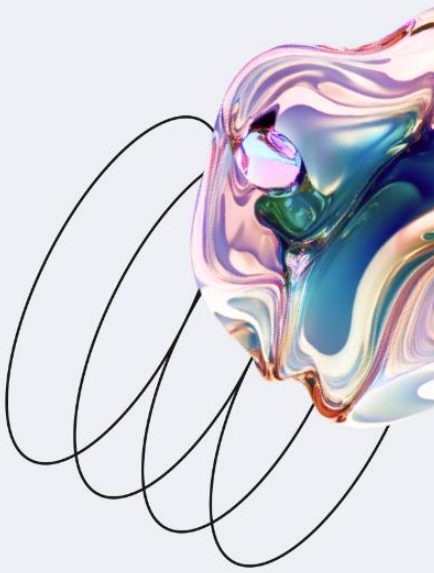
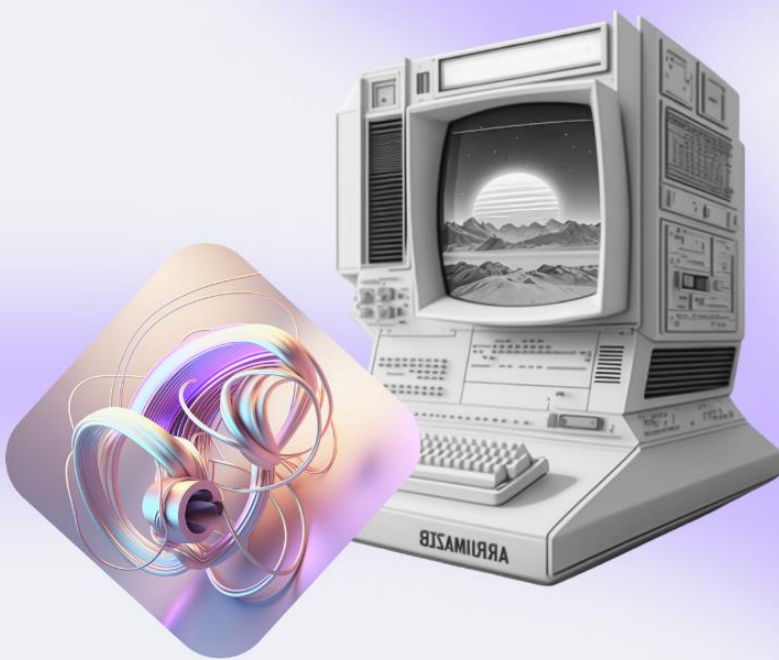
GeekBrains



Создание нейросетей с использованием Pytorch



Оглавление

Словарь терминов

Веса модели — это параметры нейронной сети, которые определяют силу связи между нейронами. Каждый вес умножается на соответствующий входной сигнал и влияет на выходное значение нейрона.

Нейрон — это базовая вычислительная единица нейронной сети.

Нейронная сеть — это математическая модель, вдохновленная биологическими нейронными сетями мозга. Она состоит из связанных между собой нейронов, организованных в слои, и используется для решения задач машинного обучения, таких как классификация, регрессия и генерация данных.

Обратное распространение ошибки (Backpropagation) — это алгоритм обучения нейронных сетей, который вычисляет градиенты функции потерь относительно весов и смещений. Алгоритм работает в два этапа: прямой проход (вычисление предсказаний) и обратный проход (корректировка параметров на основе ошибки).

Смещения модели (Biases) — это дополнительные параметры в нейронной сети, которые позволяют сдвигать функцию активации нейрона.

Функция активации — это нелинейная функция, которая применяется к выходу нейрона после взвешенного суммирования входов и добавления смещения.

Функция потерь (Loss Function) — это метрика, которая измеряет, насколько предсказания модели отличаются от истинных значений.

Введение

Нейронные сети представляют собой мощный инструмент машинного обучения, который широко используется для решения задач классификации, регрессии, генерации данных и других. Их успех обусловлен способностью автоматически извлекать сложные паттерны из данных, что делает их незаменимыми в таких областях, как компьютерное зрение, обработка естественного языка и анализ временных рядов. Для эффективного применения нейронных сетей важно понимать их основные элементы, принципы построения архитектуры, выбор функций активации и потерь, а также процесс обучения.

