Google Colab, убедитесь, что GPU выбран в настройках среды выполнения (меню `Среда выполнения -> Сменить среду выполнения`).

По умолчанию все тензоры, которые создаются в коде, размещаются на CPU. Можно переместить тензор на GPU с помощью функций `to()` или `cuda()`. Однако хорошей практикой является определение в коде объекта-устройства `device`, который указывает на GPU, если GPU доступен, а в противном случае – на CPU. Тогда можно вести разработку с учетом этого объекта, что позволит запускать один и тот же код как на системе с CPU, так и на системе с GPU. Устройство можно определить следующим образом:

```
device = torch.device("cuda") if torch.cuda.is_available() else
torch.device("cpu")
print("Device", device)
```

```
Device gpu
```

Создадим тензор и передадим его на устройство:

```
x = torch.zeros(2, 3)
x = x.to(device)
print("X", x)
```

```
X tensor([[0., 0., 0.],
        [0., 0., 0.]], device='cuda:0')
```

Если на компьютере есть GPU, в выводе рядом с тензором будет показан атрибут `device='cuda:0'`. Значение ноль рядом с `cuda` означает, что это нулевое устройство GPU на компьютере. PyTorch также поддерживает системы с несколькими GPU, но они требуются только для обучения больших сетей.

Важно помнить, что начальные значения счетчика случайных чисел не синхронизируются между процессорами CPU и GPU. Следовательно, для воспроизводимости кода нужно отдельно установить значение seed для GPU. Стоит отметить, что из-за различных архитектур GPU запуск одного и того же кода на разных GPU не гарантирует получения одинаковых значений случайных чисел. Задать значение seed для GPU можно следующим образом:

```
# Операции на GPU имеют отдельный seed,
# который также нужно задать
if torch.cuda.is_available():
    torch.cuda.manual_seed(42)
    torch.cuda.manual_seed_all(42)

# Кроме того, некоторые операции на GPU реализованы
# стохастически для повышения эффективности.
# Необходимо, чтобы все операции на GPU (если они
# используются) были детерминированы.
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

1.2. РЕШЕНИЕ ЗАДАЧИ XOR

Прежде чем построить первую нейронную сеть, важно ответить на простой вопрос: зачем вообще нужен новый алгоритм классификации, когда уже существует много алгоритмов, например деревья решений? Ответ заключается в том, что существуют некоторые задачи классификации, которые деревья решений не способны решить. Одной из таких базовых задач классификации, которую нельзя решить, используя деревья решений, является задача XOR. Задача XOR стала знаменита благодаря тому, что один нейрон, т.е. обычный линейный классификатор, не может аппроксимировать эту простую функцию. Поэтому мы построим небольшую нейронную сеть, способную выучить эту функцию. Чтобы сделать задачу немного более сложной, перенесем задачу XOR в непрерывное пространство и добавим гауссов шум к двоичных входным значениям.

Если необходимо построить нейронную сеть в PyTorch, мы можем задать все параметры (весовые матрицы, векторы смещения) с помощью функции `tensor()` (с параметром `requires_grad=True`), а затем выполнить в PyTorch вычисление градиентов и обновление параметров. Но при таком подходе объем кода может получиться очень большим, если у сети много параметров. В PyTorch доступен пакет `torch.nn`, который позволяет упростить процесс построения нейронных сетей.

Пакет `torch.nn` определяет ряд полезных классов, таких как классы для полносвязных слоев, функции активации, функции потерь и т.д. Полный список можно найти в документации к пакету `torch.nn`. Если необходимо добавить определенный слой в сеть, то сначала рекомендуется обратиться к документации пакета, прежде чем приступать к его самостоятельной реализации.

Для начала импортируем пакет `torch.nn`, как показано ниже:

```
import torch.nn as nn
```

В дополнение к пакету `torch.nn` существует модуль `torch.nn.functional`. Он содержит функции, которые используются в слоях сети. Он отличается от пакета `torch.nn`, который определяет их как контейнеры `nn.Module`, и `torch.nn` фактически использует функции из модуля `torch.nn.functional`. Функциональный модуль бывает полезен во многих случаях, поэтому мы импортируем его отдельно:

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

В PyTorch нейронная сеть строится из модулей. Модули могут содержать другие модули. Сама нейронная сеть тоже считается модулем. Основной шаблон модуля выглядит следующим образом:

```
class MyModule(nn.Module):

    def __init__(self):
        super().__init__()
        # Инициализаторы модуля

    def forward(self, x):
        # Функция для выполнения вычислений
        pass
```

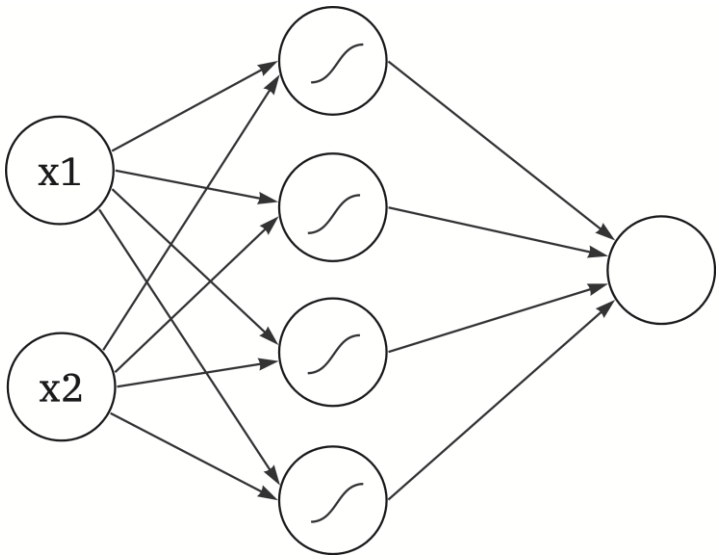


Рис. 2. Простая нейронная сеть для решения задачи XOR

Функция `forward()` – это конструкция, в которой выполняются основные операции модуля. Эта функция выполняется при вызове модуля (`nn = MyModule(); nn(x)`). В функции `init()` обычно задают параметры модуля с использованием класса `nn.Parameter`, или определяются другие модули, которые потом используются внутри функции `forward()`. Обратное вычисление выполняется автоматически, но при желании и его можно переписать.

Воспользуемся предопределенными модулями пакета `torch.nn` и реализуем собственную небольшую нейронную сеть. Создадим простую сеть с входным слоем, одним скрытым слоем с функцией активации `tanh` и выходным слоем. Другими словами, наша сеть должна выглядеть так, как показано на рис. 2.

Входные нейроны на рисунке показаны синим цветом. Они представляют собой координаты $x1$ и $x2$ входных точек данных. Скрытые нейроны с функцией активации гиперболического тангенса показаны белым цветом, а выходной нейрон – красным. В PyTorch описанную модель можно определить следующим образом:

```
class SimpleClassifier(nn.Module):

    def __init__(self, num_inputs, num_hidden, num_outputs):
        super().__init__()
  # Инициализируем модули, необходимые для построения сети
        self.linear1 = nn.Linear(num_inputs, num_hidden)
```

```
        self.act_fn = nn.Tanh()
        self.linear2 = nn.Linear(num_hidden, num_outputs)

    def forward(self, x):
        # Выполнение вычислений для определения прогноза
        x = self.linear1(x)
        x = self.act_fn(x)
        x = self.linear2(x)
        return x
```

Инициализатор требует задания числа входных нейронов (`num_inputs`), числа скрытых нейронов (`num_hidden`) и числа выходных нейронов (`num_outputs`).

Модуль `nn.Linear` является основным строительным блоком нейронных сетей в PyTorch. Он используется для создания полносвязных слоев в нейронных сетях. Модуль принимает на вход тензор размера `[batch_size, input_dim]` и преобразует его в выходной тензор размера `[batch_size, output_dim]`, где `input_dim` – размер входных данных, а `output_dim` – число нейронов в следующем слое. Этот модуль выполняет линейную операцию над входными данными. На выходе `nn.Linear` возвращает объект `nn.parameter.Parameter`, который содержит веса и смещения слоя.

Модуль `nn.Tanh` в PyTorch применяет функцию гиперболического тангенса к элементам входного тензора.

Для примера используем простую нейронную сеть с двумя входными нейронами и четырьмя нейронами в скрытом слое. Поскольку мы решаем задачу бинарной классификации, используем один выходной нейрон. Обратите внимание, что мы не применяем функцию сигмоиды в выходном нейроне. Это связано с тем, что другие функции потерь более эффективно и точно вычисляются на значениях выхода (логитах), а не на результатах применения функции сигмоиды.

```
model = SimpleClassifier(num_inputs=2, num_hidden=4,
num_outputs=1)

# Вывод всех подмодулей
print(model)
```

```
SimpleClassifier(
  (linear1): Linear(in_features=2, out_features=4, bias=True)
  (act_fn): Tanh()
  (linear2): Linear(in_features=4, out_features=1, bias=True)
)
```

Вызов функции `print()` приводит к получению списка всех содержащихся в модели подмодулей. Параметры подмодулей можно получить с помощью функций `parameters()` или `named_parameters()` для получения имени и параметров подмодулей. Для построенной нейронной сети доступны следующие параметры:

```
for name, param in model.named_parameters():
    print(f"Parameter {name}, shape {param.shape}")
```

```
Parameter linear1.weight, shape torch.Size([4, 2])
Parameter linear1.bias, shape torch.Size([4])
Parameter linear2.weight, shape torch.Size([1, 4])
Parameter linear2.bias, shape torch.Size([1])
```

Каждый линейный слой содержит матрицу весов размера `[output_dim, input_dim]` и вектор смещений размера `[output_dim]`. Функция активации `tanh` не имеет параметров. Обратите внимание, что параметры регистрируются только для объектов `nn.Module`, которые являются прямыми атрибутами объекта, т.е. `self.a = … .` Если определить список модулей, параметры этих модулей не регистрируются для внешнего модуля и могут вызвать некоторые проблемы при попытке оптимизации модуля. Существуют альтернативы, такие как `nn.ModuleList`, `nn.ModuleDict` и `nn.Sequential`, которые позволяют строить различные структуры из модулей.

Модуль `nn.Sequential` в PyTorch предназначен для последовательного объединения нескольких модулей PyTorch в один. Он позволяет создать модель глубокого обучения, состоящую из нескольких слоев, используя всего лишь один объект модуля. Он упрощает работу над моделями, которые состоят из множества слоев, и делает код более понятным и читаемым. Модуль может быть использован для создания любой архитектуры нейронной сети, которая может быть представлена в виде последовательности слоев, выполняющих раз-

личные операции, такие как свертки, операции объединения (пулинг), линейные преобразования, функции активации и т.д.

Фреймворк PyTorch также предоставляет несколько функциональных возможностей для эффективной загрузки обучающих и тестовых данных, обобщенных в пакете `torch.utils.data`:

```
import torch.utils.data as data
```

Пакет `data` определяет два класса, которые являются стандартным интерфейсом для работы с данными в PyTorch: `data.Dataset` и `data.DataLoader`. Класс `Dataset` предоставляет единый интерфейс для доступа к данным обучения/тестирования, а загрузчик данных обеспечивает эффективную загрузку и распределение данных из исходного набора в пакеты (батчи) во время обучения.

Класс `Dataset` обобщает базовую функциональность при работе с набором данных в типовом интерфейсе. Для определения набора данных в PyTorch нужно указать две функции: `__getitem__()` и `__len__()`. Функция `__getitem__()` должна возвращать *i*-тый элемент из набора данных, а функция `__len__()` возвращает размер набора данных. Для набора данных задачи XOR можно определить класс набора данных следующим образом:

```
class XORDataset(data.Dataset):

    def __init__(self, size, std=0.1):
        """
        Входные данные:
        size - Количество элементов данных, которые необходимо
сгенерировать.
        std - Стандартное отклонение шума.
        """
        super().__init__()
        self.size = size
        self.std = std
        self.generate_continuous_xor()

    def generate_continuous_xor(self):
```

```
        # Каждая точка данных в наборе данных имеет две
        # переменные, x и y, которые могут быть либо 0, либо 1.
   # Метка - это результат операции XOR.
        data = torch.randint(low=0, high=2, size=(self.size, 2),
dtype=torch.float32)
        label = (data.sum(dim=1) == 1).to(torch.long)
        # Чтобы немного усложнить задачу, мы добавим к точкам
        # данных немного гауссова шума.
        data += self.std * torch.randn(data.shape)
        self.data = data
        self.label = label

    def __len__(self):
        # Количество точек данных.
        # Альтернативно можно использовать self.data.shape[0]
        # или self.label.shape[0].
        return self.size

    def __getitem__(self, idx):
        # Вернуть idx-ю точку данных из набора данных.
        # Если есть несколько объектов для возвращения (точка
        # данных и метка), мы можем вернуть их в виде кортежа.
        data_point = self.data[idx]
        data_label = self.label[idx]
        return data_point, data_label
```

Создадим набор данных и проверим его:

```
dataset = XORDataset(size=200)
print("Size of dataset:", len(dataset))
print("Data point 0:", dataset[0])

Size of dataset: 200
Data point 0: (tensor([-0.0400,  0.9391]), tensor(1))
```

Класс `torch.utils.data.DataLoader` представляет собой Python-итератор над набором данных с поддержкой автоматической пакетной обработки, многопроцессной загрузки данных и других возможностей. Загрузчик данных взаимодействует с набором данных с помощью функции `__getitem__()`

и объединяет выходные данные в виде тензоров по первому измерению для формирования пакета. В отличие от класса `Dataset`, обычно не нужно определять свой собственный класс загрузчика данных, а можно создать его объект с набором данных в качестве входных данных. Кроме того, мы можем настроить загрузчик данных со следующими входными аргументами (представлены только некоторые из них, полный список можно посмотреть в документации):

– `batch_size`: количество образцов в пакете;

– `shuffle`: если значение равно True, данные возвращаются в случайном порядке. Необходимо во время обучения для добавления стохастичности;

– `num_workers`: число подпроцессов, используемых для загрузки данных. Значение по умолчанию, 0, означает, что данные будут загружаться в основном процессе, что может замедлить обучение для наборов, для которых загрузка данных занимает значительное время (например, большие изображения). Для таких случаев рекомендуется использовать больше подпроцессов. Для небольших наборов данных можно оставить значение 0;

– `pin_memory`: если значение равно True, то загрузчик данных будет копировать тензоры в CUDA в уже выделенную память перед их возвращением. Обычно эту технику рекомендуется использовать для тренировочного набора с целью экономии памяти GPU, но она не является обязательной для валидации и тестирования;

– `drop_last`: если значение равно True, последний пакет (батч) отбрасывается, если он меньше указанного размера пакета. Такая ситуация случается, когда размер набора данных не кратен размеру пакета. Параметр потенциально полезен только во время обучения для поддержания постоянного размера пакетов.

Ниже показано создание простого загрузчика данных:

```
data_loader = data.DataLoader(dataset, batch_size=8, shuffle=True)

# next(iter(...)) получает первый пакет из загрузчика данных.
# Если shuffle равен True, то будут возвращаться разные пакеты
# каждый раз, когда запускается код ниже.
# Для итерации по всему набору данных можно использовать
```

```
# конструкцию for batch in data_loader:.
data_inputs, data_labels = next(iter(data_loader))

# Форма выходов – [batch_size, d_1,...,d_N],
# где d_1,...,d_N – размеры элементов данных, возвращенных из
# класса набора данных.
print("Data inputs", data_inputs.shape, "\n", data_inputs)
print("Data labels", data_labels.shape, "\n", data_labels)

Data inputs torch.Size([8, 2])
 tensor([[ 0.0957, -0.0276],
        [ 0.1724, -0.0142],
        [-0.0043,  1.0355],
        [-0.0762,  1.0175],
        [ 1.0245, -0.0026],
        [ 0.9823, -0.1256],
        [ 0.9771,  1.0395],
        [ 1.0373,  0.0439]])
Data labels torch.Size([8])
      tensor([0, 0, 1, 1, 1, 1, 0, 1])
```

После определения модели и набора данных пришло время перейти к оптимизации параметров модели. Во время обучения необходимо выполнить следующие шаги:

1. Получить пакет данных из загрузчика данных.
2. Получить прогнозы модели для пакета.
3. Рассчитать потери на основе разности между предсказаниями и истинными метками.
4. Выполнить обратное распространение: вычислить градиенты для каждого параметра относительно рассчитанной потери.
5. Обновить параметры модели в направлении градиентов.

Мы рассмотрели, как можно выполнить шаги 1, 2 и 4 в PyTorch ранее. Теперь рассмотрим шаги 3 и 5.

Можно рассчитать потери для пакета, выполнив несколько тензорных операций, так как они автоматически добавляются в граф вычислений. Например, для бинарной классификации можно использовать бинарную перекрестную энтропию (Binary Cross Entropy, BCE), которая определяется следующим образом:

$$L_{BCE} = -\sum_i \left[y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i) \right],$$

где y – истинные метки данных, а $\hat{y}$ – предсказания. Однако PyTorch уже предоставляет список предопределенных функций потерь (полный список смотрите в документации).

Несколько примеров функций потерь, реализованных в PyTorch:

– `nn.MSELoss`: Mean Squared Error Loss (средняя квадратичная ошибка) – вычисляет среднеквадратичную разность между предсказанными и целевыми значениями. Обычно используется в задачах регрессии;

– `nn.CrossEntropyLoss`: Cross Entropy Loss (перекрестная энтропия) – используется для многоклассовой классификации. Вычисляет потери на основе логарифма вероятностей предсказанных классов и целевых классов;

– `nn.NLLLoss`: Negative Log Likelihood Loss (отрицательное логарифмическое правдоподобие) – также используется для задач многоклассовой классификации. Ожидает на вход логарифмы вероятностей предсказанных классов и целевых классов;

– `nn.BCELoss`: Binary Cross Entropy Loss (бинарная перекрестная энтропия) – применяется в задачах бинарной классификации. Вычисляет потери на основе логарифма вероятности предсказания истинного класса;

– `nn.BCEWithLogitsLoss`: это альтернатива `nn.BCELoss`, которая объединяет в себе функцию активации Sigmoid и функцию потерь BCELoss. Обычно используется для бинарной классификации, когда в выходном слое модели не применяется функция активации;

– `nn.L1Loss`: Mean Absolute Error Loss (средняя абсолютная ошибка) – вычисляет среднюю абсолютную разность между предсказанными и целевыми значениями. Также применяется в задачах регрессии;

– `nn.SmoothL1Loss`: так же известна как Huber Loss или L1 Loss с использованием окна сглаживания. Сочетает в себе свойства L1 и L2 Loss и обеспечивает плавный переход между ними.

Для BCE в PyTorch есть два модуля: `nn.BCELoss()`, `nn.BCEWithLogitsLoss()`. Если `nn.BCELoss` ожидает, что входные данные будут находиться в диапазоне [0, 1], т.е. они будут вычислены после примене-

ния функции сигмоиды, то `nn.BCEWithLogitsLoss` объединяет сигмоидный слой и операцию вычисления потери BCE в одном классе. Эта версия численно более устойчива, чем использование функции сигмоиды, за которой следует потеря BCE, из-за логарифмов, применяемых в функции потерь. Следовательно, рекомендуется использовать функции потерь, применяемые на «логарифмах» там, где это возможно (помните, что в этом случае не следует применять функцию сигмоиды на выходе модели). Поэтому для модели, определенной выше, используем модуль `nn.BCEWithLogitsLoss`:

```
loss_module = nn.BCEWithLogitsLoss()
```

Для обновления параметров PyTorch предоставляет пакет `torch.optim`, в котором реализовано большинство популярных оптимизаторов.

Несколько популярных оптимизаторов, реализованных в PyTorch:

– `torch.optim.SGD`: Stochastic Gradient Descent (стохастический градиентный спуск) – это один из наиболее распространенных оптимизаторов. Обновляет параметры модели, вычисляя градиенты по каждому пакету обучающих данных. Позволяет настроить скорость обучения и момент;

– `torch.optim.Adam`: Adam (адаптивный метод оптимизации с моментами) – эффективный оптимизатор, сочетающий в себе идеи стохастического градиентного спуска и адаптивного градиентного спуска. Автоматически адаптирует скорость обучения для каждого параметра на основе истории градиентов;

– `torch.optim.RMSprop`: RMSprop (адаптивный метод оптимизации с квадратичным средним градиентов) – адаптивный оптимизатор, который также адаптирует скорость обучения для каждого параметра, учитывая историю квадратичных средних градиентов;

– `torch.optim.Adagrad`: Adagrad (адаптивный градиентный метод) – оптимизатор, который адаптирует скорость обучения для каждого параметра, учитывая историю суммы квадратов градиентов. Предназначен для эффективной работы с разреженными градиентами;

– `torch.optim.Adadelta`: Adadelta – оптимизатор, основанный на методе AdaGrad, который дополнительно решает проблему уменьшения скорости обучения со временем. Использует историю средних квадратов изменений параметров для адаптации скорости обучения;

– `torch.optim.AdamW`: AdamW – вариант оптимизатора Adam, который добавляет технику сокращения весов (weight decay) в процесс оптимизации. Сокращение весов помогает предотвратить переобучение модели;

– `torch.optim.SparseAdam`: SparseAdam – оптимизатор, разработанный специально для работы с разреженными градиентами. Предоставляет более эффективный способ работы с разреженными данными, сокращая вычислительные затраты.

Каждый оптимизатор имеет свои уникальные свойства и подходит для разных типов задач и моделей. Выбор конкретного оптимизатора зависит от требований задачи, архитектуры модели.

Используем самый простой из оптимизаторов – `torch.optim.SGD`. Стохастический градиентный спуск обновляет параметры, умножая градиенты на небольшую константу – скорость обучения, и вычитая их из параметров модели (тем самым минимизируя потери). Таким образом мы медленно движемся в направлении области, минимизирующей потери. Хорошим значением скорости обучения по умолчанию для небольшой сети является значение 0.1.

```
# Входными данными для оптимизатора являются параметры модели
model.parameters()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

Оптимизатор предоставляет две полезные функции: `optimizer.step()` и `optimizer.zero_grad()`. Функция `step()` обновляет параметры на основе градиентов, как объяснялось выше. Функция `zero_grad()` устанавливает градиенты всех параметров в значение ноль. Хотя сначала эта функция кажется менее значимой, ее вызов является важным предварительным шагом перед выполнением обратного распространения. Если вызвать функцию `backward()` для потерь, когда градиенты параметров предыдущего пакета ненулевые, то новые градиенты будут добавлены к предыдущим. Такая реализация выполнена потому, что параметр может встречаться несколько раз в графе вычислений, и нам нужно суммировать градиенты в этом случае, а не заменять их. Поэтому не забывайте вызывать функцию `optimizer.zero_grad()` перед вычислением градиентов пакета.

Наконец, можно перейти к обучению модели. В качестве первого шага создадим набор данных и определим загрузчик данных с большим размером пакета.

```
train_dataset = XORDataset(size=2500)
train_data_loader = data.DataLoader(train_dataset, batch_size=128,
shuffle=True)
```

Теперь можно написать функцию обучения. Необходимо помнить про пять шагов: загрузка пакета, получение предсказаний, вычисление потерь, обратное распространение и обновление параметров. Кроме того, необходимо передать все данные и параметры модели на выбранное устройство (GPU, если есть). Для небольшой нейронной сети, которую мы построили, передача данных на GPU фактически занимает гораздо больше времени, чем выполнение операций на GPU. Для больших сетей время передачи данных значительно меньше, чем время выполнения, что делает использование GPU решающим в этих случаях. Тем не менее, для примера передадим данные на GPU:

```
# Передача модели на устройство
model.to(device)
```

```
SimpleClassifier( (linear1): Linear(in_features=2, out_features=4,
bias=True) (act_fn): Tanh() (linear2): Linear(in_features=4,
out_features=1, bias=True) )
```

Кроме того, необходимо перевести модель в режим обучения. Перевод в режим обучения выполняется с помощью вызова функции `model.train()`. Существуют некоторые модули, которые во время обучения должны выполнять одни операции, а во время тестирования – другие (например, модули `BatchNorm` и `Dropout`), и мы можем переключаться между режимам с помощью вызова функций `model.train()` и `model.eval()`.

```
def train_model(model, optimizer, data_loader, loss_module,
num_epochs=100):
    # Перевести модель в режим обучения
    model.train()
```

```
    # Цикл обучения
    for epoch in tqdm(range(num_epochs)):
        for data_inputs, data_labels in data_loader:

            # Шаг 1. Переместить входные данные на устройство.
            data_inputs = data_inputs.to(device)
            data_labels = data_labels.to(device)

            # Шаг 2. Выполнить модель на входных данных.
            preds = model(data_inputs)
            # Выход имеет размер [размер пакета, 1].
            # Но нам нужно [размер пакет].
            preds = preds.squeeze(dim=1)

            # Шаг 3: Рассчитать потери.
            loss = loss_module(preds, data_labels.float())

            # Шаг 4: Выполнить обратное распространение.
            # Перед вычислением градиентов необходимо убедиться,
            # что все они равны нулю.
            optimizer.zero_grad()

            # Выполнить обратное распространение
            loss.backward()

            # Шаг 5. Обновить параметры.
            optimizer.step()

train_model(model, optimizer, train_data_loader, loss_module)
```

После завершения обучения модели можно сохранить ее на диске, чтобы иметь возможность впоследствии загрузить те же веса. Для этого необходимо извлечь из модели так называемый словарь состояния `state_dict`, который содержит все обучаемые параметры. Для нашей модели словарь состояния содержит следующие записи:

```
state_dict = model.state_dict()
print(state_dict)
```

```
    OrderedDict([('linear1.weight', tensor([[-0.9698,  2.0019],
            [-2.9652, -2.5490],
            [-3.1848,  2.8450],
            [  0.7405,    2.0887]])),  ('linear1.bias',  tensor([
0.1846,    1.1485,  -1.5770,  -1.9533])),  ('linear2.weight',  ten-
sor([[-2.2471,  -4.3284,    4.2551,  -3.2459]])),  ('linear2.bias',
tensor([-0.0549]))])
```

Чтобы сохранить словарь состояний, можно использовать функцию `torch.save()`:

```
torch.save(state_dict, "our_model.tar")
```

Чтобы загрузить модель из словаря состояния, необходимо использовать функцию `torch.load()` для загрузки словаря состояний с диска и функцию модуля `load_state_dict()` для перезаписи текущих параметров новыми значениями:

```
# Загрузка словаря состояния с диска
state_dict = torch.load("our_model.tar")

# Создание новой модели и загрузка состояния
new_model = SimpleClassifier(num_inputs=2, num_hidden=4,
num_outputs=1)
new_model.load_state_dict(state_dict)

# Проверка того, что параметры одинаковы
print("Original model\n", model.state_dict())
print("\nLoaded model\n", new_model.state_dict())

Original model
 OrderedDict([('linear1.weight', tensor([[-0.9698,  2.0019],
        [-2.9652, -2.5490],
        [-3.1848,  2.8450],
        [ 0.7405,   2.0887]])), ('linear1.bias', tensor([ 0.1846,
1.1485, -1.5770, -1.9533])), ('linear2.weight', tensor([[-2.2471,
-4.3284,      4.2551,   -3.2459]])),   ('linear2.bias',   tensor([-
0.0549]))])
```

```
Loaded model
 OrderedDict([('linear1.weight', tensor([[-0.9698,  2.0019],
        [-2.9652, -2.5490],
        [-3.1848,  2.8450],
        [ 0.7405,  2.0887]])), ('linear1.bias', tensor([ 0.1846,
1.1485, -1.5770, -1.9533])), ('linear2.weight', tensor([[-2.2471,
-4.3284,  4.2551, -3.2459]])), ('linear2.bias', tensor([-
0.0549]))])
```

Подробное руководство по сохранению и загрузке моделей в PyTorch можно найти в документации.

После того как мы обучили модель, пришло время оценить ее на удержанном наборе. Поскольку исходный набор данных состоит из случайно сгенерированных данных, необходимо сначала создать тестовый набор с соответствующим загрузчиком данных:

```
test_dataset = XORDataset(size=500)
test_data_loader = data.DataLoader(test_dataset, batch_size=128,
shuffle=False, drop_last=False)
```

В качестве метрики будем использовать метрику Accuracy, которая рассчитывается следующим образом:

$$Accuracy = \frac{\text{число корректных предсказаний}}{\text{число всех предсказаний}} = \frac{TP + TN}{TP + TN + FP + FN},$$

где TP – число истинно положительных предсказаний; TN – число истинно отрицательных предсказаний; FP – число ложноположительных предсказаний; FN – число ложноотрицательных предсказаний.

При оценке модели не нужно следить за вычислительным графом, так как мы не собираемся вычислять градиенты. Это уменьшает объем требуемой памяти и ускоряет работу модели. В PyTorch можно отключить граф вычислений с помощью функции `with torch.no_grad()`. Не забудьте дополнительно перевести модель в режим `eval`:

```
def eval_model(model, data_loader):
    model.eval() # Перевести модель в режим оценки
```

```
    true_preds, num_preds = 0., 0.
    with torch.no_grad(): # Отключить градиенты
        for data_inputs, data_labels in data_loader:

            # Определить прогноз модели
            data_inputs, data_labels = data_inputs.to(device),
data_labels.to(device)
            preds = model(data_inputs)
            preds = preds.squeeze(dim=1)
            preds = torch.sigmoid(preds)
            # Бинаризация прогнозов
            pred_labels = (preds >= 0.5).long()
            # Учет предсказаний для метрики Accuracy
            # (true_preds=TP+TN, num_preds=TP+TN+FP+FN)
            true_preds += (pred_labels == data_labels).sum()
            num_preds += data_labels.shape[0]

    acc = true_preds / num_preds
    print(f"Accuracy of the model: {100.0*acc:4.2f}%")

eval_model(model, test_data_loader)
```

Если правильно обучить модель, то будет получен результат, близкий к значению 100%. Однако такой результат возможно получить только благодаря простой исходной задаче, и, к сожалению, таких высоких результатов невозможно добиться в более сложных задачах.

