ЛАБОРАТОРНАЯ РАБОТА №7-8.
Обучение нейронных сетей в PyTorch

ЦЕЛЬ РАБОТЫ – Решение задачи классификации с использованием нейронной сети; изучение принципов работы нейронных сетей; реализация методов обучения нейронных сетей на языке Python.

Содержание

1 Теоретические сведения

1.1 Методы машинного обучения

В зависимости от специфики процесса обучения алгоритмы машинного обучения можно разделить на два класса: **без учителя** и **с учителем**. Алгоритму обучения без учителя предъявляется набор данных, содержащий много признаков, а алгоритм должен выявить полезные структурные свойства этого набора (например, разделить набор данных на кластеры). Алгоритму обучения с учителем предъявляется набор данных, в котором каждый пример снабжен **меткой**, или **целевой переменной (классом)**.

Главная проблема машинного обучения состоит в том, что алгоритм должен хорошо работать на новых данных, которых он раньше не видел, а не только на тех, что использовались для обучения модели. Эта способность правильной работы на ранее не предъявленных данных называется **обобщением**.

В процессе обучения мы минимизирует **ошибку обучения**. Но в машинном обучении необходимо также уменьшить ошибку *обобщения*. Таким образом возникает вопрос: Как можно повлиять на качество работы на тестовом наборе,

если для наблюдения доступен только обучающий набор? Можно математически проследить связь если выполнить условия:

1) примеры в каждом наборе независимы;

2) **одинаково распределены** (выбираются из одного и того же распределения вероятности).

Мы выбираем обучающий набор, используем его для минимизации ошибки обучения, а затем выбираем тестовый набор. При таком процессе ожидаемая ошибка тестирования больше или равна ожидаемой ошибке обучения. Факторов, определяющих качество работы алгоритма машинного обучения, два:

1) сделать ошибку обучения как можно меньше;

2) сократить разрыв между ошибками обучения и тестирования.

Эти факторы соответствуют двум центральным проблемам машинного обучения: **недообучению** и **переобучению (*overfitting*)**. ***Недообучение*** происходит, когда модель не позволяет получить достаточно малую ошибку на обучающем наборе, а ***переобучение*** – когда разрыв между ошибками обучения и тестирования слишком велик.

Управлять склонностью модели к переобучению или недообучению позволяет ее **емкость** (***capacity***), т.е. число настраиваемых параметров. Модели малой емкости испытывают сложности в аппроксимации обучающего набора. Модели большой емкости склонны к переобучению, поскольку запоминают свойства обучающего набора, не присущие тестовому.

Расхождение между **ошибкой обучения** и **ошибкой обобщения** растет с ростом емкости модели, но убывает по мере увеличения количества обучающих примеров. Ожидаемая ошибка обобщения никогда не может увеличиться с ростом количества обучающих примеров.

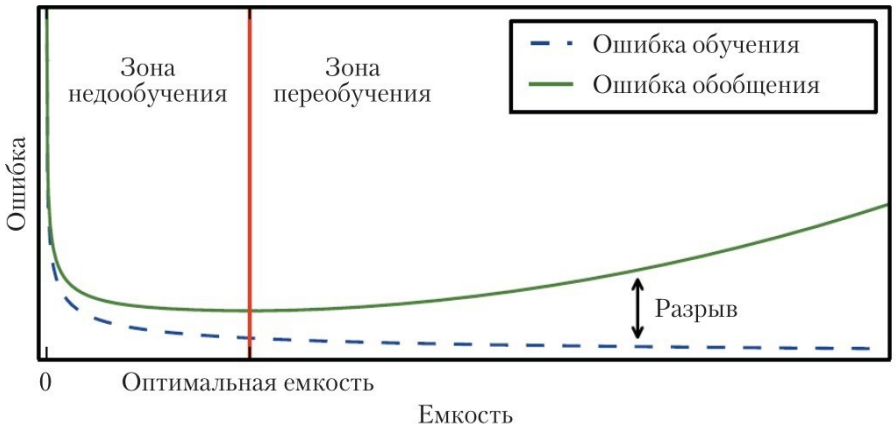

Рисунок 1 – Типичная связь между емкостью и ошибкой.

У большинства алгоритмов машинного обучения имеются гиперпараметры, управляющие поведением алгоритма. К гиперпараметрам относятся и те, что регулируют емкость модели. Подбор гиперпараметров в процессе обучения лишен смысла. При попытке подобрать их на обучающем наборе всегда выбиралась бы максимально возможная емкость модели, что приводило бы к переобучению.

1.2 ИНС: Базовые понятия

Искусственные нейронные сети (ИНС) изначально изучались с ~~не оправдавшей себя~~ надеждой разработать машины, обладающие разумным восприятием и познавательной способностью, с помощью симуляции физической структуры человеческого мозга. Эти принципы ИНС нашли множество применений в разнообразных областях, включая распознание образов и обработку сигналов. Нейронные сети состоят из множества соединенных между собой нейронов различающиеся по способу соединения (сетей прямого распространения, рекуррентные сети, самоорганизующиеся сети и т.д.)

Базовая модель нейрона имеет набор соединяющих весов w_i (соответствующих синапсам в биологическом нейроне), суммирующее устройство и функцию активации, как показано на рисунке 2.

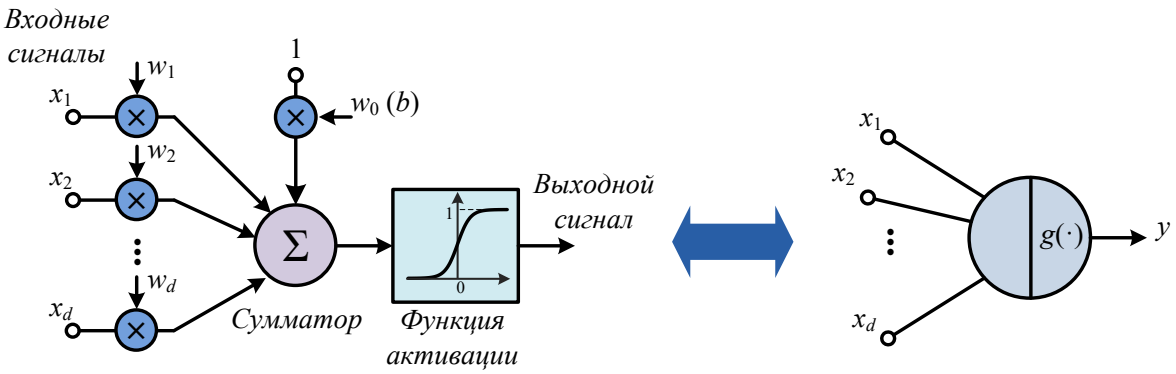

Рисунок 2 – Базовая модель нейрона

Представленная на рисунке модель нейрона описывается выражением:

$$y = g\left(\sum_{i=1}^{d} w_i x_i + b\right), \tag{1.1}$$

где x_i – выходные данные, w_i – веса, b – смещение, $g(\cdot)$ – функция активации.

