KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 2 урок > Построение нейросети и методы оптимизации

Содержание

Дополнительные материалы

Роль нелинейности в нейронных сетях

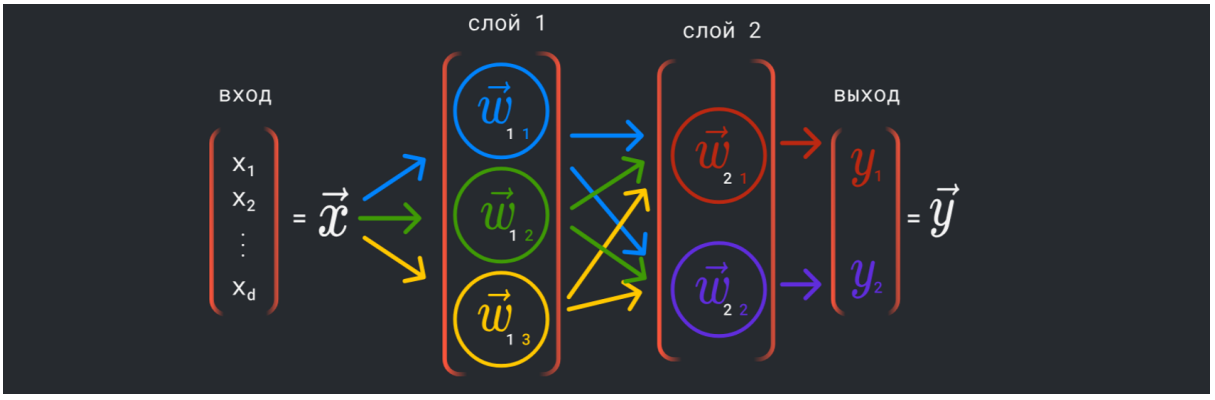
Начнем с того, что вспомним, что такое линейный слой. Он представляет собой умножение входных данных на матрицу весов: $Y = \vec{W} \cdot \vec{X}$

Далее к выходу слоя мы применяем некоторую нелинейную функцию $\sigma(Y)$. Эту функцию мы называем **нелинейностью**.

Возникает вопрос: действительно ли она нам нужна? Давайте рассмотрим это с математической точки зрения.

Пусть у нас есть некоторая нейросеть, которая принимает на вход некоторый двумерный вектор X и состоит из двух слоев:

- Первый слой состоит из трех нейронов (синий, зеленый и желтый)
- Второй слой состоит из двух нейронов(красный и фиолетовый)



Вектор X приходит на первый слой, к нейрону 1, нейрону 2, нейрону 3.

Таким образом, на выходе слоя 1 у вас будет трехмерный вектор.
Далее этот трехмерный вектор будет уходить во второй слой.
Поскольку во втором слое два нейрона, на выходе будет два числа, то есть двумерный вектор.

$$y_1 = \begin{bmatrix} \vec{w}_{11} \cdot \vec{x} \\ \vec{w}_{12} \cdot \vec{x} \\ \vec{w}_{13} \cdot \vec{x} \end{bmatrix} \cdot \vec{w}_{21} = \begin{bmatrix} w_{11}^{(1)} x_1 + w_{11}^{(2)} x_2 \\ w_{12}^{(1)} x_1 + w_{12}^{(2)} x_2 \\ w_{13}^{(1)} x_1 + w_{13}^{(2)} x_2 \end{bmatrix} \cdot \begin{bmatrix} \vec{w}_{21}^{(1)} \\ \vec{w}_{21}^{(2)} \\ \vec{w}_{21}^{(3)} \end{bmatrix} = \underbrace{(\vec{w}_{21}^{(1)} w_{11}^{(1)} + \vec{w}_{21}^{(2)} w_{12}^{(1)} + \vec{w}_{21}^{(3)} w_{13}^{(1)})}_{\vec{w}'_1} x_1 + \underbrace{(\dots)}_{\vec{w}'_2} x_2$$

Каждый нейрон в слое 1 представляет собой двумерный вектор. Первая компонента этого вектора соответствует весу, связанному с первым входом, а вторая компонента — весу, связанному со вторым входом.

Аналогичные рассуждения применимы и к другим нейронам слоя 1. Таким образом, результаты скалярного произведения для каждого нейрона первого слоя представляют собой линейные комбинации входных данных с весами соответствующих нейронов, а результаты для нейронов второго слоя — линейные комбинации выходов нейронов первого слоя.

После раскрытия скобок и группировки слагаемых мы замечаем интересное свойство: результат выглядит так, будто у нас **изначально** был только один слой нейронов с двумя весами.

То есть как будто бы мы с самого начала взяли x_1, умножили на некоторый вес ($\vec{w_1}$'), взяли x_2 , умножили на другой вес ($\vec{w_2}$') и результаты сложили.

Это подтверждает, что добавление сколь угодно большого количества дополнительных слоев не улучшает выразительные возможности модели, так как мы остаемся в пределах линейной зависимости. Однако, использование нелинейных функций активации после каждого слоя поможет избежать такой линейной зависимости и значительно расширит способности модели в выражении сложных закономерностей в данных.

Функции активации

Для того, чтобы наша нейросеть могла обучаться не только линейным зависимостям, но и более сложным закономерностям, а также для того, чтобы добавление новых слоев приносило пользу, мы используем нелинейные функции активации.

Эти функции, обозначаемые символом σ, играют важную роль в нейронных сетях, так как вносят нелинейность в процесс вычислений.

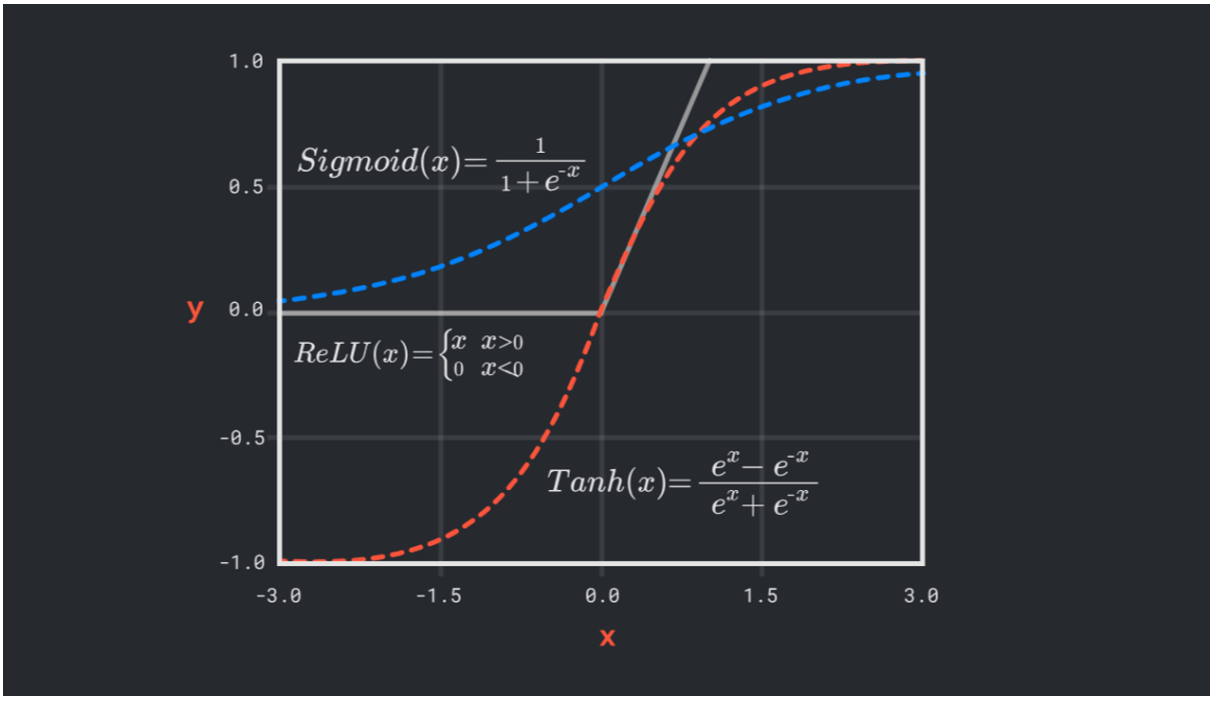
На практике функция σ обычно представляет собой какую-то нелинейную операцию, такую как возведение в квадрат, синус, логарифм и т. д. Однако при выборе функции активации важно учитывать несколько аспектов:

- Во-первых, нелинейность не должна слишком сильно увеличивать выходные значения слоя. Это важно, чтобы избежать проблемы с переполнением и большими весами, что может привести к переобучению модели.
- Во-вторых, нелинейность не должна слишком сильно уменьшать выходные значения, приближая их к нулю. Это может привести к затуханию градиентов и потере информации при прохождении через несколько слоев сети.