Перейдем к задаче логирования процесса обучения. TensorBoard – это интерактивный инструмент для визуализации, отладки и отслеживания процесса обучения модели, который поставляется с фреймворком глубокого обучения TensorFlow. TensorBoard позволяет мониторить метрики и ошибки, визуализировать графы вычислений и распределение весов моделей, а также отслеживать процесс обучения с помощью графиков и диаграмм. TensorBoard облегчает понимание того, как меняется качество модели с изменением ее параметров. Хотя изначально TensorBoard был разработан для фреймворка TensorFlow, он также интегрирован в PyTorch, что позволяет легко его использовать.

Для начала импортируем логер:

```
# Импорт логера PyTorch
from torch.utils.tensorboard import SummaryWriter

%load_ext tensorboard
```

Последняя строка необходима, если TensorBoard запускается непосредственно в Jupyter Notebook или в среде Google Colab. В противном случае можно запустить TensorBoard напрямую из терминала.

API TensorBoard в PyTorch прост в использовании. Процесс логирования начинается с создания нового объекта `writer = SummaryWriter(...)` с указанием каталога, в котором должен быть сохранен файл журнала. С помощью этого объекта можно регистрировать различные параметры модели, вызывая функции вида `writer.add_...()`. Например, мы можем визуализировать график вычислений с помощью функции `writer.add_graph()` или добавить скалярное значение, например, значение потери, с помощью функции `writer.add_scalar()`. Адаптируем ранее реализованную функцию обучения, добавив в нее логгер TensorBoard:

```
def train_model_with_logger(model, optimizer, data_loader,
loss_module, val_dataset, num_epochs=100, log-
ging_dir='runs/xor_experiment'):

    # Создать логер TensorBoard
    writer = SummaryWriter(logging_dir)
    model_plotted = False

    model.train()

    for epoch in tqdm(range(num_epochs)):
        epoch_loss = 0.0
        for data_inputs, data_labels in data_loader:

            data_inputs = data_inputs.to(device)
            data_labels = data_labels.to(device)
```

```
            # Для самого первого пакета мы визуализируем граф
            # вычислений в TensorBoard
            if not model_plotted:
                writer.add_graph(model, data_inputs)
                model_plotted = True

            preds = model(data_inputs)
            # Выход имеет размер [размер пакета, 1].
            # Но нам нужно [размер пакет].
            preds = preds.squeeze(dim=1)

            loss = loss_module(preds, data_labels.float())

            optimizer.zero_grad()

            loss.backward()

            optimizer.step()

            epoch_loss += loss.item()

        # Добавить средние потери в TensorBoard
        epoch_loss /= len(data_loader)
        writer.add_scalar('training_loss',
                          epoch_loss,
                          global_step = epoch + 1)

        # Визуализируем предсказание и добавляем изображение
        # в TensorBoard.
        # Поскольку изображения matplotlib могут медленно
        # отрисовываться, мы делаем это только каждую 10-ю
        # эпоху.
        if (epoch + 1) % 10 == 0:
            fig = visualize_classification(model,
val_dataset.data, val_dataset.label)
            writer.add_figure('predictions',
                              fig,
                              global_step = epoch + 1)

    writer.close()
```

Воспользуемся этим методом для обучения модели:

```
model = SimpleClassifier(num_inputs=2, num_hidden=4,
num_outputs=1).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
train_model_with_logger(model, optimizer, train_data_loader,
loss_module, val_dataset=dataset)
```

Файл TensorBoard в каталоге `runs/xor_experiment` будет содержать кривую потерь, график вычислений сети и визуализацию выученных предсказаний по количеству эпох (рис. 3). Чтобы запустить визуализатор TensorBoard, необходимо выполнить следующий оператор:

```
%tensorboard --logdir runs/xor_experiment
```

Визуализация TensorBoard может помочь определить возможные проблемы с моделью и выявить такие ситуации, как переобучение или недобучение. Также можно отслеживать прогресс во время обучения модели, так как логер автоматически записывает все добавленное в файл журнала.

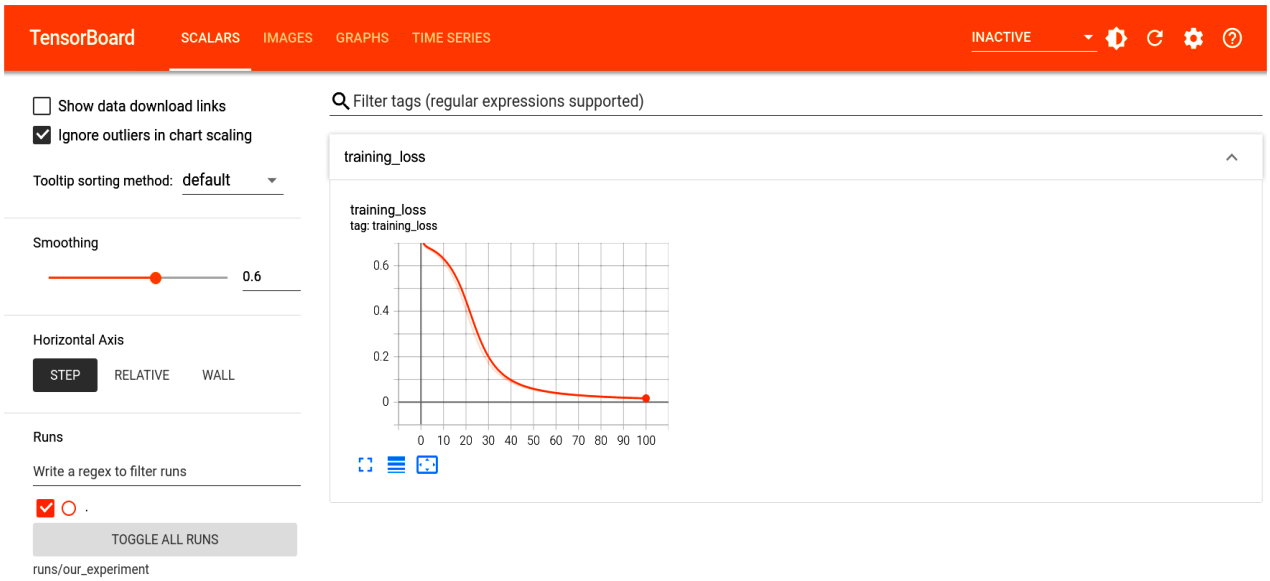


Рис. 3. Визуализация процесса обучения в TensorBoard

1.3. ФУНКЦИИ АКТИВАЦИИ

Большинство популярных функций активации можно найти в пакете `torch.nn` (полный перечень функций смотрите в документации).

Несколько популярных функции активации, реализованных в PyTorch:

– `nn.ReLU`: ReLU (Rectified Linear Unit) – одна из самых популярных функций активации в глубоком обучении. ReLU преобразует все отрицательные значения входа в ноль, а положительные значения оставляет без изменений;

– `nn.Sigmoid`: сигмоидная функция – функция активации преобразует входные значения в интервал (0, 1). Широко применяется в задачах бинарной классификации;

– `nn.Tanh`: гиперболический тангенс – ограничивает значения интервалом (–1, 1), но обладает симметричным распределением. Часто используется в рекуррентных нейронных сетях;

– `nn.LeakyReLU`: LeakyReLU – похожа на ReLU, но отличается наличием небольшого уклона для отрицательных значений входа. Это помогает избежать проблемы «мертвых нейронов» в ReLU;

– `nn.ELU`: ELU (Exponential Linear Unit) – функция активации похожа на LeakyReLU, но имеет экспоненциальное поведение для отрицательных значений, что может помочь в улучшении обобщающей способности модели;

– `nn.Softmax`: Softmax – функция активации преобразует входные значения в нормированные вероятности, которые в сумме будут давать единицу. Широко используется в задачах многоклассовой классификации, когда требуется прогнозирование вероятности каждого класса.

Для примера вручную реализуем нескольких собственных функций активации. Начнем с определения базового класса, от которого будут наследоваться все будущие реализации:

```
class ActivationFunction(nn.Module):

    def __init__(self):
        super().__init__()
        self.name = self.__class__.__name__
        self.config = {"name": self.name}
```

Каждая функция активации будет представлять собой модуль `nn.Module`, чтобы ее можно было легко интегрировать в сеть.

Далее мы реализуем две самые «старые» функции активации, которые до сих пор часто используются для различных задач: сигмоидную и функцию гиперболического тангенса. Сигмоидную и функцию гиперболического тангенса можно найти в PyTorch как в виде модулей (`nn.Sigmoid`, `nn.Tanh`), так и виде готовых функций (`torch.sigmoid`, `torch.tanh`). Мы реализуем их самостоятельно:

```
class Sigmoid(ActivationFunction):

    def forward(self, x):
        return 1 / (1 + torch.exp(-x))


class Tanh(ActivationFunction):

    def forward(self, x):
        x_exp, neg_x_exp = torch.exp(x), torch.exp(-x)
        return (x_exp - neg_x_exp) / (x_exp + neg_x_exp)
```

Другой популярной функцией активации, которая позволила обучать более глубокие сети, является «выпрямленная линейная единица» (Rectified Linear Unit, ReLU). Несмотря на простоту ReLU как кусочно-линейной функции, у нее есть одно важное преимущество по сравнению с сигмоидой и гиперболическим тангенсом: стабильный градиент для большого диапазона значений. На основе этой идеи было предложено множество вариаций ReLU, из которых мы реализуем следующие три: LeakyReLU, ELU и Swish. LeakyReLU заменяет нулевые значения в отрицательной полуплоскости на небольшие значения, определенные наклоном функции, чтобы градиенты могли «протекать» по сети и в случае таких входных значений. Аналогично, в ELU отрицательная часть функции определяется экспоненциальным спадом. Третьей функцией активации является Swish, которая фактически является результатом эксперимента с целью поиска «оптимальной» функции активации [1]. По сравнению с другими функциями активации, Swish является гладкой и немонотонной. Было показано, что эта

особенность предотвращает появление «мертвых» нейронов по сравнению со стандартной активацией ReLU, особенно в глубоких сетях.

Реализуем четыре перечисленные выше функции активации:

```
class ReLU(ActivationFunction):

    def forward(self, x):
        return x * (x > 0).float()

class LeakyReLU(ActivationFunction):

    def __init__(self, alpha=0.1):
        super().__init__()
        self.config["alpha"] = alpha

    def forward(self, x):
        return torch.where(x > 0, x, self.config["alpha"] * x)

class ELU(ActivationFunction):

    def forward(self, x):
        return torch.where(x > 0, x, torch.exp(x)-1)

class Swish(ActivationFunction):

    def forward(self, x):
        return x * torch.sigmoid(x)
```

Возьмем набор данных (на этот раз с изображениями) и построим новую нейронную сеть для экспериментов с разными функциями активации. Будем подавать изображения в виде одномерных тензоров в сеть и пропускать их через последовательность линейных слоев и заданную функцию активации:

```
class BaseNetwork(nn.Module):

    def __init__(self, act_fn, input_size=784, num_classes=10,
hidden_sizes=[512, 256, 256, 128]):
        """
```

```
        Входы:
        act_fn – Функция активации.
        input_size – Размер входных изображений в пикселях.
        num_classes – Количество классов в наборе данных.
        hidden_sizes – Размеры скрытых слоев.
        """
        super().__init__()

        # Создать сеть на основе размеров скрытых слоев.
        layers = []
        layer_sizes = [input_size] + hidden_sizes
        for layer_index in range(1, len(layer_sizes)):
            layers += [nn.Linear(layer_sizes[layer_index-1], lay-
er_sizes[layer_index]), act_fn]
        layers += [nn.Linear(layer_sizes[-1], num_classes)]
        # nn.Sequential обобщает список модулей в один последова-
тельный
        self.layers = nn.Sequential(*layers)

        # Храним все гиперпараметры в словаре
        self.config = {"act_fn": act_fn.config, "input_size": in-
put_size, "num_classes": num_classes, "hidden_sizes": hid-
den_sizes}

    def forward(self, x):
        # Преобразование изображений в плоский вектор
        x = x.view(x.size(0), -1)
        out = self.layers(x)
        return out
```

Также необходимо загрузить набор данных, на котором мы будем обучать сеть, а именно набор FashionMNIST. FashionMNIST является более сложной версией набора MNIST и содержит черно-белые изображения предметов одежды и обуви, всего – десять классов. Для загрузки этого набора данных воспользуемся еще одним пакетом PyTorch, а именно torchvision. Это пакет, представляющий собой коллекцию инструментов и библиотек для задач компьютерного зрения. Он включает в себя множество встроенных функций для работы с изображениями, видео и другими видами медиаданных. Есть несколько

ключевых особенностей, которые делают torchvision уникальной для разработчиков и исследователей в области компьютерного зрения.

Во-первых, torchvision предлагает набор заранее обученных моделей, таких как AlexNet, VGG, ResNet, SqueezeNet и DenseNet и др. Он также содержит модели для задач детекции, включая Faster R-CNN и Mask R-CNN.

Вторая ключевая особенность пакета torchvision – это коллекция наборов данных для компьютерного зрения. Эти датасеты включают в себя изображения из различных источников, таких как ImageNet, CIFAR10/100, MNIST и COCO, и могут быть использованы для обучения и тестирования моделей. Данные можно легко загрузить и использовать прямо из PyTorch, что упрощает процесс их предобработки.

Также пакет torchvision предоставляет различные преобразования, которые могут быть применены к изображениям для аугментации данных. Преобразования можно легко комбинировать и применять к изображениям, чтобы увеличить количество и разнообразие доступных данных для обучения.

Ниже приведены некоторые примеры функций аугментации изображений, предоставляемых torchvision:

1. Аугментация `ColorJitter`: изменяет яркость, контрастность и насыщенность изображения.

```
from torchvision.transforms import ColorJitter
transform = ColorJitter(brightness=0.2, contrast=0.2, satura-
tion=0.2)
augmented_image = transform(image)
```

2. Аугментация `RandomRotation`: поворачивает изображение на случайный угол.

```
from torchvision.transforms import RandomRotation
transform = RandomRotation(degrees=20)
augmented_image = transform(image)
```

3. Аугментация `RandomResizedCrop`: применяет случайную обрезку (кадрирование) масштабированного изображения.

```
from torchvision.transforms import RandomResizedCrop
transform = RandomResizedCrop(size=(64, 64))
augmented_image = transform(image)
```

4. Аугментация `RandomHorizontalFlip`: случайно отражает изображение в горизонтальном направлении.

```
from torchvision.transforms import RandomHorizontalFlip
transform = RandomHorizontalFlip()
augmented_image = transform(image)
```

Помимо этого, torchvision позволяет комбинировать различные преобразования с помощью класса `Compose`:

```
from torchvision.transforms import Compose, RandomHorizontalFlip,
RandomResizedCrop
transform = Compose([RandomHorizontalFlip(), RandomResized-
Crop(size=(64, 64))])
augmented_image = transform(image)
```

Приведенный выше код выполняет оба преобразования в указанной последовательности над одним и тем же изображением.

Загрузим набор данных FashionMNIST и визуализируем несколько изображений, чтобы получить представление о данных.

```
import torchvision
from torchvision.datasets import FashionMNIST
from torchvision import transforms

# Преобразования, применяемые к каждому изображению =>
# сначала превращаем их в тензор, затем нормализуем в диапазоне #
от -1 до 1
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,),
(0.5,))])

# Загрузка обучающего набора данных.
# Нужно разделить его на обучающую и проверочную части.
```

```
train_dataset = FashionMNIST(root=DATASET_PATH, train=True,
transform=transform, download=True)
train_set, val_set = torch.utils.data.random_split(train_dataset,
[50000, 10000])

# Загрузка тестового набора
test_set = FashionMNIST(root=DATASET_PATH, train=False,
transform=transform, download=True)

# Определение загрузчиков данных, которые можно
# использовать для различных целей позже.
# Обратите внимание, что для реального обучения модели мы будем #
использовать различные загрузчики данных с меньшим размером
# пакета.
train_loader = data.DataLoader(train_set, batch_size=1024,
shuffle=True, drop_last=False)
val_loader = data.DataLoader(val_set, batch_size=1024,
shuffle=False, drop_last=False)
test_loader = data.DataLoader(test_set, batch_size=1024,
shuffle=False, drop_last=False)
```

Обучим модель с различными функциями активации на наборе данных FashionMNIST и сравним полученные результаты. Наша конечная цель – достичь наилучшей возможной производительности на тестовом наборе данных. Поэтому напишем цикл обучения, включающий проверку после каждой эпохи и финальное тестирование лучшей модели:

```
def train_model(net, model_name, max_epochs=50, patience=7,
batch_size=256, overwrite=False):
    """
    Обучение модели на наборе FashionMNIST

    Входы:
        net – Модель.
        model_name – (str) Имя модели, используется для создания
контрольных точек.
        max_epochs – Максимальное количество эпох, в течение
которых происходит обучение модели.
```

```
        patience – Если производительность на проверочном
множестве не улучшилась за patience эпох, мы прекращаем обучение
раньше времени.
        batch_size – Размер пакетов, используемых при обучении.
        overwrite – Определяет, как поступать в случае, когда
контрольная точка уже существует. Если True, она будет перезаписана.
    """
    file_exists = os.path.isfile(_get_model_file(CHECKPOINT_PATH,
model_name))
    if file_exists and not overwrite:
        print("Model file already exists. Skipping training...")
    else:
        if file_exists:
            print("Model file exists, but will be overwritten...")

        # Определение оптимизатора, модуля потерь и загрузчика
данных
        optimizer = optim.SGD(net.parameters(), lr=1e-2,
momentum=0.9)
        loss_module = nn.CrossEntropyLoss()
        train_loader_local = data.DataLoader(train_set,
batch_size=batch_size, shuffle=True, drop_last=True,
pin_memory=True)

        val_scores = []
        best_val_epoch = -1
        for epoch in range(max_epochs):

            net.train()
            true_preds, count = 0., 0
            for imgs, labels in tqdm(train_loader_local,
desc=f"Epoch {epoch+1}", leave=False):
                imgs, labels = imgs.to(device), labels.to(device)
                # Zero-grad может быть помещен в любое место
      # перед loss.backward()
                optimizer.zero_grad()
                preds = net(imgs)
                loss = loss_module(preds, labels)
                loss.backward()
                optimizer.step()
```

```
                # Запись статистики во время обучения
                true_preds += (preds.argmax(dim=-1) =
= labels).sum()
                count += labels.shape[0]
            train_acc = true_preds / count

            val_acc = test_model(net, val_loader)
            val_scores.append(val_acc)
            print(f"[Epoch {epoch+1:2d}] Training accuracy:
{train_acc*100.0:05.2f}%, Validation accuracy:
{val_acc*100.0:05.2f}%")

            if len(val_scores) == 1 or val_acc >
val_scores[best_val_epoch]:
                print("\t   (New best performance, saving
model...)")
                save_model(net, CHECKPOINT_PATH, model_name)
                best_val_epoch = epoch
            elif best_val_epoch <= epoch - patience:
                print(f"Early stopping due to no improvement
over the last {patience} epochs")
                break

        # Построение кривой значений метрики Accuracy для
        # валидации
        plt.plot([i for i in range(1,len(val_scores)+1)],
val_scores)
        plt.xlabel("Epochs")
        plt.ylabel("Validation accuracy")
        plt.title(f"Validation performance of {model_name}")
        plt.show()
        plt.close()

    load_model(CHECKPOINT_PATH, model_name, net=net)
    test_acc = test_model(net, test_loader)
    print((f" Test accuracy: {test_acc*100.0:4.2f}% ").
center(50, "=")+"\n")
    return test_acc
```

```
def test_model(net, data_loader):
    """
    Тестирование модели на заданном наборе данных.

    Входные данные:
        net – Обученная модель.
        data_loader – Объект DataLoader набора данных
для тестирования (валидация или тест).
    """
    net.eval()
    true_preds, count = 0., 0
    for imgs, labels in data_loader:
        imgs, labels = imgs.to(device), labels.to(device)
        with torch.no_grad():
            preds = net(imgs).argmax(dim=-1)
            true_preds += (preds == labels).sum().item()
            count += labels.shape[0]
    test_acc = true_preds / count
    return test_acc
```

Представленный цикл обучения можно запустить с каждой из перечисленных выше функции активации. По результатам экспериментов модель, использующая сигмоидальную функцию активации, демонстрирует случайную производительность (Accuracy ~ 10%). Все остальные функции активации будут иметь схожую производительность. Чтобы сделать более точный вывод, необходимо обучить модели с разными начальными значениями счетчика случайных чисел и оценить средние значения производительности. Однако «оптимальная» функция активации также зависит от многих других факторов (количества слоев, типа слоев, размера слоев, задачи, набора данных, оптимизатора, скорости обучения и т.д.). На практике функции активации, которые хорошо работают с глубокими сетями, представляют собой все типы функций ReLU.

1.4. ИНИЦИАЛИЗАЦИЯ ВЕСОВ И ОПТИМИЗАЦИЯ ПАРАМЕТРОВ

При инициализации веса нейронной сети должны обладать несколькими свойствами. Во-первых, дисперсия входных данных должна оставаться одинаковой в слоях от первого до последнего, чтобы нейроны в выходном слое име-

ли аналогичное стандартное отклонение. Если дисперсия будет исчезать по мере прохождения данных через модель, то оптимизировать модель станет намного сложнее, поскольку данные превратятся в постоянное значение. Аналогично, если дисперсия увеличивается, она, скорее всего, будет «взрываться» (т.е. стремиться к бесконечности) в более глубоких слоях модели. Второе свойство, которое должно выполняться после инициализации, – сохранение одинаковой дисперсии по слоям распределения градиента. Если первый слой получает гораздо меньшие градиенты, чем последний слой, то становится трудно подобрать подходящую скорость обучения.

Попытаемся найти оптимальную инициализацию с точки зрения распределения активаций. Для этого сформулируем два требования:

1. Среднее значение активаций должно быть равно нулю.
2. Дисперсия активаций должна оставаться одинаковой в каждом слое.

Предположим, что мы хотим разработать инициализацию для следующего слоя: $y = Wx + b,\ y \in R^{d_y},\ x \in R^{d_x}$. Цель состоит в том, чтобы дисперсия каждого элемента y была равна дисперсии входов, т.е. $\mathrm{Var}(y_i) = \mathrm{Var}(x_i) = \sigma_x^2,$ и чтобы среднее значение было равно нулю. Мы предполагаем, что среднее значение x также равно нулю, потому что в глубоких нейронных сетях y будет входом другого слоя. Это условие требует, чтобы значения смещений и весов имели математическое ожидание, равное 0. На самом деле, поскольку b – это один элемент на слой, и он имеет одно и то же значение для разных входов, задаем его равным 0.

Далее нужно вычислить дисперсию, с которой необходимо инициализировать веса.

Для расчета нам понадобится следующее правило дисперсии. Если даны две независимые переменные X и Y, дисперсия их произведения равна:

$$\mathrm{Var}(XY) = \mathbb{E}(Y)^2\,\mathrm{Var}(X) + \mathbb{E}(X)^2\,\mathrm{Var}(Y) + \mathrm{Var}(X)\,\mathrm{Var}(Y) = \mathbb{E}(Y^2)\,\mathbb{E}(X^2) - \mathbb{E}(Y)^2\,\mathbb{E}(Y)^2.$$

Необходимая дисперсия весов $\mathrm{Var}\left(w_{ij}\right)$ рассчитывается следующим образом:

$$y_i = \sum_j w_{ij} x_j;$$

$$\mathrm{Var}(y_i) = \sigma_x^2 = \mathrm{Var}\sum_j w_{ij}x_j = \sum_j \mathrm{Var}\left(w_{ij}x_j\right) = \sum_j \mathrm{Var}\left(w_{ij}\right)\mathrm{Var}(x_j) =$$

$$= d_x \mathrm{Var}\left(w_{ij}\right)\mathrm{Var}(x_j) = \sigma_x^2 d_x \mathrm{Var}\left(w_{ij}\right);$$

$$\mathrm{Var}\left(w_{ij}\right) = \sigma_W^2 = \frac{1}{d_x}.$$

Таким образом, мы должны инициализировать распределение весов с дисперсией, равной обратной величине размера входных данных d_x. Реализуем эту инициализацию ниже:

```
def equal_var_init(model):
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            param.data.fill_(0)
        else:
           param.data.normal_(std=1.0/math.sqrt(param.shape[1]))

equal_var_init(model)
```

Обратите внимание, что инициализация не ограничивается нормальным распределением, а допускает любое другое распределение со средним 0 и дисперсией $1/d_x$. Часто для инициализации используется равномерное распределение. Небольшое преимущество использования равномерного распределения вместо нормального состоит в том, что мы можем исключить возможность инициализации весов очень большими или малыми значениями.

Помимо дисперсии активаций, еще одна дисперсия, которую необходимо стабилизировать – это дисперсии градиентов. Их ограничение гарантирует стабильную оптимизацию параметров глубоких сетей. Мы можем сделать тот же расчет, что и выше, начиная с $\Delta x = W\Delta y$, и прийти к выводу, что необходимо инициализировать слои с дисперсией $1/d_y$, где d_y – число выходных нейронов.

В качестве компромисса между двумя описанными выше ограничениями было предложено использовать среднее гармоническое обоих значений, что приводит нас к известной инициализации Ксавье [2]:

$$W = \sim \mathcal{N}\left(0, \frac{2}{d_x + d_y}\right).$$

Если использовать равномерное распределение, то веса инициализируются следующим образом:

$$W = \sim U\left(-\frac{\sqrt{6}}{\sqrt{d_x + d_y}}, \frac{\sqrt{6}}{\sqrt{d_x + d_y}}\right).$$

Реализуем инициализацию Ксавье:

```
def xavier_init(model):
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            param.data.fill_(0)
        else:
            bound =
math.sqrt(6)/math.sqrt(param.shape[0]+param.shape[1])
            param.data.uniform_(-bound, bound)

xavier_init(model)
```

Инициализация Ксавье хорошо работает для сетей с функцией активации гиперболического тангенса. В сетях с активацией ReLU мы не можем принять предположение о том, что нелинейность исчезает при малых значениях. Функция активации ReLU устанавливает половину входов в 0, так что и ожидание входа не равно нулю. Однако, пока ожидание W равно нулю и $b = 0$, ожидание выхода равно нулю. Часть, в которой вычисление инициализации ReLU отличается от тождественности, при определении $\mathrm{Var}\left(w_{ij} x_j\right)$ выглядит так:

$$\mathrm{Var}\left(w_{ij} x_j\right) = \underbrace{\mathbb{E}\left[w_{ij}^2\right]}_{= \mathrm{Var}(w_{ij})} \mathbb{E}\left[x_j^2\right] - \underbrace{\mathbb{E}\left[w_{ij}\right]^2}_{= 0} \mathbb{E}\left[x_j\right]^2 = \mathrm{Var}\left(w_{ij}\right)\mathbb{E}\left[x_j^2\right].$$

Если предположить, что x является выходом активации ReLU (из предыдущего слоя, $x = \max(0, \tilde{y})$, мы можем вычислить ожидание следующим образом:

$$\mathbb{E}[x^2] = \mathbb{E}[\max(0, \tilde{y})^2] = \frac{1}{2}\mathbb{E}[\tilde{y}^2] = \frac{1}{2}\mathrm{Var}(\tilde{y}).$$

Таким образом, видно, что в уравнении есть дополнительный коэффициент 1/2, поэтому желаемая дисперсия веса становится равной $2/d_x$. Этот результат дает инициализацию Кайминга [3]. Обратите внимание, что инициализация Кайминга не использует среднее гармоническое между размерами входа и выхода. В своей работе авторы утверждают, что использование d_x или d_y приводит к стабильным градиентам по всей сети и зависит только от общего размера входа и выхода сети. Следовательно, мы можем использовать только вход d_x:

```
def kaiming_init(model):
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            param.data.fill_(0)
        # Первый слой не применяет ReLU
        elif name.startswith("layers.0"):
            param.data.normal_(0, 1/math.sqrt(param.shape[1]))
        else:
            param.data.normal_(0,
math.sqrt(2)/math.sqrt(param.shape[1]))

model = BaseNetwork(act_fn=nn.ReLU()).to(device)
kaiming_init(model)
```

Инициализация Кайминга действительно хорошо работает для сетей на основе ReLU. Обратите внимание, что для Leaky-ReLU и т.д. нужно слегка подкорректировать коэффициент 2 в значении 8дисперсии, так как половина значений уже не равна нулю. PyTorch предоставляет функцию для вычисления этого коэффициента для многих функций активации (смотрите документацию к `torch.nn.init.calculate_gain`).