Рассмотрим описание (1.1) более детально. На вход функции активации $g()$ подается взвешенная комбинация входных данных (признаков):

$$v = \sum_{i=1}^{d} w_i x_i + b. \tag{1.2}$$

Каждый нейрон имеет поляризацию (связь с весом b, по которой поступает единичный сигнал), а также множество связей с весами w_i, по которым поступают входные данные. Для упрощения можно представить, что входные данные представляют собой вектор $\mathbf{x}$ с компонентами $x_1, x_2, \ldots, x_d$. В этом случае (1.2) перепишется в виде

$$v = \mathrm{w}^T \mathbf{x} + b, \tag{1.3}$$

где $\mathbf{x} = [x_1 \quad x_2 \quad \cdots \quad x_d]^T$ – вектор данных, $\mathbf{w} = [w_1 \quad w_2 \quad \cdots \quad w_d]^T$ – вектор весов.

Для большей унификации к вектору данных можно добавить искусственную переменную со значением 1, а смещение b добавить к вектору весов, т.е.

$$\mathbf{x} = [1 \quad x_1 \quad x_2 \quad \cdots \quad x_d]^T, \qquad \mathbf{w} = [b \quad w_1 \quad w_2 \quad \cdots \quad w_d]^T. \tag{1.4}$$

В этом случае выражение (1.3) упростится до вида:

$$v = \mathbf{w}^T\mathbf{x}. \tag{1.5}$$

Функция активации $g(v)$ может быть, как линейной, так и нелинейной. Популярной нелинейной функцией является логистический сигмоид:

$$g(v) = \frac{1}{1 + e^{-v}}. \tag{1.6}$$

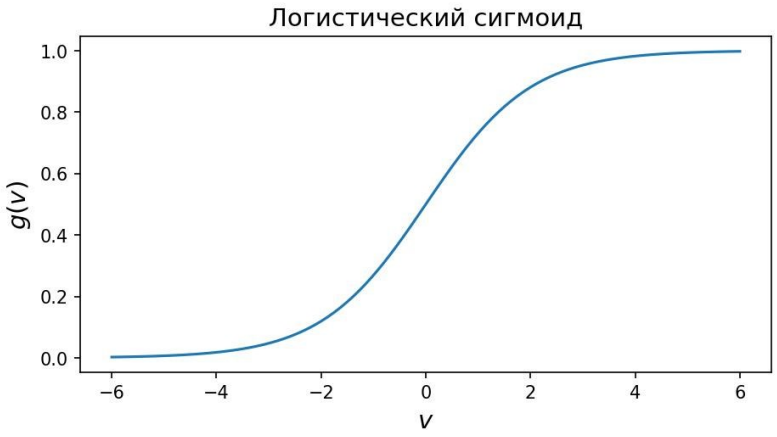

Рисунок 3 – Функция активации: логистический сигмоид

Логистический сигмоид имеет 2 преимущества: 1) непрерывность, т.е. есть возможность вычислить производную, что позволяет применять градиент; 2) его производная выражается через саму функцию:

$$\frac{dg}{dv} = \frac{e^{-v}}{(1 + e^{-v})^2} = \underbrace{\frac{1}{(1 + e^{-v})}}_{g(v)} \underbrace{\frac{e^{-v}}{(1 + e^{-v})}}_{1-g(v)} = g(v)\big(1 - g(v)\big). \tag{1.7}$$

На рисунке 4 показан график производной логистического сигмоида.

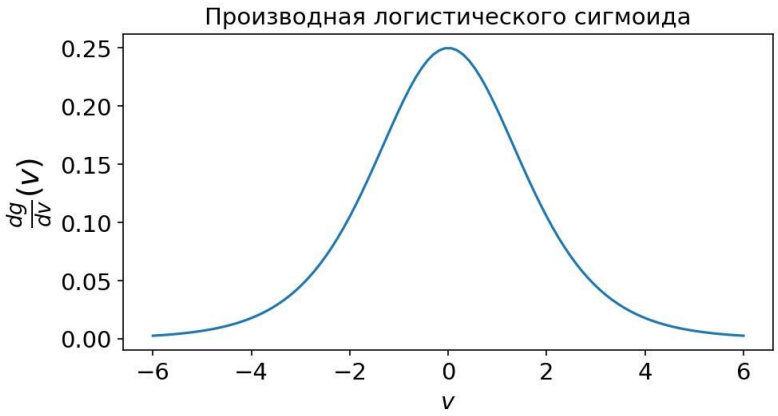

Рисунок 4 – Производная логистического сигмоида

Описанная модель нейрона (1.1) полностью совпадает с описанием логистической регрессии.

1.3 Однослойная нейронная сеть прямого распространения

Однослойную нейронную сеть образуют нейроны, расположенные в одной плоскости (рисунок 5).

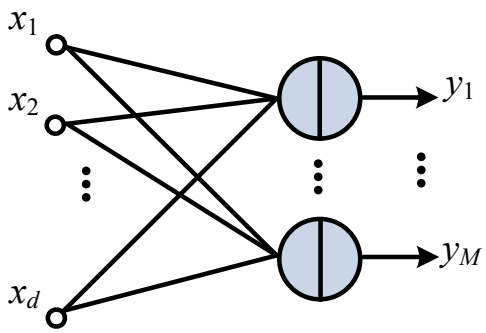

Рисунок 5 – Однослойная нейронная сеть

На рисунке не показаны веса каждого нейрона, входящего в слой. Мы можем обозначить вектор-строку весов i-го нейрона как $\mathbf{w}_i$. Тогда j-й вес i-го нейрона будет обозначаться как w_{ij}. Выход i-го нейрона сети будет иметь вид:

$$y_i = g\left(\sum_{j=0}^{d} w_{ij}x_i\right) = g(\mathbf{w}_i\mathrm{x}). \tag{1.8}$$

Обратите внимание, что в выражении (1.8) $\mathbf{w}_i$ понимается как вектор-строка, а в (1.5) w понимался как вектор-столбец.

Можно объединить веса всех нейронов, принадлежащих одному слою в матрицу:

$$\mathrm{W} = [w_{ij}], \qquad i = 1,2,..,M, \qquad j = 0,1,..,d, \tag{1.9}$$

в этом случае вычисление выходов слоя нейронной сети можно записать как

$$\mathrm{y} = g(\mathbf{W}\mathbf{x}), \tag{1.10}$$

где $\mathbf{y} = [y_1 \quad y_2 \quad \cdots \quad y_M]^T$ – вектор выходов нейронной сетей.

Выражение (1.10) необходимо понимать следующим образом: $\mathbf{x}$ – это вектор-столбец, в котором хранятся входные данные, в строках матрицы $\mathbf{W}$ хранятся вектора весов нейронов; после вычисления $\mathbf{Wx}$ образуется вектор-столбец $\mathbf{v} = [v_1 \quad v_2 \quad \cdots \quad v_M]^T$ значений скалярных произведений входных данных на веса нейронов, затем для каждого значения v_i вычисляется функция активации $y_i = g(v_i)$.

1.4 Классификатор на базе нейронной сети

Многоклассовый классификатор на базе однослойной нейронной сети имеет математическое описание:

$$y_i(\mathbf{x}) = g(\mathbf{w}_i\mathbf{x}), \ i = 1,..M. \tag{1.11}$$

где M – количество классов.