- В-третьих, функция активации должна быть дифференцируемой, чтобы обеспечить корректное обновление весов в процессе обучения с помощью градиентного спуска.

Среди самых часто используемых в нейросетях функций активации можно выделить следующие:

1. Сигмоида
2. Гиперболический тангенс
3. ReLU (Rectified Linear Unit)



Популярные функции активации

Остановимся на них подробнее.

Сигмоида (Sigmoid)

Сигмоида — это нелинейная функция, являющаяся самой распространенной функцией активации. Её формула:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Где:

- $\sigma(x)$ — значение сигмоиды для входа x
- e — основание натурального логарифма
- x — входное значение

Сигмоида возвращает значения в диапазоне от 0 до 1 и обладает важным свойством **интерпретируемости** вывода как **вероятности**. Часто используется на выходе последнего слоя нейросети в задачах классификации.

Преимущества	**Недостатки**
Простая интерпретация. Значения сигмоиды лежат в диапазоне от 0 до 1, что удобно интерпретировать как вероятности.	Проблема затухания градиентов. При глубоких сетях градиенты могут затухать, что затрудняет обучение модели.
Гладкая производная. Производная сигмоиды легко вычисляется и гладкая по всему диапазону.	Нецентрированность. Сигмоида не является центрированной относительно нуля, что может приводить к затуханию градиентов на больших отрицательных и положительных значениях.
Отсутствие "взрывающихся градиентов": При обратном распространении ошибки градиенты могут быть управляемыми, избегая проблем с "взрывающимися градиентами".	

Гиперболический тангенс (Tanh)

Гиперболический тангенс — это одна из гиперболических функций, которая также широко используется в нейронных сетях. Его формула:

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

Гиперболический тангенс изменяется в диапазоне от -1 до 1, что делает его подходящим для различных задач.

Преимущества	**Недостатки**
Центрированность. Гиперболический тангенс центрирован относительно нуля, что может уменьшить проблему затухания градиентов.	Проблема затухания градиентов. При глубоких сетях все еще может возникнуть проблема затухания градиентов.

Преимущества	Недостатки
Нелинейность. Тангенс обладает нелинейными свойствами, позволяя модели аппроксимировать сложные функции.	Экспоненциальные вычисления. Использование экспоненциальной функции в формуле может быть вычислительно затратным.

ReLU (Rectified Linear Unit)

ReLU — это функция активации, которая стала очень популярной в современных нейронных сетях. Её формула:

$$ReLU(x) = \max(0, x)$$

Где:

- ReLU(x) — значение функции ReLU для входа x
- x — входное значение

ReLU оставляет положительные значения без изменений и зануляет отрицательные, делая функцию нелинейной.

Преимущества	Недостатки
Высокая скорость обучения. Работает быстрее, чем сигмоида и тангенс, так как активируется только для положительных значений.	Проблема «мёртвых нейронов»: Для отрицательных значений ReLU всегда выдаёт ноль, что может привести к «мёртвым нейронам», когда они перестают обновляться.
Решение проблемы затухания градиентов. Избегает проблемы затухания градиентов для положительных значений, что облегчает обучение глубоких сетей.	Нецентрированность. ReLU также не является центрированной, что может вызывать проблемы при обучении.
Простота. Простота вычисления и отсутствие вычислительно затратных операций, таких как экспоненциальные функции.	

Полносвязная сеть

В предыдущих уроках мы обсуждали необходимость использования нелинейных функций в нейронных сетях. Теперь давайте объединим это с нашим пониманием линейных моделей и линейных слоев.

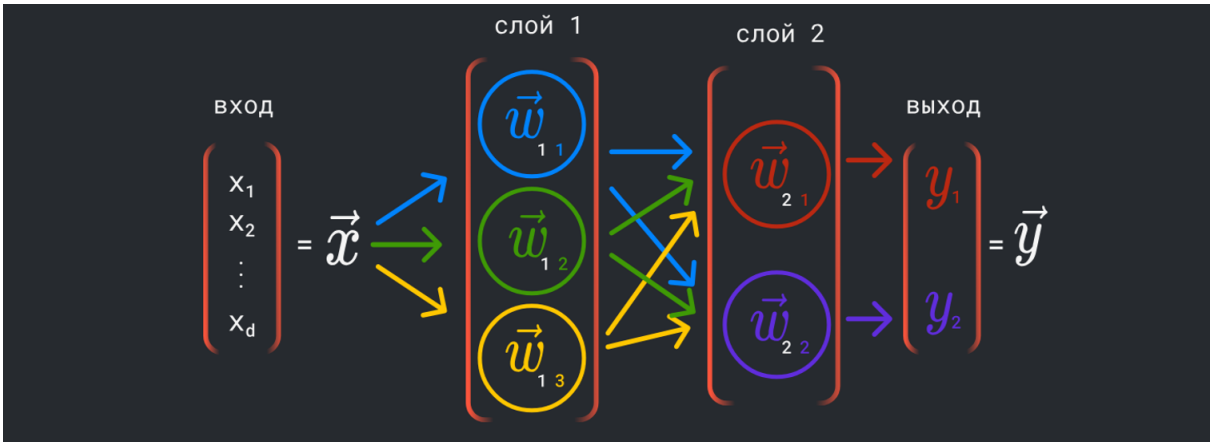
Fully connected слой или полносвязный слой состоит из линейного слоя и нелинейности. Линейный слой представляет собой умножение входных данных на матрицу весов. Формально это можно записать следующим образом: $FC(X) = \sigma(XW)$.

Fully connected слой часто называют полносвязной сетью, так как все нейроны в одном слое связаны со всеми нейронами в следующем слое.

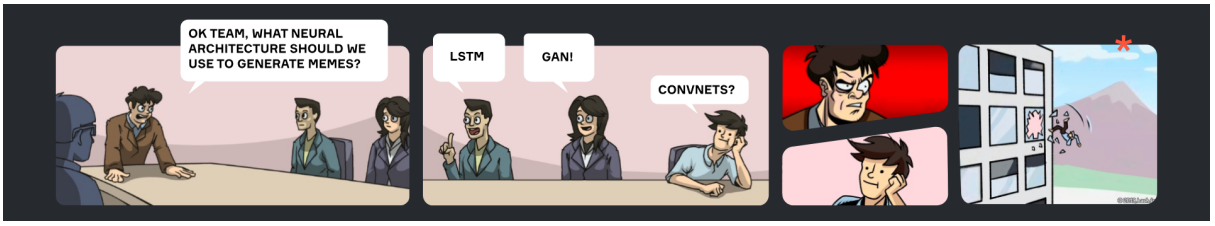
Для активации функций в нейронных сетях обязательно должна присутствовать нелинейность. Это означает, что после каждого линейного слоя должна быть применена нелинейная функция активации, которая вводит нелинейность в модель.

Архитектура нейронной сети состоит из слоев, которые могут быть линейными с активациями или более продвинутыми слоями, которые мы изучим позже. Эти слои могут быть комбинированы между собой, и если правильно применять нелинейности, модель сможет изучать сложные закономерности.

Если представить, что слои — это куски хлеба, то их можно складывать друг на друга, как в бутерброде. Этот процесс называется **стэкингом** или сложением. Именно такой процедурой мы можем создавать сети любой сложности.



Как выбрать слои для нашей модели? Это зависит от конкретной задачи. Есть много различных типов слоев, которые могут быть объединены в блоки. Например, блок может содержать 4-5 последовательных слоев. В каждой задаче используется свой набор слоев, который оптимален для данной задачи. Нет универсального рецепта, и выбор слоев зависит от логики их работы и требований задачи.