В этой лекции рассматриваются ключевые аспекты работы с нейронными сетями. Мы начнем с анализа основных элементов нейронных сетей, таких как нейроны, веса, смещения и функции активации. Затем перейдем к практической части, где разберем, как определить архитектуру нейронной сети с использованием библиотеки PyTorch. Далее рассмотрим различные функции активации и функции потерь, а также методы оптимизации, которые играют важную роль в обучении модели. Наконец, мы обсудим процесс обучения нейронной сети, включая алгоритм обратного распространения ошибки.

Основные элементы нейронных сетей

Нейронные сети — это вычислительные модели, вдохновленные биологическими нейронными сетями мозга. Они широко используются в машинном обучении для решения задач классификации, регрессии, генерации данных и других. Основные элементы нейронных сетей можно разделить на несколько ключевых компонентов.

Нейроны (узлы)

Нейроны — это базовые единицы нейронной сети. Каждый нейрон принимает входные данные, обрабатывает их и выдает выходное значение. В математическом смысле нейрон – функция $z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$, где:

- x_i — входные значения,
- w_i — веса, которые определяют важность каждого входа,
- b — смещение (bias), которое позволяет сдвигать функцию активации.

Выход нейрона вычисляется как $a = f(z)$, где f — функция активации.

Функции активации

Функции активации добавляют нелинейность в модель, что позволяет нейронным сетям решать сложные задачи.

Слои

Нейронные сети состоят из слоев, каждый из которых выполняет определенную роль:

- Входной слой (Input Layer) получает исходные данные (например, пиксели изображения или признаки объекта).
- Скрытые слои (Hidden Layers) выполняют основную обработку данных. Число слоев и нейронов в них определяет сложность сети.
- Выходной слой (Output Layer) генерирует конечный результат (например, метку класса или числовое значение).

Веса и смещения

Веса (Weights) – параметры, которые умножаются на входные значения и определяют силу связи между нейронами. В процессе обучения веса корректируются для минимизации ошибки.

Смещения (Biases) – дополнительные параметры, которые позволяют сдвигать функцию активации. Без смещений сеть была бы менее гибкой.

Функция потерь (Loss Function)

Функция потерь измеряет, насколько предсказания сети отличаются от истинных значений. Цель обучения — минимизировать эту функцию. Примеры:

- MSE (Mean Squared Error)для задач регрессии.
- Cross-Entropy Loss для задач классификации.
- Binary Cross-Entropy для бинарной классификации.

Оптимизатор

Оптимизатор — это алгоритм, который корректирует веса и смещения сети для минимизации функции потерь. Наиболее популярные оптимизаторы: SGD (Stochastic Gradient Descent), Adam, RMSprop

Обратное распространение ошибки (Backpropagation)

Это ключевой механизм обучения нейронных сетей. Алгоритм включает:

1. Прямой проход (Forward Pass) для вычисления выходных значений сети.
2. Вычисление ошибки – сравнение предсказаний с истинными значениями через функцию потерь.
3. Обратный проход (Backward Pass) – расчет градиентов функции потерь относительно весов и смещений.
4. Обновление параметров – корректировка весов и смещений с использованием оптимизатора.

Архитектуры нейронных сетей

Различные типы нейронных сетей используются для разных задач:

- Полносвязные сети (Fully Connected Networks) - все нейроны одного слоя соединены со всеми нейронами следующего слоя, например, многослойный перцептрон.
- Сверточные сети (Convolutional Neural Networks, CNN) используются для анализа изображений благодаря сверткам.
- Рекуррентные сети (Recurrent Neural Networks, RNN) обрабатывают последовательные данные (например, текст или временные ряды).
- Генеративные сети (Generative Adversarial Networks, GAN) используются для генерации новых данных.
- Трансформеры (Transformers) широко применяются в задачах обработки естественного языка.

Регуляризация

Регуляризация помогает предотвратить переобучение (overfitting):

- L1/L2-регуляризация накладывает штраф на величину весов.
- Dropout – случайным образом отключает часть нейронов во время обучения.
- Batch Normalization нормализует данные внутри сети для стабилизации обучения.