Для обучения нейронной сети необходимо иметь тренировочный набор $\mathcal{D} = \{(\mathrm{x}^n, c^n) \mid n = 1, \dots, N\}$, где c^n определяет метку класса образца x^n). Необходимо определить целевые переменные для задачи классификации. В идеальном случае для образца k-го класса, k-ый выход нейронной сети должен иметь нулевой значение, а все остальные выходы должны быть нулевыми. По этой причине выходное (целевое) значение для определенного образца (x^n, c^n) определяется как

$$t_i^n = \begin{cases} 1, & i = c^n, \\ 0, & \text{иначе.} \end{cases} \tag{1.12}$$

Задача обучения состоит в подстройке весов $\mathbf{w}_i$ (или матрицы (1.9), что тоже самое) таким образом, чтобы сумма квадратов ошибок на тренировочном наборе

$$E = \frac{1}{2}\sum_{n=1}^{N}\sum_{i=1}^{M}[y_i(\mathbf{x}^n) - t_i^n]^2 \tag{1.13}$$

была минимальной.

Функцию (1.13) также называют ***функцией потерь*** (англ. *loss function*).

Для минимизации функции потерь (1.13) можно использовать метод градиентного спуска. Для простоты рассмотрим однослойную сеть. Функция потерь имеет следующий вид:

$$E = \frac{1}{2}\sum_{n=1}^{N}\sum_{i=1}^{M}[g(\mathbf{w}_i\mathbf{x}^n) - t_i^n]^2. \quad (1.14)$$

Для градиентного спуска требуется вычислить частные производные функции E по коэффициентам $w_{i,j}$, объединение всех частных производных в один вектор и будет градиентом E. Чтобы упростить задачу представим, что у нас есть только один обучающий пример $\mathbf{x}^1$ и один выходной нейрон сети, т.е. $M = 1$. В этом случае функция потерь примет вид:

$$E = \frac{1}{2}[g(\mathbf{w}_1\mathbf{x}^1) - t_1^1]^2. \quad (1.15)$$

Производная (1.15) по коэффициентам $w_{1,j}$ имеет вид

$$\frac{\partial E}{\partial w_{1,j}} = (g(\mathbf{w}_1\mathbf{x}^1) - t_1^1) \cdot \frac{d(g(\mathbf{w}_1\mathbf{x}^1) - t_1^1)}{dw_{1,j}}. \quad (1.16)$$

Чтобы завершить вычисление (1.16) нам требуется производная логистической функции, которая была посчитана в (1.7). Учитывая это получаем:

$$\begin{aligned}\frac{\partial E}{\partial w_{1,j}} &= (g(\mathbf{w}_1\mathbf{x}^1) - t_1^1) \cdot \frac{d(g(\mathbf{w}_1\mathbf{x}^1) - t_1^1)}{dw_{1,j}} \\ &= (g(\mathbf{w}_1\mathbf{x}^1) - t_1^1) \cdot g(\mathbf{w}_1\mathbf{x}^1) \cdot \big(1 - g(\mathbf{w}_1\mathbf{x}^1)\big) \cdot \frac{d(\mathbf{w}_1\mathbf{x}^1)}{dw_{1,j}} \\ &= (g(\mathbf{w}_1\mathbf{x}^1) - t_1^1) \cdot g(\mathbf{w}_1\mathbf{x}^1) \cdot \big(1 - g(\mathbf{w}_1\mathbf{x}^1)\big) \cdot x_j\end{aligned} \quad (1.17)$$

Выражение (1.17) каким образом находится градиент функции ошибки в случае, когда у нас имеется всего один обучающий образец (x^1, c^1). Основной прием, который мы применяли в (1.17) – это известное из математического анализа правило дифференцирования сложных функций:

$$\frac{df(g(x))}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}. \quad (1.18)$$

Выражение (1.17) достаточно легко обобщить на случай, когда имеется N обучающих примеров и M нейронов однослойной сети:

$$\frac{\partial E}{\partial w_{i,j}} = \sum_{n=1}^{N}\sum_{i=0}^{M}(g(\mathbf{w}_i\mathbf{x}^n) - t_i^n) \cdot g(\mathbf{w}_i\mathbf{x}^n) \cdot \big(1 - g(\mathbf{w}_i\mathbf{x}^n)\big)x_j^n \quad (1.19)$$

Используя метод градиентного спуска изменение весов выполняется следующим образом:

$$w_{i,j} = w_{i,j} - \eta\frac{\partial E}{\partial w_{i,j}}, \quad (1.20)$$

где η – параметр, отвечающий за скорость сходимости $(0 < \eta < 1)$.

1.5 Многослойный персептрон

Описанная в разделе 1.3 однойслойная нейронная сеть может использоваться, как строительный блок для получения многослойной нейронной сети.

Рассматриваемая в данном разделе ИНС относится к сетям прямого распространения (*feedforward neural network*), и ещё более конкретно – это архитектура, называемая многослойным персептроном (*MultiLayer Perceptron, MLP*). Многослойный персептрон (МСП) показан на рисунке 6.

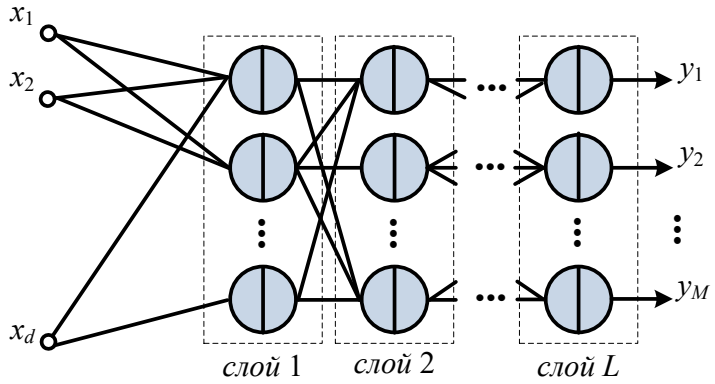

Рисунок 6 – Многослойный персептрон (МСП)

Как видим МСП представляет собой комбинацию полносвязанных (*full connected*) слоев нейронов. В зависимости от функции активации, используемой на последнем (выходном) слое МСП может быть моделью регрессии или классификации.

В случае, когда МСП осуществляет классификацию в качестве функции активации на последнем слое выбирается ***функция softmax***:

$$g(\boldsymbol{v}) = [g_1, \dots, g_M]^T, \;\; g_i(\boldsymbol{v}) = \frac{e^{v_i}}{\sum_{k=1}^{M} e^{v_k}}. \tag{1.21}$$

Функция softmax – это обобщение сигмоидальной функции на многомерные выходы. Она имеет свойство $\sum_{k=1}^{M} g_k = 1$ и $g_k > 0$ для всех k.

Важной особенностью функции softmax является то, что её производная может быть легко посчитана через значения самой функции:

$$\frac{\partial g_i}{\partial v_i} = g_i(\boldsymbol{v})(1 - g_i(\boldsymbol{v})). \tag{1.22}$$

Когда МСП решает задачу классификации, то в качестве функции потерь лучше вместо суммы квадратов ошибок (1.13) использовать перекрестную энтропию:

$$CE = -\sum_{n=1}^{N}\sum_{i=1}^{M} t_i^n \log y_i^n, \tag{1.23}$$

где t_i^n – это целевое значение на i-м выходе сети для n-го примера, y_i^n – выходное значение нейронной сети на i-м выходе сети для n-го примера.