Источник картинки: https://www.meme-arsenal.com/create/chose

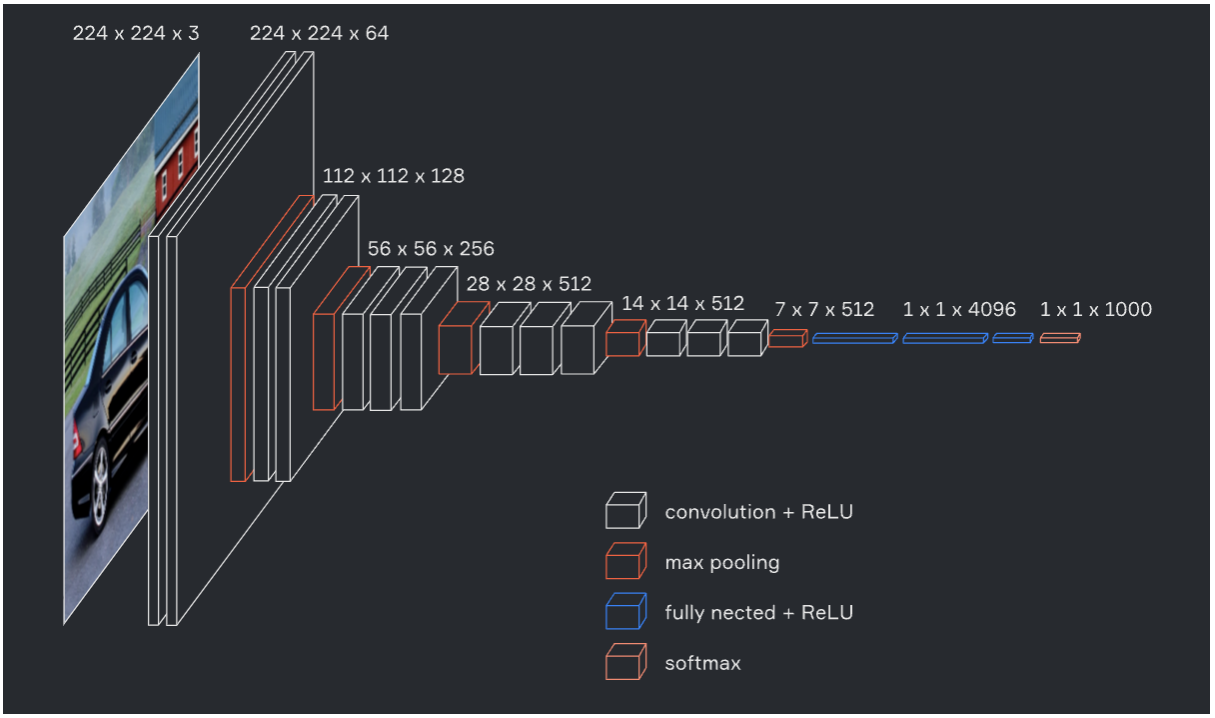
Архитектура нейронной сети

Давайте разберемся, что такое архитектура нейронной сети и как ее визуализировать. Архитектура представляет собой **описание того, как уровни (слои) нейронной сети связаны между собой**. Они могут быть сложными или простыми, но чаще всего они довольно сложные. Поэтому, чтобы лучше понять структуру, ее часто пытаются изобразить. Однако нет какого-то стандартного метода для этого, каждый может рисовать архитектуру так, как ему удобно.

Визуализация архитектуры нейронной сети может выглядеть по-разному. Например, для изображений это могут быть квадратики, переходящие друг в друга, или стандартные схемы с блоками и стрелочками, иногда с вложенными текстовыми описаниями, которые не всегда понятны до конца и требуют дополнительного объяснения в тексте статьи.

Когда вы работаете с архитектурой нейронной сети, важно также обратить внимание на то, как соотносятся размерности тензоров между слоями. Например, один слой может уменьшить изображение в два раза, но увеличить количество каналов в три раза.

Давайте рассмотрим пример архитектуры нейронной сети под названием VGG, которая использовалась для анализа изображений и решала задачу классификации на тысячу классов. Эта сеть принимала изображение размером 224 на 224 пикселя с тремя каналами. Затем проходила через несколько слоев, уменьшая размерность и изменяя количество каналов.



Архитектура сети VGG (Источник картинки: https://questu.ru/articles/439897/)

Визуально это можно представить как несколько прямоугольников, расположенных друг за другом. Внутри этих прямоугольников могут быть разные слои: например, черный прямоугольник представляет собой сверточный слой с функцией активации ReLU, красный — другой слой, синий — полносвязанный слой, а затем идет слой Softmax.

Softmax — это функция активации, которая преобразует вектор чисел в вектор вероятностей, где каждый элемент на выходе представляет собой вероятность принадлежности к определенному классу. Он принимает этот вектор и масштабирует его значения так, чтобы они лежали в интервале от 0 до 1, и чтобы их сумма составляла 1.

Если вспомнить сигмоиду, которая ограничивала значения от 0 до 1, то Softmax делает нечто похожее, но для вектора из тысячи чисел. Таким образом, значения вектора интерпретируются как вероятности, где более высокие значения соответствуют более высокой уверенности в принадлежности к классу.

Если, например, сеть выдает очень большие числа для одного класса и очень маленькие для остальных, то Softmax назначит высокую вероятность именно этому классу.

Промежуточные итоги

- Мы узнали, что нелинейности позволяют нам учить сложные зависимости с помощью линейных слоев
- Мы разобрали, что полносвязная сеть состоит из линейного слоя, за которым следует нелинейная функция активации
- Затем мы увидели, что нейронные сети в PyTorch* представляют собой комбинацию слоев,
  и то, как эти слои взаимодействуют, определяет их архитектуру
- Архитектуру часто изображают на диаграммах, где слои связаны друг с другом.
  Нет универсальной архитектуры; ее выбор зависит от конкретной задачи и области знаний

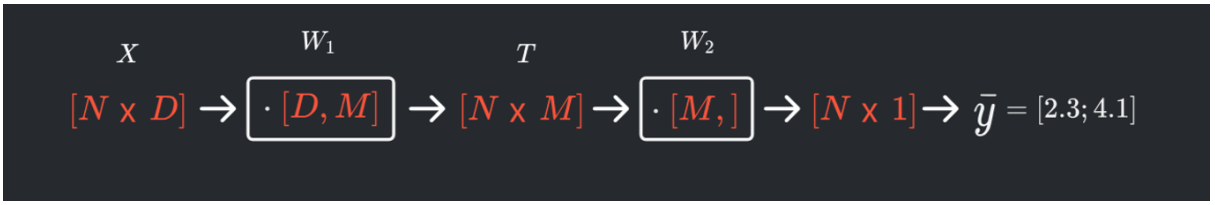
Backpropagation. Математический аспект.

Backpropagation (обратное распространение) — это фундаментальный метод обучения нейронных сетей. Этот метод позволяет нам эффективно обновлять веса модели, минимизируя функцию потерь, и тем самым улучшать производительность сети.
Начнем с рассмотрения основных концепций, которые лежат в основе backpropagation.

Пусть у нас есть некоторая матрица X, которая представляет собой обучающую выборку. Предположим, что эта матрица имеет N примеров, где каждый пример имеет размерность D. Это можно представить как обычный датасет в машинном обучении, где у нас есть N записей, каждая из которых содержит D признаков.

Предположим также, что мы хотим пропустить эту выборку через двухслойную нейронную сеть. Первый слой должен иметь размерность d, чтобы мы могли перемножить матрицы X и W_1, где W_1 — матрица весов первого слоя.
Размерность второго слоя может быть любой; давайте обозначим ее как M, что представляет собой размерность промежуточного представления.

Когда мы перемножаем матрицы X и W_1, мы получаем матрицу размера $N \times M$. Затем мы умножаем это промежуточное представление на матрицу W_2, которая на самом деле будет вектором. Это означает, что размерность M исчезает, и у нас остается вектор размера $N \times 1$, в котором для каждого объекта обучающей выборки предсказывается одно число.



Например, это может быть ситуация, когда у нас есть фичи пользователя, и мы хотим предсказать, сколько денег мы заработаем с этого пользователя в ближайший месяц, основываясь на различных признаках пользователя, таких как его активность на платформе, история покупок и т.д.

Для обучения нейронной сети с использованием метода обратного распространения ошибки (backpropagation) необходимо определить функцию потерь, которую мы стремимся минимизировать. В данном случае мы выберем среднеквадратичное отклонение (MSE) в качестве функции потерь.

Функция потерь L для нашего примера может быть записана следующим образом:

$$L = \sum_{i=1}^{n} (\hat{Y}_i - Y_i)^2 = \sum_{i=1}^{n} (T_i W_i - Y_i)^2,$$

где Y_i — это истинное значение для i-го примера, а $\hat{Y}_i$ — это предсказанное значение для i-го примера.

- Мы начнем с формулы для производной L по W_2:

$$\frac{\partial L}{\partial W_2} = \sum 2(T_j W_2 - Y_j) T_i$$

Для того, чтобы найти производную функции потерь по W_1 понадобится производная по промежуточному выходу T размерности $N \times M$:

$$\text{chain rule: } \frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial T} \frac{\partial T}{\partial W_1}$$

Его можно рассчитать аналогично:

$\frac{\partial L}{\partial T_j} = 2(T_j W_2 - Y_i)W_2,$

где T_j — вектор размерности $[M,]$ (j — индекс по первой оси)

$\frac{\partial L}{\partial T} = [\frac{\partial L}{\partial T_1}, \ldots, \frac{\partial L}{\partial T_N}]$

- Строгий подсчет $\frac{\partial T}{\partial W_1}$ довольно долгий и ресурсоемкий, поэтому пойдем другим путем:

Так как производная имеет такую же размерность как параметр,

$\frac{\partial L}{\partial W_1}$ имеет ту же размерность, что $W_1 W_1 [D, M]$

$\frac{\partial L}{\partial T}$ имеет ту же размерность, что T — $[N, M]$

Также мы знаем, T представляет собой матричное произведение входного слоя на веса:

$T = X \cdot W_1$, где X — матрица входных данных (размерности $n \times d$), W_1 — матрица весов первого слоя (размерности $d \times m$).