Помимо инициализации важным, выбором для глубоких нейронных сетей может стать выбор подходящего алгоритма оптимизации. Прежде чем рассмотреть их подробнее, необходимо написать код для обучения моделей. Бóльшая часть следующего кода скопирована из предыдущих глав пособия и лишь слегка изменена:

```
def train_model(net, model_name, optim_func, max_epochs=50,
batch_size=256, overwrite=False):
    """
    Обучение модели на наборе FashionMNIST

    Входы:
        net – Модель.
        model_name – (str) Имя модели, используется для создания
контрольных точек.
        max_epochs – Максимальное количество эпох, в течение
которых происходит обучение модели.
        patience – Если производительность на проверочном
множестве не улучшилась за patience эпох, мы прекращаем обучение
раньше времени.
        batch_size – Размер пакетов, используемых при обучении.
        overwrite – Определяет, как поступать в случае, когда
контрольная точка уже существует. Если True, она будет перезаписана.
    """
    file_exists = os.path.isfile(_get_model_file(CHECKPOINT_PATH,
model_name))
    if file_exists and not overwrite:
        print(f"Model file of \"{model_name}\" already exists.
Skipping training...")
        with open(_get_result_file(CHECKPOINT_PATH, model_name),
"r") as f:
            results = json.load(f)
    else:
        if file_exists:
            print("Model file exists, but will be overwritten...")

        # Определение оптимизатора, функции потерь и
        # загрузчика данных.
        optimizer =  optim_func(net.parameters())
        loss_module = nn.CrossEntropyLoss()
        train_loader_local = data.DataLoader(train_set,
batch_size=batch_size, shuffle=True, drop_last=True,
pin_memory=True)

        results = None
        val_scores = []
```

```
        train_losses, train_scores = [], []
        best_val_epoch = -1
        for epoch in range(max_epochs):
            ############
            # Обучение #
            ############
            net.train()
            true_preds, count = 0., 0
            t = tqdm(train_loader_local, leave=False)
            for imgs, labels in t:
                imgs, labels = imgs.to(device), labels.to(device)
                optimizer.zero_grad()
                preds = net(imgs)
                loss = loss_module(preds, labels)
                loss.backward()
                optimizer.step()
                # Record statistics during training
                true_preds += (preds.argmax(dim=-1) =
= labels).sum().item()
                count += labels.shape[0]
                t.set_description(f"Epoch {epoch+1}:
loss={loss.item():4.2f}")
                train_losses.append(loss.item())
            train_acc = true_preds / count
            train_scores.append(train_acc)

            ############
            # Валидация #
            ############
            val_acc = test_model(net, val_loader)
            val_scores.append(val_acc)
            print(f"[Epoch {epoch+1:2d}] Training accuracy:
{train_acc*100.0:05.2f}%, Validation accuracy:
{val_acc*100.0:05.2f}%")

            if len(val_scores) == 1 or val_acc >
val_scores[best_val_epoch]:
                print("\t   (New best performance, saving
model...)")
                save_model(net, CHECKPOINT_PATH, model_name)
```

```
                best_val_epoch = epoch

    if results is None:
        load_model(CHECKPOINT_PATH, model_name, net=net)
        test_acc = test_model(net, test_loader)
        results = {"test_acc": test_acc, "val_scores": val_scores,
"train_losses": train_losses, "train_scores": train_scores}
        with open(_get_result_file(CHECKPOINT_PATH, model_name),
"w") as f:
            json.dump(results, f)

    # Построение кривой Accuracy валидации
    sns.set()
    plt.plot([i for i in range(1,len(results["train_scores"])+1)],
results["train_scores"], label="Train")
    plt.plot([i for i in range(1,len(results["val_scores"])+1)],
results["val_scores"], label="Val")
    plt.xlabel("Epochs")
    plt.ylabel("Validation accuracy")
    plt.ylim(min(results["val_scores"]),
max(results["train_scores"])*1.01)
    plt.title(f"Validation performance of {model_name}")
    plt.legend()
    plt.show()
    plt.close()

    print((f" Test accuracy: {results['test_acc']*100.0:4.2f}%
").center(50, "=")+"\n")
    return results


def test_model(net, data_loader):
    """
    Тестирование модели

    Входы:
        net - Обученная модель типа BaseNetwork
        data_loader - Объект DataLoader набора данных
для тестирования (валидация или тест)
    """
```

```
    net.eval()
    true_preds, count = 0., 0
    for imgs, labels in data_loader:
        imgs, labels = imgs.to(device), labels.to(device)
        with torch.no_grad():
            preds = net(imgs).argmax(dim=-1)
            true_preds += (preds == labels).sum().item()
            count += labels.shape[0]
    test_acc = true_preds / count
    return test_acc
```

Во-первых, разберемся, для чего нужен оптимизатор. Оптимизатор отвечает за обновление параметров сети с учетом градиентов. Следовательно, мы реализуем функцию $w^t = f(w^{t-1}, g^t, ...)$, где w – параметры, а $g^t = \nabla_{w(t-1)} L^{(t)}$ – градиенты на шаге времени t. Дополнительным параметром этой функции является скорость обучения, обозначаемая η. Обычно скорость обучения можно рассматривать как «размер шага» обновления. Бóльшая скорость обучения означает, что мы изменяем веса в направлении градиентов большими шагами, меньшая – что мы делаем более короткие шаги.

Поскольку большинство оптимизаторов отличаются только реализацией функции f, мы можем определить шаблон оптимизатора в PyTorch. В качестве входных данных используем параметры модели и скорость обучения. Функция `zero_grad()` устанавливает градиенты всех параметров в ноль. Эту операцию мы должны сделать перед вызовом `loss.backward()`. Наконец, функция `step()` оптимизатора выполняет обновление всех весов на основе их градиентов. Шаблон оптимизатора может быть реализован следующим образом:

```
class OptimizerTemplate:

    def __init__(self, params, lr):
        self.params = list(params)
        self.lr = lr

    def zero_grad(self):
        # Установить градиенты всех параметров в ноль
        for p in self.params:
```

```
            if p.grad is not None:
                # Для оптимизаторов второго порядка
                p.grad.detach_()
                p.grad.zero_()

    @torch.no_grad()
    def step(self):
        # Применить шаг обновления ко всем параметрам
        for p in self.params:
            # Пропускаем параметры без градиентов
            if p.grad is None:
                continue
            self.update_param(p)

    def update_param(self, p):
        # Должны быть реализованы в классах,
        # специфичных для оптимизатора
        raise NotImplementedError
```

Первый оптимизатор, который мы реализуем, – стандартный стохастический градиентный спуск (Stochastic Gradient Descent, SGD). SGD обновляет параметры, используя следующее уравнение:

$$w^{(t)} = w^{(t-1)} - \eta g^{(t)}.$$

Реализация SGD:

```
class SGD(OptimizerTemplate):

    def __init__(self, params, lr):
        super().__init__(params, lr)

    def update_param(self, p):
        p_update = -self.lr * p.grad
        # Обновление на месте для экономии памяти, не создает
        # вычислительный граф
        p.add_(p_update)
```

В некоторых оптимизаторах существует понятие импульса, который заменяет градиент в обновлении экспоненциальным средним всех прошлых градиентов, включая текущий:

$$m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1) g^{(t)};$$

$$w^{(t)} = w^{(t-1)} - \eta m^{(t)}.$$

Реализация:

```
class SGDMomentum(OptimizerTemplate):

    def __init__(self, params, lr, momentum=0.0):
        super().__init__(params, lr)
        # Соответствует beta_1 в уравнении выше
        self.momentum = momentum
        # Словарь для хранения m_t
        self.param_momentum = {p: torch.zeros_like(p.data) for p
in self.params}

    def update_param(self, p):
        self.param_momentum[p] = (1 - self.momentum) * p.grad +
self.momentum * self.param_momentum[p]
        p_update = -self.lr * self.param_momentum[p]
        p.add_(p_update)
```

Оптимизатор Adam объединяет идею импульса с адаптивной скоростью обучения, которая основана на экспоненциальном среднем квадратичном значении градиентов, т.е. норме градиентов. Кроме того, добавляется коррекция смещения для импульса и адаптивной скорости обучения на первых итерациях [4]:

$$m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1) g^{(t)};$$

$$v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2)(g^{(t)})^2;$$

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta_1^t}, \ \hat{v}^{(t)} = \frac{v^{(t)}}{1-\beta_2^t};$$

$$w^{(t)} = w^{(t-1)} - \frac{\eta}{\sqrt{\hat{v}^{(t)}} + \varepsilon} \circ \hat{m}^{(t)}.$$

В выражениях выше ε – это небольшая константа, используемая для улучшения численной стабильности при очень малых нормах градиента. Адаптивная скорость обучения не заменяет гиперпараметр скорости обучения η, а действует как дополнительный фактор и гарантирует, что градиенты различных параметров имеют схожую норму.

Реализация:

```
class Adam(OptimizerTemplate):

    def __init__(self, params, lr, beta1=0.9, beta2=0.999,
eps=1e-8):
        super().__init__(params, lr)
        self.beta1 = beta1
        self.beta2 = beta2
        self.eps = eps
        # Запоминает t каждого параметра для коррекции
        # смещения
        self.param_step = {p: 0 for p in self.params}
        self.param_momentum = {p: torch.zeros_like(p.data) for p
in self.params}
        self.param_2nd_momentum = {p: torch.zeros_like(p.data)
for p in self.params}

    def update_param(self, p):
        self.param_step[p] += 1

        self.param_momentum[p] = (1 - self.beta1) * p.grad +
self.beta1 * self.param_momentum[p]
        self.param_2nd_momentum[p] = (1 - self.beta2) *
(p.grad)**2 + self.beta2 * self.param_2nd_momentum[p]

        bias_correction_1 = 1 - self.beta1 ** self.param_step[p]
        bias_correction_2 = 1 - self.beta2 ** self.param_step[p]

        p_2nd_mom = self.param_2nd_momentum[p] / bias_correction_2
        p_mom = self.param_momentum[p] / bias_correction_1
        p_lr = self.lr / (torch.sqrt(p_2nd_mom) + self.eps)
        p_update = -p_lr * p_mom

        p.add_(p_update)
```

Все оптимизаторы одинаково хорошо работают с нашей моделью. Различия слишком малы, чтобы можно было сделать какой-либо значимый вывод. Однако следует помнить, что результат может быть связан с выбранной инициализацией. При изменении инициализации в худшую сторону (например, инициализация константным значением) Adam обычно показывает себя более устойчивым благодаря адаптивной скорости обучения.

В PyTorch планировщики скорости (learning rate schedulers) являются важным инструментом при обучении нейронных сетей. Они позволяют динамически изменять скорость обучения во время обучения модели. Такой подход позволяет улучшить сходимость обучения, предотвратить переобучение и повысить обобщающую способность модели. Несколько популярных планировщиков скорости в PyTorch:

1. Планировщик `StepLR`: уменьшает скорость обучения на заданный коэффициент `gamma` каждые `step_size` эпох обучения. Этот планировщик позволяет динамически изменять скорость обучения во время обучения модели.

Параметры:

– `optimizer` (обязательный): оптимизатор, для которого будет применяться планировщик скорости;

– `step_size` (обязательный): число, которое определяет, через сколько эпох нужно уменьшить скорость обучения;

– `gamma` (обязательный): коэффициент, на который будет уменьшаться скорость обучения. Например, если `gamma=0.1`, то скорость обучения уменьшится в 10 раз;

– `last_epoch` (по умолчанию –1): число, которое позволяет указать последнюю эпоху, на которой применяется планировщик. По умолчанию –1 означает, что планировщик будет применяться сразу после инициализации.

Пример использования:

```
import torch
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR

optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
# Создание планировщика
scheduler = StepLR(optimizer, step_size=10, gamma=0.5)
```

Мы создаем планировщик StepLR с параметрами `step_size=10` и `gamma=0.5`. Это означает, что скорость обучения будет уменьшаться в 2 раза каждые 10 эпох обучения.

2. Планировщик `MultiStepLR`: позволяет задать несколько точек, на которых будет происходить изменение скорости обучения. Этот планировщик позволяет более гибко управлять скоростью обучения во время обучения модели.

Параметры:

– `optimizer` (обязательный): оптимизатор, для которого будет применяться планировщик скорости;

– `milestones` (обязательный): список, содержащий эпохи, на которых нужно изменить скорость обучения. Например, `milestones=[10, 20, 30]` означает, что скорость обучения будет изменена на указанный коэффициент на эпохах 10, 20 и 30;

– `gamma` (обязательный): коэффициент, на который будет уменьшаться скорость обучения. Например, если `gamma=0.1`, то скорость обучения уменьшится в 10 раз;

– `last_epoch` (по умолчанию –1): число, которое позволяет указать последнюю эпоху, на которой применяется планировщик. По умолчанию –1 означает, что планировщик будет применяться сразу после инициализации.

Пример использования:

```
import torch
import torch.optim as optim
from torch.optim.lr_scheduler import MultiStepLR

optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
# Создание планировщика
scheduler = MultiStepLR(optimizer, milestones=[10, 20, 30], gam-
ma=0.1)
```

Мы создали планировщик MultiStepLR с параметрами `milestones=[10, 20, 30]` и `gamma=0.1`. Это означает, что скорость обучения будет уменьшаться в 10 раз на эпохах 10, 20 и 30.

3. Планировщик `ExponentialLR`: уменьшает скорость обучения с экспоненциальным затуханием по заданному коэффициенту `gamma` на каждой эпохе обучения.

Параметры:

– `optimizer` (обязательный): оптимизатор, для которого будет применяться планировщик скорости;

– `gamma` (обязательный): коэффициент, на который будет уменьшаться скорость обучения. Например, если `gamma=0.1`, то скорость обучения уменьшится в 10 раз;

– `last_epoch` (по умолчанию –1): число, которое позволяет указать последнюю эпоху, на которой применяется планировщик. По умолчанию –1 означает, что планировщик будет применяться сразу после инициализации.

Пример использования:

```
import torch
import torch.optim as optim
from torch.optim.lr_scheduler import ExponentialLR

optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
# Создание планировщика
scheduler = ExponentialLR(optimizer, gamma=0.95)
```

Мы создали планировщик ExponentialLR с параметром `gamma=0.95`. Это означает, что скорость обучения будет уменьшаться на 5% на каждой эпохе.

4. Планировщик `CosineAnnealingLR`: изменяет скорость обучения по косинусной функции. Он позволяет динамически уменьшать скорость обучения в процессе обучения согласно заданной косинусной кривой.

Параметры:

– `optimizer` (обязательный): оптимизатор, для которого будет применяться планировщик скорости;

– `T_max` (обязательный): количество эпох, на которое рассчитывается период изменения скорости обучения;

– `eta_min` (по умолчанию 0): минимальное значение скорости обучения. По умолчанию равно 0.

Пример использования:

```
import torch
import torch.optim as optim
from torch.optim.lr_scheduler import CosineAnnealingLR

optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
# Создание планировщика
scheduler = CosineAnnealingLR(optimizer, T_max=50)
```

Мы создали планировщик CosineAnnealingLR с параметром `T_max=50`. Это означает, что период изменения скорости обучения составит 50 эпох.

5. Планировщик `CosineAnnealingWarmRestarts`: работает подобно планировщику `CosineAnnealingLR`, но с добавленной функциональностью «разогрева» (warm restarts). Это позволяет менять скорость обучения согласно косинусной кривой в течение нескольких эпох, а затем сбрасывать скорость обучения обратно к начальному значению, повторяя этот процесс в течение нескольких циклов.

Параметры:

– `optimizer` (обязательный): оптимизатор, для которого будет применяться планировщик скорости;

– `T_0` (обязательный): период разогрева в эпохах, в течение которого скорость обучения увеличивается от `eta_min` до исходного значения;

– `T_mult` (по умолчанию 1): коэффициент, определяющий, во сколько раз увеличивается T_0 после каждого цикла разогрева;

– `eta_min` (по умолчанию 0): минимальное значение скорости обучения. По умолчанию равно 0;

– `last_epoch` (по умолчанию –1): число, которое позволяет указать последнюю эпоху, на которой применяется планировщик. По умолчанию –1 означает, что планировщик будет применяться сразу после инициализации.

Пример использования:

```
import torch
import torch.optim as optim
```

```
from torch.optim.lr_scheduler import CosineAnnealingWarmRestarts
optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
# Создание планировщика
scheduler = CosineAnnealingWarmRestarts(optimizer, T_0=50,
T_mult=2, eta_min=0.001)
```

Мы создали планировщик `CosineAnnealingWarmRestarts` с параметрами `T_0=50`, `T_mult=2` и `eta_min=0.001`. Это означает, что скорость обучения будет увеличиваться от `eta_min` до исходного значения (`lr=0.1`) в течение 50 эпох, после чего она будет сбрасываться обратно к значению `eta_min`, а затем процесс будет повторяться, увеличивая `T_0` в `T_mult` раз после каждого цикла разогрева.

Планировщики скорости используются во время обучения следующим образом:

```
for epoch in range(num_epochs):
    # Обучение модели
    train(...)

    # Выполнение планировщика скорости на каждую эпоху
    scheduler.step()
```

При вызове метода `step()` планировщик скорости обновляет скорость обучения в зависимости от определенного расписания, которое было указано при его инициализации. Таким образом скорость обучения будет изменяться по указанному плану на каждую эпоху обучения. Это помогает улучшить сходимость и качество обучения модели. Важно также правильно подобрать параметры планировщика в зависимости от задачи и архитектуры модели.

В PyTorch есть несколько планировщиков, которые применяются не после эпох, а после каждого пакета (батча). Эти планировщики позволяют более гибко изменять скорость обучения на уровне отдельных пакетов (батчей) данных. Примеры таких планировщиков:

1. Планировщик `LambdaLR`: позволяет определить пользовательскую функцию, которая зависит от номера текущего пакета (батча) и изменяет

скорость обучения соответствующим образом в зависимости от начальной скорости:

$$lr_{epoch} = lr_{init} - 1 * Lambda(epoch).$$

Параметры:

– `optimizer` (обязательный): оптимизатор, для которого будет применяться планировщик скорости;

– `lr_lambda` (обязательный): функция, которая принимает номер текущего пакета (батча) и возвращает значение коэффициента, на который будет умножаться скорость обучения на данном пакете (батче). Функция должна быть определена пользователем.

Пример использования:

```
import torch
import torch.optim as optim
from torch.optim.lr_scheduler import LambdaLR

optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
# Определение функции, которая будет изменять скорость обучения
def lr_lambda(batch):
    return 0.95 ** batch

# Создание планировщика
scheduler = LambdaLR(optimizer, lr_lambda=lr_lambda)

# Цикл обучения
for epoch in range(num_epochs):
    for batch_idx, data in enumerate(train_loader):
        # Обучение модели на текущем пакете (батче) данных
        train(data)

        # Выполнение планировщика скорости на каждом пакете (бат-
че) данных
        scheduler.step()
```

Мы определили функцию `lr_lambda`, которая принимает номер текущего пакета (батча) данных `batch_idx` и возвращает значение 0.95 ** `batch_idx`. Затем мы создаем планировщик LambdaLR с параметром `lr_lambda`, который

ссылается на определенную функцию `lr_lambda`. Во вложенном цикле обучения для каждого пакета (батча) данных вызываем метод `step()` планировщика, который обновляет скорость обучения в соответствии с заданной функцией. Таким образом, скорость обучения будет уменьшаться на уровне отдельных пакетов (батчей) данных согласно указанной пользовательской функции.

2. Планировщик `MultiplicativeLR`: умножает скорость обучения на заданный коэффициент на каждом шаге обучения (батче). Позволяет уменьшать скорость обучения согласно выбранному коэффициенту на каждом шаге, что может быть полезно, если требуется экспоненциальное уменьшение скорости обучения:

$$lr_{epoch} = lr_{epoch-1} * Lambda(epoch).$$

Параметры:

– `optimizer` (обязательный): оптимизатор, для которого будет применяться планировщик скорости;

– `lr_lambda` (обязательный): функция, которая принимает номер текущего шага обучения (`batch_idx`) и возвращает значение коэффициента, на который будет умножаться скорость обучения на данном шаге. Функция должна быть определена пользователем;

– `last_epoch` (по умолчанию –1): число, которое позволяет указать последний шаг обучения, на котором применяется планировщик. Значение по умолчанию –1 означает, что планировщик будет применяться сразу после инициализации.

Пример использования:

```
import torch
import torch.optim as optim
from torch.optim.lr_scheduler import MultiplicativeLR

optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)

# Определение функции, которая будет изменять скорость обучения
def lr_lambda(batch_idx):
    return 0.95
```

```
# Создание планировщика
scheduler = MultiplicativeLR(optimizer, lr_lambda=lr_lambda)
# Цикл обучения
for epoch in range(num_epochs):
    for batch_idx, data in enumerate(train_loader):
        # Обучение модели на текущем батче данных
        train(data)

        # Выполнение планировщика скорости на каждом шаге обучения (батче данных)
        scheduler.step()
```

3. Планировщик `CyclicLR`: представляет собой циклическую функцию изменения скорости обучения на протяжении определенного количества шагов обучения (батчей). Позволяет создавать циклические колебания скорости обучения, что может помочь в поиске более оптимальных значений.

Параметры:

– `optimizer` (обязательный): оптимизатор, для которого будет применяться планировщик скорости;

– `base_lr` (обязательный): начальное значение скорости обучения, которое будет использоваться в цикле;

– `max_lr` (обязательный): максимальное значение скорости обучения, которое будет использоваться в цикле. Значение скорости обучения будет изменяться в интервале от `base_lr` до `max_lr`;

– `step_size_up` (обязательный): количество шагов обучения (батчей), за которое скорость обучения будет увеличиваться от `base_lr` до `max_lr`;

– `mode` (по умолчанию `triangular`): режим изменения скорости обучения. Может принимать значения `triangular`, `triangular2`, `exp_range`. В режиме `triangular` скорость обучения будет меняться линейно, в режиме `triangular2` она будет меняться синусоидально, а в режиме `exp_range` будет меняться по экспоненциальной функции;

– `gamma` (по умолчанию 1.0): параметр, используемый в режиме `exp_range`, задает множитель для экспоненциальной функции изменения скорости обучения.

Пример использования:

```
import torch
import torch.optim as optim
from torch.optim.lr_scheduler import CyclicLR

optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)

# Определение параметров планировщика
base_lr = 0.01
max_lr = 0.1
step_size_up = 100

# Создание планировщика
scheduler = CyclicLR(optimizer, base_lr=base_lr, max_lr=max_lr,
step_size_up=step_size_up)

# Цикл обучения
for epoch in range(num_epochs):
    for batch_idx, data in enumerate(train_loader):
        # Обучение модели на текущем пакете (батче) данных
        train(data)

        # Выполнение планировщика скорости на каждом шаге обучения
(батче данных)
        scheduler.step()
```

В приведенном примере `base_lr=0.01`, `max_lr=0.1`, и `step_size_up=100`, что означает, что скорость обучения будет изменяться от значения 0.01 до 0.1 в течение 100 шагов обучения, затем снова вернется к значения 0.01 и т.д., создавая циклическую функцию изменения скорости обучения.

2. ИСПОЛЬЗОВАНИЕ ФРЕЙМВОРКА PYTORCH LIGHTNING ДЛЯ РЕШЕНИЯ ЗАДАЧ КОМПЬЮТЕРНОГО ЗРЕНИЯ

Фреймворк PyTorch предоставляет различные слои, которые широко используются для создания сверточных нейронных сетей. Ниже – краткая информация о каждом из них:

1. `torch.nn.Conv2d`: представляет собой сверточный слой, который выполняет свертку двумерных входных данных с ядрами свертки для извлечения признаков. Он принимает на вход количество входных каналов (in_channels), количество выходных каналов (out_channels), размер ядра свертки (kernel_size) и другие параметры, такие как шаг (stride) и заполнение (padding). Пример использования: `torch.nn.Conv2d(in_channels, out_channels, kernel_size)`.

2. Слой `torch.nn.MaxPool2d`: представляет собой слой пулинга (уменьшения размерности – дискретизации), который выбирает максимальное значение в каждой области входных данных. Слой выполняет операцию максимального пулинга на двумерных входных данных и позволяет уменьшить размерность данных. Пример использования:

```
torch.nn.MaxPool2d(kernel_size).
```

3. Слой `torch.nn.AvgPool2d`: представляет собой слой пулинга, но вместо выбора максимального значения вычисляет среднее значение в каждой области входных данных. Слой выполняет операцию усредняющего пулинга на двумерных входных данных и также уменьшает размерность данных. Пример использования: `torch.nn.AvgPool2d(kernel_size)`.

4. Слой `torch.nn.BatchNorm2d`: представляет собой слой пакетной нормализации, который применяет нормализацию по пакету (батчу) к входным данным. Слой нормализует значения активаций на основе статистики по всему пакету (батчу), улучшая стабильность и скорость обучения модели. Пример использования: `torch.nn.BatchNorm2d(num_features)`.

5. Слой `torch.nn.Dropout`: представляет собой слой отключения (dropout), который случайным образом отключает нейроны во время обучения. Это помогает предотвратить переобучение и улучшить обобщающую способность модели. Пример использования: `torch.nn.Dropout(p)`.

Эти слои являются основными строительными блоками при создании сверточных нейронных сетей в PyTorch. Комбинируя их с другими слоями, функциями активации и операциями, можно создавать разнообразные архитектуры сверточных нейронных сетей для различных задач компьютерного зрения.

2.1. ПОСТРОЕНИЕ ПРОСТОЙ МОДЕЛИ

Далее мы обучим и оценим модель сверточной нейронной сети на наборе данных CIFAR10. В качестве первого шага рассчитаем среднее и стандартное отклонение по набору данных CIFAR:

```
train_dataset = CIFAR10(root=DATASET_PATH, train=True,
download=True)
DATA_MEANS = (train_dataset.data / 255.0).mean(axis=(0,1,2))
DATA_STD = (train_dataset.data / 255.0).std(axis=(0,1,2))
print("Data mean", DATA_MEANS)
print("Data std", DATA_STD)
```

Будем использовать эту информацию для определения модуля `transforms.Normalize`, который соответствующим образом будет нормализовать входные данные. Кроме того, используем технику увеличения данных (аугментации) во время обучения. Это позволит снизить риск переобучения и поможет сети стать инвариантной к изображениям разных классов. В частности, используем два случайных дополнения.

Во-первых, будем переворачивать каждое изображение по горизонтали с вероятностью 50% (`transforms.RandomHorizontalFlip`). Класс объекта обычно не меняется при переворачивании изображения, и мы не ожидаем, что какая-либо информация об объекте будет зависеть от горизонтальной ориентации. Однако ситуация будет иной, если мы попытаемся решить задачу обнару-

жения цифр или букв на изображении, поскольку они имеют строго определенную ориентацию.

Второе используемое дополнение – `transforms.RandomResizedCrop`. Это преобразование масштабирует изображение в небольшом диапазоне, изменяя при этом соотношение сторон, а затем обрезает его в прежнем размере. Таким образом, фактические значения пикселей меняются, а содержание или общая семантика изображения остаются неизменными.

Случайным образом разделим набор данных на обучающий и проверочный поднаборы. Проверочное множество будет использоваться для определения эпохи ранней остановки. После завершения обучения протестируем модель на тестовом наборе.

```
test_transform = transforms.Compose([transforms.ToTensor(),
                        transforms.Normalize(DATA_MEANS, DATA_STD)
                                    ])
# Добавление аугментаций
train_transform = transforms.Compose([transforms.
RandomHorizontalFlip(),
trans-
forms.RandomResizedCrop((32,32),scale=(0.8,1.0),ratio=(0.9,1.1)),
transforms.ToTensor(), transforms.Normalize(DATA_MEANS, DATA_STD)
                                    ])
# Загрузка обучающего набор данных, разделение его на обучающую
и проверочную части.
# Для проверочного набора не нужно использовать аугментации.
train_dataset = CIFAR10(root=DATASET_PATH, train=True,
transform=train_transform, download=True)
val_dataset = CIFAR10(root=DATASET_PATH, train=True,
transform=test_transform, download=True)
set_seed(42)
train_set, _ = torch.utils.data.random_split(train_dataset,
[45000, 5000])
set_seed(42)
_, val_set = torch.utils.data.random_split(val_dataset,
[45000, 5000])
# Загрузка тестового набора.
```

```
test_set = CIFAR10(root=DATASET_PATH, train=False,
transform=test_transform, download=True)

# Определение загрузчиков данных.
train_loader = data.DataLoader(train_set, batch_size=128,
shuffle=True, drop_last=True, pin_memory=True, num_workers=4)
val_loader = data.DataLoader(val_set, batch_size=128,
shuffle=False, drop_last=False, num_workers=4)
test_loader = data.DataLoader(test_set, batch_size=128,
shuffle=False, drop_last=False, num_workers=4)
```

Для проверки нормализации вычислим среднее и стандартное отклонение одного пакета. Среднее значение должно быть близко к 0, а стандартное отклонение – близко к 1 для каждого канала:

```
imgs, _ = next(iter(train_loader))
print("Batch mean", imgs.mean(dim=[0,2,3]))
print("Batch std", imgs.std(dim=[0,2,3]))

Batch mean tensor([-0.0143, -0.0204, -0.0017])
Batch std tensor([0.9772, 0.9650, 0.9809])
```

В этом примере и последующих используется библиотека PyTorch Lightning. PyTorch Lightning – это фреймворк, который упрощает написание кода, необходимого для обучения, оценки и тестирования модели в PyTorch. Он также имеет поддержку TensorBoard, включает набор инструментов визуализации экспериментов, и позволяет сохранять контрольные точки модели автоматически без написания лишнего кода. На момент написания пособия версия фреймворка 2.1. В будущих версиях могут быть изменены интерфейсы и, следовательно, часть кода может не работать.

Для начала импортируем PyTorch Lightning:

```
try:
    import pytorch_lightning as pl
except ModuleNotFoundError:
    !pip install --quiet pytorch-lightning>=2.1
```

```
import pytorch_lightning as pl
```

В PyTorch Lightning есть много полезных функций, например, функция для задания начального значения счетчика случайных числе:

```
pl.seed_everything(42)
```

Таким образом, в будущем больше не придется определять собственную функцию `set_seed()`.

В PyTorch Lightning необходимо определить модуль `pl.LightningModule` (наследующийся от `torch.nn.Module`), с применением которого код организуется в пяти основных секциях:

1. Секция инициализации (`__init__`), в которой определяются необходимые параметры/модели

2. Секция оптимизаторов (`configure_optimizers`), в которой определяются оптимизаторы, планировщик скорости обучения и т.д.

3. Цикл обучения (`training_step`), в котором необходимо определить способ вычисления потерь для одного пакета (циклы `optimizer.zero_grad()`, `loss.backward()` и `optimizer.step()`, а также все операции регистрации/сохранения выполняются в фоновом режиме).

4. Цикл проверки (`validation_step`), в котором, как и в случае с циклом обучения, необходимо определить, что должно происходить на каждом шаге.

5. Цикл тестирования (`test_step`), который является таким же, как и цикл валидации, только выполняется на тестовом наборе.