Перекрестную энтропию используют совместно с функцией softmax в выходном слое. В этом случае:

$$\frac{\partial CE}{\partial v_i} = -\sum_{n=1}^{N}\sum_{i=1}^{M} \frac{\partial CE}{\partial g_i}\frac{\partial g_i}{\partial v_i} = \sum_{n=1}^{N} y_i^n - t_i^n \tag{1.24}$$

Для обучения МСП, как и других глубоких нейронных сетей используется алгоритм ***обратного распространения ошибки***, который целиком и полностью зиждется на правиле дифференцирования сложных функций (1.18). Для изучения деталей этого алгоритма лучше обратиться к специальной литературе, например, [1, стр. 75].

1.6 Использование библиотеки PyTorch

`PyTorch` – это библиотека для задач машинного обучения на языке Python, основанный фремворке Torch, который имеет открытый исходный код. Он используется для таких приложений, как компьютерное зрение и обработка естественного языка.

Если на компьютере установлен Python, то для установки библиотек `PyTorch` в командной строке нужно ввести команды

```
pip install torch
pip install torchvision
```

Главное преимущество `PyTorch`, что он позволяет определить вычислительный граф нейронной сети, который определяет прямой путь прохождения данных через нейронную сеть. В дальнейшем этот граф может быть использован для автоматического дифференцирования для расчета градиента и обучения модели.

1.6.1 Тензоры

Тензор – это базовая структура данных в PyTorch. В математике и физике тензоры связывают с понятиями пространств, систем отсчета и преобразованиями между ними, одна в машинном обучении тензор это всего лишь обобщение понятий вектора и матрицы на произвольную размерность. Поэтому, когда вы сталкиваетесь с тензорами в контексте задач машинного обучения следует знать, что речь идет о *многомерных массивах*.

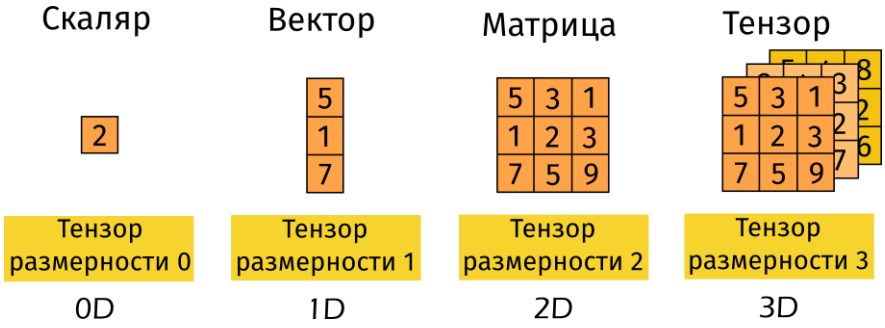

Рисунок 7 – Тензоры, как обобщение многомерных массивов

Тензор размерности (или ранга) 0 – просто число, или скаляр. Тензор размерности 1 – есть вектор, а тензор размерности 2 – матрица (рис. 7).

Чтобы создать тензор в PyTorch нужно импортировать модуль `torch`, а затем обратится к конструктору `torch.tensor`, передав в него список значений, который должен быть записан в тензор. Приведем пример

```
import torch
a = torch.tensor([1, 2, 3])
>> tensor([1, 2, 3])
```

В результате был создан одномерный тензор на три элемента. Создать двумерный тензор (матрицы) на две строки и три столбца можно сделать следующим образом:

```
b = torch.tensor([[1,2,3], [4,5,6]])
>> tensor([[1, 2, 3],
           [4, 5, 6]])
```

Обратится к отдельным элементам можно по индексу, как и в случае с обычными массивами. Например, обратится ко второму элементу нулевой строки матрицы `b` можно так

```
b[0,2]
>> tensor(3)
```

Чтобы узнать размер тензора нужно вызвать метод `size()`, как показано в примере ниже

```
b.size()
>> torch.Size([2, 3])
```

PyTorch также позволяет изменять размерность тензоров, используя метод `view`. Например, рассматриваемый двумерный тензор `b` можно превратить в одномерный используя команду

```
c = b.view(1,6)
>> tensor([[1, 2, 3, 4, 5, 6]])
```

Можно не указывать вторую размерность в методе `view`, тогда PyTorch вычислит её исходя из размера исходного тензора, поэтому аналогичный результат дает следующая код:

```
c = b.view(1,-1)
>> tensor([[1, 2, 3, 4, 5, 6]])
```

1.6.2 Автоматическое вычисление градиента

В целом тензоры PyTorch напоминают массивы NumPy, их главной отличительной чертой является возможность отслеживать породивший их граф вычислений и автоматически рассчитывать градиент, что чрезвычайно важно для задач глубокого обучения. Возможность автоматически вычислять градиент означает легкость в выполнении алгоритма обратного распространения ошибки, поскольку сложная задача вычисления производных от параметров модели, которая раньше лежала на разработчике, теперь может быть автоматизирована.

Рассмотрим данную особенность на конкретном примере. Для этого, покажем, как можно использовать механизмы автоматического дифференцирования PyTorch для того, чтобы обучить модель линейной регрессии. Допустим у нас есть данные

```
X = torch.tensor([0.4, 2.5, 3.2, 3.9, 4.3, 6.0, 6.3, 7.0, 7.4, 7.6, 8.0, 8.5,])
y = torch.tensor([6.4, 6.0, 5.8, 4.7, 5.2, 4.9, 4.6, 3.9, 4.2, 3.5, 4.0, 3.6,])
```

которые мы хотим аппроксимировать линейной моделью:

```
def model(x, w, b):
    y_out = x*w + b
    return y_out
```

Расхождение между реальными данными `y` и теми, что будет предсказывать модель будем оценивать при помощи функции потерь:

```
def loss_fn(y_true, y_pred):
    MSE = torch.mean((y_true-y_pred)**2)
    return MSE
```

Теперь создадим тензор с параметрами модели:

```
params = torch.tensor([1.0, 0.0], requires_grad=True)
```

Мы будем использовать `params`, чтобы хранить в нем параметры модели `w` и `b`. Таким образом, в начальный момент `w=params[0]=1.0`, а `b=params[1]=0.0`.

Атрибут `requires_grad=True` указывает PyTorch отслеживать все тензоры, которые получаются в результате операций над params. Т.е. у любого тензора, произошедшего от params, будет доступ к цепочке функций, вызывавшихся для получения из params этого тензора. Если эти функции дифференцируемые, то величина производной будет автоматически занесена в атрибут `grad` тензора `params`.

Раскроем сказанное на примере.

```
y_pred = model(X[0], *params)
```

```
loss = loss_fn(y_pred, y[0])
loss.backward()
```

В первой строке мы передели в модель линейной регрессии первое значение из обучающей выборки, чтобы получить её прогноз `y_pred`. В этом коде используется одна из уловок Python – *распаковка аргументов*: выражение *params указывает компилятору передавать элементы `params` в виде отдельных аргументов. Поэтому `model(X[0], *params)` эквивалентно `model(X[0], params[0], params[1])`.