Гиперпараметры

Гиперпараметры — это параметры, которые настраиваются перед обучением:

- Размер батча (batch size),
- Скорость обучения (learning rate),
- Количество эпох (epochs),
- Архитектура сети (число слоев и нейронов).

Нейронные сети — это мощный инструмент машинного обучения, который состоит из множества взаимосвязанных элементов. Часть их них мы рассмотрим в этой лекции

Функции активации

Функция активации нейрона определяет выходной сигнал. Он определяется входным сигналом или набором входных сигналов. Функцию активации используют, чтобы получить выходные данные узла.

Функции активации делятся на два типа:

- линейные — не могут аппроксимировать нелинейные функции;
- нелинейные — используются в большинстве современных сетей.

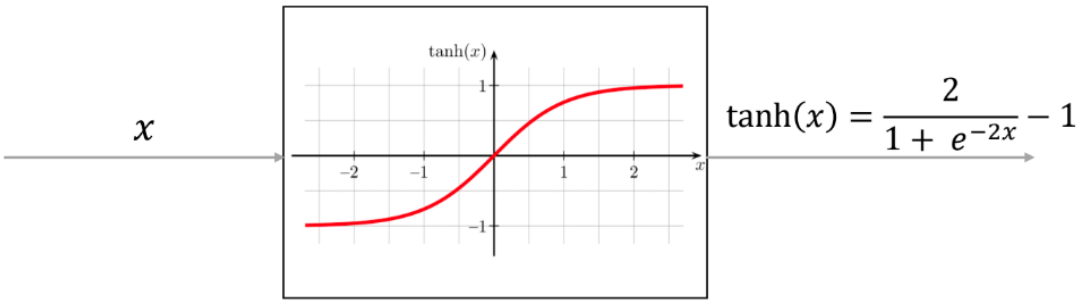
1. **Сигмоида (Sigmoid)** — возрастающая нелинейная функция, имеющая форму буквы S.

При использовании в сетях с большим числом слоёв нейроны с сигмоидой могут насыщаться и приводить к угасающим градиентам — то есть сеть перестаёт обучаться. Другая сложность — нецентрированность в нуле: значения функции попадают в диапазон (0,1), что может усложнять обучение.

Кроме того, вычисления с экспонентой требуют много ресурсов и времени.

2. **Функция Tanh** — гиперболический тангенс.

   Значения функции попадают в диапазон (-1,1) и центрированы в нуле — это отличает от сигмоиды. Но сохраняется проблема с затухающими градиентами при больших или малых значениях.



Здесь и далее — изображения Geek Brains

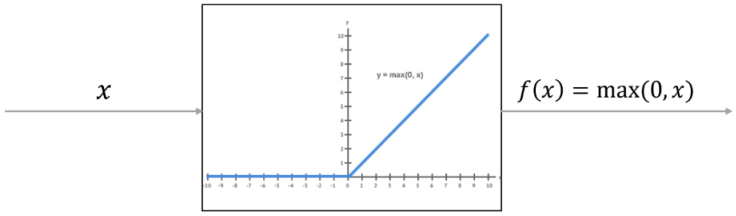
3. **ReLU (rectified linear unit)** — наиболее распространённая функция активации. Она используется практически во всех свёрточных нейронных сетях и для Deep Learning.

$$f(x) = max(0, x)$$

   ReLU быстро вычисляется, поскольку, в отличие от сигмоиды, не содержит экспоненты.

   Градиент равен 1 для положительных x и 0 для отрицательных — то есть проблема с затуханием градиента не возникает.

   ReLU хорошо ускоряет обучение.



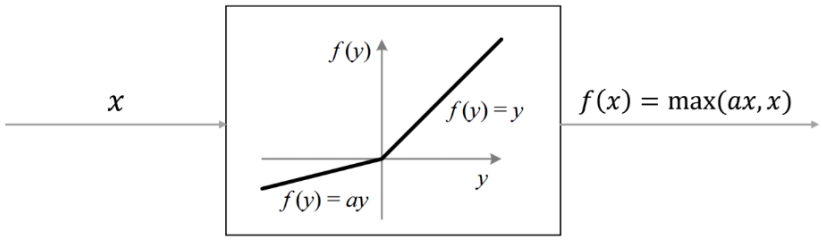
Но у ReLU есть недостатки. Если вход $x <= 0$, то и градиенты равны нулю, а значит, нейрон перестаёт обновляться. Функция не центрирована в нуле и выход нейрона всегда неотрицательный, что может усложнить обучение.