Таким образом, мы не реализуем абстракцию над кодом PyTorch, а скорее, упорядочиваем его и определяем некоторые часто используемые операции по умолчанию. Если необходимо что-то изменить в цикле обучения/валидации/тестирования, то существует множество возможных функций, которые можно переписать (подробности смотрите в документации).

Теперь рассмотрим пример того, как выглядит модуль Lightning для обучения сверточной нейронной сети:

```
class CIFARModule(pl.LightningModule):
```

```
    def __init__(self, model_name, model_hparams, optimizer_name,
optimizer_hparams):
    """
        Входы:
            model_name – имя модели
            model_hparams – гиперпараметры модели, словарь
            optimizer_name – имя используемого оптимизатора
            optimizer_hparams – гиперпараметры оптимизатора,
словарь
        """
        super().__init__()

        self.save_hyperparameters()

        self.model = create_model(model_name, model_hparams)

        self.loss_module = nn.CrossEntropyLoss()

        self.example_input_array = torch.zeros((1, 3, 32, 32),
dtype=torch.float32)

    def forward(self, imgs):

        return self.model(imgs)

    def configure_optimizers(self):

        if self.hparams.optimizer_name == "Adam":
            optimizer = optim.AdamW(
                self.parameters(),
**self.hparams.optimizer_hparams)
        elif self.hparams.optimizer_name == "SGD":
            optimizer = optim.SGD(self.parameters(),
**self.hparams.optimizer_hparams)
        else:
```

```
            assert False, f"Unknown optimizer:
\"{self.hparams.optimizer_name}\""

        # Уменьшить скорость обучения в 10 раз после 100 и 150
эпох.
        scheduler = optim.lr_scheduler.MultiStepLR(
            optimizer, milestones=[100, 150], gamma=0.1)
        return [optimizer], [scheduler]

    def training_step(self, batch, batch_idx):
        # "batch" is the output of the training data loader.
        imgs, labels = batch
        preds = self.model(imgs)
        loss = self.loss_module(preds, labels)
        acc = (preds.argmax(dim=-1) == labels).float().mean()

        # Логировать метрику каждую эпоху в tensorboard
        # (средневзвешенное значение по пакетам)
        self.log('train_acc', acc, on_step=False, on_epoch=True)
        self.log('train_loss', loss)
        return loss  # Вернуть тензор для вызова функции backward

    def validation_step(self, batch, batch_idx):
        imgs, labels = batch
        preds = self.model(imgs).argmax(dim=-1)
        acc = (labels == preds).float().mean()
        # Логировать метрику каждую эпоху
        # (средневзвешенное значение по пакетам)
        self.log('val_acc', acc)

    def test_step(self, batch, batch_idx):
        imgs, labels = batch
        preds = self.model(imgs).argmax(dim=-1)
        acc = (labels == preds).float().mean()
        # Логировать метрику каждую эпоху
        # (средневзвешенное значение по пакетам)
        self.log('test_acc', acc)
```

Другой важной частью PyTorch Lightning является концепция обратных вызовов (callback). Обратные вызовы – это функции, которые содержат несущественную логику модуля Lightning. Обычно они вызываются после завершения эпохи обучения, но могут также влиять на другие части цикла обучения. Например, используем следующие два предопределенных обратных вызова: `LearningRateMonitor` и `ModelCheckpoint`. Обратный вызов `LearningRateMonitor` добавляет текущую скорость обучения в TensorBoard, что помогает проверить, правильно ли работает планировщик скорости обучения. Обратный вызов `ModelCheckpoint` позволяет настроить процедуру сохранения контрольных точек. Например, сколько контрольных точек сохранять, когда именно сохранять, на какую метрику обращать внимание и т.д. Мы импортируем их ниже:

```
# Обратные вызовы
from pytorch_lightning.callbacks import LearningRateMonitor,
ModelCheckpoint
```

Если бы мы передавали классы или объекты непосредственно в качестве аргумента модулю Lightning, то не смогли бы воспользоваться преимуществами автоматического сохранения и загрузки гиперпараметров PyTorch Lightning.

Помимо основного модуля Lightning, вторым по важности модулем в PyTorch Lightning является `Trainer`. Он отвечает за выполнение шагов обучения, определенных в модуле Lightning, и завершает работу фреймворка. Как и в модуле Lightning, можно переопределить любую ключевую часть, но стандартные реализации часто являются наилучшим выбором. Ниже приведены наиболее важные функции, которые будут использоваться в примерах:

– `trainer.fit`: принимает на вход модуль `lightning` – модель, набор данных для обучения и (необязательно) набор данных для проверки. Эта функция обучает модуль на тренировочном наборе данных с периодической проверкой (по умолчанию один раз за эпоху, но частоту можно изменить);

– `trainer.test`: принимает на вход модель и набор данных, на котором необходимо провести тестирование. Возвращает метрику на тестовом наборе данных.

Во время процедур обучения и тестирования не нужно явно для модели определять режим оценивания (`model.eval()`), поскольку PyTorch Lightning выполняет эту операцию автоматически. Ниже показано определение функции обучения модели:

```
def train_model(model_name, save_name=None, **kwargs):
    """
    Входы:
        model_name – имя модели
        save_name (необязательно) – имя модели для сохранения
    """
    if save_name is None:
        save_name = model_name

    # Создать trainer PyTorch Lightning с обратным вызовом
    trainer = pl.Trainer(default_root_dir = \
              os.path.join(CHECKPOINT_PATH, save_name), \
              accelerator="gpu" if \
              str(device).startswith("cuda") else "cpu", \
              devices=1, max_epochs=180,
              callbacks=[ModelCheckpoint(save_weights_only=True, \
              mode="max", monitor="val_acc"), \
              LearningRateMonitor("epoch")], ena-
ble_progress_bar=True)
    trainer.logger._log_graph = True
    trainer.logger._default_hp_metric = None

    pl.seed_everything(42) # To be reproducable
    model = CIFARModule(model_name=model_name, **kwargs)
    trainer.fit(model, train_loader, val_loader)
    model = CIFARMod-
ule.load_from_checkpoint(trainer.checkpoint_callback.
best_model_path)

    val_result = trainer.test(model, val_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {"test": test_result[0]["test_acc"], \
             "val": val_result[0]["test_acc"]}

    return model, result
```

2.2. РЕАЛИЗАЦИЯ АРХИТЕКТУРЫ ResNet

Статья ResNet [5] является одной из наиболее цитируемых статей по нейронным сетям. Несмотря на свою простоту, идея остаточных связей очень эффективна, поскольку она обеспечивает стабильное распространение градиента по нейронной сети. Вместо моделирования выражения $x_{l+1} = F(x_l)$, сеть моделирует $x_{l+1} = x_l + F(x_l)$, где F – нелинейное отображение (обычно последовательность модулей нейронной сети, таких как свертки, функции активации и нормализации), а l – номер слоя. Если выполнить обратное распространение через такие остаточные связи, то получим

$$\frac{\partial x_{l+1}}{\partial x_l} = I + \frac{\partial F(x_l)}{\partial x_l}.$$

Наличие единичной матрицы гарантирует стабильное распространение градиента, на которое меньше влияет сама функция F. Было предложено множество вариантов сети ResNet, которые в основном касаются вида функции F, или операций, применяемых к сумме.

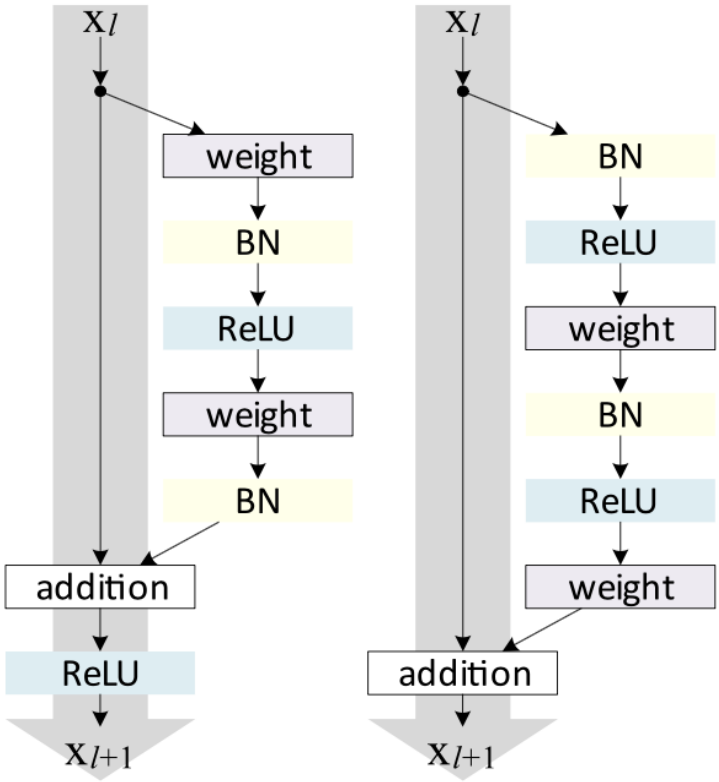


Рис. 4. Сравнение блоков ResNet и Pre-Activation ResNet

Далее в пособии рассматриваются два варианта блоков ResNey: оригинальный блок ResNet и блок Pre-Activation ResNet. На рисунке 4 показано сравнение этих блоков [6].

В оригинальном блоке ResNet нелинейная функция активации, обычно это ReLU применяется после пропускных соединений. В отличие от него, блок Pre-Activation ResNet с предварительной активацией применяет нелинейность в начале функции *F*. Оба варианта имеют свои преимущества и недостатки. Для очень глубоких сетей, однако, сеть с блоками Pre-Activation показывает себя лучше, поскольку градиентный поток гарантированно имеет единичную матрицу, как было показано выше, и ему не повредит никакая примененная к нему нелинейная активация.

Начнем с оригинального блока ResNet. Визуализация выше уже показывает, какие слои включены в функцию *F*. Единственный особый случай – необходимость уменьшения размеров изображения по ширине и высоте. Базовый блок ResNet требует, чтобы результат $F(x_l)$ имел ту же форму, что и x_l. Таким образом, необходимо изменить размер x_l перед добавлением к $F(x_l)$. В оригинальной реализации использовалось тождественное отображение с шагом 2 и добавление дополнительных признаков с нулевыми значениями. Однако более распространенной реализацией является использование свертки 1×1 с шагом 2, поскольку она позволяет изменять размер признака, при этом являясь эффективной по параметрам и вычислительным затратам. Код блока ResNet относительно прост и показан ниже:

```
class ResNetBlock(nn.Module):

    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        """
        Входы:
            c_in - Количество входных признаков.
            act_fn - Конструктор класса активации (например,
nn.ReLU).
            subsample - Если True, применяется страйд внутри блока
и выходная форма уменьшается в 2 раза по высоте и ширине.
```

```
            c_out – Количество выходных элементов. Обратите внима-
ние, что это имеет значение, только если subsample равен True, так
как в противном случае c_out = c_in
        """
        super().__init__()
        if not subsample:
            c_out = c_in

        # Представление F
        self.net = nn.Sequential(
            nn.Conv2d(c_in, c_out, kernel_size=3, padding=1,
stride=1 if not subsample else 2, bias=False),
            nn.BatchNorm2d(c_out),
            act_fn(),
            nn.Conv2d(c_out, c_out, kernel_size=3, padding=1,
bias=False),
            nn.BatchNorm2d(c_out)
        )

        self.downsample = nn.Conv2d(c_in, c_out, kernel_size=1,
stride=2) if subsample else None
        self.act_fn = act_fn()

    def forward(self, x):
        z = self.net(x)
        if self.downsample is not None:
            x = self.downsample(x)
        out = z + x
        out = self.act_fn(out)
        return out
```

Второй блок, который мы реализуем, – это блок Pre-Activation ResNet. Для этого нужно изменить порядок слоев в `self.net` и не применять функцию активации на выходе. Кроме того, для операции понижения размерности необходимо применить нелинейную функцию, так как входные данные, x_l, еще не были обработаны ей. Следовательно, блок будет выглядеть следующим образом:

```
class PreActResNetBlock(nn.Module):

    def __init__(self, c_in, act_fn, subsample=False, c_out=-1):
        """
        Входы:
            c_in - Количество входных признаков.
            act_fn - Конструктор класса активации (например,
nn.ReLU).
            subsample - Если True, применяется страйд внутри блока
и выходная форма уменьшается в 2 раза по высоте и ширине.
            c_out - Количество выходных элементов. Обратите
внимание, что это имеет значение, только если subsample равен
True, так как в противном случае c_out = c_in
        """
        super().__init__()
        if not subsample:
            c_out = c_in

        # Представление F
        self.net = nn.Sequential(
            nn.BatchNorm2d(c_in),
            act_fn(),
            nn.Conv2d(c_in, c_out, kernel_size=3, padding=1,
stride=1 if not subsample else 2, bias=False),
            nn.BatchNorm2d(c_out),
            act_fn(),
            nn.Conv2d(c_out, c_out, kernel_size=3, padding=1,
bias=False)
        )

        self.downsample = nn.Sequential(
            nn.BatchNorm2d(c_in),
            act_fn(),
            nn.Conv2d(c_in, c_out, kernel_size=1, stride=2,
bias=False)
        ) if subsample else None

    def forward(self, x):
```

```
        z = self.net(x)
        if self.downsample is not None:
            x = self.downsample(x)
        out = z + x
        return out
```

Общая архитектура сети ResNet состоит из множества блоков, некоторые из которых понижают дискретизацию входных данных. Блоки ResNet обычно группируют по одинаковой форме выхода. Следовательно, если мы говорим, что сеть ResNet имеет [3, 3, 3] блоков, это означает, что мы имеем три группы при три блока ResNet, а дискретизация входных данных происходит в четвертом и седьмом блоке. Сеть ResNet с блоками [3, 3, 3] для набора данных CIFAR10 показана на рис. 5.

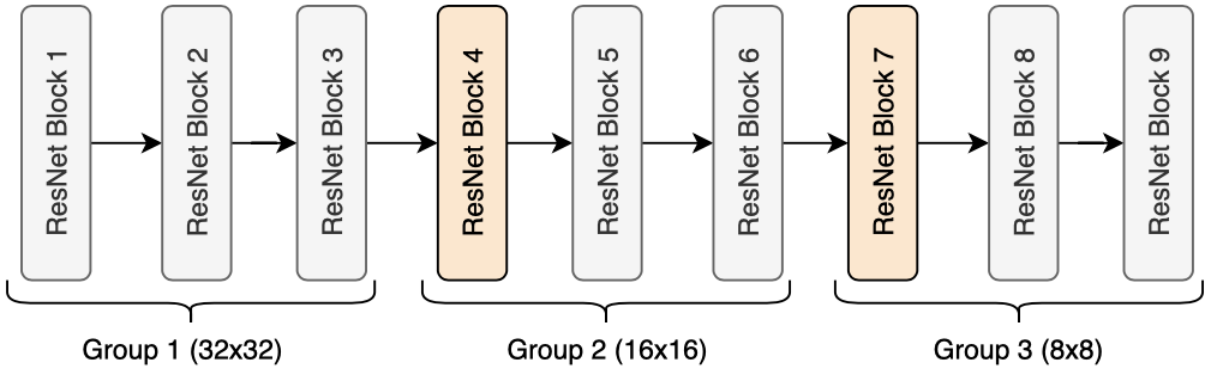


Рис. 5. Сеть ResNet с блоками [3, 3, 3]

В группах блоков используются данные с разрешениями 32×32, 16×16 и 8×8 пикселей соответственно. Блоки оранжевого цвета на рис. 5 обозначают блоки ResNet с дискретизацией. Та же нотация используется во многих других реализациях, например, в библиотеке torchvision из PyTorch. Таким образом, итоговый код будет выглядеть следующим образом:

```
class ResNet(nn.Module):

    def __init__(self, num_classes=10, num_blocks=[3,3,3],
c_hidden=[16,32,64], act_fn_name="relu", block_name="ResNetBlock",
**kwargs):
        """
```

```
        Входы:
            num_classes – Количество выходов классификатора
(10 для CIFAR10).
            num_blocks – Список с количеством блоков ResNet.
Первый блок каждой группы кроме первой группы использует
downsampling.
            c_hidden – Список со скрытыми размерностями
в различных блоках. Обычно умножается на 2 по мере увеличения
глубины.
            act_fn_name – Имя используемой функции активации,
из act_fn_by_name.
            block_name – Имя блока ResNet,
из resnet_blocks_by_name.
        """
        super().__init__()
        assert block_name in resnet_blocks_by_name
        self.hparams = SimpleNamespace(num_classes=num_classes,
                                       c_hidden=c_hidden,
                                       num_blocks=num_blocks,
                                       act_fn_name=act_fn_name,

act_fn=act_fn_by_name[act_fn_name],

block_class=resnet_blocks_by_name[block_name])
        self._create_network()
        self._init_params()

    def _create_network(self):
        c_hidden = self.hparams.c_hidden

        if self.hparams.block_class == PreActResNetBlock:
            self.input_net = nn.Sequential(
                nn.Conv2d(3, c_hidden[0], kernel_size=3,
padding=1, bias=False)
            )
        else:
            self.input_net = nn.Sequential(
```

```
                nn.Conv2d(3, c_hidden[0], kernel_size=3,
padding=1, bias=False),
                nn.BatchNorm2d(c_hidden[0]),
                self.hparams.act_fn()
            )

        blocks = []
        for block_idx, block_count
in enumerate(self.hparams.num_blocks):
            for bc in range(block_count):
                subsample = (bc == 0 and block_idx > 0)
                blocks.append(

self.hparams.block_class(c_in=c_hidden[block_idx if not subsample
else (block_idx-1)],

act_fn=self.hparams.act_fn,
                                                subsample=subsample,

c_out=c_hidden[block_idx])
                )
        self.blocks = nn.Sequential(*blocks)

        self.output_net = nn.Sequential(
            nn.AdaptiveAvgPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(c_hidden[-1], self.hparams.num_classes)
        )

    def _init_params(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity=self.hparams.act_fn_name)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
```

```
        x = self.input_net(x)
        x = self.blocks(x)
        x = self.output_net(x)
        return x
```

Наконец, мы можем обучить модели ResNet. Использование оптимизатора Adam часто приводит к немного худшей точности на простых, неглубоких сетях ResNet. На 100% не ясно, почему Adam работает хуже, но одно из возможных объяснений связано с поверхностью потерь ResNet. Было показано [7], что ResNet создает более гладкие поверхности потерь, чем сети без пропускных соединений. Возможная визуализация поверхности потерь с пропускными соединениями и без них приведена на рис. 6.

Обучим приведенную выше модель с помощью оптимизатора SGD:

```
resnet_model, resnet_results = train_model(model_name="ResNet",
                                           model_hparams={"num_classes": 10,
                                                          "c_hidden": [16,32,64],
                                                          "num_blocks": [3,3,3],
                                                          "act_fn_name": "relu"},
                                           optimizer_name="SGD",
                                           optimizer_hparams={"lr": 0.1,
                                                              "momentum": 0.9,
                                                              "weight_decay": 1e-4})
```

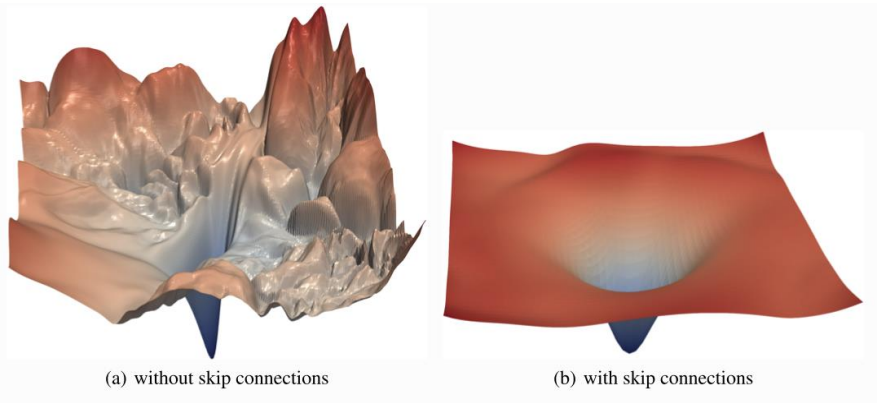


Рис. 6. Визуализация поверхности потерь с пропускными соединениями и без них

2.3. ИСПОЛЬЗОВАНИЕ СЛОЖНЫХ АУГМЕНТАЦИЙ

Аугментации MixUp, CutMix и Cutout являются популярными методами для расширения наборов данных при обучении нейронных сетей. Они помогают улучшить обобщающую способность модели, предотвратить переобучение и повысить ее устойчивость. Разберем каждую их техник более подробно.

MixUp представляет собой метод, при котором изображения и их метки классов линейно комбинируются с другими изображениями и их метками классов. Пусть x_i и y_i соответствуют i-му изображению и его метке класса в обучающем наборе данных. Затем для случайной пары (x_i, y_i) и (x_j, y_j), где $i \neq j$, случайным образом выбирается коэффициент смешивания $\lambda \sim Beta(\alpha, \alpha)$, где $Beta(\alpha, \alpha)$ – бета-распределение с параметром α (обычно $\alpha = 0{,}2$). Тогда смешанное изображение x_m будет задано следующим образом:

$$x_m = \lambda x_i + (1 - \lambda)\, x_j,$$

а метка y_m для смешанного изображения будет

$$y_m = \lambda y_i + (1 - \lambda)\, y_j.$$

Таким образом, происходит интерполяция между двумя изображениями и их метками.

Затем смешанное изображение x_m вместе с метками y_i и y_j используется для вычисления значения функции потерь на основе предсказаний модели для смешанного изображения. Обычно для задачи классификации используется кросс-энтропийная функция потерь:

$$\text{loss} = \lambda * nn.CrossEntropyLoss(model(x_m), y_i) +$$
$$+ (1 - \lambda) * nn.CrossEntropyLoss(model(x_m), y_j).$$

Пример реализации аугментации MixUp в PyTorch:

```
import numpy as np

def mixup_data(x, y, alpha=0.2):
    lam = np.random.beta(alpha, alpha)
    batch_size = x.size(0)
    index = torch.randperm(batch_size)
    mixed_x = lam * x + (1 - lam) * x[index, :]
    y_a, y_b = y, y[index]
    return mixed_x, y_a, y_b, lam

criterion = torch.nn.CrossEntropyLoss()
```

```
# Пример использования в цикле обучения
for inputs, targets in dataloader:
    inputs, targets = inputs.to(device), targets.to(device)
    mixed_inputs, targets_a, targets_b, lam = mixup_data(inputs,
targets)
    outputs = model(mixed_inputs)
    loss = lam * criterion(outputs, targets_a) + (1 - lam) *
criterion(outputs, targets_b)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Аугментация CutMix представляет собой метод, в котором реализуется случайное вырезание и смешивание части изображения с другим изображением. Такой подход позволяет создавать новые образцы данных, которые содержат части из одного изображения, добавленные в другое изображение, а также соответствующие метки классов, полученные путем линейной комбинации меток исходных изображений. Этот метод также уменьшает переобучение и повышает устойчивость модели к малым вариациям в данных.

Пусть x_i и y_i соответствуют i-му изображению и его метке класса в обучающем наборе данных. Затем для случайной пары (x_i, y_i) и (x_j, y_j), где $i \neq j$, случайным образом выбираются размер вырезаемой области и координаты области $bbx1$, $bby1$, $bbx2$, $bby2$. Затем эта область изображения x_i заменяет вырезанную область изображения x_j и соответствующим образом пересчитываются метки.

Математически, смешанное изображение x_m и соответствующая метка класса y_m вычисляются следующим образом:

$$x_m = \begin{cases} x_i, & \text{если для точки } (x, y)\ bbx1 \le x < bbx2,\ bby1 \le y < bby2; \\ x_j, & \text{в противном случае.} \end{cases}$$

$$y_m = \lambda y_i + (1-\lambda) y_i,$$

где λ – это площадь вырезанной области по отношению к общей площади изображения. Значение λ может быть вычислено как

$$\lambda = 1 - ((bbx2 - bbx1)(bby2 - bby1) / (width \cdot height)).$$

Для аугментации CutMix вычисление значения функции потерь основано на смешанном изображении x_m и соответствующей метке класса y_m. Используется смешанное изображение x_m вместе с метками y_i и y_j для вычисления значения функции потерь на основе предсказаний модели для смешанного изображения. Обычно для задачи классификации используется кросс-энтропийная функция потерь:

$$\text{loss} = \lambda \cdot nn.CrossEntropyLoss(model(x_m), y_i) +$$
$$+ (1 - \lambda) \cdot nn.CrossEntropyLoss(model(x_m), y_j).$$

Пример реализации аугментации CutMix в PyTorch:

```
import numpy as np
import random

def cutmix_data(x, y, beta=1.0):
    lam = np.random.beta(beta, beta)
    batch_size = x.size(0)
    index = torch.randperm(batch_size)
    bbx1, bby1, bbx2, bby2 = rand_bbox(x.size(), lam)
    mixed_x = x.clone()
    mixed_x[:, :, bbx1:bbx2, bby1:bby2] = x[index, :, bbx1:bbx2,
bby1:bby2]
    lam = 1 - ((bbx2 - bbx1) * (bby2 - bby1) / (x.size()[-1] *
x.size()[-2]))
    y_a, y_b = y, y[index]
    return mixed_x, y_a, y_b, lam

def rand_bbox(size, lam):
    width = size[2]
    height = size[3]
    cut_rat = np.sqrt(1. - lam)
    cut_w = np.int(width * cut_rat)
    cut_h = np.int(height * cut_rat)
    cx = np.random.randint(width)
    cy = np.random.randint(height)
    bbx1 = np.clip(cx - cut_w // 2, 0, width)
```

```
    bby1 = np.clip(cy - cut_h // 2, 0, height)
    bbx2 = np.clip(cx + cut_w // 2, 0, width)
    bby2 = np.clip(cy + cut_h // 2, 0, height)
    return bbx1, bby1, bbx2, bby2

criterion = torch.nn.CrossEntropyLoss()

# Пример использования в цикле обучения
for inputs, targets in dataloader:
    inputs, targets = inputs.to(device), targets.to(device)
    mixed_inputs, targets_a, targets_b, lam = cutmix_data(inputs,
targets)
    outputs = model(mixed_inputs)
    loss = lam * criterion(outputs, targets_a) + (1 - lam) *
criterion(outputs, targets_b)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Аугментация Cutout представляет собой метод, основанный на случайном удалении или замене пикселей внутри случайно выбранной области изображения. Эта операция приводит к удалению части информации из изображения и принудительному использованию модели для работы с неполными или частично зашумленными данными. Такая аугментация помогает сделать модель более устойчивой к шуму и повышает ее обобщающую способность.

Пусть x_i представляет собой i-е изображение в обучающем наборе данных, а y_i – соответствующая метка класса. Для применения аугментации Cutout случайно выбирается координаты начальной точки (x, y) области обрезания на изображении, а также размер обрезанной области. В этой области все пиксели заменяются нулевыми значениями или заменяются на некоторые фиксированные значения (например, черный цвет).

Формально, аугментированное изображение x_m с применением аугментации Cutout вычисляется следующим образом:

$$x_m = \begin{cases} 0, & \text{если } (x, y) \text{ принадлежит области вырезания;} \\ x_i, & \text{в противном случае.} \end{cases}$$

Для аугментации Cutout вычисление значения функции потерь основано на обучении модели на аугментированных данных и оценке ее производительности на них. Поскольку аугментация Cutout применяется во время обучения, значение функции потерь вычисляется на основе предсказаний модели для аугментированных данных и сравнивается с истинными метками классов.

Обычно для задачи классификации используется кросс-энтропийная функция потерь:

$$\text{loss} = nn.CrossEntropyLoss(model(x_m), y_i).$$

Пример реализации аугментации Cutout в PyTorch:

```
import numpy as np

def cutout_data(x, length=16):
    batch_size, n_channels, height, width = x.size()
    mask = np.ones((batch_size, n_channels, length, length))
    for i in range(batch_size):
        y = np.random.randint(height - length)
        x = np.random.randint(width - length)
        mask[i, :, y:y+length, x:x+length] = 0
    mask = torch.from_numpy(mask).to(x.device)
    x = x * mask
    return x

# Пример использования в цикле обучения
for inputs, targets in dataloader:
    inputs, targets = inputs.to(device), targets.to(device)
    cutout_inputs = cutout_data(inputs)
    outputs = model(cutout_inputs)
    loss = criterion(outputs, targets)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Значение параметра length можно изменить для контроля размера вырезаемой области, что позволяет варьировать степень аугментации Cutout и контро-

лировать влияние этой операции на обучение модели. Обычно используют значения от 8 до 32 пикселей в зависимости от размера изображений в обучающем наборе данных и сложности задачи.

Во время обучения модели который можно случайным образом применять аугментации MixUp, CutMix и Cutout. Приведенный ниже код основан на предположении, что уж есть загруженные данные и определена модель, функция потерь и оптимизатор.