Далее происходит вызов функции потерь `loss_fn(y_pred, y[0])`, благодаря которой мы можем узнать насколько сильно расходится прогноз модели `y_pred` и целевое значение `y[0]` в точке `X[0]`. После этого необходимо вызнать метод `backward` тензора `loss`, чтобы PyTorch рассчитал производные функции потерь для всех параметров модели. Если после выполнения данных действий обратится к атрибуту grad тензора `param`, то там будут находится производные функции потерь

```
params.grad
>> tensor([ -4.8000, -12.0000])
```

В математических терминах мы сделали следующее. У нас есть модель линейной регрессии

$$f(x) = wx + b. \tag{1.25}$$

Далее мы объявили функцию потерь

$$L(f(\mathbf{x}), \mathbf{y}) = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - y_i)^2. \tag{1.26}$$

Для обучения модели регрессии необходимо знать градиент функции потерь по параметрам модели, т.е.

$$\frac{\partial L}{\partial w} \text{ и } \frac{\partial L}{\partial b}.$$

Рассмотрим значения функции $L(f(\mathbf{x}), \mathbf{y})$ и её производные в точке X[0]=0.4, где истинное значение равно y[0]=6.4, для модели, у которой начальные значения $w = 1$, а $b = 0$.

$$\frac{\partial L}{\partial w}(x_0) = 2 \cdot (f(x_0) - y_0) \cdot \frac{\partial f}{\partial w} = \Big|_{f(x) = w \cdot x + b} =$$

$$= 2(wx_0 + b - y_0)(x_0).$$

Учитывая, что изначально параметры модели $w = 1$, а $b = 0$, то получим

$$\frac{\partial L}{\partial w}(0{,}4) = 2 \cdot (1 \cdot 0{,}4 + 0 - 6{,}4) \cdot (0{,}4) = -4{,}8$$

Это как раз то значение, которое позволил найти метод `backward`, когда мы применили его к тензору `loss`. Значение $\frac{\partial L}{\partial b}(0{,}4) = -12$ вычисляется аналогичным образом. Самое главное, что требуется уяснить из данного примера это то, что PyTorch избавляет разработчика от необходимости прописывать в коде операции вычисления градиента для функции потерь, за счет использования библиотеки автоматического дифференцирования `autograd`.

1.6.3 Обучение линейной регрессии с использованием PyTorch

В данном разделе будет показано, как используя возможности PyTorch автоматического вычисления градиента функции ошибки выполнить «обучение» линейной регрессии. Кратко напомним, что модель линейной регрессии $f(x)$ в нашем примере зависит от параметров w и b и описывается выражением (1.25). Функция потерь L описывается выражением (1.26). Под процессом обучения понимается минимизация функции потерь методом градиентного спуска, который в данном случае описывается выражениями:

$$\begin{aligned} w &= w - \eta \frac{\partial L}{\partial w}, \\ b &= b - \eta \frac{\partial L}{\partial b}, \end{aligned} \tag{1.27}$$

где η – скорость обучения.

Градиентный спуск – итеративный процесс, поэтому выражение (1.27) описывает правило обновления параметров модели на каждой итерации.

Ниже приведен код, реализующий обучение линейной регрессии, с использованием библиотеки PyTorch.

```
def train_loop(X, y, params, n_epochs=10, lr=0.01):
# Функция обучения (градиентный спуск)
# X -- входные данные (предикторы)
# y -- целевые значения
```

```
# params -- параметры модели
# n_epochs -- число эпох обучения
# lr -- параметр скорости обучения
    # Цикл по эпохам
    for epoch in range(n_epochs):
        # Получение текущих предсказаний модели
        y_pred = model(X, *params)
        # Вычисление функции потерь
        loss = loss_fn(y, y_pred)
        # Вывод отладочной информации
        if (epoch%60 == 0):
            print(f'Epoch #{epoch}: loss = {loss:.4f}')

        # Вычисление градиента по параметрам w и b
        loss.backward()
        # Обновление параметров w и b, формула (1.27)
        with torch.no_grad():
            params = params - lr*params.grad

        # Обнуление атрибута grad
        params.requires_grad = True
        if params.grad is not None:
            params.grad.zero_()
    return params

# Определяем скорость обучения
learning_rate = 0.02
# Задаем начальные значения параметров модели
params = torch.tensor([1.0, 0.0], requires_grad=True)

# Вызов процедуры обучения
params = train_loop(X, y, params, n_epochs=600, lr=learning_rate)
print(params)
```

Комментарии подробно объясняют, что происходит в каждом участке кода. В результате вызова этого блока в отладочной консоли должна отразиться следующая информация:

```
Epoch #0: loss = 11.4358
Epoch #60: loss = 3.5310
Epoch #120: loss = 1.6571
Epoch #180: loss = 0.8016
Epoch #240: loss = 0.4110
Epoch #300: loss = 0.2327
Epoch #360: loss = 0.1513
Epoch #420: loss = 0.1142
Epoch #480: loss = 0.0972
Epoch #540: loss = 0.0895
tensor([-0.3385,  6.5482], requires_grad=True)
```

Можно заметить, что значение функции потерь в результате обучения постоянно падает, что означает, что модель постепенно все лучше и лучше описывает обучающий набор.

На рис. 8, а показан обучающий набор, а также вид линейной регрессии в начальный момент обучения. Рядом на рис. 8,б показан вид линейной регрессии после обучения.

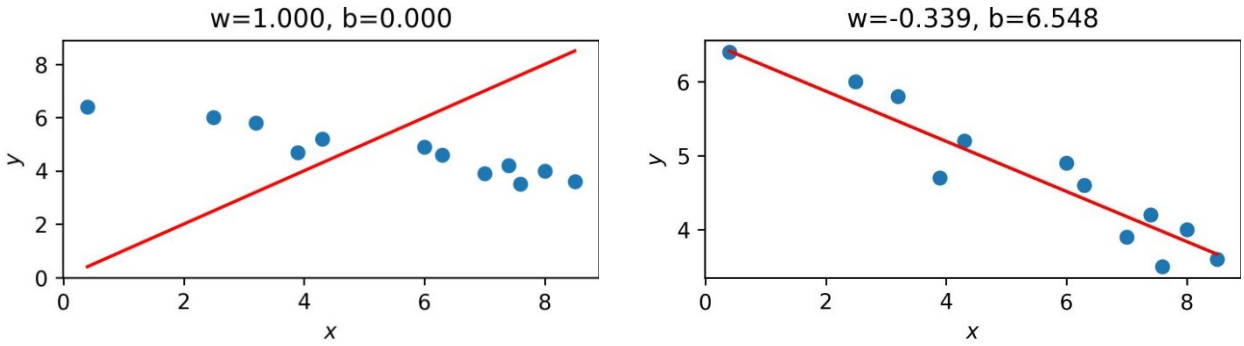

Рисунок 8 – Линейная регрессия: а) начальное приближение, б) после обучения

1.6.4 Автоматизация процесса оптимизации

В предыдущем разделе был использован простой метод градиентного спуска, известный как стохастический градиентный спуск (*SGD – stochastic gradient descent*). Существует большое число способов и приемов, позволяющих улучшить процесс оптимизации параметров моделей. PyTorch представляет возможность абстрагировать процесс оптимизации. Для этого можно воспользоваться модулем `optim`, который содержит классы, реализующие различные алгоритмы оптимизации.

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

Конструкторы всех оптимизаторов получают в качестве первого аргумента список параметры оптимизируемой модели, которые затем изменяются при вызове метода `step` оптимизатора.

Приведем пример кода, показывающий как создать оптимизатор и передать в него параметры модели.

```
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 0.02   # скорость обучения
optimizer = optim.SGD([params],lr=learning_rate)   # создание оптимизатора
```

При создании объекта `optimizer` в качестве второго аргумента мы передали скорость обучения.