Поскольку мы хотим вычислить производную T по W_1, которая не зависит от X, то X условно можно считать константой (по аналогии с производной $(kx)' = k$, где k — константа).

Таким образом, производная T по W_1 равна транспонированной матрице X, обозначаемой как X^T.

То есть второй член нашего выражения для производной L по W_1 теперь выглядит следующим образом: $\frac{\partial T}{\partial W_1} = X^T$

- Теперь мы можем использовать это выражение, а также ранее вычисленный градиент $\frac{\partial L}{\partial T}$, чтобы получить полный градиент $\frac{\partial L}{\partial W_1}$ и обновить веса W_1 в процессе обучения:

$\frac{\partial L}{\partial W_1} = \frac{\partial T}{\partial W_1} \frac{\partial L}{\partial T} = X^T \frac{\partial L}{\partial T}$

Таким образом мы можем научить каждый слой считать производную функции потерь по своему входу, зная производную по своему выходу, что позволит пробрасывать градиент от конца сети в начало.

А что с нелинейностью?

В предыдущем примере мы имели последовательность линейного слоя, за которым следовал другой линейный слой, а затем — ответ. **Теперь мы просто добавляем между этими двумя линейными слоями слой с нелинейностью**.

Как это влияет на процесс обучения? Нам нужно уметь вычислять градиенты для этого слоя с нелинейностью. Поскольку мы можем записать формулу для градиента по выходам этого слоя, мы также можем выразить градиент по его входам. Мы можем сделать это, используя математику, ведь у нас есть конкретная формула для нашей нелинейности. Мы просто вычисляем производные.

Итак, мы научили эту нелинейность «пробрасывать» градиенты дальше. Когда приходит время для обратного прохода, градиент по выходам этой нелинейности передается в обратном направлении. Таким образом, нелинейность пересчитывает градиент по своим входам, используя тот же механизм, что и для линейных слоев.

Следующий линейный слой в этой последовательности уже не знает, была ли между ним и предыдущим линейным слоем нелинейность. Все, что он получает, это градиент потерь по своему выходу. И вот здесь вся красота алгоритма: **слой не обращает внимания на историю применения нелинейности или ее отсутствие. Он просто получает градиенты и обновляет свои параметры в соответствии с ними, передавая градиент дальше.**

Таким образом, добавление нелинейности в сеть не сильно изменяет процесс обучения. Просто добавляется еще один слой, который нужно аккуратно обработать во время обратного прохода. Это соответствует идее Pytorch* о том, что нелинейность можно рассматривать как добавление еще одного слоя к сети.

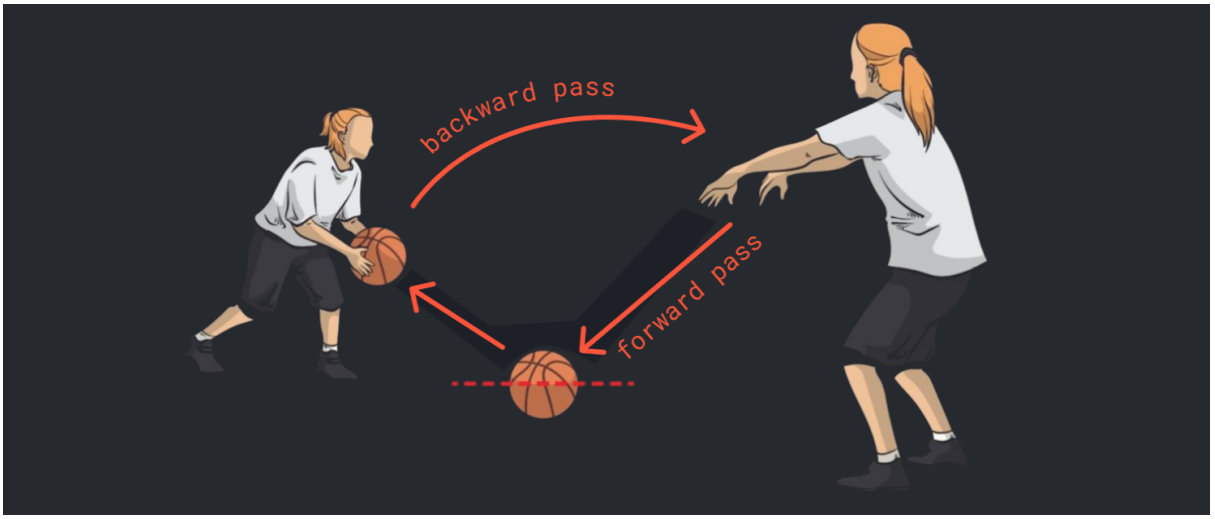
Backpropagation в деталях

Каждый слой передает эту информацию следующему слою в цепочке, обновляя веса и параметры в соответствии с градиентами. Этот процесс продолжается до тех пор, пока градиенты не достигнут самого начала сети. Этот процесс, когда градиенты передаются от конца сети к началу, **называется обратным распространением ошибки (backpropagation)**. Back означает движение назад.

Для лучшего понимания представьте себе нейросеть, которая вначале передает входные данные через несколько слоев, каждый из которых обрабатывает информацию и передает ее следующему слою. После прохождения через все слои

сети вы вычисляете функцию потерь. Затем начинается обратный проход: вы вычисляете градиенты функции потерь, а затем передаете их обратно через сеть.

Итак, сначала вы идете вперед по сети, рассчитывая ответ, **что называется прямым проходом (forward pass)**. Затем, уже имея выходные данные, вы обрабатываете функцию потерь и ищете ошибки, чтобы затем вернуться назад через сеть, чтобы рассчитать градиенты. **Этот шаг называется обратным проходом (backward pass).**

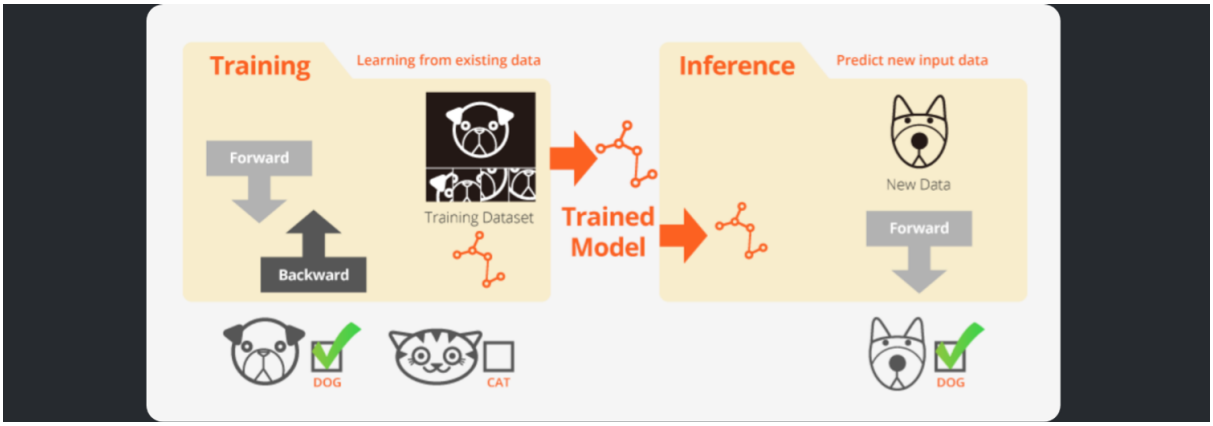


Источнник картинки: https://mas-alahrom.my.id/images/teknik_bermain_bola_basket/bounce_passing.jpg

Важно помнить, что для выполнения обратного прохода необходимо хранить все промежуточные данные. Это требует дополнительной памяти, особенно для больших сетей. Поэтому для нейросетей разработаны два режима работы:

1. Режим обучения (**train**)
2. Режим вывода (**inference** или **eval**).

В режиме обучения нейросеть запоминает все промежуточные выходы, так как вам нужно будет вычислять градиенты. В режиме вывода нейросеть уже обучена и не требует вычисления градиентов, поэтому нет необходимости хранить промежуточные данные, что позволяет сэкономить память.



Источник картинки: https://www.qnap.com/solution/openvino/hu-hu/

Промежуточные итоги

- Backpropagation начинается с конечного слоя нейронной сети и движется обратно к началу. Каждый слой сети принимает на вход градиент по своему выходу и использует его для обновления своих параметров (весов) и передачи градиента дальше.
- Важно сохранять промежуточные значения (тензоры), полученные во время прямого прохода (forward pass), так как они потребуются для вычисления производных во время обратного прохода (backward pass).
- Существуют два режима работы нейронных сетей: обучение (train) и выполнение (inference). В режиме обучения сеть запоминает промежуточные значения для вычисления градиентов, в то время как в режиме выполнения это не требуется, и промежуточные значения не сохраняются.
- Процесс обучения сети включает в себя два этапа: прямой проход (forward pass) и обратный проход (backward pass). Во время прямого прохода данные проходят через сеть и вычисляется ответ сети. Затем, во время обратного прохода, вычисляются градиенты и происходит обновление параметров сети на основе этих градиентов.
- Нелинейности, такие как функции активации, рассматриваются как дополнительные слои между линейными слоями. Они также участвуют в процессе обратного распространения ошибки, передавая градиенты от выходов к входам.