```
import torch
import numpy as np

# Функция для аугментации MixUp
def mixup_data(x, y, alpha=0.2):
    lam = np.random.beta(alpha, alpha)
    batch_size = x.size(0)
    index = torch.randperm(batch_size)
    mixed_x = lam * x + (1 - lam) * x[index, :]
    y_a, y_b = y, y[index]
    return mixed_x, y_a, y_b, lam

# Функция для аугментации CutMix
def cutmix_data(x, y, beta=1.0):
    lam = np.random.beta(beta, beta)
    batch_size = x.size(0)
    index = torch.randperm(batch_size)
    bbx1, bby1, bbx2, bby2 = rand_bbox(x.size(), lam)
    mixed_x = x.clone()
    mixed_x[:, :, bbx1:bbx2, bby1:bby2] = x[index, :, bbx1:bbx2,
bby1:bby2]
    lam = 1 - ((bbx2 - bbx1) * (bby2 - bby1) / (x.size()[-1] *
x.size()[-2]))
    y_a, y_b = y, y[index]
    return mixed_x, y_a, y_b, lam

# Функция для аугментации Cutout
def cutout_data(x, length=16):
    batch_size, n_channels, height, width = x.size()
```

```
    mask = np.ones((batch_size, n_channels, length, length))
    for i in range(batch_size):
        y = np.random.randint(height - length)
        x = np.random.randint(width - length)
        mask[i, :, y:y+length, x:x+length] = 0
    mask = torch.from_numpy(mask).to(x.device)
    x = x * mask
    return x

# Пример использования аугментаций в цикле обучения
for inputs, targets in dataloader:
    inputs, targets = inputs.to(device), targets.to(device)

    # Случайным образом выбираем аугментацию
    augmentation = np.random.choice(['mixup', 'cutmix', 'cutout'])

    if augmentation == 'mixup':
        mixed_inputs, targets_a, targets_b, lam =
mixup_data(inputs, targets)
        outputs = model(mixed_inputs)
        loss = lam * criterion(outputs, targets_a) + (1 - lam) *
criterion(outputs, targets_b)

    elif augmentation == 'cutmix':
        mixed_inputs, targets_a, targets_b, lam = cut-
mix_data(inputs, targets)
        outputs = model(mixed_inputs)
        loss = lam * criterion(outputs, targets_a) + (1 - lam) *
criterion(outputs, targets_b)

    else:  # augmentation == 'cutout'
        cutout_inputs = cutout_data(inputs)
        outputs = model(cutout_inputs)
        loss = criterion(outputs, targets)

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

3. ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ PYTORCH IMAGE MODELS ДЛЯ РЕШЕНИЯ ЗАДАЧ КОМПЬЮТЕРНОГО ЗРЕНИЯ

PyTorch Image Models (timm) – это библиотека для решения задачи классификации изображений, которая содержит собственную обширную коллекцию моделей нейронных сетей, оптимизаторов, планировщиков, аугментаций и многое чего еще.

Одной из основных особенностей timm является большая и постоянно пополняющаяся коллекция архитектур моделей нейронных сетей. Многие из этих моделей содержат предварительно обученные веса – либо выученные с использованием фреймворка PyTorch, либо перенесенные из других библиотек, таких как Jax и TensorFlow.

Для начала оценим размер коллекции доступных моделей:

```
import torch
import timm
len(timm.list_models('*'))
```

Мы также можем использовать аргумент `pretrained` для выбора предварительно обученных моделей:

```
len(timm.list_models(pretrained=True))
```

Остановимся на использовании моделей семейства ResNet. Перечислим различные доступные варианты моделей ResNet, используя подстановочный знак в качестве фильтра:

```
timm.list_models('resnet*', pretrained=True)
```

```
['resnet18',
 'resnet18d',
 'resnet26',
 'resnet26d',
 'resnet26t',
 'resnet32ts',
```

```
 'resnet33ts',
 'resnet34',
 'resnet34d',
 'resnet50',
 'resnet50_gn',
 'resnet50d',
 'resnet51q',
 'resnet61q',
 'resnet101',
 'resnet101d',
 'resnet152',
 ...
 ]
```

Самый простой способ создания модели – использование функции `create_model()`, которая может быть использована для создания любой модели в библиотеке timm.

3.1. РЕАЛИЗАЦИЯ АРХИТЕКТУРЫ ResNet-D

Продемонстрируем использование этой функции на примере создания модели ResNet-D [8]. Это модификация архитектуры ResNet, в которой для дискретизации используется усредняющий пулинг (Average pooling).

```
model = timm.create_model('resnet50d', pretrained=True)
model
```

```
ResNet(
  (conv1): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), 
bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, 
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), 
bias=False)
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, 
track_running_stats=True)
    (5): ReLU(inplace=True)
```

```
    (6): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
  )
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (act1): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (act1): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (act2): ReLU(inplace=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (act3): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Identity()
        (1): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (act1): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (act2): ReLU(inplace=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (act3): ReLU(inplace=True)

... (часть вывода опущена)

    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (act1): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (act2): ReLU(inplace=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (act3): ReLU(inplace=True)
    )
  )
  (global_pool): SelectAdaptivePool2d (pool_type=avg, flat-
ten=Flatten(start_dim=1, end_dim=-1))
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
```

Как мы видим, это обычная модель формата PyTorch. Мы можем получить доступ к конфигурации модели, которая содержит статистику для нормализации входных данных, количество выходных классов и имя классификационной части сети.

```
model.default_cfg
```

```
{'url': 'https://github.com/rwightman/pytorch-image-
models/releases/download/v0.1-weights/resnet50d_ra2-464e36ba.pth',
 'num_classes': 1000,
 'input_size': (3, 224, 224),
 'pool_size': (7, 7),
 'crop_pct': 0.875,
 'interpolation': 'bicubic',
 'mean': (0.485, 0.456, 0.406),
 'std': (0.229, 0.224, 0.225),
 'first_conv': 'conv1.0',
 'classifier': 'fc',
 'architecture': 'resnet50d'}
```

Одна из менее известных, но невероятно полезных особенностей моделей библиотеки timm заключается в том, что модели способны использовать входные изображения с разным количеством каналов, что представляет проблему для большинства других библиотек.

Мы можем указать количество каналов входных изображений, передав аргумент `in_chans` в функцию `create_model()`:

```
model = timm.create_model('resnet50d', pretrained=True,
in_chans=1)

# Изображение с одним каналом
x = torch.randn(1, 1, 224, 224)

model(x).shape

torch.Size([1, 1000])
```

Используя случайный тензор для представления одноканального изображения в данном случае, мы видим, что модель обработала изображение и вернула ожидаемую форму на выходе.

Важно отметить, что, хотя такая особенность позволяет использовать предварительно обученную модель, входные данные значительно отличаются от данных, на которых обучалась модель. Поэтому не следует ожидать такого же уровня производительности, и перед использованием модели необходимо провести ее дообучение на новом наборе данных.

Помимо создания моделей со стандартными архитектурами, функция `create_model()` также поддерживает ряд аргументов, позволяющих настроить модель для частной задачи.

Поддерживаемые аргументы функции `create_model()` могут зависеть от базовой архитектуры модели, например:

– `global_pool`: определяет тип слоя глобального пулинга, который будет использоваться перед слоями классификации. Параметр зависит от того, используется ли в архитектуре слой глобального пулинга или нет. Например, не имеет смысла использовать его в модели ViT, в которой его просто нет.

Параметры, которые могут быть использованы практически для всех моделей:

– `drop_rate`: задает вероятность Dropout при обучении (по умолчанию – 0).

– `num_classes`: количество выходных нейронов, соответствующих классам.

Изучая конфигурацию модели, которую мы видели ранее, можно заметить, что имя классификационной части (головной части) – fc. Мы можем использовать это имя для получения прямого доступа к соответствующему модулю.

```
model.fc
```

```
Linear(in_features=2048, out_features=1000, bias=True)
```

Однако это имя, скорее всего, будет меняться в зависимости от используемой архитектуры модели. Чтобы обеспечить согласованный интерфейс для различных моделей, модели timm имеют метод `get_classifier`, который можно использовать для получения доступа к классификационной части без необходимости поиска имени модуля.

```
model.get_classifier()
```

```
Linear(in_features=2048, out_features=1000, bias=True)
```

Поскольку модель была предварительно обучена на наборе данных ImageNet, последний слой имеет 1000 нейронов по количеству классов. Можно изменить это число с помощью аргумента `num_classes`:

```
timm.create_model('resnet50d', pretrained=True,
num_classes=10).get_classifier()
Linear(in_features=2048, out_features=10, bias=True)
```

Если необходимо полностью избавиться от последнего слоя, можно установить число классов, равным 0, что приведет к созданию модели с функцией идентичной последнему слою. Эта особенность может быть полезна для проверки вывода предпоследнего слоя.

```
timm.create_model('resnet50d', pretrained=True,
num_classes=0).get_classifier()
Identity()
```

Из конфигурации модели также видно, что установлен параметр `pool_size`, информирующий о том, что перед классификатором используется слой глобального пулинга. Можно проверить это следующим образом:

```
model.global_pool
SelectAdaptivePool2d (pool_type=avg, flatten=Flatten(start_dim=1,
end_dim=-1))
```

Видно, что возвращается экземпляр `SelectAdaptivePool2d`, который является слоем, реализованным в timm, который поддерживает различные конфигурации пулинга и спрямления. На момент написания пособия поддерживаются следующие варианты пулинга:

– `avg`: усредняющий пулинг;

– `max`: максимальный пулинг;

– `avgmax`: сумма усредняющего и максимального пулингов, отмасштабированная до значения 0.5;

– `catavgmax`: объединение выходов усредняющего и максимального пулингов по размеру признака. При этом размер признака удваивается;

– `''`: пулинг не используется. Слой пулинга заменяется операцией Identity.

Можно визуализировать формы выходных данных при различных вариантах пулинга:

```
pool_types = ['avg', 'max', 'avgmax', 'catavgmax', '']

for pool in pool_types:
    model = timm.create_model('resnet50d', pretrained=True,
num_classes=0, global_pool=pool)
    model.eval()
    feature_output = model(torch.randn(1, 3, 224, 224))
    print(feature_output.shape)

torch.Size([1, 2048])
torch.Size([1, 2048])
torch.Size([1, 2048])
torch.Size([1, 4096])
torch.Size([1, 2048, 7, 7])
```

Можно изменить слои классификатора и пулинга существующей модели, используя метод `reset_classifier`:

```
m = timm.create_model('resnet50d', pretrained=True)

print(f'Original pooling: {m.global_pool}')
print(f'Original classifier: {m.get_classifier()}')
print('--------------------')
m.reset_classifier(10, 'max')
print(f'Modified pooling: {m.global_pool}')
print(f'Modified classifier: {m.get_classifier()}')
```

```
Original pooling: SelectAdaptivePool2d (pool_type=avg,
flatten=Flatten(start_dim=1, end_dim=-1))
Original classifier: Linear(in_features=2048, out_features=1000,
bias=True)
--------------------
Modified pooling: SelectAdaptivePool2d (pool_type=max,
flatten=Flatten(start_dim=1, end_dim=-1))
Modified classifier: Linear(in_features=2048, out_features=10,
bias=True)
```

Создадим модель ResNet с 10 классами. Воспользуемся пулингом `catavgmax` для объединения, чтобы предоставить больше информации на вход классификатору:

```
model = timm.create_model('resnet50d', pretrained=True,
num_classes=10, global_pool='catavgmax')
from torch import nn
```

Из существующего классификатора можем получить количество входных признаков:

```
num_in_features = model.get_classifier().in_features;
print(num_in_features)
```

```
4096
```

Теперь можно заменить последний слой модифицированной классификационной частью, обратившись к классификатору напрямую.

```
model.fc = nn.Sequential(
    nn.BatchNorm1d(num_in_features),
    nn.Linear(in_features=num_in_features, out_features=512,
bias=False),
    nn.ReLU(),
    nn.BatchNorm1d(512),
    nn.Dropout(0.4),
    nn.Linear(in_features=512, out_features=10, bias=False))
```

При тестировании модели получаем выход ожидаемой формы:

```
model.eval()
print(model(torch.randn(1, 3, 224, 224)).shape)

torch.Size([1, 10])
```

Снова создадим модель ResNet-D:

```
model = timm.create_model('resnet50d', pretrained=True)
model.default_cfg
{'url': 'https://github.com/rwightman/pytorch-image-
models/releases/download/v0.1-weights/resnet50d_ra2-464e36ba.pth',
 'num_classes': 1000,
 'input_size': (3, 224, 224),
 'pool_size': (7, 7),
 'crop_pct': 0.875,
 'interpolation': 'bicubic',
 'mean': (0.485, 0.456, 0.406),
 'std': (0.229, 0.224, 0.225),
 'first_conv': 'conv1.0',
 'classifier': 'fc',
 'architecture': 'resnet50d'}
```

Если нас интересует только конечная карта признаков, т.е. выход конечного сверточного слоя до объединения в данном случае, мы можем использовать метод `forward_features()`, чтобы обойти глобальные слои пулинга и классификации:

```
feature_output = model.forward_features(image)
```

3.2. ИСПОЛЬЗОВАНИЕ АУГМЕНТАЦИЙ

Библиотека timm включает в себя множество преобразований для аугментации данных, которые можно объединять в цепочки для создания конвейеров. Подобно torchvision, эти конвейеры ожидают на входе изображение формата PIL.

Самый простой способ использования агументаций – использовать функцию `create_transform ()`:

```
from timm.data.transforms_factory import create_transform
create_transform(224,)

Compose(
    Resize(size=256, interpolation=bilinear, max_size=None,
antialias=None)
    CenterCrop(size=(224, 224))
    ToTensor()
    Normalize(mean=tensor([0.4850, 0.4560, 0.4060]),
std=tensor([0.2290, 0.2240, 0.2250]))
)
create_transform(224, is_training=True)

Compose(
    RandomResizedCropAndInterpolation(size=(224, 224),
scale=(0.08, 1.0), ratio=(0.75, 1.3333), interpolation=bilinear)
    RandomHorizontalFlip(p=0.5)
    ColorJitter(brightness=[0.6, 1.4], contrast=[0.6, 1.4],
saturation=[0.6, 1.4], hue=None)
    ToTensor()
    Normalize(mean=tensor([0.4850, 0.4560, 0.4060]),
std=tensor([0.2290, 0.2240, 0.2250]))
)
```

Мы создали несколько основных конвейеров аугментаций, включая изменение размера, нормализацию и преобразование изображения в тензор. При установке `is_training=True` включаются дополнительные преобразования, такие как горизонтальное переворачивание и цветовой джиттер. Величину этих преобразований можно контролировать с помощью таких аргументов, как `hflip`, `vflip` и `color_jitter`.

В то время как при проверке используются стандартные аугментации `Resize` и `CenterCrop`, во время обучения используется аугментация `RandomResizedCropAndInterpolation`. Реализация этого преобразования в timm также позволяет устанавливать различные методы интерполяции изображения.

Приступая к новой задаче, бывает трудно понять, какие дополнения использовать и в каком порядке. При том количестве аугментаций, которое сейчас доступно, количество комбинаций огромно.

Хорошим началом является использование конвейера аугментаций, который продемонстрировал хорошую производительность в других похожих задачах. Одним из таких методов является `RandAugment` – автоматизированный метод аугментации данных, который равновероятно выбирает операции из набора аугментаций – таких как цветовой джиттер, постеризация, изменение контраста, изменение яркости, изменение резкости, сдвиг и т.д. – и последовательно применяет несколько из них к изображению. Более подробную информацию можно найти в оригинальной статье.

В timm параметры политики RandAugment определяются с помощью строки конфигурации, которая состоит из нескольких секций, разделенных знаком тире (–).

Первая секция определяет конкретный вариант RandAugment (в настоящее время поддерживается только `rand`). Остальные секции, которые могут быть расположены в любом порядке, следующие:

– `m` (int): магнитуда аугментации;

– `n` (int): количество операций аугментации, выбранных для каждого изображения. Это необязательное значение, по умолчанию – 2;

– `mstd` (float): стандартное отклонение магнитуды шума;

– `mmax` (int): устанавливает верхнюю границу магнитуды шума, по умолчанию – 10;

– `w` (int): индекс веса вероятности (индекс набора весов, влияющих на выбор операции);

– `inc` (bool): использовать увеличение сложности аугментаций. Это необязательное значение, по умолчанию – 0.

Например:

– `'rand-m9-n3-mstd0.5'`: RandAugment с магнитудой 9, три аугментации на изображение, magnitude_std – 0.5;

– `'rand-mstd1-w0'`: RandAugment с magnitude_std – 1.0, весом – 0, величиной `m` по умолчанию 10 и двумя аугментациями на изображение.

Передавая строку конфигурации в `create_transform()`, мы видим, что она обрабатывается объектом RandAugment, и мы можем увидеть названия всех доступных операций:

```
create_transform(224, is_training=True,
auto_augment='rand-m9-mstd0.5')

Compose(
    RandomResizedCropAndInterpolation(size=(224, 224),
scale=(0.08, 1.0), ratio=(0.75, 1.3333), interpolation=bilinear)
    RandomHorizontalFlip(p=0.5)
    RandAugment(n=2, ops=
     AugmentOp(name=AutoContrast, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Equalize, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Invert, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Rotate, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Posterize, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Solarize, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=SolarizeAdd, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Color, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Contrast, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Brightness, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Sharpness, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=ShearX, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=ShearY, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=TranslateXRel, p=0.5, m=9, mstd=0.5)
```

```
     AugmentOp(name=TranslateYRel, p=0.5, m=9, mstd=0.5))
    ToTensor()
    Normalize(mean=tensor([0.4850, 0.4560, 0.4060]),
std=tensor([0.2290, 0.2240, 0.2250]))
)
```

Можно создать такой объект для использования в конвейере с помощью функции `rand_augment_transform()`, как показано ниже:

```
from timm.data.auto_augment import rand_augment_transform

tfm = rand_augment_transform(
    config_str='rand-m9-mstd0.5',
    hparams={'img_mean': (124, 116, 104)}
)
tfm

RandAugment(n=2, ops=
     AugmentOp(name=AutoContrast, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Equalize, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Invert, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Rotate, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Posterize, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Solarize, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=SolarizeAdd, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Color, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Contrast, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Brightness, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=Sharpness, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=ShearX, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=ShearY, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=TranslateXRel, p=0.5, m=9, mstd=0.5)
     AugmentOp(name=TranslateYRel, p=0.5, m=9, mstd=0.5))
```

Библиотека timm предоставляет гибкую реализацию аугментаций CutMix и Mixup в виде класса `Mixup`, который реализует обе аугментации и предоставляет возможность переключения между ними.

Используя класс `Mixup`, можно выбирать из множества различных стратегий смешивания:

– `batch`: выбор между CutMix и Mixup, значение λ и области CutMix выбираются для всего пакета;

– `pair`: обе аугментации, значение λ и области выбираются для выборочных пар в пределах пакета;

– `elem`: обе аугментации, значение λ и области выбираются для каждого изображения в пакете;

– `half`: то же самое, что и с `elem`, но одно из каждой пары смешанных изображений отбрасывается, так что каждый образец просматривается один раз за эпоху.

Создадим загрузчик данных DataLoader, выполним итерирование по нему и применим аугментации к пакетам:

```
from timm.data import ImageDataset
from torch.utils.data import DataLoader

def create_dataloader_iterator():
    dataset = ImageDataset('pets/images',
transform=create_transform(224))
    dl = iter(DataLoader(dataset, batch_size=2))
    return dl

dataloader = create_dataloader_iterator()
inputs, classes = next(dataloader)
```

Теперь создадим аугментацию MixUp. Класс `Mixup` поддерживает следующие аргументы:

– `mixup_alpha` (float): альфа-значение MixUp. MixUp применяется, если значение > 0., (по умолчанию: 1);

– `cutmix_alpha` (float): альфа-значение CutMix. CutMix применяется, если значение > 0. (по умолчанию: 0);

– `cutmix_minmax` (List[float]): отношение min/max значений аугментации CutMix. CutMix использует это значение вместо с `cutmix_alpha`, если оно не None;

– `prob` (float): вероятность применения MixUp или CutMix на пакет или элемент (по умолчанию: 1);

– `switch_prob` (float): вероятность переключения на CutMix вместо MixUp, когда обе аугментации применяются (по умолчанию: 0,5);

– `mode` (str): способ применения параметров MixUp/CutMix (по умолчанию: `batch`);

– `label_smoothing` (float): «сила» операции сглаживания меток, применяемое к тензору смешанных целевых значений (по умолчанию: 0.1);

– `num_classes` (int): количество классов целевой переменной.

Определим набор аргументов, чтобы применять либо MixUp, либо CutMix к пакету изображений, чередуя их с вероятностью 1, и используем их для создания преобразования `Mixup`:

```
from timm.data.mixup import Mixup

mixup_args = {
    'mixup_alpha': 1.,
    'cutmix_alpha': 1.,
    'prob': 1,
    'switch_prob': 0.5,
    'mode': 'batch',
    'label_smoothing': 0.1,
    'num_classes': 2}

mixup_fn = Mixup(**mixup_args)
```

Поскольку MixUp и CutMix выполняются для пакета изображений, можно поместить пакет на GPU до применения аугментации в целях ускорения процесса:

```
mixed_inputs, mixed_classes =
mixup_fn(inputs.to(torch.device('cuda:0')), clas-
ses.to(torch.device('cuda:0')))
out = torchvision.utils.make_grid(mixed_inputs)
imshow(out, title=mixed_classes)
```

Библиотека timm предоставляет ряд полезных утилит для работы с различными типами наборов данных. Самый простой способ начать работу – использовать функцию `create_dataset()`, которая создаст подходящий набор данных.

Функция `create_dataset()` всегда принимает два аргумента:

– `name`: имя набора данных, который необходимо загрузить;

– `root`: корневой каталог с набором данных в локальной файловой системе.

Но есть и дополнительные аргументы, которые можно использовать для указания опций, например, для загрузки обучающего или проверочного наборов.

Можно использовать функцию `create_dataset()` для загрузки данных из нескольких разных источников:

– из наборов данных, доступных в torchvision;

– из наборов данных, доступных в TensorFlow datasets;

– из наборов данных, хранящихся в локальных каталогах.

Рассмотрим некоторые из этих возможностей.

Чтобы загрузить набор данных, включенный в torchvision, достаточно указать префикс `torch/` перед именем набора данных. Если данных нет на диске, можно загрузить их, задав параметр `download=True`. Кроме того, можно указать, что необходимо загрузить обучающий набор данных с помощью аргумента `split`:

```
from timm.data import create_dataset

ds = create_dataset('torch/cifar10', 'cifar10', download=True,
split='train')
Files already downloaded and verified

ds, type(ds)

(Dataset CIFAR10
     Number of datapoints: 50000
     Root location: cifar10
     Split: Train,
 torchvision.datasets.cifar.CIFAR10)
```

Можно получить доступ к набору данных, как обычно, с помощью индекса:

```
ds[0]
```

```
(<PIL.Image.Image image mode=RGB size=32x32 at 0x7F74A150D100>, 6)
```

Можно загружать данные из локальных каталогов. В этих случаях достаточно использовать пустую строку (' ') в качестве имени набора данных.

Помимо возможности загрузки из каталогов в форме ImageNet, функция `create_dataset()` также позволяет извлекать данные из одного или нескольких tar-архивов.

Кроме того, до сих пор мы загружали необработанные изображения, поэтому воспользуемся аргументом `transform` для применения аугментаций:

```
from timm.data.transforms_factory import create_transform

ds = create_dataset(name='', root='imagenette/
imagenette2-320.tar', transform=create_transform(224))
image, label = ds[0]
image.shape
```

```
torch.Size([3, 224, 224])
```

3.3. ИСПОЛЬЗОВАНИЕ ОПТИМИЗАТОРОВ

Библиотека timm содержит большое количество оптимизаторов, некоторые из которых недоступны в фреймворке PyTorch. Например:

– AdamP;

– RMSPropTF: реализация RMSProp, основанная на оригинальной реализации в TensorFlow, с небольшими доработками; часто приводит к более стабильному обучению, чем версия PyTorch;

– LAMB: вариант оптимизатора FusedLAMB, который совместим с TPU при использовании PyTorch XLA;

– AdaBelief;

– MADGRAD;

– AdaHessian.

Оптимизаторы в timm поддерживают тот же интерфейс, что и оптимизаторы в `torch.optim`, и в большинстве случаев могут быть просто добавлены без каких-либо изменений.

Чтобы увидеть все оптимизаторы, которые реализован в timm, можно просмотреть модуль `timm.optim`:

```
import inspect
import timm.optim

[cls_name for cls_name, cls_obj in inspect.getmembers(timm.optim)
if inspect.isclass(cls_obj) if cls_name !='Lookahead']

['AdaBelief',
 'Adafactor',
 'Adahessian',
 'AdamP',
 'AdamW',
 'Lamb',
 'Lars',
 'MADGRAD',
 'Nadam',
 'NvNovoGrad',
 'RAdam',
 'RMSpropTF',
 'SGDP']
```

Самый простой способ создать оптимизатор – использовать функцию `create_optimizer_v2()`, которая ожидает следующие параметры:

- модель или набор параметров;
- имя оптимизатора;
- любые аргументы для передачи оптимизатору.

Можно использовать эту функцию для создания любой реализации оптимизатора из timm, а также популярных оптимизаторов из `torch.optim` и оптимизаторов из Apex (если они установлены).

Рассмотрим несколько примеров:

```
import torch

model = torch.nn.Sequential(
    torch.nn.Linear(2, 1),
    torch.nn.Flatten(0, 1)
)
optimizer = timm.optim.create_optimizer_v2(model, opt='sgd',
lr=0.01, momentum=0.8); optimizer, type(optimizer)

(SGD (
 Parameter Group 0
     dampening: 0
     lr: 0.01
     momentum: 0.8
     nesterov: True
     weight_decay: 0.0
 ),
 torch.optim.sgd.SGD)
```

Поскольку timm не содержит реализации SGD, он создал оптимизатор, используя реализацию из `torch.optim`.

Попробуем создать один из оптимизаторов, реализованных в timm:

```
optimizer = timm.optim.create_optimizer_v2(model,
                                           opt='lamb',
                                           lr=0.01,
                                           weight_decay=0.01)
optimizer, type(optimizer)
(Lamb (
 Parameter Group 0
     always_adapt: False
     betas: (0.9, 0.999)
     bias_correction: True
     eps: 1e-06
     grad_averaging: True
     lr: 0.01
     max_grad_norm: 1.0
```

```
    trust_clip: False
    weight_decay: 0.0

Parameter Group 1
    always_adapt: False
    betas: (0.9, 0.999)
    bias_correction: True
    eps: 1e-06
    grad_averaging: True
    lr: 0.01
    max_grad_norm: 1.0
    trust_clip: False
    weight_decay: 0.01
),
timm.optim.lamb.Lamb)
```

Была использована реализация Lamb из timm, и сокращение весов (`weight_decay`) было применено к группе параметров с номеров 1.

3.4. ИСПОЛЬЗОВАНИЕ ПЛАНИРОВЩИКОВ СКОРОСТИ

Перейдем к планировщикам скорости. На момент написания пособия timm содержит следующие планировщики:

– `StepLRSScheduler`: скорость обучения снижается каждые n шагов; аналогичен `torch.optim.lr_scheduler.StepLR`;

– `MultiStepLRSScheduler`: планировщик, поддерживающий несколько вех, на которых можно снизить скорость обучения; аналогичен `torch.optim.lr_scheduler.MultiStepLR`;

– `PlateauLRSScheduler`: уменьшает скорость обучения на заданный коэффициент каждый раз, когда заданная метрика достигает плато; аналогичен `torch.optim.lr_scheduler.ReduceLROnPlateau`;

– `CosineLRScheduler`: график косинусного затухания с перезапусками; аналогичен `torch.optim.lr_scheduler.CosineAnnealing-WarmRestarts`;

– `TanhLRScheduler`: график гиперболически-тангенциального затухания с перезапусками;

– `PolyLRScheduler`: график полиномиального затухания.

Хотя многие планировщики, реализованные в timm, имеют аналоги в PyTorch, версии timm часто имеют другие гиперпараметры по умолчанию, а также предоставляют дополнительные опции. Все планировщики timm поддерживают эпохи разогрева, а также имеют возможность использования случайного шума. Кроме того, планировщик `CosineLRS` и планировщик `PolyLRS` поддерживают опцию затухания, известную как k-decay.

Рассмотрим, как можно использовать планировщики из библиотки timm в сценарии обучения, а затем рассмотрим некоторые опции, которые предоставляют эти планировщики.

В отличие от планировщиков, включенных в PyTorch, хорошая практика заключается в обновлении планировщиков timm дважды за эпоху:

– метод `step_update()` должен вызываться после каждого обновления оптимизатора, с индексом следующего обновления;

– метод `step()` должен вызываться в конце каждой эпохи, с индексом следующей эпохи.

Явное указание количества обновлений и индексов эпох позволяет планировщикам timm устранить поведение last_epoch и –1, наблюдаемое в планировщиках PyTorch.

Пример использования планировщика timm:

```
training_epochs = 300
cooldown_epochs = 10
num_epochs = training_epochs + cooldown_epochs

optimizer = timm.optim.AdamP(my_model.parameters(), lr=0.01)
scheduler = timm.scheduler.CosineLRScheduler(optimizer,
t_initial=training_epochs)

for epoch in range(num_epochs):

    num_steps_per_epoch = len(train_dataloader)
    num_updates = epoch * num_steps_per_epoch

    for batch in training_dataloader:
        inputs, targets = batch
        outputs = model(inputs)
```

```
        loss = loss_function(outputs, targets)

        loss.backward()
        optimizer.step()
        scheduler.step_update(num_updates=num_updates)

        optimizer.zero_grad()

    scheduler.step(epoch + 1)
```

Чтобы продемонстрировать некоторые возможности, которые предлагает timm, рассмотрим некоторые из доступных гиперпараметров и то, как их изменение влияет на график скорости обучения.

Остановимся на планировщике CosineLRS, поскольку именно этот планировщик используется по умолчанию в timm. Однако, как описано выше, такие функции, как добавление разогрева и шума, присутствуют во всех перечисленных планировщиках.

Чтобы показать, чем планировщик `CosineAnnealingWarmRestarts` в timm отличается от планировщика, включенного в PyTorch, сначала посмотрим, как использовать реализацию `CosineAnnealingWarmRestarts` из torch.

Этот класс поддерживает следующие параметры:

– `T_0` (int): Количество итераций для первого перезапуска;

– `T_mult` (int): Коэффициент, увеличивающий `T_{i}` после перезапуска (по умолчанию: 1);

– `eta_min` (float): Минимальная скорость обучения (по умолчанию: 0);

– `last_epoch` (int): Индекс последней эпохи (по умолчанию: –1).