В следующем примере показано, как изменить цикл обучения линейной регрессии с использованием встроенного оптимизатора SGD.

```
def train_loop(X, y, params, optimizer, n_epochs=10):
    # Цикл по эпохам
    for epoch in range(n_epochs):
        y_pred = model(X, *params)

        loss = loss_fn(y, y_pred)
        # Вывод отладочной информации
        if (epoch%60 == 0):
            print(f'Epoch #{epoch}: loss = {loss:.4f}')

        # Обнуление атрибута grad параметров модели
        optimizer.zero_grad()
        # Вычисление градиента по параметрам модели
        loss.backward()
        # Обновление параметров модели
        optimizer.step()
    return params


# Задаем начальные значения параметров модели
params = torch.tensor([1.0, 0.0], requires_grad=True)

learning_rate = 0.02   # скорость обучения
optimizer = optim.SGD([params],lr=learning_rate)   # создание оптимизатора

# Вызов процедуры обучения
params = train_loop(X, y, params, optimizer, n_epochs=600)
print(params)
```

В данном коде вы не увидите операторов непосредственно изменяющих значения параметров модели. Обновление коэффициентов выполняется в методе `optimizer.step()`. Вызов метода `optimizer.zero_grad()` необходим, чтобы обнулись градиенты параметров перед вызовом процедуры обратного распространения ошибки. Если запустить приведенный код, то в отладочной консоли будет выведена информация о процессе обучения.

```
Epoch #0: loss = 11.4358
Epoch #60: loss = 3.5310
Epoch #120: loss = 1.6571
Epoch #180: loss = 0.8016
Epoch #240: loss = 0.4110
Epoch #300: loss = 0.2327
Epoch #360: loss = 0.1513
Epoch #420: loss = 0.1142
Epoch #480: loss = 0.0972
```

```
Epoch #540: loss = 0.0895
tensor([-0.3385,  6.5482], requires_grad=True)
```

Полученные результаты в точности повторяют значения, которые были ранее получены при непосредственном описании метода градиентного спуска в предыдущем разделе. Таким образом, мы получили доступ к мощной библиотеки оптитимизаторов `torch.optim`, которую можно использовать для обучения сложных глубоких моделей, содержащих сотни тысяч параметров.

В следующем разделах мы коснемся более сложной задачи и поговорим о том, каким образом построить нейронную сеть, которая способна распознавать рукописные цифры. Однако в начале требуется разобраться в одном дополнительном вопросе, а именно, каким образом в PyTorch хранятся данные и как получить к ним доступ.

1.6.5 Классы dataset и dataloader

Самый известный в области машинного обучения набор данных MNIST содержит изображения рукописных цифр. Этот набор будет использоваться в данной лабораторной работе. Всего в MNIST содержится 70000 размеченных черно-белых изображений размером 28x28 пикселей. MNIST настолько широко распространен, что включен во многие библиотеки языка Python. На рис. 9 показаны первые семь изображений из базы MNIST.

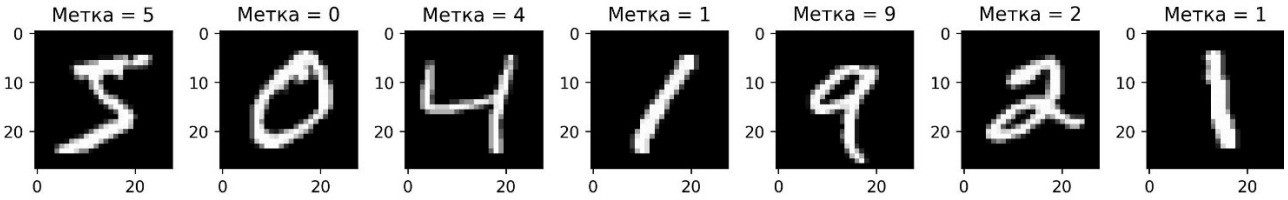

Рисунок 9 – Первые семь изображений тренировочной выборки MNIST

Для загрузки базы MNIST нужно импортировать из библиотеки `torchvision` модули `datasets` и `transforms`.

```
import torch
from torchvision import datasets, transforms

train_set = datasets.MNIST('c:/Datasets/', download=True, train=True)
```

Класс `datasets.MNIST` является производным от класса `torch.utils.data.Dataset`, который представляет из себя специальную абстракцией PyTorch, используемую для хранения набора данных. При создании объекта набора данных первый аргумент (`'c:/Datasets/'`) является путем к папке, откуда необходимо загрузить данные. Параметр `download=True` означает, что в случае, если в папке нет файлов с изображениями базы MNIST, их нужно скачать из сети Интернет (PyTorch сделает это автоматически). Параметр `train=True` означает,

что будет загружен набор с тренировочными данными, если указать `train=False`, то будут загрузится набор с изображениями для теста.

Главной особенностью класса `torch.utils.data.Dataset` является наличие двух методов `__getitem__` и `__len__`. Когда у объекта Python есть метод `__len__`, его можно передать в качестве аргумента встроенной функции `len`. В нашем случае это позволит узнать число изображений в базе. Применим функцию `len` для загруженного набора данных `train_set`:

```
print('Dataset size = ', len(train_set))
>> Dataset size = 60000
```

Для доступа к элементам набора данных можно использовать обычную индексацию:

```
img, label = train_set[0]
```

В результате такого обращения вернется кортеж, состоящей из первого изображения из базы и его метки (т.е. числа, которое изображено на изображении).

По умолчанию загружаемые изображения имеют тип `PIL Image`, однако для работы с PyTorch требуется преобразовать их в тензоры. Для этого воспользуемся классом-функцией `PILToTensor()` из модуля `transforms`:

```
transform = transforms.PILToTensor()
img_tensor = transform(img)
```

После выполнения данных действий в `img_tensor` будет записан тензор, который будет иметь размер 1x28x28. Удобство PyTorch состоит в том, что при создании объекта `Dataset` мы можем передать в его конструктор функцию, которая будет применятся к данными перед тем, как они будут возвращены методом `__getitem__`. В следующем примере показано, как переделать код для загрузки MNIST, что бы возвращаемые изображения имели тип `Tensor`.

```
transform_mnist = transforms.Compose([ transforms.PILToTensor() ])
train_set = datasets.MNIST('c:/Datasets/', download=False, train=True,
transform=transform_mnist)

img, label = train_set[25]

print('img type = ', img.dtype)
print('img shape = ', img.shape)

>> img type =  torch.uint8
>> img shape =  torch.Size([1, 28, 28])
```

Класс `transforms.Compose` позволяет создать композицию из функций преобразующих данные. В данном случае, композиция состояла только из одного класса-функции `transforms.PILToTensor()`. Рекомендованной практикой при обучении нейронных сетей является выполненные операции нормализации данных. Используя механизм составления композиции функций нормализацию данных, можно выполнить следующим образом.

```
transform_mnist = transforms.Compose([
      transforms.ToTensor(),
      transforms.Normalize(mean=0,std=1)
      ])
norm_train_set    =    datasets.MNIST('c:/Datasets/',    download=False,
train=True, transform=transform_mnist)
```

В данном случае строится композиция из двух функций `ToTensor()`, которая переводит изображение в тензор и преобразует его к типу `torch.float32`. Второе преобразование – `transforms.Normalize(mean=0,std=1)`, обеспечивает данным нулевое среднее и единичное стандартное отклонение. На рис. 10 показан пример изображения, загруженного из набора данных `train_set` и `norm_train_set`.