Чтобы решить проблему выхода ReLU, создали **Leaky ReLU**.

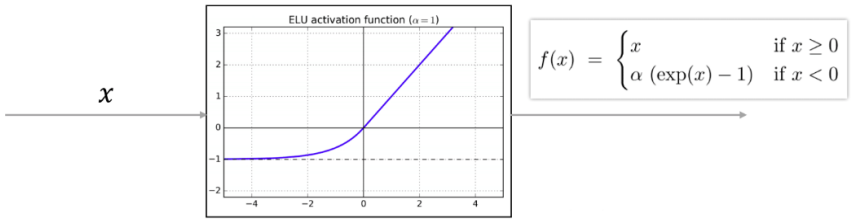
$f(x) = max(\alpha \cdot x, x)$, где α — небольшой коэффициент.

Характеристики Leaky ReLU-активации:

- всегда будут обновления,
- центрирована примерно в нуле,
- $\alpha \neq 1$.

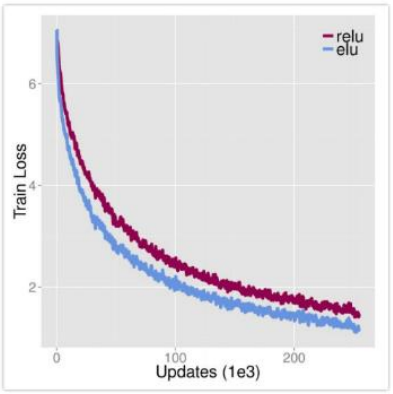


Функция активации ELU (Exponential Linear Unit), по результатам исследований, быстрее сводит к нулю и даёт более точные результаты. В отрицательной части аргументов использует экспоненту.



ELU центрирована в нуле — это помогает ускорить обучение. Для отрицательных значений создаётся плавный наклон, что предотвращает появление «мёртвых» нейронов.

Несмотря на присутствие экспоненты, ELU обеспечивает быструю сходимость, но требует дополнительных ресурсов.



C.M. Bartle / tungmphung.com

Кроме этих функций, отметим функций Softmax. Она преобразует значения в вероятности классов и используется в последнем слое для задач многоклассовой классификации.

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Функция	**Диапазон**	**Градиенты при x→±∞**	**Центрированность**	**Вычислительная сложность**	**Основные проблемы**
Sigmoid	(0, 1)	Затухают	Нет	Высокая	Затухание градиентов
Tanh	(-1, 1)	Затухают	Да	Высокая	Затухание градиентов
ReLU	[0, ∞)	Не затухают (x>0x > 0x>0)	Нет	Низкая	«Мёртвые» нейроны

Leaky ReLU	(-∞, ∞)	Не затухают (x>0x > 0x>0)	Нет	Низкая	Нецентрированность
ELU	(-α, ∞)	Не затухают	Да	Средняя	Вычисление экспоненты
Softmax	(0, 1)	Не затухают	Да	Высокая	Чувствительность к выбросам

Инициализация весов нейронной сети

Инициализация весов — важный этап, который определяет успешное обучение нейронной сети. Инициализация весов происходит в начале обучения, когда нейросети нужно взять какие-то значения, которые необходимо корректировать во время обучения на тренировочных данных. Неправильный выбор начальных весов может привести к медленной сходимости или не позволит обучить модель.

Вы уже знаете, что нельзя инициализировать веса нулями. Если все веса в нейросети изначально равны нулю, градиенты для всех параметров будут одинаковыми. Это приводит к симметрии: все нейроны обучаются одинаково, и это делает сеть менее мощной. Чтобы избежать этого, нужно добавить случайный шум в начальные веса.