Чтобы задать планировщик, нужно определить следующие значения: количество эпох, количество обновлений, которые происходят за эпоху, и – если необходимо использовать перезапуск – количество шагов, через которое скорость обучения должна вернуться к исходному значению. Поскольку мы не используем никаких данных, можем задать эти параметры произвольно.

```
num_epochs=300
num_epoch_repeat = num_epochs//2
num_steps_per_epoch = 10
```

Создадим планировщик скорости обучения. Поскольку `T_0` требует, чтобы время до первого перезапуска было указано в количестве итераций (каждая итерация является пакетом), мы вычисляем значение, умножая индекс эпохи, в которую необходимо выполнить перезапуск, на количество шагов за эпоху. Мы также указываем, что скорость обучения никогда не должна опускаться ниже значения 1e-6.

```
model, optimizer = create_model_and_optimizer()
scheduler =
torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer,
     T_0=num_epoch_repeat*num_steps_per_epoch,
     T_mult=1,
     eta_min=1e-6,
     last_epoch=-1)
```

Теперь можно смоделировать использование этого планировщика в цикле обучения. Поскольку мы используем реализацию PyTorch, нужно вызывать функцию `step()` только после каждого обновления оптимизатора, т.е. один раз за цикл.

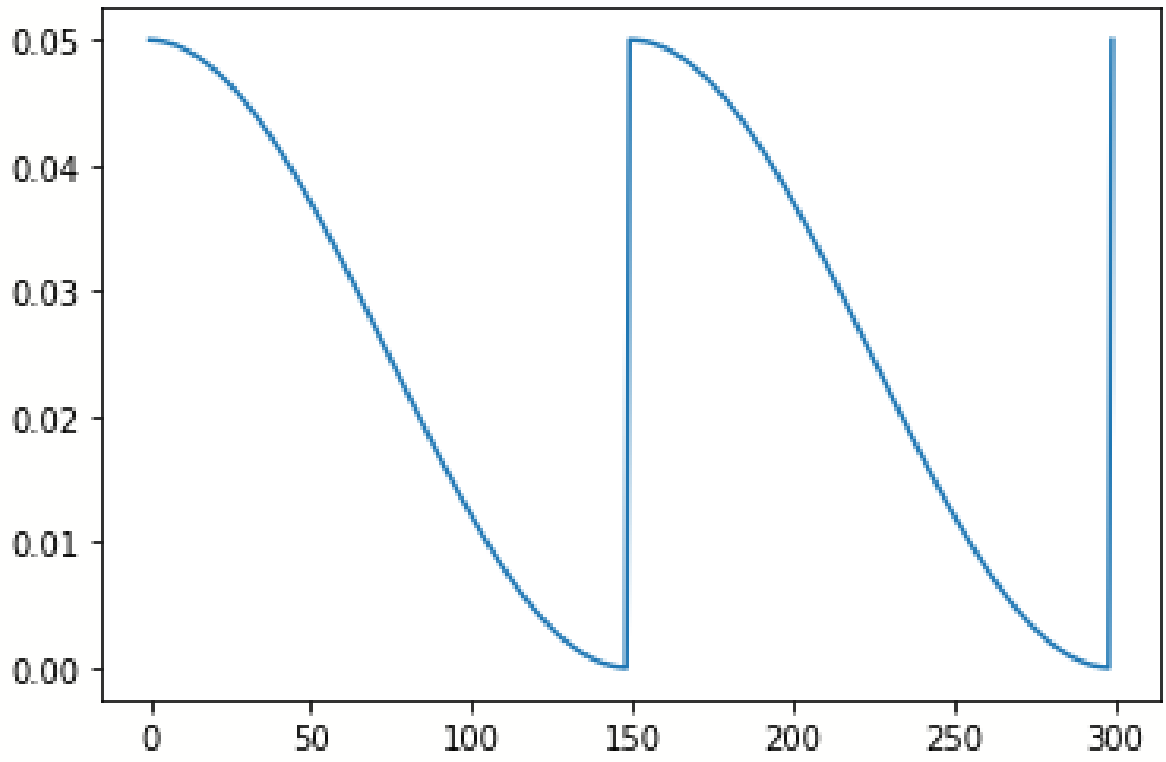


Рис. 7. График планирования скорости `CosineAnnealingWarmRestarts`

Из графика на рис. 7 видно, что скорость обучения снижалась до эпохи 150, после чего она вернулась к своему начальному значению и снова снизилась, как мы и ожидали.

Теперь сравним этот планировщик с реализацией, включенной в timm, и рассмотрим дополнительные опции, которые предлагаются библиотекой. Для начала воспроизведем предыдущий график, используя реализацию планировщика скорости обучения косинуса в timm `CosineLRScheduler`.

Некоторые аргументы `CosineLRScheduler` аналогичны тем, что мы видели ранее:

– `t_initial` (int): количество итераций для первого перезапуска; эквивалентно `T_0` в реализации Torch;

– `lr_min` (float): минимальная скорость обучения; эквивалентно `eta_min` в реализации torch (по умолчанию: 0);

– `cycle_mul` (float): фактор, увеличивающий `T_{i}` после перезапуска; эквивалентно `T_mult` в реализации PyTorch (по умолчанию: 1).

Однако, чтобы наблюдать поведение, соответствующее PyTorch, также потребуется установить следующие параметры:

– `cycle_limit` (int): ограничение количества перезапусков в цикле обучения (по умолчанию: 1);

– `t_in_epochs` (bool): указание количество итераций в эпохах, а не в количестве пакетов (по умолчанию: True).

Определим то же самое планирование скорости, что и ранее:

```
num_epochs=300
num_epoch_repeat = num_epochs/2
num_steps_per_epoch = 10
```

Теперь можно создать экземпляр планировщика. Мы выражаем количество итераций в количестве шагов обновления, а также увеличиваем `cycle_limit`, чтобы он превышал желаемое количество перезапусков; таким образом, параметры будут такими же, как мы использовали в реализации PyTorch ранее:

```
model, optimizer = create_model_and_optimizer()
```

```
scheduler = timm.scheduler.CosineLRScheduler(optimizer,
t_initial=num_epoch_repeat*num_steps_per_epoch,
                                             lr_min=1e-6,
     cycle_limit=num_epoch_repeat+1,
                                             t_in_epochs=False)
```

Определим новую функцию для имитации использования планировщика timm для обучения и зафиксируем обновления скорости обучения:

```
def plot_lrs_for_timm_scheduler(scheduler):
    lrs = []
    for epoch in range(num_epochs):
        num_updates = epoch * num_steps_per_epoch
        for i in range(num_steps_per_epoch):
            num_updates += 1
            scheduler.step_update(num_updates=num_updates)
        scheduler.step(epoch + 1)
        lrs.append(optimizer.param_groups[0]["lr"])
    plt.plot(lrs)
```

До сих пор мы выражали количество итераций в обновлениях оптимизатора, что требовало от нас вычисления количества итераций для первого перезапуска с помощью num_epoch_repeat * num_steps_per_epoch. Однако, задавая итерации в терминах эпох, что является значением по умолчанию в timm – мы можем избежать этого вычисления. Используя настройки по умолчанию, можно передать индекс эпохи, в которой необходимо произвести первый перезапуск, как показано ниже.

```
model, optimizer = create_model_and_optimizer()
scheduler = timm.scheduler.CosineLRScheduler(optimizer,
                              t_initial=num_epoch_repeat,
                              lr_min=1e-6,
                              t_in_epochs=True,
      cycle_limit=num_epoch_repeat+1)
plot_lrs_for_timm_scheduler(scheduler)
```

Еще одной особенностью всех оптимизаторов timm является то, что они поддерживают добавление разогрева и шума в график скорости обучения.

С помощью аргументов `warmup_t` и `warmup_lr_init` можно указать количество эпох разогрева и начальную скорость обучения, которая будет использоваться во время разогрева. Изменим планировщик, указав, что нужно 20 эпох разогрева:

```
model, optimizer = create_model_and_optimizer()

scheduler = timm.scheduler.CosineLRScheduler(optimizer,
                                     t_initial=num_epoch_repeat,
                                     lr_min=1e-5,
                                     warmup_lr_init=0.01,
                                     warmup_t=20,
cycle_limit=num_epoch_repeat+1)
plot_lrs_for_timm_scheduler(scheduler)
```

Мы также можем добавить шум в некоторое количество эпох, используя аргументы `noise_range_t` и `noise_pct`. Добавим небольшое количество шума к первым 150 эпохам:

```
model, optimizer = create_model_and_optimizer()
scheduler = timm.scheduler.CosineLRScheduler(optimizer,
     t_initial=num_epoch_repeat,
     lr_min=1e-5,
     noise_range_t=(0, 150),
                          noise_pct=0.1,
                          cycle_limit=num_epoch_repeat+1)
```

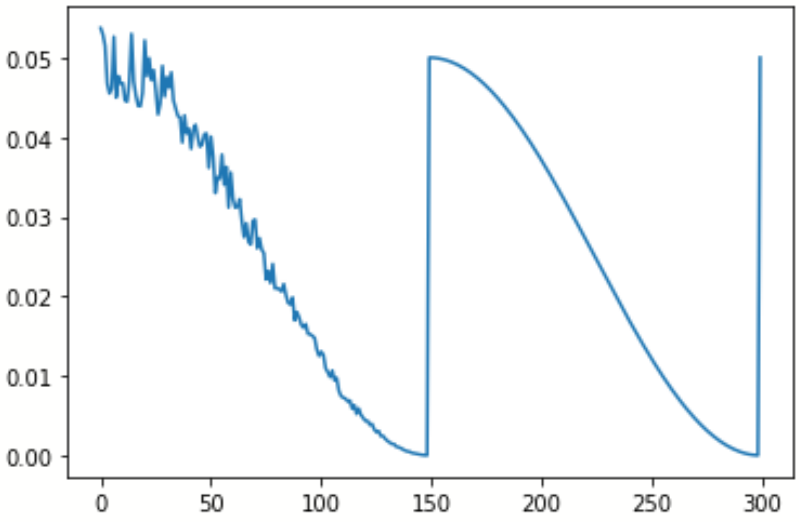


Рис. 8. График планирования скорости `CosineLRScheduler` с шумом

Видно (рис. 8), что до эпохи 150 добавленный шум влияет на график так, что скорость обучения не уменьшается по плавной кривой. Можем увеличить силу шума, увеличив значение `noise_pct` (рис. 9):

```
model, optimizer = create_model_and_optimizer()
scheduler = timm.scheduler.CosineLRScheduler(optimizer,
     t_initial=num_epoch_repeat,
                              lr_min=1e-5,
                              noise_range_t=(0, 150),
                              noise_pct=0.8,
     cycle_limit=num_epoch_repeat+1)
```

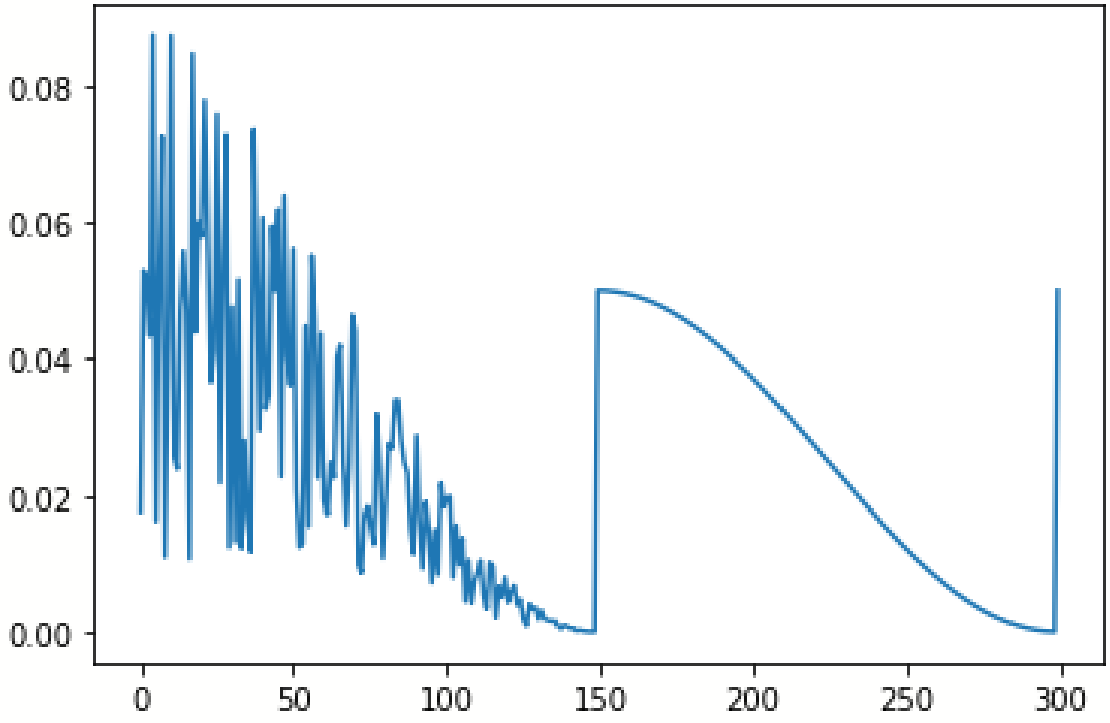


Рис. 9. График планирования скорости `CosineLRScheduler` с большим значением шума

4. ВВЕДЕНИЕ В АРХИТЕКТУРУ TRANSFORMER

В первой части этого раздела мы реализуем архитектуру Transformer. Поскольку эта архитектура так популярна, уже существует модуль PyTorch `nn.Transformer`. Однако, чтобы разобраться в деталях, реализуем архитектуру самостоятельно.

4.1. МЕХАНИЗМ ATTENTION

Механизм Attention («внимание») [9] описывает новую группу слоев в нейронных сетях, которая за последние несколько лет привела к достижению значимых результатов в разных задачах, особенно в задачах обработки последовательностей. В литературе существует множество различных определений понятия «внимание», но мы будем использовать следующее: механизм внимания позволяет получить средневзвешенное значение элементов последовательности с весами, динамически вычисляемыми на основе входного запроса (Query) и ключей (Key) элементов. Цель состоит в том, чтобы получить среднее значение по признакам нескольких элементов последовательности. Вместо взвешивания каждого элемента последовательности с одним и тем же значением веса, происходит взвешивание в зависимости от их значений. Другими словами, нейронная сеть динамически определяет, на какие входы сети следует «обратить внимание» больше, чем на другие элементы. Технически в механизме внимания обычно используют четыре элемента:

– Query (запрос). Это вектор признаков, описывающий, что необходимо найти в последовательности, т.е. на что, возможно, необходимо обратить внимание;

– Key (ключ). Для каждого входного элемента определяется ключ, который является вектором признаков. Этот вектор признаков описывает, что представляет из себя данный входной элемент, или когда он может быть важен. Ключи должны быть разработаны таким образом, чтобы можно было определить элементы, на которые необходимо обратить внимание, на основе запроса;

– Value (значение). Для каждого входного элемента определяется вектор значений Value. По этому вектору значений и выполняется усреднение;

– Score function (функция оценки). Для того, чтобы оценить, на какие элементы последовательности необходимо обратить внимание, необходимо задать функцию оценки f_{attn}. Функция оценки принимает на вход вектор Query и вектор Key элементов и выдает оценку (вес) внимания пары Query-Key. Обычно она реализуется с помощью простых метрик сходства, таких как скалярное произведение или небольшой многослойный перцептон.

Средние веса вычисляются путем применения функции Softmax ко всем выходам функции оценки внимания. В итоге больший вес присваивается тем векторам значений, чей соответствующий ключ Key наиболее похож (выше значение оценки внимания) на запрос Query.

Если формализовать сказанное вше, получим следующее выражение:

$$\alpha_i = \frac{\exp\left(f_{attn}\left(\text{key}_i, \text{query}\right)\right)}{\sum_j \exp\left(f_{attn}\left(\text{key}_j, \text{query}\right)\right)}, \quad \text{out} = \sum_i \alpha_i \cdot \text{value}_i.$$

Визуально можно показать применение механизма внимания к последовательности слов так, как показано на рис. 10.

Для каждого слова определяется один ключ Key и один вектор значений Value. Запрос Query сравнивается со всеми ключами с помощью функции оценки (в данном случае скалярного произведения) для определения весов. Функция Softmax на рисунке выше не показана. Наконец, векторы значений Value всех слов усредняются с использованием весов внимания.

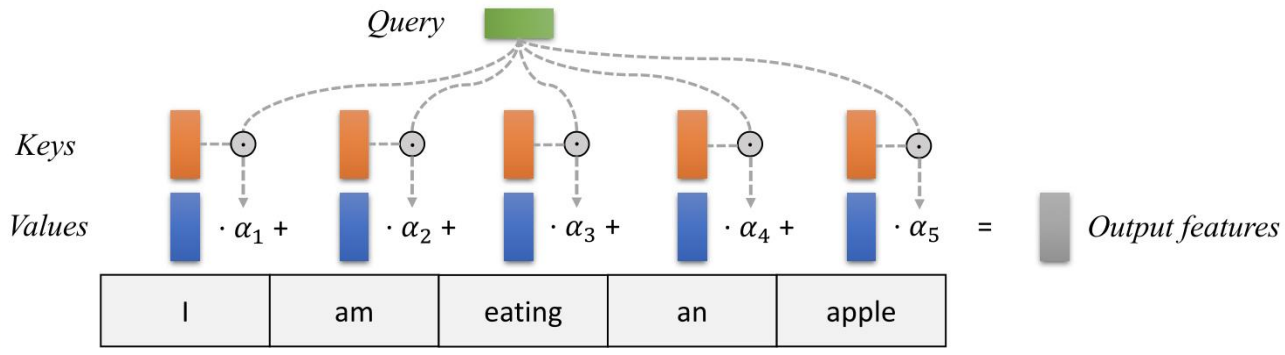


Рис. 10. Визуализация применения механизма внимания к последовательности

Большинство механизмов внимания различаются тем, какие используются запросы Query, как определяются векторы Key и Value и какая используется функция оценки. Внимание, применяемое в архитектуре Transformer, называется Self-Attention («самовнимание»). При самовнимании каждый элемент последовательности содержит ключ, значение и запрос. Для каждого элемента применяется слой внимания: на основе запроса проверяется сходство ключей всех элементов последовательности и каждого элемента последовательности, возвращается усредненный вектор значений. Немного углубимся в детали. Сначала рассмотрим конкретную реализацию механизма внимания, который в случае Transformer является вниманием с масштабированным скалярным произведением.

Теперь обсудим широкоиспользуемый механизм Self-Attention (самовнимание), известный так же как «внимание на основе масштабированного скалярного произведения».

Основной концепцией самовнимания является масштабированное скалярное произведение. Цель его состоит в том, чтобы получить механизм внимания, с помощью которого любой элемент в последовательности может «обращать внимание» на любой другой и при этом механизм внимания должен эффективно вычисляться. Внимание на основе скалярного произведения принимает на вход набор Query

$$Q \in R^{T \times d_k},\ \text{Key}\, K \in R^{T \times d_k}\ \text{и Value}\ V \in R^{T \times d_v},$$

где T – длина последовательности, а d_k и d_v – скрытая размерность запросов/ключей и значений соответственно. Для простоты пренебрегаем размером пакета. Значение внимания от элемента последовательности i к j основывается на сходстве запроса Q_i и ключа K_j и определяется скалярным произведение в качестве метрики сходства. Скалярное произведение в механизме внимания вычисляется следующим образом:

$$\text{Attention}\ (Q, K, V) = \text{Soft}\max\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

Матричное умножение QK^T – скалярное произведение для каждой возможной пары Query и Key. В результате чего получается матрица вида $T \times T$.

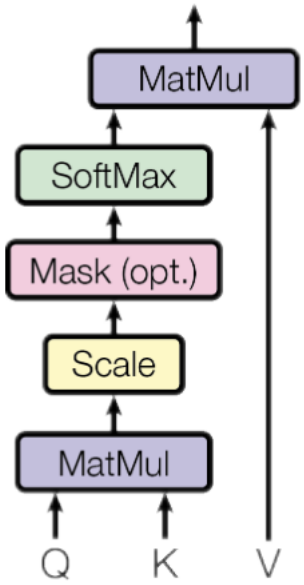


Рис. 11. Граф вычислений механизма внимания

Каждая строка в матрице представляет собой логиты внимания для конкретного элемента i ко всем остальным элементам в последовательности. К этим значениям применяется функция Softmax и результат умножается на вектор значений для получения средневзвешенного значения (веса определяются вниманием). Альтернативное представление механизма внимания – граф вычислений, который показан на рис. 11.

Один аспект, который мы еще не обсуждали, – это коэффициент масштабирования $1/\sqrt{d_k}$. Этот коэффициент имеет решающее значение для ограничения дисперсии значений оценки внимания после инициализации. Слои нейронной сети необходимо инициализировать таким образом, чтобы получить равную дисперсию во всех слоях модели, и, следовательно, значения векторов Q и K могут иметь дисперсию, близкую к 1. Однако выполнение скалярного произведения над двумя векторами с дисперсией σ^2 приводит к скаляру с дисперсией в d_k раз большей:

$$q_i \sim \mathcal{N}(0, \sigma^2), \quad k_i \sim \mathcal{N}(0, \sigma^2) \rightarrow \mathrm{Var}\left(\sum_{i=1}^{d_k} q_i k_i\right) = \sigma^4 d_k.$$

Если не уменьшить дисперсию до значения $\sim\sigma^2$, то выполненная над логитами функция Softmax выведет значение 1 для одного элемента и 0 для всех остальных. В этом случае градиенты, проходящие через функцию Softmax, будут близки к нулю, поэтому сеть не сможет обучиться должным образом.

Заметим, что дополнительный фактор σ^2, т.е. наличие σ^4 вместо σ^2, обычно не является проблемой, поскольку мы все равно сохраняем исходную дисперсию σ^2 близкой к 1.

Блок Mask (опциональный) на изображении выше представляет собой опциональную маскировку определенных записей в матрице внимания. Этот механизм используется в случае сложения нескольких последовательностей разной длины в один пакет. Для распараллеливания операций в PyTorch выполняется разбиение предложения на подпредложения одинаковой длины и маскирование лексем во время вычисления значений оценки внимания. Обычно этот механизм реализуется путем установки соответствующих логитов внимания в очень небольшие значения.

После обсуждения деталей блока внимания с масштабированным скалярным произведением можно написать функцию, которая вычисляет выходные признаки, учитывая тройку Query, Key и Value:

```
def scaled_dot_product(q, k, v, mask=None):
    d_k = q.size()[-1]
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
    attn_logits = attn_logits / math.sqrt(d_k)
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, -9e15)
    attention = F.softmax(attn_logits, dim=-1)
    values = torch.matmul(attention, v)
    return values, attention
```

Обратите внимание, что приведенный выше код поддерживает дополнительную первую размерность перед длиной последовательности, так что его можно использовать для пакетов последовательностей. Для лучшего понимания процесса сгенерируем несколько случайных Query, Key и Value и рассчитаем значения внимания:

```
seq_len, d_k = 3, 2
pl.seed_everything(42)
q = torch.randn(seq_len, d_k)
k = torch.randn(seq_len, d_k)
v = torch.randn(seq_len, d_k)
```

```
values, attention = scaled_dot_product(q, k, v)
print("Q\n", q)
print("K\n", k)
print("V\n", v)
print("Values\n", values)
print("Attention\n", attention)
```

```
Q
 tensor([[ 0.3367,  0.1288],
        [ 0.2345,  0.2303],
        [-1.1229, -0.1863]])
K
 tensor([[ 2.2082, -0.6380],
        [ 0.4617,  0.2674],
        [ 0.5349,  0.8094]])
V
 tensor([[ 1.1103, -1.6898],
        [-0.9890,  0.9580],
        [ 1.3221,  0.8172]])
Values
 tensor([[ 0.5698, -0.1520],
        [ 0.5379, -0.0265],
        [ 0.2246,  0.5556]])
Attention
 tensor([[0.4028, 0.2886, 0.3086],
        [0.3538, 0.3069, 0.3393],
             [0.1303, 0.4630, 0.4067]])
```

4.2. МЕХАНИЗМ MULTIHEAD ATTENTION

Взвешенное скалярное произведение механизма внимания позволяет сети следить за последовательностью. Однако для отдельно взятого элемента последовательности часто бывает необходимо обратить внимание на несколько различных элементов последовательности, и одно результирующее взвешенное среднее не подходит для этого. Поэтому механизмы внимания можно расширить на несколько выходов – «голов», т.е. на несколько различных триплетов Query-Key-Value по одним и тем же признакам.

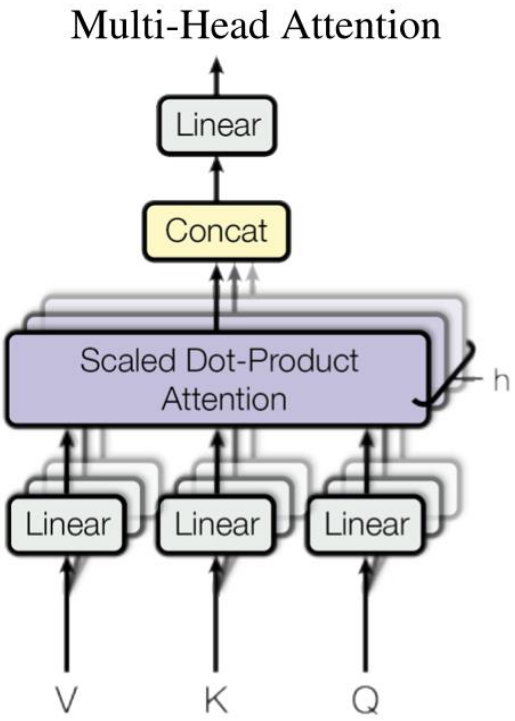


Рис. 12. Граф вычислений Multihead Attention

Если говорить точнее, то, получив матрицу Query, Key и Value, мы преобразуем их в h подзапросов, подключей и подзначений, которые пропускаются через масштабированное скалярное произведение внимания независимо друг от друга. После этого мы конкатенируем выходы и объединяем их с итоговой матрицей весов. Математически можно выразить эту операцию следующим образом:

$$Multihead(Q, K, V) = Concat(head_1, ..., head_h)W^O;$$
$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V).$$

Такую реализацию называют слоем Multi-Head Attention с обучаемыми параметрами: $W_{i...h}^Q \in R^{D\times d_k}$, $W_{i...h}^K \in R^{D\times d_k}$, $W_{i...h}^V \in R^{D\times d_v}$ и $W^O \in R^{hd_v\times d_{out}}$ (D – размерность входа). Соответствующий вычислительный граф мы можем представить так, как показано на рис. 12.

В случае применения слоя Multihead Attention в нейронной сети возникает логичный вопрос, как задать векторы Query, Key и Value? Простая, но эффективная реализация заключается в том, чтобы задать текущую карту признаков $X \in R^{B\times T\times d_{\text{model}}}$ как Q, K и V (B – размер пакета, T – длина последовательности, d_{model} – скрытая размерность X). Последовательные весовые матрицы W^Q, W^K и W^V могут преобразовать X в соответствующие векторы признаков,

которые представляют Query, Key и Value для входных данных. Используя этот подход, мы можем реализовать модуль Multi-Head Attention:

```
class MultiheadAttention(nn.Module):

    def __init__(self, input_dim, embed_dim, num_heads):
        super().__init__()
        assert embed_dim % num_heads == 0, "Embedding dimension
must be 0 modulo number of heads."

        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        # Сложение весовых матриц 1...h для эффективности.
        # Во многих реализациях используется bias=False, что
        # совсем необязательно.
        self.qkv_proj = nn.Linear(input_dim, 3*embed_dim)
        self.o_proj = nn.Linear(embed_dim, embed_dim)

        self._reset_parameters()

    def _reset_parameters(self):
        # Оригинальная инициализация.
        nn.init.xavier_uniform_(self.qkv_proj.weight)
        self.qkv_proj.bias.data.fill_(0)
        nn.init.xavier_uniform_(self.o_proj.weight)
        self.o_proj.bias.data.fill_(0)

    def forward(self, x, mask=None, return_attention=False):
        batch_size, seq_length, _ = x.size()
        qkv = self.qkv_proj(x)

        # Отделение Q, K, V от линейного выхода.
        qkv = qkv.reshape(batch_size, seq_length, self.num_heads,
3*self.head_dim)
        qkv = qkv.permute(0, 2, 1, 3) # [Batch, Head, SeqLen,
Dims]
        q, k, v = qkv.chunk(3, dim=-1)
```

```
        # Определение выходных значений.
        values, attention = scaled_dot_product(q, k, v, mask=mask)
        values = values.permute(0, 2, 1, 3) # [Batch, SeqLen,
Head, Dims]
        values = values.reshape(batch_size, seq_length,
self.embed_dim)
        o = self.o_proj(values)

        if return_attention:
            return o, attention
        else:
            return o
```

Одной из важнейших характеристик Multihead Attention является то, что оно эквивариантно к перестановкам по отношению к своим входам. Это означает, что если мы поменяем местами два входных элемента в последовательности, например, $X1 \leftrightarrow X2$ (пока пренебрегая размерностью пакета), то на выходе будет точно такой же результат. Следовательно, Multihead Attention на самом деле рассматривает вход не как последовательность, а как набор элементов. Это свойство делает блок Multihead Attention и архитектуру Transformer применимым и для решения других задач, например, компьютерного зрения.

4.3. АРХИТЕКТУРА КОДЕРА

Далее рассмотрим, как применить блок Multihead Attention в архитектуре Transformer. Изначально модель Transformer была разработана для задачи машинного перевода. Поэтому она имеет структуру кодер-декодер, в которой кодер принимает на вход предложение на языке оригинала и генерирует представление, основанное на механизме внимания. С другой стороны, декодер обращает внимание на закодированную (кодером) последовательность и генерирует переведенное предложение авторегрессионным способом, как в стандартной рекуррентной нейронной сети. Хотя такая структура чрезвычайно полезна для задач «последовательность-последовательность» (Seq2Seq), когда необходимо авторегрессивное декодирование, сосредоточимся на кодирующей части. Многие достижения в области обработки естественного были совершены с использованием только кодирующих моделей трансформеров (к ним относятся семейство моделей BERT, моделей Vision Transformer и др.). Полная архитектура трансформера выглядит следующим образом, показанным на рис. 13.