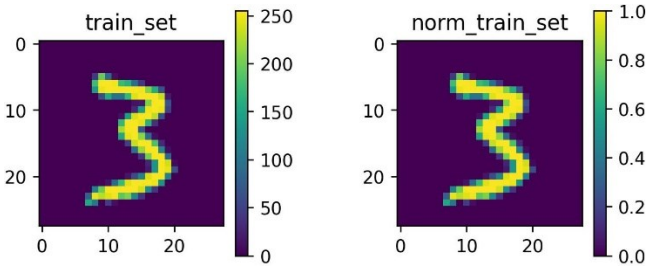

Рисунок 10 – Изображения из базы MNIST (до и после нормализации)

Внешне изображения не изменились, однако обратите внимание на диапазон значений, показанных на цветовой карте, он изменился.

Осталось сказать про ещё один важный класс, который необходим для организации процесса обучения нейронной сети (или любой другой модели). Речь идет о классе `DataLoader` из модуля `torch.utils.data`. Он используется для того, чтобы перебирать набора данных не по одному изображению, и не все сразу, а небольшими порциями, которые называются ***мини-батчами***. Необходимость этого связана в первую очередь с особенностями метода градиентного спуска – SGD. Оказывается, что этот метод лучше всего работает, если градиент в нем рассчитывается по небольшой выборке (мини-батчу) из набора данных. В этом случае получающийся градиент является аппроксимацией того градиента, который мог бы быть получен, если бы вычисление выполнялось по всему набору данных.

Приводимый далее пример показывает, как создать загрузчик для набора данных MNIST.

```
train_loader = torch.utils.data.DataLoader(norm_train_set, batch_size=64,
shuffle=True)
```

Первым параметром передается объект класса DataSet, второй параметр – размер мини-батча, т.е. одна «порция» данных в процессе обучения. Значение `shuffle=True` говорит о том, что данные нужно перетасовывать в начале каждой эпохи.

Можно проверить, что какие «порции» данных будет возвращать `train_loader` в процессе обучения.

```
batch_imgs, batch_labels = next(iter(train_loader))
print('Size of batch_imgs = ', batch_imgs.shape)
>> Size of batch_imgs =  torch.Size([64, 1, 28, 28])
```

Тензор `batch_imgs` имеет размерность $B \times C \times H \times W$, где B – размер мини-батча, C – число каналов, а H и W это высота и ширина, соответственно.

Чтобы подавать изображения, которые в нашем случае имеют размерность $1 \times 28 \times 28$ нужно преобразовать их к вектору размерности 784×1. Это можно сделать, если «вытянуть» изображение в один длинный столбец. В PyTorch для выполнения этой операции можно использовать метод `view`. Чтобы преобразовать все изображения из одного мини-батча можно поступить следующим образом:

```
batch_size = batch_imgs.shape[0]
batch_imgs = batch_imgs.view(batch_size, -1)
print('Size of batch_imgs = ', batch_imgs.shape)
>> Size of batch_imgs = torch.Size([64, 784])
```

Здесь в методе `view` указывается, что нулевая размерность, которая определяет число изображений в мини-батче должна оставаться прежней, а значение `-1` , которое относится к первой размерности должна вычисляться автоматически на основании исходного тензора `batch_imgs`.

Мы освоили уже много материала, научились оптимизировать модели, загружать данные и манипулировать ими. Теперь мы готовы, чтобы перейти к самому интересному – созданию и обучению нейронной сети.

1.6.6 Создание нейронная сеть для распознавания рукописных цифр

Наша сеть не будет слишком сложной – это будет однослойный персептрон (рисунок 6) с выходной функцией активации – softmax.

В PyTorch существует несколько способов описать нейронную сеть. Однако, наиболее распространенный – это создание собственного подкласса `nn.Module`. В минимальной «комплектации» при создании подкласса nn.Module

нужно описать функцию прямого прохода – `forward`, которая будет принимать входные данные и формировать выход модели.

Обычно описываемая модель должна использовать и другие подмодули (*submodules*), такие как свертки, полносвязные слои и т.д. Они, как правило, описываются в конструкторе `__init__` и присваиваются `self`, чтобы их в дальнейшем использовать в функции `forward`.

Ниже показано, как описать нейронную сеть в виде подмодуля.

```
from torch import nn

class simple_NN(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(784, 10)
        self.act = nn.Softmax()

    def forward(self,x):
        out = self.fc(x)
        out = self.act(out)

        return out
```

В строке `self.fc = nn.Linear(784, 10)` создается линейный слой, который преобразует входной вектор с размерностью 784 в выходной вектор размерностью 10. Работа этого слоя описывается следующим образом:

$$\mathbf{y}_{10\times1} = \mathbf{W}_{10\times784} \cdot \mathbf{x}_{784\times1} + \mathbf{b}_{784\times1}, \tag{1.28}$$

где $\mathbf{W}_{10\times784}$ и $\mathbf{b}_{784\times1}$ – обучаемые параметры слоя (модели).

Далее в строке `self.act = nn.Softmax()` создается слой, реализующий вычисление активационной функции softmax, которая описывается выражением (1.21).

В методе forward описывается, как должно преобразовывать выходное изображение $\mathbf{x}$ при прохождении через нейронную сеть.

В качестве теста подадим на нашу новенькую модель изображение из базы MNIST.

```
img_norm, _ = norm_train_set[0]
x = img_norm.view(1,-1)  # вытягивает изображение 28x28 в строку 1x784
model = simple_NN()  # создает экземпляр класса
y = model(x)         # подача изображения на вход нейросети
print(y)             # вывод результата
>> tensor([[0.089, 0.111, 0.085, 0.129, 0.099, 0.113, 0.112, 0.103, 0.074,
0.084]], grad_fn=<SoftmaxBackward0>)
```

В результате прогона изображения через необученную нейронную сеть мы получили вектор размерности 1×10. Пока что полученные значения не имеют никакого смысла, однако мы убедились, что все работает именно так, как и было задумано.

1.6.7 Обучение нейронной сети

До получения результата остается последний шаг – нужно разработать цикл, который будет осуществлять обучение разработанной нами нейронной сети. Что-то похожее мы уже ранее делали для линейной регрессии. Теперь нужно повторить те же действия, однако учесть специфику, как наших данных, так и новой модели.

Ниже приведен код, в котором реализована процедура обучения разработанной нами модели нейронной сети.

```
model = simple_NN() # создание модели

optimizer = optim.SGD(model.parameters(),lr=0.003) # создание оптимизатора

loss_fn = nn.CrossEntropyLoss()  # функция потерь

n_epochs = 100 # число эпох обучения
# цикл по эпохам
for epoch in range(n_epochs):
    total_loss = 0
    for X, y_true in train_loader:
        X = X.view(X.shape[0],-1)
        y_pred = model(X)

        loss = loss_fn(y_pred, y_true) # вычисление функции потерь

        # Обнуление атрибута grad параметров модели
        optimizer.zero_grad()
        # Вычисление градиента по параметрам модели
        loss.backward()
        # Обновление параметров модели
        optimizer.step()

        total_loss += loss.item()

    # Вывод отладочной информации
    if (epoch%10 == 0):
        print(f'Epoch #{epoch}: loss = {total_loss:.4f}')
```

Поясним некоторые моменты, которые могут вызвать затруднение. Обратите внимание, что в подклассе `nn.Module` реализован метод `parameters()`, кото-

рый возвращает список всех тензоров модели, имеющих значение атрибута `requires_grad=True`. По сути, это все настраиваемые параметры модели нейронной сети. Именно по этой причине мы обращаемся к этому методу, когда создаем оптимизатор для нашей модели:

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

Далее в программе мы создаем функцию потерь (`loss_fn = nn.CrossEntropyLoss()`), которой в данном случае является перекрестная энтропия, вычисляемая по формуле (1.23).