Выбор масштаба шума

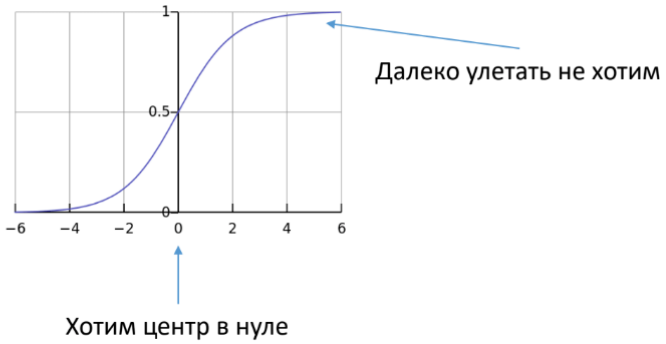
Мы генерируем случайные веса: например, из стандартного нормального распределения (среднее = 0, дисперсия = 1). Но слишком большие значения могут ухудшить обучение, поэтому веса нужно умножить на маленькое число, например, 0.01 или 0.001. Но чтобы сеть работала корректно, важно выбрать правильный масштаб.

Нейрон до активации

Линейная комбинация входов нейрона (до применения активации) определяется как сумма произведений входов и весов x_i w_i. Если входы отцентрированы (их среднее равно нулю) и веса также имеют среднее 0, то среднее значение выхода линейной комбинации тоже будет 0. Но важно учитывать **дисперсию** выхода, чтобы избежать проблемы с активацией.

Значение дисперсии

Если дисперсия слишком большая, то значения линейной комбинации выйдут за «рабочую зону» функции активации. Например, для сигмоиды это приводит к насыщению, где градиенты близки к нулю, а обучение замедляется. Чтобы избежать этого, нужно контролировать масштаб весов.



Вычисление подходящего масштаба

Чтобы сохранить дисперсию выхода нейрона на уровне дисперсии его входов, нужно нормализовать веса.
Если входы и веса имеют одинаковую дисперсию σ^2 , то дисперсия линейной комбинации зависит от числа входов nnn и выражается как $n \cdot \sigma^{2n}$. Чтобы дисперсия не росла с увеличением числа входов, веса масштабируются на коэффициент $1/\sqrt{n}$.
Практические методы инициализации

1. **Инициализация Хавьера (Glorot Initialization)** подходит для функций активации, подобных гиперболическому тангенсу. Коэффициент масштабирования равен $\sqrt{2/(n_{in} + n_{out}}$), где n_{in} — число входов, n_{out}— число выходов.
2. **Инициализация Хе (He Initialization)** используется для функций ReLU. Коэффициент масштабирования равен $\sqrt{\frac{2}{n_{in}}}$, где n_{in} — число входов.

Корректная инициализация помогает сохранить стабильную дисперсию активаций и градиентов на всех слоях. Это предотвращает затухание или взрыв градиентов (происходит, когда значения градиентов, которые используются для обновления весов модели, становятся слишком большими) и ускоряет обучение. Например, с инициализацией Хавьера распределение градиентов остаётся примерно одинаковым на всех слоях — это способствует быстрой сходимости.

Инициализация весов — ключевой шаг для построения эффективно обучающихся нейронных сетей.

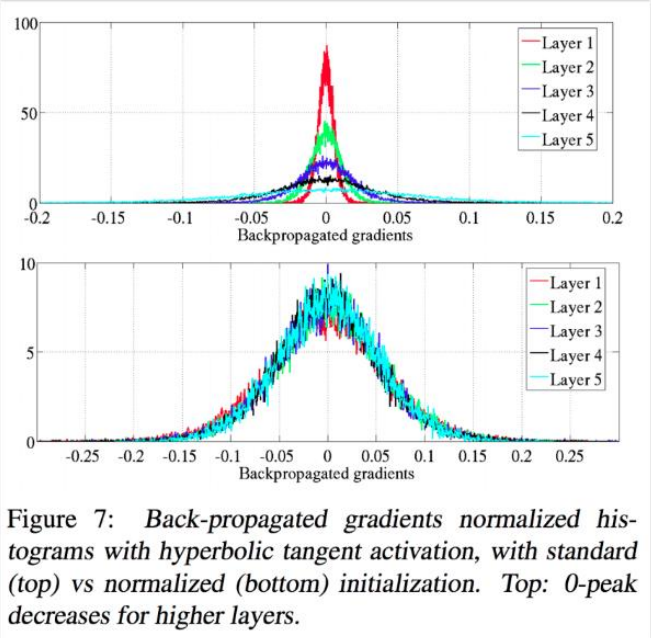


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Xavier Glorot, Yoshua Bengio / researchgate.net