Градиентный спуск

Теперь поговорим о методах оптимизации, а именно о градиентном спуске и его улучшениях.

Градиентный спуск (Gradient descent) — это основной метод оптимизации в машинном обучении.

Он работает следующим образом: мы берем некоторое подмножество наших данных, прогоняем их через модель, после чего считаем функцию потерь и находим градиент этой функции по отношению к параметрам модели на этих конкретных данных.

Этот набор данных называется батчем.

Затем мы обновляем параметры модели в направлении, противоположном градиенту, чтобы минимизировать функцию потерь:

$$\theta_t = \theta_{t-1} + \eta\vec{\nabla}L(\theta_t)$$

Где:

θ_t— новое значение параметра после обновления на шаге t

θ_{t-1} — предыдущее значение параметра на шаге t-1

η — скорость обучения (learning rate), коэффициент, определяющий размер шага обновления параметров

$\vec{\nabla}L(\theta_t)$ — градиент функции потерь L по параметру θ_t, указывает направление наибольшего убывания функции потерь.

Важно помнить, что мы движемся в направлении **антиградиента**. Мы делаем это на каждом шаге, обновляя параметры модели, чтобы получить новые предсказания. Затем мы повторяем процесс: снова берем батч, считаем градиенты, обновляем параметры, пока не выполнится условие остановки (критерий останова).

Критерий останова — это условие, определяющее, когда следует завершить процесс оптимизации. Этот критерий обычно используется для того, чтобы определить, достигли ли мы достаточно хорошего приближения к минимуму функции потерь или когда дальнейшая оптимизация не оправдана.

Вот некоторые типичные примеры критериев останова:

1. **Ограничение на количество итераций.** В этом случае оптимизация завершается после определенного числа шагов. Например, мы можем решить, что 50 итераций достаточно для получения хороших результатов. Это простой и часто используемый критерий, особенно когда точное значение минимума неизвестно.
2. **Проверка изменения параметров.** Мы можем остановить оптимизацию, когда изменение параметров модели на следующей итерации становится

незначительным. Например, если разница между весами на текущей и предыдущей итерациях меньше определенного порога, например, 10^-4, то мы можем считать оптимизацию завершенной.

3. **Достижение заданного значения функции потерь.** Если мы заранее знаем значение функции потерь, при котором мы считаем задачу решенной, мы можем использовать это значение как критерий останова. Например, если наша функция потерь достигает значения менее 0.1, мы можем остановить оптимизацию.
4. **Расходимость.** Если оптимизация начинает расходиться (т.е., значение функции потерь начинает увеличиваться), мы можем остановиться, чтобы избежать потери времени на дальнейшие бесполезные вычисления.

Проблемы градиентного спуска:

- Выбор шага: если мы выберем слишком большой шаг, мы можем проскочить минимум или максимум функции потерь. Если шаг слишком маленький, процесс может быть очень медленным.
- Используем только часть данных для вычисления градиента, что может привести к случайным блужданиям и затруднить сходимость к оптимуму.

Для улучшения градиентного спуска были разработаны различные методы.

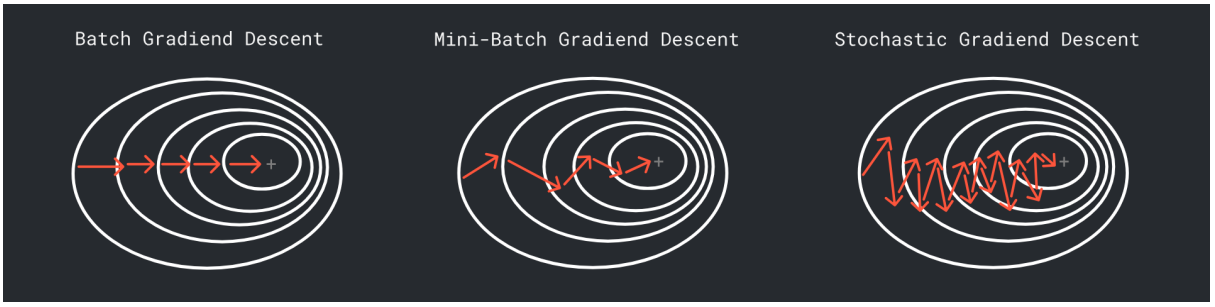
Стохастический градиентный спуск (SGD)

Стохастический градиентный спуск (SGD, Stochastic Gradient Descent) — это вариант градиентного спуска, который обновляет параметры модели на основе градиента функции потерь, вычисленного на случайном подмножестве данных (**батче**). Вот формула для обновления параметров в SGD:

$$\theta_{t+1} = \theta_t + \alpha \vec{\nabla} L(\theta_t, x_i, y_i)$$

Где:

- θ_t — вектор параметров модели на предыдущем шаге
- θ_{t+1} — обновленный вектор параметров модели
- α — скорость обучения (learning rate), коэффициент, определяющий величину шага при обновлении параметров
- $\vec{\nabla} L(\theta_t, x_i, y_i)$ — градиент функции потерь L по параметрам модели на конкретном наблюдении (x_i, y_i)



Виды градиентного спуска

Сильные стороны SGD	Слабые стороны SGD
Эффективность использования памяти. SGD требует гораздо меньше памяти, чем полный градиентный спуск, потому что он работает с одним случайным наблюдением за раз.	Нестабильность обучения. Из-за случайности выбора батчей SGD может быть менее стабильным в сходимости к оптимуму.
Быстрая сходимость: Благодаря частым обновлениям параметров модели, SGD может быстро сойтись к локальному оптимуму.	Зависимость от выбора скорости обучения. Неправильный выбор скорости обучения может привести к расходимости или медленной сходимости SGD.
Легко интерпретируется и иллюстрируется	

Чтобы решить проблемы SGD был разработан ряд улучшений, о который поговорим далее.

Методы оптимизации SGD

Momentum (добавление инерции)

Одна из идей была придумана на основе случайности, присутствующей в выборке батчей. При использовании SGD градиент может "вести" модель в разные стороны из-за случайного выбора данных.

Инерция — это концепция сохранения движения модели в том же направлении, что и на предыдущем шаге. Это аналогично инерции тяжелого объекта: если он двигается в определенном направлении, он будет сохранять скорость движения в этом направлении.

В математической формулировке, скорость из предыдущей итерации умножается на коэффициент (обычно 0.9), а скорость из текущей итерации на другой коэффициент (обычно 0.1), чтобы сохранить движение модели.

В методе Momentum добавляется инерционный член $v_t + 1$, который представляет собой "накопленный импульс" градиента. Этот член позволяет учитывать предыдущие изменения параметров и направление градиента при обновлении весов.

$\theta_{t+1} = \theta_t - \eta\vec{\nabla}L(\theta_t) = \theta_t + v_{t+1}$

$v_{t+1} = \mu v_t - \eta\vec{\nabla}L(\theta_t)$, где μ — гиперпараметр, отвечающий за то, как быстро забываем прошлую скорость

$\eta\vec{\nabla}L(\theta_t)$ — это стандартное обновление параметров на основе градиента функции потерь L по параметрам θ_t, где η — скорость обучения, а $\vec{\nabla}L(\theta_t)$— градиент функции потерь по параметрам θ_t.

v_{t+1} — это накопленный импульс, который вычисляется на основе предыдущего импульса v_t и текущего градиента. v_{t+1} учитывает изменения параметров, произошедшие на предыдущих итерациях, и добавляет их к текущему обновлению параметров. Формула для показывает, что он вычисляется как предыдущий импульс, умноженный на коэффициент затухания μ, минус скорость обучения η умноженная на градиент функции потерь по параметрам θ_t.

Таким образом, метод Momentum помогает ускорить сходимость и обеспечить более плавное движение по градиентному пространству за счет учета накопленного импульса.

Однако данный метод имеет ряд существенных недостатков:

- **Склонность к “перепрыгиванию”.** Иногда Momentum может "перепрыгнуть" оптимальное значение из-за накопленного импульса, особенно на плато функции потерь.
- **Чувствительность к гиперпараметру.** Эффективность метода Momentum зависит от выбора **гиперпараметра** (коэффициента скорости обучения). Неправильный выбор может привести к медленной сходимости или расходимости.
- **Трудность настройки.** Подбор правильного значения для коэффициента скорости обучения и начального импульса может быть сложной задачей.
- **Потенциальные проблемы с быстрым снижением скорости обучения.** В некоторых случаях метод Momentum может привести к тому, что скорость обучения снизится слишком быстро, что замедлит сходимость или застрянет в локальном минимуме.