Внутри цикла по эпохам мы используем загрузчик train_loader, который мы создали ранее:

```
for X, y_true in train_loader:
```

На каждой итерации в переменные `X, y_true` будут помещаться очередные «порции» данных из базы MNIST. Далее в строке

```
X = X.view(X.shape[0],-1)
```

Происходит вытягивание изображений размера 28x28 в длинный вектор 1x784, чтобы его можно было подать на вход модели. Остальные шаги программы имеют тот же смысл, что и в случае обучения линейной регрессии. Отдельно скажем про переменную `total_loss`, она нужна для того, чтобы рассчитать суммарное значение ошибки обучения на всем наборе данных. Отслеживание значений `total_loss` от эпохи к эпохи позволяет нам видеть, как проходит процесс обучения. В частности, по мере увеличения номера эпохи суммарное значение функции потерь должно падать.

После завершения процесса обучения в консоль будет выведена следующая информация:

```
Epoch #0:  loss = 2141.30
Epoch #10: loss = 1701.75
Epoch #20: loss = 1626.61
Epoch #30: loss = 1586.71
Epoch #40: loss = 1555.50
Epoch #50: loss = 1539.09
Epoch #60: loss = 1528.52
Epoch #70: loss = 1520.83
Epoch #80: loss = 1514.91
Epoch #90: loss = 1510.14
```

Можно заметить, что суммарное значение функции потерь падает, что говорит о том, что сеть с каждым разом делает предсказание все лучше.

1.6.8 Тестирование нейронной сети

После завершения процесса обучения нейронной сети самое время узнать, насколько хорошо она способна предсказывать рукописные цифры на изображениях, которые она не «видела» в процессе обучения. В следующем примере показано, как загрузить тестовый набор изображений, создать на его основе `DataLoader` и вычислить точность предсказаний модели.

```
transform_mnist = transforms.Compose([ transforms.ToTensor(), transforms.Normalize(mean=0,std=1)])
norm_test_set    =    datasets.MNIST('c:/Datasets/',    download=False, train=False, transform=transform_mnist)

test_loader         =         torch.utils.data.DataLoader(norm_train_set, batch_size=len(norm_test_set)//10, shuffle=True)

correct = 0
total = 0

model.eval()
for X, y_true in test_loader:
    X = X.view(X.shape[0],-1)
    outputs = model(X)
    _,y_pred = torch.max(outputs, dim=1)
    total +=y_true.shape[0]
    correct +=int((y_pred==y_true).sum())

print(f"Accuracy = {correct/total: .3f}")
```

В результате выполнения кода был получен следующий результат:

```
Accuracy =  0.894
```

Очень неплохо, особенно если учесть, что на вход сети подавались изображения, которые не участвовали в процессе обучения. Разумеется, мы сделали лишь самый первый шаг в мир глубокого обучения, однако те знания, которые вы получили изучая данный материал можно использовать для построения гораздо более сложных и интересных моделей.

2 Порядок выполнения работы

1) Постройте классификатор на основе однослойной нейронной сети, согласно варианту (см. таблицу 1). В качестве функции потерь используйте.

Таблица 1 – Варианты заданий

Номер варианта	Функция классификатора	Функция потерь
1	Классификатор должен распознавать следующие 5 классов: – Класс 1 (цифры 0 и 6) – Класс 2 (цифры 1 и 7) – Класс 3 (цифры 2 и 3) – Класс 4 (цифры 4 и 9) – Класс 5 (цифры 5 и 8)	MSE – Среднеквадратичная ошибка – (1.13)
2	Классификатор должен распознавать следующие 2 класса: – Класс 1 (цифры 0, 2, 4, 6, 8) – Класс 2 (цифры 1, 3, 5, 7, 9)	MSE – Среднеквадратичная ошибка – (1.13)
3	Классификатор должен распознавать следующие 3 класса: – Класс 1 (цифры 0) – Класс 2 (цифры 6) – Класс 3 (все оставшиеся цифры)	CE – перекрестная энтропия – (1.23).
4	Классификатор должен распознавать следующие 4 класса: – Класс 1 (цифры 1) – Класс 2 (цифры 3) – Класс 3 (2, 4, 7, 9) – Класс 4 (0, 5, 6, 8)	CE – перекрестная энтропия – (1.23).
5	Классификатор должен распознавать следующие 2 класса: – Класс 1 (цифры 0, 1, 2, 3, 4) – Класс 2 (цифры 5, 6, 7, 8, 9)	MSE – Среднеквадратичная ошибка – (1.13)
6	Классификатор должен распознавать следующие 3 класса: – Класс 1 (цифры 2) – Класс 2 (цифры 3) – Класс 3 (все оставшиеся цифры)	MSE – Среднеквадратичная ошибка – (1.13)
7	Классификатор должен распознавать следующие 3 класса:	CE – перекрестная энтропия – (1.23).

Номер варианта	Функция классификатора	Функция потерь
	– Класс 1 (цифра 0, 9) – Класс 2 (цифра 1, 7) – Класс 3 (все остальные цифры)	
8	Классификатор должен распознавать следующие 3 класса: – Класс 1 (цифры 1, 2, 7, 9) – Класс 2 (цифры 3, 5,8) – Класс 3 (цифры 0, 4, 6)	CE – перекрестная энтропия – (1.23).

2) Для первого и для второго задания постройте кривые, показывающие изменение ошибки обучения на тренировочном на тестовом наборе (по аналогии, как это сделано на рис. 1).

3) Выполните задания в соответствии с таблицей 2

Таблица 1 – Варианты заданий

Номер варианта	Задание
1, 3, 4, 6–8	Опишите модель двухслойной нейронной сети согласно вашему варианту. Число нейронов внутреннего слоя определите самостоятельно.
2, 5,	Измените вашу модель, используя в выходном слое вместо функции softmax логистическую функцию.

Оценить, как изменилась точность распознавания модели после модификации?

4)* Для классификаторов из заданий 1 и 3 постройте матрицы спутывания (*confusion matrix*).

5) Оформить отчет в соответствии с СТП 01-2017.

3 Дополнительные задания

1) Продемонстрируйте как выглядят первые 6 изображений из тренировочной выборки на внутреннем слое нейронной сети.

2) Разработайте функцию, которая на основе матрицы спутывания рассчитывает правильность классификатора.

3) Визуализируйте вид изображения на внутри нейронной сети.

* Дополнительное задание, необязательное

4) Разработайте функцию Python выполняющую ту же действие, что и `keras.utils.to_categorical()`.

4 Литература

1. Николенко, С., Кадурин, А., Архангельская, Е. *Глубокое обучение* – СПб.: Питер, 2019. – 480 с.

2. Стивенс Э., Антига Л., Виман Т. PyTorch. Освещая глубокое обучение. – СПб.: Питер, 2022. – 576 с.