Функции потерь и оптимизация

Нейронная сеть представляет из себя большой набор матриц, на которые последовательно умножается входной вектор. Числа в этих матрицах носят название параметры сети.

Окончательным выходом нейронной сети служит некоторый иной тензор произвольной формы, зависящей от её архитектуры. Это может быть тензор высокого ранга, вектор или просто число. Сам выход нейронной сети служит входными данными для некоторой функции. Можно сказать, что оптимизация параметров нейронной сети — это процесс настройки изменения параметров сети таким образом, чтобы минимизировать значение этой функции для каждого возможного её входного тензора. В принципе, эта функция может быть любой.

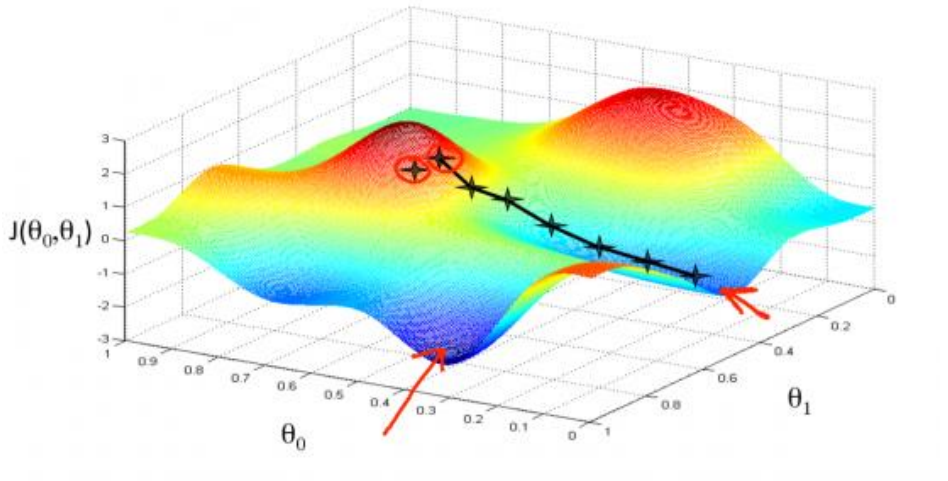
Например, выход последнего слоя нейронной сети может суммироваться и тогда процесс оптимизации будет сводится к минимизации этой суммы. Очевидно, что в таком случае решением является выставление таких весов чтобы сумма начала стремится к минус бесконечности. Если же, например, выход нейронной сети перед суммированием возводится в квадрат, то результатом оптимизации параметров нейронной сети будет 0.

В реальных задачах функция, применяемая к выходу нейронной сети подобрана таким образом, чтобы оценивать производительность сети в некоторой задаче. Например, насколько хорошо сеть предсказывает или классифицирует данные и носит имя функции ошибки или функции потерь.

Если мы рассмотрим каждый параметр как независимую переменную, тот можем представить себе пространство очень высокой размерности, каждая ось которого соответствует одному из параметров. Для наглядного представления, как изменение параметров влияет на поведение модели и её оценку ошибок, можно использовать визуализации. Эти визуализации часто изображают функцию потерь как ландшафт, где оси соответствуют параметрам модели, а высота каждой точки — значению функции потерь при этих параметрах.

Параметры нейросети играют роль координат в этом пространстве. То есть в реальности пространство параметров нейронной сети – это пространство не с 3 измерениями как привычное нам, а с миллионами независимых измерений. В процессе обучения, изменения этих параметров исследуют пространство возможных решений в поисках оптимальной конфигурации, которая минимизирует ошибку на обучающих данных.