- **Память о предыдущих градиентах.** Использование предыдущих градиентов может иногда замедлить сходимость, особенно если данные нестационарны или функция потерь имеет изменяющийся градиент.

AdaGrad (Adaptive Gradient Algorithm)

Вторая идея заключается в том, чтобы учесть разные масштабы градиента по разным направлениям. Вот как это работает:

- **Идея адаптивного масштабирования.** В основе Adagrad лежит идея штрафовать большие компоненты градиента и увеличивать шаг обучения для маленьких компонент. Это становится важным, когда градиенты по разным направлениям сильно отличаются по величине. Например, если вдоль одной оси градиент очень большой, а вдоль другой — маленький, это может привести к неэффективной оптимизации.
- **Адаптивное масштабирование по элементам.** Для реализации этой идеи Adagrad берет каждый элемент градиента и умножает его сам на себя (то есть возводит в квадрат). Это делается поэлементно, и результаты остаются в том же размере, что и исходные градиенты. Таким образом, большие значения градиента становятся еще больше, а маленькие — меньше.
- **Аккумуляция градиентов.** Adagrad хранит сумму квадратов градиентов для каждого параметра на протяжении всего обучения. Это позволяет алгоритму учитывать историю изменений градиентов и адаптировать скорость обучения для каждого параметра на основе его прошлых значений градиентов.
- **Проблемы с долгосрочной памятью.** Одним из недостатков Adagrad является его чувствительность к выбросам в данных. Поскольку он накапливает квадраты градиентов, даже небольшие изменения в градиентах могут привести к значительным изменениям в скорости обучения. Это может привести к проблемам с сходимостью или переобучению модели.

$g_t = g_{t-1} + \vec{\nabla} L(\theta_t) \odot \vec{\nabla} L(\theta_t)$: Накопление квадрата градиента функции потерь L по параметру θ_t на текущем шаге t поэлементно.

$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \vec{\nabla} L(\theta_t)$: Обновление параметра θ на следующем шаге $t+1$ с использованием шага обучения η, деленного на квадратный корень из накопленного квадрата градиента G_t с добавлением эпсилона для стабильности, поэлементно умножая на градиент функции потерь L по параметру θ_t.

В целом, Adagrad предлагает эффективное решение для адаптивного масштабирования скорости обучения, но имеет одну серьезную проблему - он сохраняет и накапливает информацию о градиентах на протяжении всего обучения. Это означает, что если какая-то компонента градиента станет очень большой на одной итерации, она будет оставаться большой и в последующих итерациях, что может привести к замедлению сходимости или даже к расходимости алгоритма.

Для решения этой проблемы была предложена третья идея, реализованная в оптимизаторе **RMSProp**. Этот метод использует скользящее среднее для постепенного забывания информации о градиентах в прошлом.

$G_t = \mu G_{t-1} + (1-\mu)\vec{\nabla} L(\theta_t) \odot \vec{\nabla} L(\theta_t)$: Обновление аккумулированного квадрата градиента G_t на текущем шаге t с использованием экспоненциального скользящего среднего предыдущего G_{t-1} и квадрата градиента функции потерь L по параметру θ_t, умноженного поэлементно на себя.

$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \vec{\nabla} L(\theta_t)$: Обновление параметра аналогично как и в Adagrad

Таким образом, если какая-то компонента градиента внезапно становится очень большой, она будет забыта со временем, что позволяет избежать проблемы сохранения больших значений градиента навсегда.

Adam

Adam (Adaptive Moment Estimation) — это метод оптимизации, который сочетает в себе несколько ключевых концепций для эффективного обучения нейронных сетей. Давайте разберемся подробнее, как работает этот оптимизатор.

Оптимизатор Adam объединяет лучшие идеи из предыдущих методов. Он включает в себя инерцию для ускорения обучения, штраф за большие компоненты градиента и механизм забывания старых значений градиента.

В основе Adam лежит идея адаптивной инерции. Это означает, что он учитывает как направление, так и скорость изменения параметров модели при обновлении весов. По сути, Adam сохраняет скользящее среднее предыдущих градиентов и их квадратов для каждого параметра.

Как это работает на практике? Давайте представим, что мы обучаем нейронную сеть для распознавания изображений. После каждой итерации обучения, мы вычисляем градиенты по всем параметрам сети. Затем Adam подсчитывает скользящее среднее для градиентов и их квадратов.

Пример:

- Предположим, у нас есть параметр модели, который отвечает за распознавание границ объектов на изображении. После вычисления градиента для этого параметра, Adam сохраняет скользящее среднее его значений.
- Затем мы также вычисляем квадрат градиента для этого параметра и сохраняем скользящее среднее квадратов.
- Adam использует эти скользящие средние для корректировки скорости обновления параметров. Если какая-то компонента градиента велика, Adam уменьшает скорость обновления, чтобы избежать слишком больших изменений параметров.
- В то же время, если некоторые компоненты градиента малы, Adam увеличивает скорость обновления, чтобы ускорить обучение в этих направлениях.

Другим ключевым аспектом Adam является штраф за большие компоненты градиента. Это помогает избежать проблемы, когда одна компонента градиента становится слишком большой и "перекрывает" влияние других параметров на обучение модели.

Пример:

- Предположим, что у нас есть параметр, который отвечает за определение цвета объекта на изображении. Если величина градиента для этого параметра слишком велика, Adam применяет штраф, чтобы уменьшить его влияние на обновление весов, обеспечивая более стабильное обучение.

Математически Adam может быть представлен следующим образом:

$$v_{t+1} = \beta_1 v_t + (1 - \beta_1)\vec{\nabla} L(\theta_t) :$$

Где:

- v_{t+1} — обновленный первый момент на текущем шаге $t + 1$, учитывающий текущий градиент функции потерь.
- β_1 — коэффициент затухания для первого момента (обычно 0.9).
- v_t — предыдущее значение первого момента.
- $\vec{\nabla} L(\theta_t)$ — градиент функции потерь L по параметру θ_t.

Возьмем инерцию по градиенту, добавим штраф:

$$g_t = \mu g_{t-1} + (1 - \mu)\nabla L(\theta_t) \odot \nabla L(\theta_t),$$

Где:

- g_t: накопленные квадраты градиентов на шаге t
- μ: коэффициент затухания, обычно выбирается близким к 1 (например, 0.9)

- g_{t-1}: накопленные квадраты градиентов на предыдущем шаге $t-1$
- $\nabla L(\theta_t)$: градиент функции потерь по параметрам на шаге t
- $\odot$: покомпонентное умножение (произведение Адамара)

Обновим веса:

$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{g_t + \epsilon}} \odot v_{t+1},$

Где:

- θ_t: параметры на шаге t
- v_{t+1}: коррекция по инерции на шаге $t+1$
- g_t: накопленные квадраты градиентов до шага t
- η: скорость обучения (learning rate)
- ϵ: значение, добавленное для численной стабильности

Таким образом, Adam сочетает в себе идеи адаптивной инерции и штрафа за большие градиенты для эффективного обучения нейронных сетей. Он позволяет алгоритму быстро сходиться к оптимальным значениям параметров модели, даже при наличии больших различий в масштабах компонентов градиента.

Практика

Задача классификации изображений. Общий пайплайн.

Разберем на примере небольшого датасета как строить и обучать модели в PyTorch*, и закрепим наши знания про слои.

Будем решать задачу по классификации английских букв от A до J на изображениях. Изображения имеют фиксированный размер 28x28 пикселей, черно-белые.

Будем придерживаться следующего пайплайна:

- постановка задачи;
- определимся с метриками;
- соберем данные;
- построим бейзлайн, обучим его;
- проверим качество бейзлайна;

Бейзлайн мы построим на известных нам слоях: Linear и нелинейности.

В качестве финальной метрики бизнес интересует **accuracy** — точность распознавания класса.

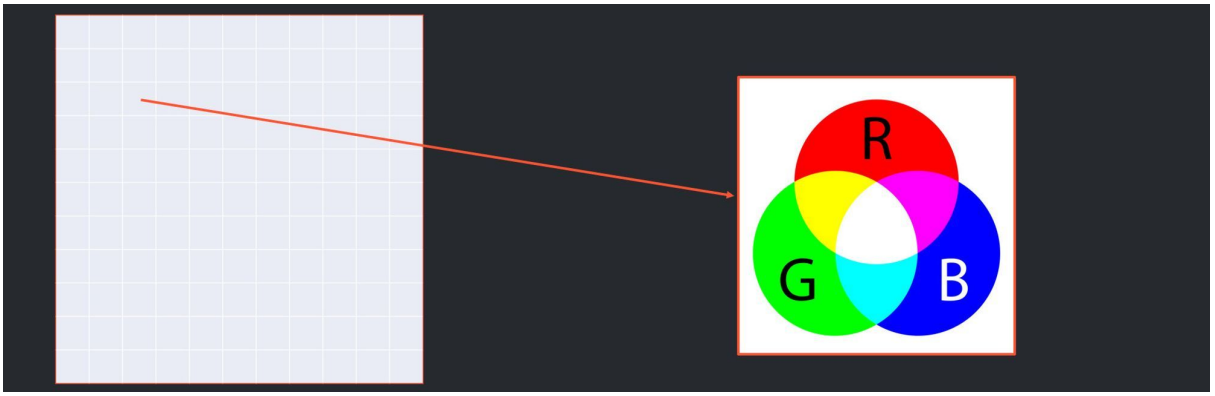
Для обучения и валидации будем использовать датасет notMNIST, который можно скачать по ссылке.