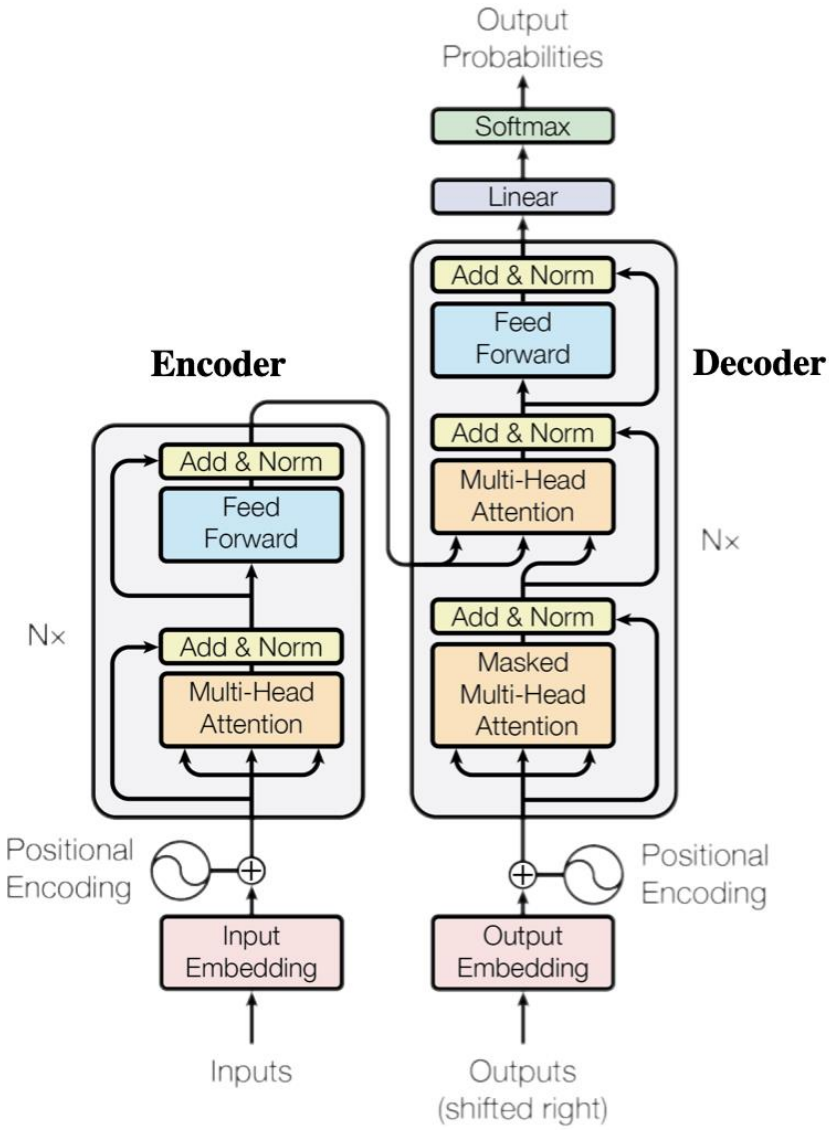


Рис. 13. Архитектура Transfromer из оригинальной статьи

Кодер состоит из N одинаковых блоков, которые применяются последовательно. Входной сигнал x сначала проходит через блок Multi-Head Attention (реализован выше). Выходной сигнал добавляется к исходному входному с помощью остаточного соединения, и после этого применяется операция последовательной нормализации суммы. В итоге получается операция LayerNorm(x + Multihead(x, x, x)) (x – это Q, K и V, входящие в слой внимания). Остаточная связь имеет решающее значение в архитектуре трансформера по двум причинам:

1. Подобно сетям ResNet, трансформеры разработаны для очень глубоких сетей. Некоторые модели содержат более 24 блоков в кодере. Следовательно, остаточные связи имеют решающее значение для обеспечения плавного потока градиента через модель.

2. Без остаточной связи информация об исходной последовательности теряется. Слой многоголового внимания игнорирует положение элементов в последовательности и может определить его только на основе входных признаков. Удаление остаточных связей означает, что эта информация теряется после первого слоя внимания (после инициализации), а при случайной инициализации векторов Query и Key, выходные векторы для позиции i не имеют никакого отношения к исходным входным данным. Все выходы внимания, скорее всего, будут представлять схожую/одинаковую информацию, и у модели нет шансов отличить, какая информация получена от какого входного элемента. Альтернативным вариантом остаточной связи может быть фиксация хотя бы одной головы для фокусировки на ее исходном входе, но это очень неэффективно и не дает преимущества улучшенного градиентного потока.

Нормализация слоя (LayerNorm) также играет важную роль в архитектуре трансформер, поскольку она позволяет ускорить обучение и обеспечивает небольшую регуляризацию. Кроме того, она гарантирует, что признаки имеют одинаковую магнитуду для всех элементов в последовательности. Пакетная нормализация не используется, поскольку она зависит от размера пакета, который часто имеет небольшое значение в трансформерах (так как сами модели требуют большого объема памяти GPU), и еще потому, что она особенно плохо работает в задачах обработки естественного языке, поскольку признаки слов имеют гораздо более высокую дисперсию (много очень редких слов, которые необходимо учитывать для хорошей оценки распределения).

Дополнительно к модели Multi-Head Attention добавляется небольшая полносвязная сеть прямого распространения, которая применяется к каждой позиции независимо. В частности, в модели используется линейный многослойный перцептрон ($Linear \rightarrow ReLU \rightarrow Linear$). Полное преобразование, включая остаточную связь, может быть выражено так:

$$FFN(x) = \max(0,\ xW_1 + b_1)W_2 + b_2;$$
$$x = LayerNorn(x + FFN(x)).$$

Этот многослойный перцептрон добавляет дополнительную сложность в модель и позволяет преобразовывать каждый элемент последовательности отдельно. Можно добавить, что это позволяет модели «постобработать» новую информацию, добавленную предыдущей частью сети, и подготовить ее для

следующего блока внимания. Обычно внутренняя размерность многослойного перцептрона в 2 – 8 раз больше, чем d_{model}, т.е. размера исходного входного сигнала x. Общее преимущество полносвязного слоя заключается в более быстром параллельном выполнении.

Наконец, рассмотрев все части архитектуры кодера, можно приступить к его реализации. Начнем с реализации одного блока кодера. В дополнение к описанным выше слоям добавим слои Dropout внутри многослойного перцептона и на выходе многослойного перцептона и Multi-Head Attention для регуляризации:

```
class EncoderBlock(nn.Module):

    def __init__(self, input_dim, num_heads, dim_feedforward,
dropout=0.0):
        """
        Входы:
            input_dim - Размер входных данных.
            num_heads - Количество голов для использования в блоке
внимания.
            dim_feedforward - Размер скрытого слоя в MLP.
            dropout - Вероятность Dropout для использования
в слоях Dropout.
        """
        super().__init__()

        # Слой внимания
        self.self_attn = MultiheadAttention(input_dim, input_dim,
num_heads)

        # Двуслойный MLP
        self.linear_net = nn.Sequential(
            nn.Linear(input_dim, dim_feedforward),
            nn.Dropout(dropout),
            nn.ReLU(inplace=True),
            nn.Linear(dim_feedforward, input_dim)
        )

        # Слои, добавляемые между основными слоями
```

```
        self.norm1 = nn.LayerNorm(input_dim)
        self.norm2 = nn.LayerNorm(input_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Часть внимания
        attn_out = self.self_attn(x, mask=mask)
        x = x + self.dropout(attn_out)
        x = self.norm1(x)

        # MLP-часть
        linear_out = self.linear_net(x)
        x = x + self.dropout(linear_out)
        x = self.norm2(x)

        return x
```

На основе этого блока можно реализовать модуль всего кодера. В дополнение к функции `forward()`, которая выполняет итерирование по последовательности блоков кодера, также реализуем функцию `get_attention_maps()`. Идея этой функции заключается в том, чтобы вернуть вероятности (оценки) внимания для всех блоков Multi-Head Attention в кодере.

Текущая реализация:

```
class TransformerEncoder(nn.Module):

    def __init__(self, num_layers, **block_args):
        super().__init__()
        self.layers = nn.ModuleList([EncoderBlock(**block_args)
for _ in range(num_layers)])

    def forward(self, x, mask=None):
        for l in self.layers:
            x = l(x, mask=mask)
        return x

    def get_attention_maps(self, x, mask=None):
        attention_maps = []
        for l in self.layers:
```

```
            _, attn_map = l.self_attn(x, mask=mask,
return_attention=True)
            attention_maps.append(attn_map)
            x = l(x)
        return attention_maps
```

4.4. ПОЗИЦИОННОЕ КОДИРОВАНИЕ

Мы уже обсуждали, что блок многоголового внимания является эквивариантным к перестановкам и не может различать, находится ли одно входное значение в последовательности перед другим или нет. Однако в таких задачах, как понимание языка, позиция важна для интерпретации входных слов. Поэтому информация о позиции может быть добавлена за счет входных признаков. Можно был бы выучить эмбеддинги для каждой возможной позиции, но такое решение не подходит для входной последовательности переменной длины. Следовательно, лучшим вариантом является использование шаблонов признаков, которые сеть может определить по признакам и потенциально обобщить на большие последовательности. Конкретный паттерн, выбранный в оригинальной статье, представляет собой функции синуса и косинуса значений различных частот, как показано ниже:

$$PE_{(pos,i)} = \begin{cases} \sin\left(\dfrac{pos}{10\,000^{i/d_{model}}}\right), & \text{если } i \bmod 2 = 0; \\ \cos\left(\dfrac{pos}{10\,000^{(i-1)/d_{model}}}\right), & \text{в противном случае.} \end{cases}$$

$PE_{(pos,i)}$ представляет результат позиционного кодирования элемента в позиции *pos* в последовательности и скрытой размерности *i*. Эти значения, конкатенированные для всех скрытых размерностей, добавляются к исходным входным признакам и определяют информацию о позиции. Мы различаем четные ($i$mod2 = 0) и нечетные ($i$mod2 = 1) скрытые размерности, в которых применяются операции синус/косинус соответственно. Интуиция, стоящая за этим кодированием, заключается в том, что можно представить $PE_{(pos+k,:)}$ как линейную функцию $PE_{(pos,:)}$, что может позволить модели легко учитывать

относительные положения. Длины волн в различных измерениях варьируются от 2π до $10\,000 \cdot 2\pi$.

Позиционное кодирование реализовано ниже. Код взят с сайта PyTorch.

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model, max_len=5000):
        """
        Входы
            d_model – Скрытая размерность входных данных.
            max_len – Максимальная длина ожидаемой
последовательности.
        """
        super().__init__()

        # Создать матрицу [SeqLen, HiddenDim], представляющую
        # позиционное кодирование для входов max_len.
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_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(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)

        # register_buffer => Tensor, который не является
        # параметром, но должен быть частью состояния модуля.
        # Используется для тензоров, которые должны находиться
        # на том же устройстве, что и модуль.
        # persistent=False указывает PyTorch не добавлять буфер
        # в словаь состояний (например, когда мы сохраняем
        # модель).
        self.register_buffer('pe', pe, persistent=False)

    def forward(self, x):
        x = x + self.pe[:, :x.size(1)]
        return x
```

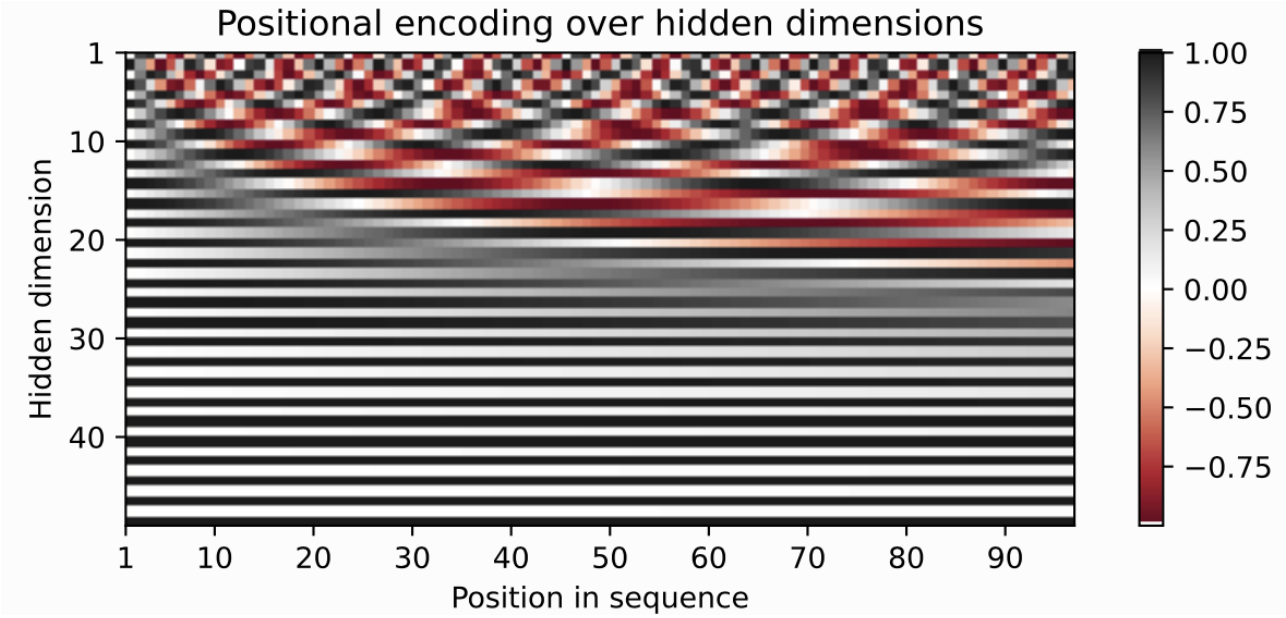


Рис. 14. Визуализация позиционного кодирования

Чтобы лучше понять позиционное кодирование, можно его визуализировать. Создадим изображение схемы позиционного кодирования по скрытой размерности и позиции в последовательности. Каждый пиксель на рис. 14 представляет собой изменение входного признака, которое выполняется для кодирования конкретной позиции.

4.5. ПЛАНИРОВАНИЕ СКОРОСТИ ОБУЧЕНИЯ С РАЗОГРЕВОМ

Одним из часто используемых методов обучения трансформера является «разогрев» (warmup) скорости обучения. Это означает, что в течение первых нескольких итераций мы постепенно увеличиваем скорость обучения от 0 до заданной скорости обучения. Таким образом, мы медленно начинаем обучение, т.е. не делаем очень большие шаги с самого начала. Обучение глубокого трансформера без разогрева скорости обучения может привести к расхождению модели и гораздо худшей производительности при обучении и тестировании.

Почему разогрев так важен? В настоящее время существует два распространенных объяснения. Во-первых, оптимизатор Adam использует коэффициенты коррекции смещения, что, однако, может привести к увеличению дисперсии адаптивной скорости обучения во время первых итераций. Было показано,

что улучшенные оптимизаторы, такие как RAdam, преодолевают эту проблему, не требуя разогрева для обучения трансформеров. Во-вторых, итеративно применяемая послойная нормализация по слоям может привести к очень большим градиентам во время первых итераций, что может быть решено использованием предварительной послойной нормализации (аналогично предварительной активации ResNet) или заменой послойной нормализации другими техниками (Adaptive Normalization, Power Normalization).

Тем не менее, во многих реализациях и статях по-прежнему используют оригинальную архитектуру трансформер с оптимизатором Adam, так как разогрев – это простой, но эффективный способ решения проблемы с градиентами на первых итерациях. Существует множество различных планировщиков, которые можно использовать. Например, в оригинальной работе использовался планировщик экспоненциального затухания с разогревом. Однако в настоящее время наиболее популярным планировщиком является косинусоидальный планировщик с разогревом, который сочетает разогрев с косинусоидальным затуханием скорости обучения. Реализуем его ниже и визуализируем коэффициент скорости обучения по эпохам.

```
class CosineWarmupScheduler(optim.lr_scheduler._LRScheduler):

    def __init__(self, optimizer, warmup, max_iters):
        self.warmup = warmup
        self.max_num_iters = max_iters
        super().__init__(optimizer)

    def get_lr(self):
        lr_factor = self.get_lr_factor(epoch=self.last_epoch)
        return [base_lr * lr_factor for base_lr in self.base_lrs]

    def get_lr_factor(self, epoch):
        lr_factor = 0.5 * (1 + np.cos(np.pi * epoch /
self.max_num_iters))
        if epoch <= self.warmup:
            lr_factor *= epoch * 1.0 / self.warmup
        return lr_factor
```

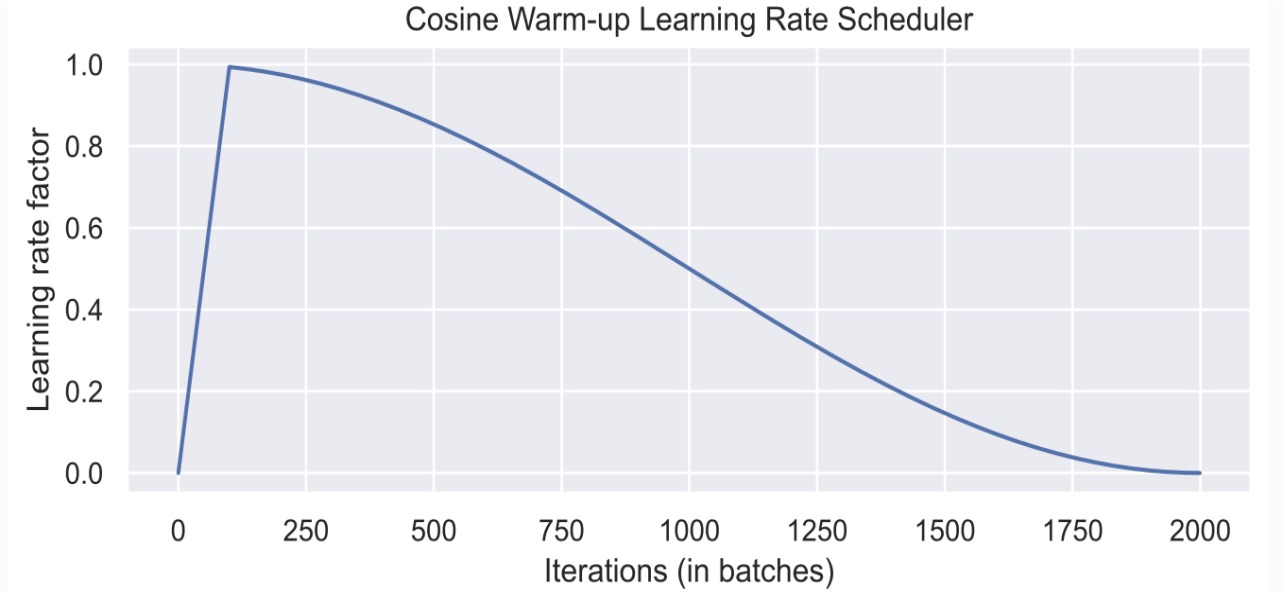


Рис. 15. Визуализация косинусоидального планировщика с разогревом

В течение первых 100 итераций мы увеличиваем значение скорости обучения от 0 до 1, в то время как во всех последующих итерациях скорость затухает по косинусоиде (рис. 15). Реализацию такого планировщика можно найти в популярной библиотеке Transformer от HuggingFace.

4.6. РЕАЛИЗАЦИЯ АРХИТЕКТУРЫ TRANSFORMER

Наконец, можно встроить архитектуру трансформер в модуль PyTorch Lightning. Использование PyTorch Lightning позволяет упростить код процедур обучения и тестирования, а также определить код в отдельные функции. Реализуем шаблон для классификатора, основанного на кодере трансформера. Таким образом, на выходе будут предсказания для каждого элемента последовательности. Если нужен классификатор для всей последовательности, то типовой подход заключается в добавлении к последовательности дополнительного токена [CLS], представляющего токен классификатора.

Дополнительно к архитектуре трансформер добавим небольшую входную сеть (сопоставляет размеры входа с размерами модели), позиционное кодирование и выходную сеть (преобразует выходные кодированные представления в предсказания). Также можно добавить планировщик скорости обучения, который выполняет изменение скорости на каждой итерации, а не один раз

за эпоху. Это необходимо для разогрева и реализации плавного косинусоидального затухания. Шаги обучения, валидации и тестирования пока остаются пустым и будут реализованы для конкретных моделей.

```
class TransformerPredictor(pl.LightningModule):

    def __init__(self, input_dim, model_dim, num_classes,
num_heads, num_layers, lr, warmup, max_iters, dropout=0.0,
input_dropout=0.0):
        """
        Входы:
            input_dim – Скрытая размерность входных данных.
            model_dim – Скрытая размерность для использования
внутри трансформера.
            num_classes – Число классов.
            num_heads – Количество голов для использования
в блоках многоголового внимания.
            num_layers – Количество используемых блоков
кодировщика.
            lr – Скорость обучения в оптимизаторе.
            warmup – Количество шагов разогрева.
Обычно от 50 до 500.
            max_iters – Количество максимальных итераций,
за которые обучается модель. Необходимо для планировщика
CosineWarmup.
            dropout – Dropout для применения внутри модели.
            input_dropout – Dropout для применения к входным
признакам.
        """
        super().__init__()
        self.save_hyperparameters()
        self._create_model()

    def _create_model(self):
        self.input_net = nn.Sequential(
            nn.Dropout(self.hparams.input_dropout),
            nn.Linear(self.hparams.input_dim,
self.hparams.model_dim)
```

```
        )
        # Позиционное кодирование.
        self.positional_encoding =
PositionalEncoding(d_model=self.hparams.model_dim)
        # Трансформер
        self.transformer = TransformerEncod-
er(num_layers=self.hparams.num_layers,

input_dim=self.hparams.model_dim,

dim_feedforward=2*self.hparams.model_dim,
                                    num_heads=self.hparams.num_heads,
                                     dropout=self.hparams.dropout)
        # Выходной классификатор.
        self.output_net = nn.Sequential(
            nn.Linear(self.hparams.model_dim,
self.hparams.model_dim),
            nn.LayerNorm(self.hparams.model_dim),
            nn.ReLU(inplace=True),
            nn.Dropout(self.hparams.dropout),
            nn.Linear(self.hparams.model_dim,
self.hparams.num_classes)
        )

    def forward(self, x, mask=None, add_positional_encoding=True):
        """
        Входы:
            x - Входные признаки формы [Batch, SeqLen, input_dim].
            mask - Маска для применения к выводам внимания
(необязательно).
            add_positional_encoding - Если True, добавляем
позиционное кодирование к входным данным. Может быть нежелательно
для некоторых задач.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        x = self.transformer(x, mask=mask)
```

```
        x = self.output_net(x)
        return x

    @torch.no_grad()
    def get_attention_maps(self, x, mask=None,
add_positional_encoding=True):
        """
        Функция для извлечения матриц внимания всего трансформера
для одного пакета.
        Входные аргументы те же, что и при прямом проходе.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        attention_maps = self.transformer.
get_attention_maps(x, mask=mask)
        return attention_maps

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(),
lr=self.hparams.lr)

        # Применение планировщика lr на каждом шаге.
        lr_scheduler = CosineWarmupScheduler(optimizer, \
                       warmup=self.hparams.warmup, \
                       max_iters=self.hparams.max_iters)
        return [optimizer], [{'scheduler': lr_scheduler,
'interval': 'step'}]

    def training_step(self, batch, batch_idx):
        raise NotImplementedError

    def validation_step(self, batch, batch_idx):
        raise NotImplementedError

    def test_step(self, batch, batch_idx):
        raise NotImplementedError
```

4.7. ПРОВЕДЕНИЕ ЭКСПЕРИМЕНТОВ

После завершения реализации архитектуры можно начать экспериментировать и применять ее для решения различных задач. В этом разделе сосредоточимся на задаче преобразования последовательности в последовательность.

4.7.1. Преобразование последовательности в последовательность

Задача Sequence-to-Sequence представляет собой задачу, в которой вход и выход – это последовательности, не обязательно одинаковой длины. Популярные задачи в этой области включают машинный перевод и резюмирование текстов. Обычно используется кодер для интерпретации входной последовательности и декодер для генерации выходного сигнала авторегрессионным способом. Однако вернемся к гораздо более простому примеру и будем использовать только кодировщик. Дана последовательность чисел N от 0 до M. Задача состоит в том, чтобы обратить входную последовательность. В нотации Numpy это означает, что если на входе x, то на выходе должно получиться x [::–1]. Хотя задача выглядит очень простой, с помощью рекуррентных нейронных сетей решить ее будет трудно, поскольку задача требует хранения долгосрочных связей. Трансформеры созданы для решения именно такого рода задач.

Сначала создадим класс набора данных:

```
class ReverseDataset(data.Dataset):

    def __init__(self, num_categories, seq_len, size):
        super().__init__()
        self.num_categories = num_categories
        self.seq_len = seq_len
        self.size = size

        self.data = torch.randint(self.num_categories,
size=(self.size, self.seq_len))

    def __len__(self):
        return self.size
```

```
    def __getitem__(self, idx):
        inp_data = self.data[idx]
        labels = torch.flip(inp_data, dims=(0,))
        return inp_data, labels
```

Мы создаем произвольное количество случайных последовательностей чисел от 0 до `num_categories-1`. Метка – это тензор, перевернутая последовательность. Создадим соответствующие загрузчики данных:

```
dataset = partial(ReverseDataset, 10, 16)
train_loader = data.DataLoader(dataset(50000), batch_size=128,
shuffle=True, drop_last=True, pin_memory=True)
val_loader   = data.DataLoader(dataset(1000), batch_size=128)
test_loader  = data.DataLoader(dataset(10000), batch_size=128)
```

Рассмотрим произвольную выборку из набора данных:

```
inp_data, labels = train_loader.dataset[0]
print("Input data:", inp_data)
print("Labels:    ", labels)
```

```
Input data: tensor([9, 6, 2, 0, 6, 2, 7, 9, 7, 3, 3, 4, 3, 7, 0,
9])
Labels:     tensor([9, 0, 7, 3, 4, 3, 3, 7, 9, 7, 2, 6, 0, 2, 6,
9])
```

Во время обучения пропускаем входную последовательность через кодер и предсказываем выход для каждого входного токена. Для этого используем стандартную кросс-энтропию. Каждое число представляется в виде унитарного вектора. Очевидно, что представление категорий в виде единичных скаляров уменьшает выразительность модели, поскольку 0 и 1 не являются более близкими значением, чем 0 и 9. Альтернативой унитарному вектору является использование выученного вектора эмбеддинга, как это предусмотрено модулем PyTorch `nn.Embedding`. Однако использование унитарного вектора с дополнительным линейным слоем, как в нашем случае, имеет тот же эффект, что и слой эмбеддинга (`self.input_net` отображает унитарный вектор в плот-

ный вектор, при котором каждая строка весовой матрицы представляет эмбеддинг для определенной категории).

Для реализации обучения создадим новый класс, наследующийся от `TransformerPredictor` и перепишем этапы обучения, проверки и тестирования:

```
class ReversePredictor(TransformerPredictor):

    def _calculate_loss(self, batch, mode="train"):
        # Получение данных и преобразование категорий в
        # унитарные векторы/
        inp_data, labels = batch
        inp_data = F.one_hot(inp_data,
num_classes=self.hparams.num_classes).float()

        # Выполнение прогнозирования и рассчет потери
        # и Accuracy.
        preds = self.forward(inp_data,
add_positional_encoding=True)
        loss = F.cross_entropy(preds.view(-1,preds.size(-1)),
labels.view(-1))
        acc = (preds.argmax(dim=-1) == labels).float().mean()

        # Логирование.
        self.log(f"{mode}_loss", loss)
        self.log(f"{mode}_acc", acc)
        return loss, acc

    def training_step(self, batch, batch_idx):
        loss, _ = self._calculate_loss(batch, mode="train")
        return loss

    def validation_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="val")

    def test_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="test")
```

Наконец, создадим функцию обучения, подобную той, что использовали ранее. Создаем объект `pl.Trainer`, выполняем обучение в течение N эпох, логируем процесс в TensorBoard и сохраняем лучшую модель после оценки на валидационном наборе. После этого тестируем модели на тестовом наборе. Дополнительным параметром, который передается в `Trainer`, является `gradient_clip_val`. Он обрезает норму градиентов для всех параметров перед выполнением шага оптимизатора и предотвращает расхождение модели. Для трансформеров обрезка градиента может помочь дополнительно стабилизировать обучение во время первых нескольких итераций и во всем цикле обучения. Во фреймворке PyTorch можно применить обрезку градиентов с помощью функции `torch.nn.utils.clip_grad_norm_(...)` (смотрите документацию). Значение `clip` обычно составляет от 0,5 до 10, в зависимости от того, насколько сильно необходимо обрезать большие градиенты.

Реализуем функцию обучения:

```
def train_reverse(**kwargs):

    root_dir = os.path.join(CHECKPOINT_PATH, "ReverseTask")
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,

callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",
monitor="val_acc")],
                         accelerator="gpu" if
str(device).startswith("cuda") else "cpu",
                         devices=1,
                         max_epochs=10,
                         gradient_clip_val=5)
    trainer.logger._default_hp_metric = None #

    model = ReversePredic-
tor(max_iters=trainer.max_epochs*len(train_loader), **kwargs)
    trainer.fit(model, train_loader, val_loader)

    val_result = trainer.test(model, val_loader, verbose=False)
```

```
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {"test_acc": test_result[0]["test_acc"], "val_acc":
val_result[0]["test_acc"]}

    model = model.to(device)
    return model, result
```

Теперь обучить модель. Используем один блок кодирования и одну голову в многоголовом внимании. Выбор обусловлен простотой задачи.