Как хранятся картинки в памяти компьютера

Компьютер хранит картинку в памяти как набор пикселей. Каждый пиксель — это квадрат одного цвета. Пиксели достаточно маленькие, поэтому глаз не замечает, что изображение состоит из квадратов.

В черно-белых изображениях пиксель представляет из себя одно число — интенсивность белого цвета. 0 — это черный цвет, 255 — белый, посередине — оттенки серого.

В цветных изображениях пиксель — это набор из трех чисел: (R, G, B). R — интенсивность красного цвета, G — интенсивность зелёного цвета, B — синего. Числа тоже в диапазоне от 0 до 255 (обычно).



В нашем датасете только ч/б изображения, поэтому каждая картинка будет представлять из себя матрицу (28, 28) — 28 пикселей в ширину и 28 в высоту.

Для работы нам понадобится импортировать следующие библиотеки:

```
import http.client
import tarfile
from pathlib import Path
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
from sklearn.preprocessing import LabelEncoder
```

Для воспроизводимости результатов можно зафиксировать генератор случайных чисел, задав seed:

```
seed = 0
torch.cuda.manual_seed_all(seed)
torch.manual_seed(seed)
np.random.seed(seed)
```

Функция для загрузки датасета:

```
def prepare_data():
"""Скачивает данные и распаковывает их."""
target_file = "notMNIST_small.tar.gz"
if Path(target_file).exists():
```

```
print("Файл уже загружен, не загружаю снова")
else:
conn = http.client.HTTPConnection("yaroslavvb.com", 80)
conn.request("GET", "/upload/notMNIST/notMNIST_small.tar.gz")
data = conn.getresponse().read()
with open(target_file, "wb") as f:
f.write(data)
with tarfile.open(target_file) as f:
f.extractall()
print("Данные были скачены и распакованы")
```

Функция, считывающая файлы формата формата png по указанному пути с помощью функции `plt.imread()` и возвращающая в два массива — с изображениями и соответствующими им лэйблами:

```
def read_notmnist_data(
data_dir: str = "notMNIST_small",
) -> tuple[np.ndarray, np.ndarray]:
"""Прочитать картинки датасета notMNIST и положить их в numpy-масс
ив.
:returns: пару numpy-массивов (изображения, соответствующие метки)
"""
images, labels = [], []
for img_path in Path(data_dir).glob("**/*.png"):
    # Имя папки - это метка класса
    img_label = img_path.parts[1]
    try:
        image = plt.imread(img_path)
    except SyntaxError:
        print(
            f"Изображение не читается по пути {img_path} (это ок,
но таких должно быть < 10)"
        )
        continue
    labels.append(img_label)
    images.append(image)
return np.stack(images, axis=0), np.stack(labels, axis=0)
```

```
prepare_data()
X, y = read_notmnist_data()
assert X.shape[0] == y.shape[0] #проверка, что количество лэйблов со
```

```
print("Размеры данных:", X.shape, y.shape)
print(f"Пример класса: {y[0]}")
# видим, что метки - это буквы, закодируем их числами
ohe = LabelEncoder()
y = ohe.fit_transform(y)
```

Разделение данных на тренировочную, валидационную и тестовую выборки:

```
from sklearn.model_selection import train_test_split
X_train_val, X_test, y_train_val, y_test = train_test_split(
X, y, test_size=0.2, shuffle=True, random_state=seed
)
X_train, X_val, y_train, y_val = train_test_split(
X_train_val, y_train_val, test_size=0.2, shuffle=True, random_state=
)
```

Визуализация данных из трейна:

```
fig, ax = plt.subplots(4, 4, figsize=(16, 16))
for row in range(4):
    for col in range(4):
        idx = 4 * row + col
        ax[row][col].imshow(X_train[idx])
        ax[row][col].set_title(f"label={y_train[idx]}")
```

Построение модели

Чтобы объявить сеть в PyTorch*, нужно:

- создать класс, наследующийся от `torch.nn.Module`
- в методе `forward(self, x)` прописать, сколько тензоров сеть принимает и что с ними нужно сделать
- в конструкторе (метод `__init__`) объявить все необходимые слои, вызвав `super().__init__()` в начале

```
class SimpleModel(nn.Module):
    # В __init__ объявим все слои, которые нам нужны
    def __init__(self, num_classes: int):
```

```
        super().__init__()

        # NOTE: Linear принимает (N, C), картинки заданы как (N, W,
        # Будем из 28 * 28 переводить в 256
        hidden_dim = 256 #промежуточное значение слоя
        self.linear_1 = nn.Linear(in_features=28 * 28, out_features=

        # Нелинейность иногда называют активацией. Мы возьмем ReLU(x
        self.act_1 = nn.ReLU()

        # Второй линейный слой, принимать будет выход из первого, на
        self.linear_2 = nn.Linear(in_features=hidden_dim, out_featur

        # Слой активации Softmax выдает числа от 0 до 1, которые в с
        self.act_2 = nn.Softmax(dim=1)
        # NOTE: какой выход куда пойдет - это будет прописано в forw

    def forward(self, x: torch.Tensor):
        # В этом методе нужно сделать все преобразования над тензоро
        # Мы сначала поменяем форму, как это требует linear_1
        x = x.reshape((-1, 28 * 28)) #схлопывает две последние разме
        # Потом последовательно применим все слои
        for one_transform in (self.linear_1, self.act_1, self.linear
            x = one_transform(x)
        return x
```

В PyTorch нет полносвязного слоя. Вместо этого используется комбинация из двух слоев: линейного (`nn.Linear`) и нелинейности. Их много, например `nn.Tanh`.

Важно отметить, что PyTorch* модели не умеют работать с numpy-массивами, только с `torch.Tensor`. Поэтому массивы необходимо конвертировать через `torch.from_numpy()`.

Желательно также прогнать созданную сеть над примером данных — чтобы убедиться, что все работает:

```
model = SimpleModel(num_classes=len(ohe.classes_))
some_output = model(torch.from_numpy(X_train[[0]]))
some_output
```

Градиент

У выхода сети есть атрибут `grad_fn`. Это неспроста: мы использовали все слои из `torch.nn`, поэтому PyTorch* сумел **самостоятельно** разобраться, как считать градиент по всем слоям и всем параметрам. Нам не нужно расписывать формулы и считать градиенты — это делает автоматика!

Давайте убедимся в этом: подсчитаем функцию потерь и попросим от нее градиент.
Будем использовать Cross Entropy в качестве функции потерь — хороший вариант для задачи классификации.

```
import torch.nn.functional as F

loss = F.cross_entropy(some_output, torch.from_numpy(y[[0]]))
print(loss)
# Хм, а реально ли градиенты все сами подсчитаются? Посмотрим
print(model.linear_1.weight.grad)
# Градиента нет - мы лишь прогнали данные через модель
```

```
# Теперь просим подсчитать градиент и распространить его назад по вс
loss.backward()
# Смотрим на градиент
print(model.linear_1.weight.grad)
```

Когда мы вызвали `loss.backward()`, PyTorch* сделал сделал следующее:

1. Подсчитал напрямую производную лосса

$L = -\sum_{c=1}^{M} y_c \log p_c$ по p_c

1. Отследил цепочку преобразований, через которую получился этот p_c — вероятность каждого класса.
2. Прошелся по этой цепочке с конца в начало, применил chain rule и подсчитал градиент по каждому из обучаемых параметров.
3. Записал этот градиент в тензоры weight и bias каждого слоя, чтобы не забыть.

И все это произошло "под капотом" одной функции: `loss.backward()` — чудеса!

Обучение модели

Когда мы собрали данные, построили модель, выбрали функцию потерь, научились считать градиент по всем параметрам модели, дело осталось за малым - пройтись градиентным спуском по данным и оптимизировать параметры.

PyTorch* делает четкое разделение: модели в одном месте, алгоритм спуска - в другом.
Более того, сам спуск вам понадобится написать вручную - не бойтесь, это несложно.

Чтобы не вызывать везде `torch.from_numpy` можно преобразовать все данные в `torch.Tensor`:

```
X_train, y_train = torch.from_numpy(X_train), torch.from_numpy(y_tra
X_val, y_val = torch.from_numpy(X_val), torch.from_numpy(y_val)
X_test, y_test = torch.from_numpy(X_test), torch.from_numpy(y_test)
```

Для "обучения" модели есть модуль
`torch.optim` — в нем инструменты для обучения параметров.