```
reverse_model, reverse_result =
train_reverse(input_dim=train_loader.dataset.num_categories,
                                                model_dim=32,
                                                num_heads=1,

num_classes=train_loader.dataset.num_categories,
                                                num_layers=1,
                                                dropout=0.0,
                                                lr=5e-4,
                                                warmup=50)
```

5. ИСПОЛЬЗОВАНИЕ АРХИТЕКТУРЫ TRANSFORMER ДЛЯ РЕШЕНИЯ ЗАДАЧ КОМПЬЮТЕРНОГО ЗРЕНИЯ

Рассмотрим пример использования архитектуры Transformer для решения задачи компьютерного зрения на наборе данных CIFAR10. Используем те же варианты дополнения данных, что использовались ранее для сверточных нейронных сетей, чтобы обеспечить объективное сравнение. Константы в трансформации `transforms.Normalize` соответствуют значениям, которые позволяют выполнить масштабирование и сдвиг данных к среднему значения, равному нулю, и стандартному отклонению, равному единице:

```
test_transform = transforms.Compose([transforms.ToTensor(),
                                     trans-
forms.Normalize([0.49139968, 0.48215841, 0.44653091], [0.24703223,
0.24348513, 0.26158784])
                                     ])
train_transform = trans-
forms.Compose([transforms.RandomHorizontalFlip(),
                                      trans-
forms.RandomResizedCrop((32,32),scale=(0.8,1.0),ratio=(0.9,1.1)),
                                      transforms.ToTensor(),

transforms.Normalize([0.49139968, 0.48215841, 0.44653091],
[0.24703223, 0.24348513, 0.26158784])
                                     ])
# Загрузка обучающего набора данных.
# Разделение его на обучающую и проверочную части.
# Проверочный набор не должен использовать аугментацию.
train_dataset = CIFAR10(root=DATASET_PATH, train=True,
transform=train_transform, download=True)
val_dataset = CIFAR10(root=DATASET_PATH, train=True,
transform=test_transform, download=True)
pl.seed_everything(42)
```

```
train_set, _ = torch.utils.data.random_split(train_dataset,
[45000, 5000])
pl.seed_everything(42)
_, val_set = torch.utils.data.random_split(val_dataset,
[45000, 5000])

# Загрузка тестового набора.
test_set = CIFAR10(root=DATASET_PATH, train=False,
transform=test_transform, download=True)

# Определение набора загрузчиков данных.
train_loader = data.DataLoader(train_set, batch_size=128,
shuffle=True, drop_last=True, pin_memory=True, num_workers=4)
val_loader = data.DataLoader(val_set, batch_size=128,
shuffle=False, drop_last=False, num_workers=4)
test_loader = data.DataLoader(test_set, batch_size=128,
shuffle=False, drop_last=False, num_workers=4)

# Визуализация образцов.
NUM_IMAGES = 4
CIFAR_images = torch.stack([val_set[idx][0] for idx
in range(NUM_IMAGES)], dim=0)
img_grid = torchvision.utils.make_grid(CIFAR_images, nrow=4,
normalize=True, pad_value=0.9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("Image examples of the CIFAR10 dataset")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```

5.1. РЕАЛИЗАЦИЯ АРХИТЕКТУРЫ VISION TRANSFORMER

Трансформеры были первоначально предложены для обработки последовательностей, поскольку это инвариантная к перестановкам входных данных архитектура. Чтобы применить трансформеры к последовательностям, исполь-

зуют добавление позиционного кодирования к входным векторам признаков, и модель учиться учитывать позиции при обучении. Так почему бы не сделать то же самое с изображениями? Именно это и предложили Алексей Досовицкий и др. в своей работе [10].

В частности, Vision Transformer – это модель классификации изображений, которая рассматривает изображения как последовательности небольших блоков (патчей). В качестве предварительного этапа обработки изображение, например, 48×48 пикселей, разбивается на 9 патчей размером 16×16 пикселей. Каждый из этих патчей рассматривается как слово/лексема и проецируется в пространство признаков. Добавив позиционное кодирование и патч-лексему для классификации, можно применить трансформер, как к последовательности, и обучить новой задачи. Визуализация архитектуры показана на рис. 16.

Рассмотрим шаг за шагом использование Vision Transformer и реализуем все его элементы. Сначала выполним предварительную обработку изображения: изображение размером $N \times M$ должно быть разбито на $(N \times M)^2$ патчей размером $M \times M$. Набор патчей представляет собой входную последовательность для трансформера.

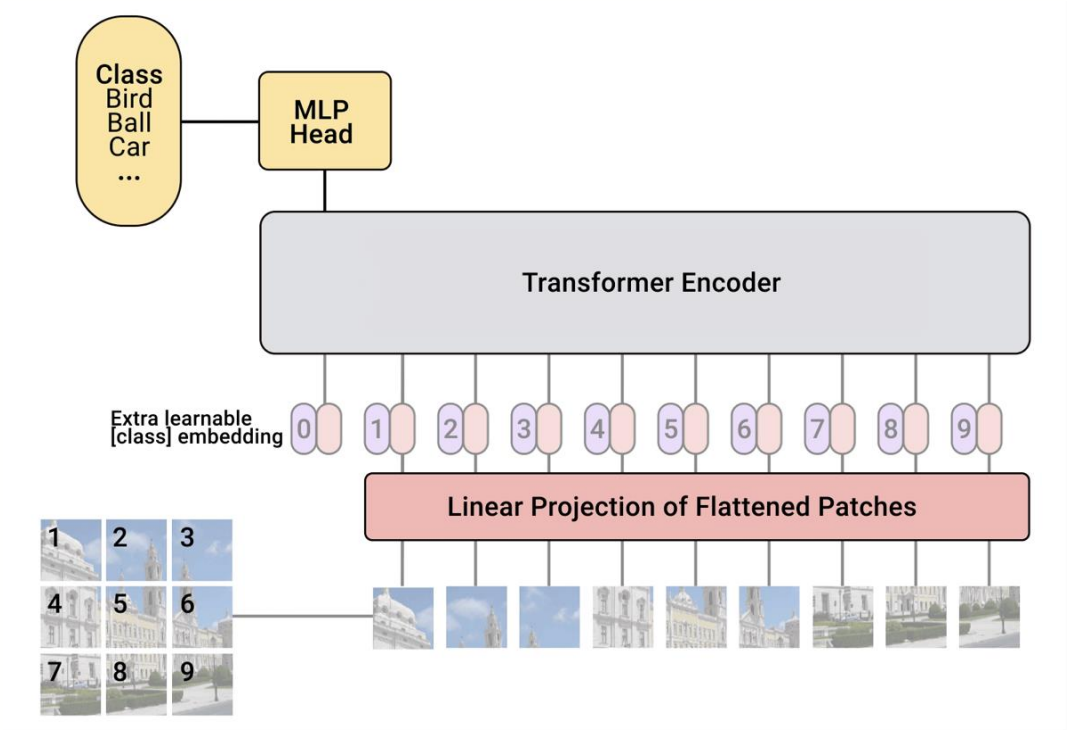


Рис. 16. Архитектура Vision Transformer

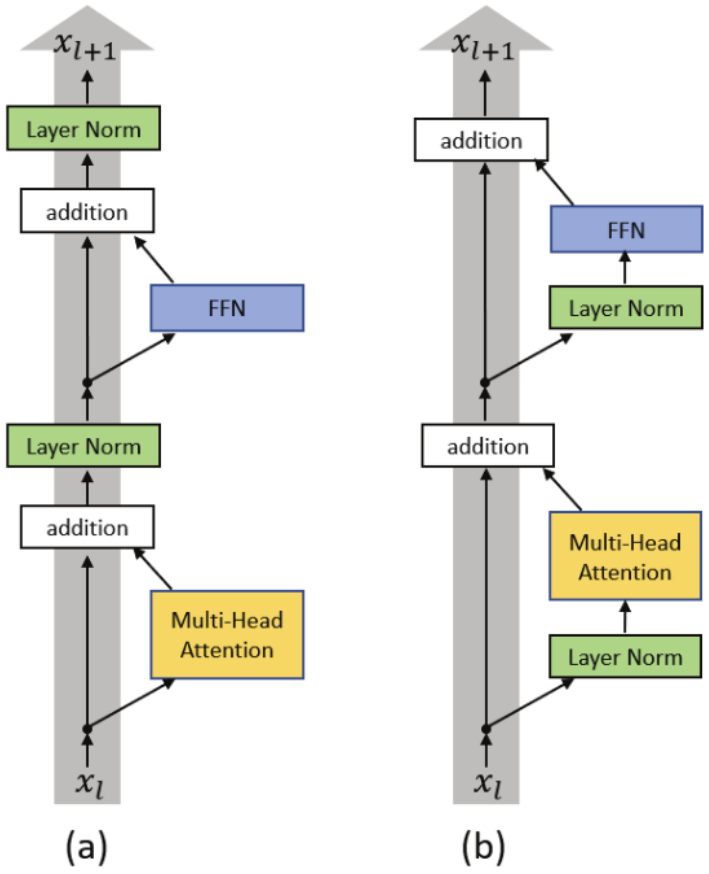
```
def img_to_patch(x, patch_size, flatten_channels=True):
    """
    Входы:
        x - torch.Tensor, представляющий изображение формы [B, C,
H, W].
        patch_size - Количество пикселей в каждом измерении патчей
(целое число).
        flatten_channels - Если True, патчи будут возвращены
в выпрямленном формате в виде вектора признаков тензора.
    """
    B, C, H, W = x.shape
    x = x.reshape(B, C, H//patch_size, patch_size, W//patch_size,
patch_size)
    x = x.permute(0, 2, 4, 1, 3, 5) # [B, H', W', C, p_H, p_W]
    x = x.flatten(1,2)              # [B, H'*W', C, p_H, p_W]
    if flatten_channels:
        x = x.flatten(2,4)          # [B, H'*W', C*p_H*p_W]
    return x
```

Рассмотрим пример использования этого подхода для набора данных CIFAR. Для исходных изображений размером 32×32 выберем размер патча 4. Таким образом получатся последовательности из 64 патчей размером 4×4.

По сравнению с исходными изображениями, распознать объекты на патчах последовательности стало гораздо сложнее. Тем не менее, это входные данные, которые передаются на вход трансформеру для решения задачи классификации изображений. Модель должна сама научиться тому, как ей скомбинировать патчи для распознавания объектов на изображениях. Интуиция, стоящая за сверточными нейронными сетями, в том, что изображение представляет собой сетку пикселей, при таком формате входных данных не применима.

После того как мы рассмотрели этап предварительной обработки исходного изображения, можно приступать к построению модели трансформера. Поскольку мы ранее подробно рассмотрели основы MultiHead Attention, используем готовый модуль PyTorch `nn.MultiheadAttention`. Далее используем версию предварительной нормализации слоя (LayerNorm) блоков трансформера [11]. Идея заключается в том, чтобы применять нормализацию слоев

не между остаточными блоками, а в качестве первого слоя в остаточных блоках. Такая реорганизация слоев обеспечивает лучший градиентный поток и устраняет необходимость в этапе разогрева. Визуализация разницы между стандартной версией Post-LN и версией Pre-LN показана на рис. 17.



(a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

Рис. 17. Сравнение версий Post-LN и Pre-LN слоев трансформера

Реализация блока внимания Pre-LN выглядит следующим образом:

```
class AttentionBlock(nn.Module):

    def __init__(self, embed_dim, hidden_dim, num_heads,
dropout=0.0):
        """
        Входные данные:
            embed_dim - Размер векторов признаков входа
и внимания.
```

```
            hidden_dim – Размер скрытого слоя в сети прямого
распространения (обычно в 2-4 раза больше, чем embed_dim).
            num_heads – Количество голов для использования в блоке
Multi-Head Attention.
            dropout – Вероятность Dropout для применения в сети
с прямым распространением.
        """
        super().__init__()

        self.layer_norm_1 = nn.LayerNorm(embed_dim)
        self.attn = nn.MultiheadAttention(embed_dim, num_heads,
                                          dropout=dropout)
        self.layer_norm_2 = nn.LayerNorm(embed_dim)
        self.linear = nn.Sequential(
            nn.Linear(embed_dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, embed_dim),
            nn.Dropout(dropout)
        )


    def forward(self, x):
        inp_x = self.layer_norm_1(x)
        x = x + self.attn(inp_x, inp_x, inp_x)[0]
        x = x + self.linear(self.layer_norm_2(x))
        return x
```

На данный момент есть все модули, необходимые для создания собственной реализации Vision Transformer. Кроме кодера трансформера, необходимы следующие модули:

– слой линейной проекции, который отображает входные патчи в вектор признаков большего размера. Он реализуется в виде простого линейного слоя, который принимает на вход каждый патч $M \times M$;

– классификационный токен, который добавляется к входной последовательности. Будем использовать выходной вектор признаков с классификационным токенов (CLS) для получения предсказания;

– обучаемые позиционные кодеры, которые добавляются к токенам перед обработкой трансформером. Они необходимы для изучения информации, зависящей от позиции, и преобразования набора патчей в последовательность. Поскольку мы работаем с фиксированным разрешением изображений, можно выучить значения позиционных кодировок вместо использования готового шаблона с функциями синуса и косинуса;

– головной блок с многослойным перцептроном, который принимает выходной вектор признаков токена CLS и преобразует его в предсказание классификатора. Обычно он реализуется в виде небольшой сети прямого распространения или даже одним линейным слоем.

Реализуем целиком модель Vision Transfromer:

```
class VisionTransformer(nn.Module):

    def __init__(self, embed_dim, hidden_dim, num_channels,
num_heads, num_layers, num_classes, patch_size, num_patches,
dropout=0.0):
        """
        Входы:
            embed_dim – Размер входных векторов признаков
для трансформера.
            hidden_dim – Размер скрытого слоя в сетях с прямым
распространением в трансформере.
            num_channels – Количество каналов входного сигнала
(3 для RGB).
            num_heads – Количество голов для использования в блоке
Multi-Head Attention.
            num_layers – Количество слоев для использования
в трансформере.
            num_classes – Количество классов для прогнозирования.
            patch_size – Количество пикселей, которое имеет патч
в каждом измерении.
            num_patches – Максимальное количество патчей, которое
может иметь изображение.
            dropout – Вероятность Dropout, применяемая в сети
с прямым распространением.
```

```
        """
        super().__init__()

        self.patch_size = patch_size

        # Слои/сети
        self.input_layer = nn.Linear(num_channels*(patch_size**2),
embed_dim)
        self.transformer =
nn.Sequential(*[AttentionBlock(embed_dim, hidden_dim, num_heads,
dropout=dropout) for _ in range(num_layers)])
        self.mlp_head = nn.Sequential(
            nn.LayerNorm(embed_dim),
            nn.Linear(embed_dim, num_classes)
        )
        self.dropout = nn.Dropout(dropout)

        # Параметры/эмбеддинги.
        self.cls_token = nn.Parameter(torch.randn(1,1,embed_dim))
        self.pos_embedding =
nn.Parameter(torch.randn(1,1+num_patches,embed_dim))


    def forward(self, x):
        # Предварительная обработка входных данных.
        x = img_to_patch(x, self.patch_size)
        B, T, _ = x.shape
        x = self.input_layer(x)

        # Добавление маркера CLS и позиционное кодирование.
        cls_token = self.cls_token.repeat(B, 1, 1)
        x = torch.cat([cls_token, x], dim=1)
        x = x + self.pos_embedding[:,:T+1]

        # Применение трансформера
        x = self.dropout(x)
        x = x.transpose(0, 1)
        x = self.transformer(x)
```

```
        # Выполнение классификационного прогнозирования.
        cls = x[0]
        out = self.mlp_head(cls)
        return out
```

Наконец, можно поместить весь код в модуль PyTorch Lightning. В качестве оптимизатора используем `torch.optim.AdamW`, который представляет собой Adam с исправленной реализацией сокращения весов (weight decay). Поскольку мы используем версию Pre-LN трансформера, больше не нужно использовать стадию разогрева скорости обучения. Вместо этого используем тот же планировщик скорости обучения, что и для сверточной нейронной сети в предыдущем примере классификации изображений:

```
class ViT(pl.LightningModule):

    def __init__(self, model_kwargs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.model = VisionTransformer(**model_kwargs)
        self.example_input_array = next(iter(train_loader))[0]

    def forward(self, x):
        return self.model(x)

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(),
lr=self.hparams.lr)
        lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer,
milestones=[100,150], gamma=0.1)
        return [optimizer], [lr_scheduler]

    def _calculate_loss(self, batch, mode="train"):
        imgs, labels = batch
        preds = self.model(imgs)
        loss = F.cross_entropy(preds, labels)
        acc = (preds.argmax(dim=-1) == labels).float().mean()
```

```
        self.log(f'{mode}_loss', loss)
        self.log(f'{mode}_acc', acc)
        return loss

    def training_step(self, batch, batch_idx):
        loss = self._calculate_loss(batch, mode="train")
        return loss

    def validation_step(self, batch, batch_idx):
        self._calculate_loss(batch, mode="val")

    def test_step(self, batch, batch_idx):
        self._calculate_loss(batch, mode="test")
```

5.2. ПРОВЕДЕНИЕ ЭКСПЕРИМЕНТОВ

Обычно сети Vision Transformer применяются к крупномасштабным задачам классификации изображений, таким как ImageNet, так как только на сложных задачах они способны продемонстрировать свой потенциал. Но можно ли с помощью Vision Transformer успешно решать задачи на небольших классических наборах данных, таких как CIFAR10? Чтобы выяснить это, обучим Vision Transformer с нуля на наборе данных CIFAR10. Сначала создадим функцию обучения для модуля PyTorch Lightning.

```
def train_model(**kwargs):
    trainer =
pl.Trainer(default_root_dir=os.path.join(CHECKPOINT_PATH, "ViT"),
                         accelerator="gpu" if
str(device).startswith("cuda") else "cpu",
                         devices=1,
                         max_epochs=180,

callbacks=[ModelCheckpoint(save_weights_only=True, mode="max",
monitor="val_acc"),
                                    LearningRateMonitor("epoch")])
    trainer.logger._log_graph = True
```

```
    trainer.logger._default_hp_metric = None # Optional logging
argument that we don't need

    pl.seed_everything(42)
    model = ViT(**kwargs)
    trainer.fit(model, train_loader, val_loader)
    model = ViT.load_from_checkpoint(trainer.checkpoint_callback.
    best_model_path)

    # Тестирование лучшей модели на валидационном
    # и тестовом множествах.
    val_result = trainer.test(model, val_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {"test": test_result[0]["test_acc"], "val":
val_result[0]["test_acc"]}

    return model, result
```

Теперь можно начать обучение модели. Как видно из реализации, есть несколько гиперпараметров, которые необходимо задать заранее.

Во-первых, рассмотрим размер патча. Чем меньше размер патчей, тем длиннее становятся входные последовательности. Хотя в целом это позволяет трансформеру моделировать более сложные зависимости, но при этом потребует больше времени на вычисления из-за квадратичного использования памяти в слое внимания. Кроме того, небольшие патчи могут усложнить задачу, поскольку трансформеру приходится изучать, какие патчи находятся рядом друг с другом, а какие – далеко. Эксперименты с размерами патчей 2, 4 и 8 пикселей, которые дают длину входной последовательности 256, 64 и 16 соответственно, показывают, что использование патчей размера 4 пикселя приводит к лучшим результатам.

Эмбеддинги и размер скрытого слоя оказывают такое же влияние на трансформер, как и на многослойный перцептрон. Чем больше размер, тем сложнее становится модель, и тем больше времени требуется на обучение. Однако в трансформерах необходимо учитывать еще один аспект: размер ключей-запросов в слоях MultiHead Attention. Каждый ключ имеет размер

embed_dim × num_heads. Учитывая, что длина входной последовательности равна 64, минимальный разумный размер для векторов ключей – 16 или 32. Меньшая размерность может слишком сильно ограничить возможные карты внимания. По результатам экспериментов выбрана размерность эмбеддингов 256. Скрытая размерность в сетях прямого распространения обычно в 2 – 4 раза больше, чем размерность эмбеддингов, поэтому выбрано значение 512.

Наконец, скорость обучения трансформеров обычно относительно мала, и в статьях обычно используется значение 3e-5. Однако, поскольку мы работаем с меньшим набором данных и имеем потенциально более простую задачу, можно увеличить скорость обучения без каких-либо последствий. Чтобы уменьшить переобучение, используем значение вероятности Dropout 0,2. Будем также использовать небольшие дополнения изображений в качестве регуляризации во время обучения.

В целом, на наборе данных CIFAR10 Vision Transformer не показывает себя слишком чувствительным к выбору гиперпараметров.

```
model, results = train_model(model_kwargs={
                                'embed_dim': 256,
                                'hidden_dim': 512,
                                'num_heads': 8,
                                'num_layers': 6,
                                'patch_size': 4,
                                'num_channels': 3,
                                'num_patches': 64,
                                'num_classes': 10,
                                'dropout': 0.2
                            },
                            lr=3e-4)
```

Модель Vision Transformer достигает производительности, равной примерно 75% при проверке и тестировании. Для сравнения, почти все архитектуры сверточных нейронных сетей позволяют получить эффективность классификации около 90%. Это значительный разрыв. Он показывает, что хотя модели вида Vision Transformer демонстрируют высокие результаты на наборе данных

ImageNet при предварительном обучении, они не могут приблизиться к простым сверточным нейронным сетям на наборе CIFAR10 при обучении с нуля. Различия между сверточными нейронными сетями и трансформерами хорошо видны на кривых обучения (рис. 18).

Из рисунка видно, что сеть ResNet обучается гораздо быстрее на первых итерациях. Сеть ResNet достигает лучших показателей по сравнению с трансформерами всего через 5 эпох (2000 итераций). В то время как потери при обучении ResNet и точность валидации имеют схожую тенденцию к изменению во времени, производительность Vision Transformer при валидации лишь незначительно изменяется после 10 тысяч итераций, в то время как потери при обучении только начинают снижаться. Тем не менее, Vision Transformer также способен достичь точности, близкой к 100%, на обучающем множестве.

Наблюдаемые явления можно объяснить с помощью концепции, которую мы уже рассматривали ранее: априорные знания о природе данных. Сверточные нейронные сети были разработаны исходя из предположения, что изображения инвариантны к трансформации.

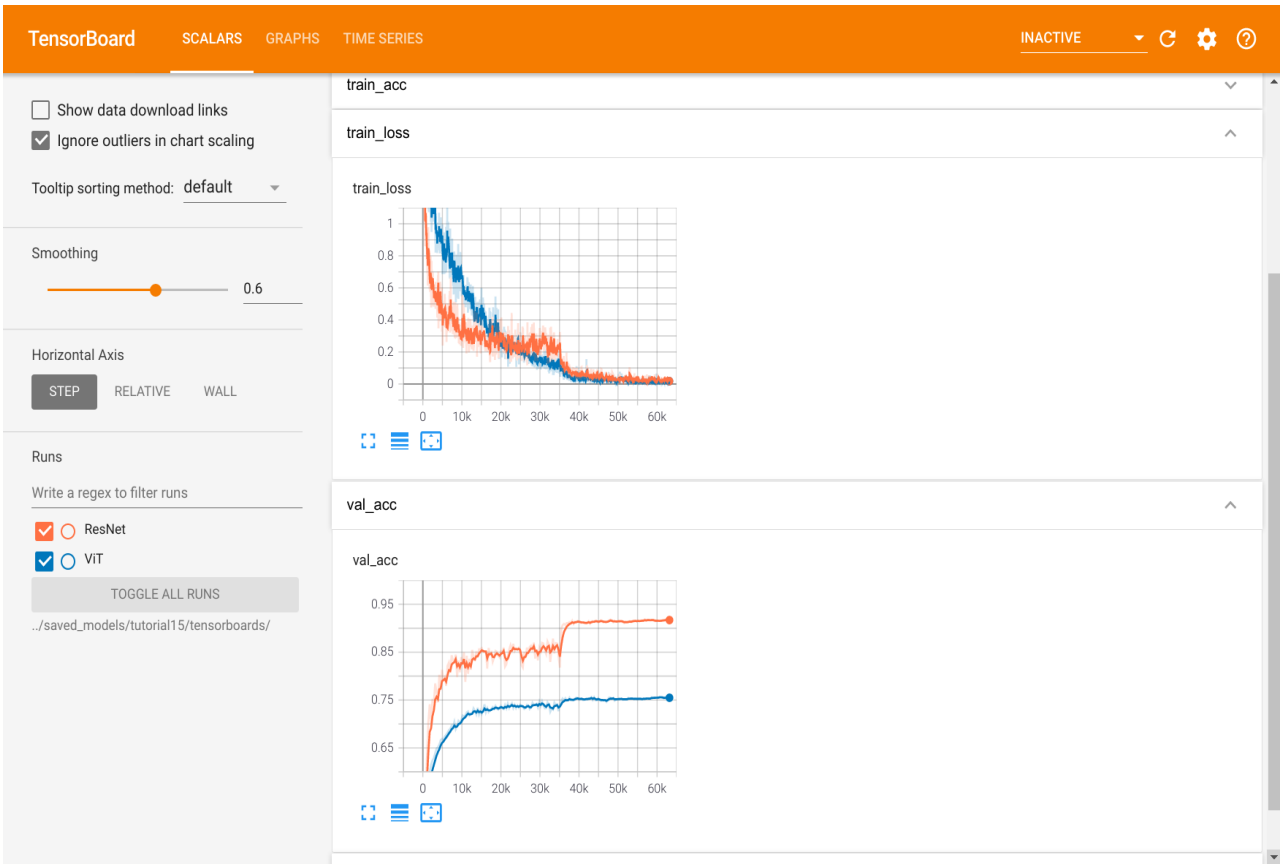


Рис. 18. Визуализация кривых обучения ResNet и ViT на наборе CIFAR10

Следовательно, мы применяем свертки с общими фильтрами ко всему изображению. Кроме того, архитектура сверточной нейронной сети учитывает концепцию расстояния на изображении: два пикселя, расположенные близко друг к другу, более связаны, чем два удаленных друг от друга пикселя. Локальные паттерны объединяются в более крупные до тех пор, пока сеть не выполнит предсказание.

В отличие от сказанного выше, Vision Transformer не знает, какие два пикселя находятся близко друг к другу, а какие далеко друг от друга. Он должен получить эту информацию исключительно из разреженных обучающих данных. Это огромный недостаток в случае небольшого набора данных, поскольку такая информация имеет решающее значение для обобщения на тестовом наборе данных. При достаточно больших наборах данных и(или) хорошем предварительном обучении трансформер может выучить эту информацию без индуктивных знаний. К тому же, сама архитектура является более гибкой, чем сверточные нейронные сети. Особенно трудно обрабатывать в сверточных нейронных сетях связи между локальными паттернами, находящимися на больших удалениях друг от друга, в то время как в трансформерах все паттерны имеют расстояние, равное единице. Именно поэтому Vision Transformer показывают такие хорошие результаты на крупномасштабных наборах данных, таких как ImageNet, но сильно проигрывают сверточным нейронным сетям при применении к небольшим наборам данных, таким как CIFAR10.

ЗАКЛЮЧЕНИЕ

Фреймворки PyTorch и PyTorch Lightning являются удачным выбором для решения задач глубокого обучения благодаря их гибкости, эффективности и поддержке сообщества. Они обеспечивают богатые возможности для создания, обучения и тестирования моделей, а также для визуализации данных и результатов.

Ожидается, что представленные в пособии материалы способствуют освоению читателями указанных инструментов и технологий, а также стимулируют дальнейшее изучение и исследование в области глубокого обучения. Следует отметить, что область искусственного интеллекта характеризуется быстрым развитием, и непрерывное обучение является неотъемлемым элементом успешной профессиональной деятельности в этой сфере.

СПИСОК ЛИТЕРАТУРЫ

1. Ramachandran, Prajit, Barret Zoph, and Quoc V. Le. “Searching for activation functions.” arXiv preprint arXiv:1710.05941, 2017.

2. Glorot, Xavier, and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks.” Proceedings of the thirteenth international conference on artificial intelligence and statistics, 2010.

3. He, Kaiming, et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.” Proceedings of the IEEE international conference on computer vision, 2015.

4. Kingma, Diederik P. & Ba, Jimmy. “Adam: A Method for Stochastic Optimization.” Proceedings of the third international conference for learning representations (ICLR), 2015

5. He, Kaiming, et al. “Deep residual learning for image recognition.” Proceedings of the IEEE conference on computer vision and pattern recognition, 2016.

6. He, Kaiming, et al. “Identity mappings in deep residual networks.” Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14. Springer International Publishing, 2016.

7. Li, Hao, et al. “Visualizing the loss landscape of neural nets.” Advances in neural information processing systems 31, 2018.

8. He, Tong, et al. “Bag of tricks for image classification with convolutional neural networks.” Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2019.

9. Vaswani, Ashish, et al. “Attention is all you need.” Advances in neural information processing systems 30, 2017.

10. Dosovitskiy, Alexey, et al. “An image is worth 16x16 words: Transformers for image recognition at scale.”, 2020.

11. Xiong, Ruibin, et al. “On layer normalization in the transformer architecture.” International Conference on Machine Learning. PMLR, 2020.

СОДЕРЖАНИЕ

Учебное электронное издание

ЕЛИСЕЕВ Алексей Игоревич
МИНИН Юрий Викторович
КУЛАКОВ Юрий Владимирович

РЕШЕНИЕ ЗАДАЧ ГЛУБОКОГО ОБУЧЕНИЯ С ИСПОЛЬЗОВАНИЕМ ФРЕЙМВОРКОВ PYTORCH И PYTORCH LIGHTNING

Учебное пособие

Редактирование И. В. Калистратовой
Графический и мультимедийный дизайнер Т. Ю. Зотова
Обложка, упаковка, тиражирование И. В. Калистратовой

ISBN 978-5-8265-2659-0

Подписано к использованию 08.11.2023.
Тираж 50 шт. Заказ № 142

Издательский центр ФГБОУ ВО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106, к. 14
Тел./факс (4752) 63-81-08.
E-mail: izdatelstvo@tstu.ru