Как работает Optimizer:
- вы говорите ему, какие параметры в его зоне ответственности;
- затем вы говорите ему, что пришло время сделать шаг спуска;
- optimizer проходит по всем параметрам, за которые отвечает, берет их градиент и что-то с ним делает;
- что именно делает — зависит от конкретного оптимизатора;

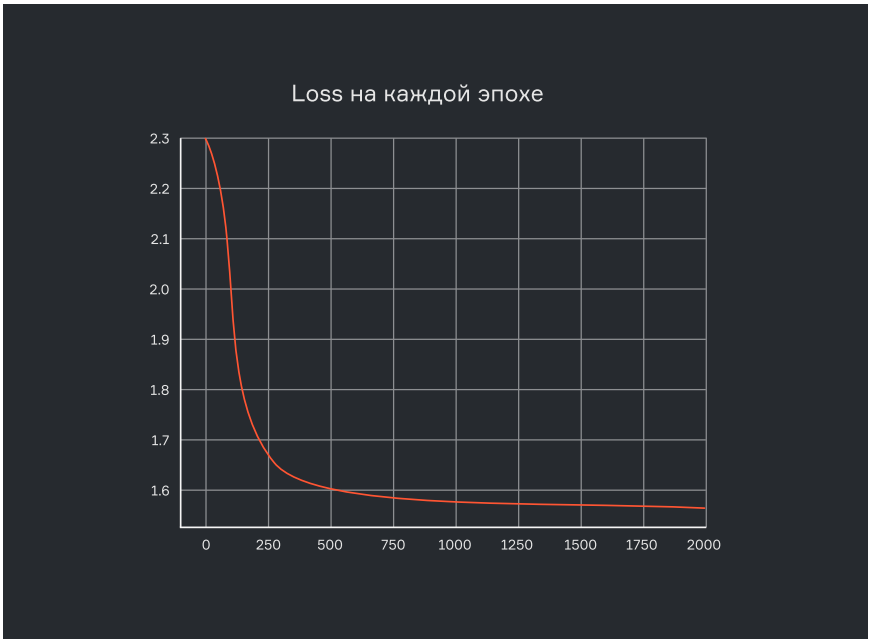
Конкретно SGD делает обновление весов по формуле w = w - eta * gradient, то есть обычный градиентный спуск.
Есть более продвинутые алгоритмы (Adam, RMSprop), с ними познакомимся позже.

```
import tqdm

# все методы оптимизации лежат в torch.optim
from torch.optim.sgd import SGD

# создаем optimizer и говорим "ты отвечаешь за все параметры модели"
optimizer = SGD(params=model.parameters(), lr=1e-1)
```

```
# model.parameters() возвращает все обучаемые параметры модели (в ви
# Откуда функция все эти параметры узнает, ведь мы нигде их явно не
# Это магия PyTorch, он умеет многое делать сам :)


def train_model(model: nn.Module, optimizer: SGD, x: torch.Tensor, y
    losses = []
    # Пройдемся 2000 раз по всем данным
    for _ in tqdm.trange(2_000):
        # optimizer по умолчанию "помнит" градиенты с прошлых итерац
        # Так сделано ради продвинутых техник обучения.
        # Нам это не надо, поэтому в каждой итерации явно зануляем в
        optimizer.zero_grad()

        # Это блок уже знаем: считаем выход, потери, градиенты по по
        output = model(x)
        loss = F.cross_entropy(output, y)
        loss.backward()

        # Просим оптимизатор пройтись по параметрам и сделать градие
        # Оптимизатор сам обновит веса, вручную этого делать не надо
        optimizer.step()

        # Запомним loss
        losses.append(loss.detach().item())
    return losses


losses = train_model(model, optimizer, X_train, y_train)
```

Посмотреть, действительно ли модель обучилась, можно построив график изменения функции потерь:

```
def plot_losses(losses: list[float]):
    plt.plot(losses)
    plt.title("Loss на каждой эпохе")
    plt.grid()
    plt.show()


plot_losses(losses)
```



Также важно проверить качество классификации:

```
def accuracy(model: SimpleModel, x: torch.Tensor, y: torch.Tensor):
    return torch.sum(torch.argmax(model(x), dim=1) == y) / len(y)


print(f"accuracy на train данных: {accuracy(model, X_train, y_train)
print(f"accuracy на val данных: {accuracy(model, X_val, y_val)}")
print(f"accuracy на test данных: {accuracy(model, X_test, y_test)}")
```

В качестве возможных улучшений модели можно попробовать следующее:

- использовать AdaM вместо SGD;
- добавить больше слоев;
- попробовать новые слои (в следующем уроке пополним наш арсенал).

Задача регрессии и оптимизатор Adam

Задача регрессии в PyTorch* решается по тому же пайплайну, как и задача классификации, меняется только последний слой — вместо 10 классов модель регрессии будет предсказывать одно число.

Еще один способ, которым можно задавать слои в нейросети — передавать их в `nn.Sequential`, последовательный контейнер в PyTorch*, который позволяет

добавлять модули в определенном порядке и выполнять их последовательно:

```
self.transform = nn.Sequential(
nn.Linear(in_features=num_features, out_features=hidden_dim),
nn.ReLU(),
nn.Linear(in_features=hidden_dim, out_features=1)
```

Еще один полезный метод `.squeeze()` в PyTorch* используется для удаления размерностей с единичной длиной из тензора. Он удаляет все размерности с единичной длиной (размерностью 1) из тензора.

Это полезно, когда нужно привести размерность тензора к желаемому виду, например, перед подачей на полносвязный слой.

Например, если у вас есть тензор размера (10,1), то `.squeeze()` приведет его к размеру (10,):

```
 def forward(self, x: torch.Tensor):
        # .squeeze(1) "съест" лишнюю размерность: (N, 1) -> (N, )
        return self.transform(x).squeeze(1)
```

Импортируем и создадим объект оптимайзер:

```
from torch.optim.adam import Adam
reg_optimizer = Adam(params=reg_model.parameters())
```

Обучение модели на 3000 эпохах:

```
# Перед началом обучения разместим модель на нужном устройстве
reg_model = reg_model.to(device)
# Оптимизатор не надо на устройство переносить

# И идем учиться, 3000 эпох
# Данных мало, так что сразу на всех их будем учиться
reg_losses = []
for _ in tqdm.trange(3000):
    # Не забываем обнулить градиенты
    reg_optimizer.zero_grad()

    # Переносим новый батч на GPU
    # Вообще, раз учимся на всех данных, можно было 1 раз перенести
```

```
    # Но в реальной ситуации будут разные данные на каждой итерации
    batch_x = t_train_features.clone().to(device)
    batch_y = t_train_target.clone().to(device)

    # Считаем выход, лосс, градиенты - и делаем шаг
    output = reg_model(batch_x)
    # MSE loss лежит в том же месте, где cross_entropy
    # И используется точно так же
    loss = F.mse_loss(output, batch_y)
    loss.backward()
    reg_optimizer.step()

    # .cpu() перенесет loss на ЦПУ, .item() вернет как число
    reg_losses.append(loss.cpu().item())
```

Функция потерь регрессии в PyTotch*:

```
# Считаем финальную метрику на тестовых данных
with torch.no_grad():
    # Помним, что все тензоры должны быть на одном устройстве
    out = reg_model(t_test_features.to(device)).cpu()
    mse_loss = F.mse_loss(out, t_test_target).item()
    # MAPE нету в F, придется руками делать
    mape_loss = torch.mean(
        # clamp обрезает все значения тензора с двух концов.
        torch.abs(out - t_test_target) / torch.clamp(torch.abs(out),
    ).item()
    print("Финальный MSE-лосс:", mse_loss)
    print("Финальный MAPE-лосс:", mape_loss)
```

Резюме

1. Узнали, что картинки хранятся в компьютере как матрица, каждый ее элемент описывает интенсивность цвета.
2. Научились создавать нейросеть через класс, наследующийся от `nn.Module`.
3. Научились прогонять данные через нейросеть, считать функцию потерь и градиенты по всем параметрам.
4. Познакомились с оптимизаторами (optimizer), с их помощью обучили нейросеть, используя SGD.

5. Посчитали метрику качества обученной модели, получили рабочий бейзлайн.
6. Научились применять оптимизатор AdaM.
7. Разобрались, как использовать GPU во время обучения.

> 💡 PyTorch является продуктом, финансируемым компанией Meta. Компания Meta Platforms Inc., по решению Тверского районного суда города Москвы от 21.03.2022, признана экстремистской организацией, её деятельность на территории России запрещена.

Дополнительные материалы

1) Туториал из TensorFlow

2) Хорошая демонстрация того, как ведет себя градиентный спуск с инерцией: тут