Такая визуализация помогает понять, какие параметры приводят к высокой ошибке (высокие точки на ландшафте) и какие минимизируют её (низкие точки или «долины» на ландшафте). Задача оптимизации таким образом сведётся к тому, чтобы, начав двигаться со случайно точки этого ландшафта прийти в итоге к минимуму.



Конечно, самым простым способом нахождение минимума было бы не двигаться, а сразу переместится в точку с наименьшим значением функции. То есть, в идеальном случае необходимо было бы проверить все точки пространства и просто выбрать ту, в которой значение функции минимально. Разумеется, это не представляется вычислительно возможным, так как количества возможных комбинаций параметров чрезвычайно велико. На самом деле оно настолько велико что никакие сравнения с размерами вселенной или числом атомов в ней будут неуместны. Это число гораздо больше любого числа, встречающегося в реальном мире при счёте объектов. Вместо того, чтобы перебирать параметры нейронной сети оптимизация осуществляется итеративно и похожа на движение по ландшафту, в котором тело пытается найти самую низкую точку в долине катясь по действию силы тяжести.

Как же осуществляется этот алгоритм поиска на практике? Градиент функции многих переменных – это вектор, каждая компонента которого является частной производной

функции по одной из её переменных. Градиент указывает направление наискорейшего возрастания значения функции. Соответственно, противоположное градиенту направление указывает на наискорейшее убывание функции, что используется в алгоритмах оптимизации для поиска минимума функции. Основная идея градиентного спуска заключается в обновлении параметров модели в направлении, противоположном градиенту функции потерь по этим параметрам. Это делается для того, чтобы шаг за шагом приближаться к минимуму функции потерь. Математически это можно описать следующей формулой обновления параметров:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla J(\theta)$$

θ параметры модели,
η скорость обучения или learning rate
$\nabla J(\theta)$ — градиент функции потерь

Скорость обучения является важным гиперпараметром:

- если она слишком мала, процесс обучения будет проходить очень медленно;
- если слишком велика – модель может «перескакивать» через минимум или даже расходиться.

SGD

В теории всё кажется простым, но как в принципе определить производную функции потерь по миллионам параметров нейронной сети, найти градиент и определить направление изменения параметров, ведь производная сложной функции достаточно сложна для вычисления? Для этого применяется обратное распространение ошибки (**backpropagation**). В его основе лежит цепное правило из дифференциального исчисления.

Цепное правило позволяет выразить производную сложной функции через производные составляющих её функций. В контексте нейронных сетей, это означает, что производная функции потерь по отношению к весу или смещению в любом слое может быть выражена через производные функций активации и других операций, проводимых в сети. Процесс обратного распространения начинается с последнего слоя сети, где первоначально вычисляется производная функции потерь по выходным значениям сети. Затем, применяя цепное правило, производные передаются к предыдущим слоям. Например, если у нас есть двуслойная сеть с последовательно соединёнными слоями, где выход сети и функция потерь определяется следующим образом:

$$y = f(g(x)) \text{ и } L = L(y, \text{target}),$$

то производная функции потерь по весам первого слоя g будет вычисляться как:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial g} \cdot \frac{\partial g}{\partial w}$$

Таким образом, градиенты «проталкиваются» обратно через сеть от выхода к входу, что позволяет пошагово адаптировать веса для минимизации потерь. Это последовательное применение цепного правила и делает обратное распространение эффективным способом для обучения глубоких нейронных сетей.

Большинство алгоритмов оптимизации основано на методе градиентного спуска, который обновляет параметры модели в направлении, противоположном градиенту функции потерь.

Выбор подходящего алгоритма оптимизации зависит от конкретной задачи, структуры нейронной сети и характеристик данных. Поэтому при создании нейронной сети необходимо подбирать алгоритм оптимизации — это отдельный гиперпараметр всего алгоритма.

Есть разные варианты оптимизации через градиентный спуск, в этом материале рассмотрим три из них.

Существует несколько разновидностей градиентного спуска, каждая из которых имеет свои особенности и области применения. При стандартном градиентном спуске обновление параметров происходит после вычисления градиента по всему набору данных. Это может быть очень неэффективно на больших данных.

Вычислить градиентный спуск — дорогая и долгая задача, поэтому в нейросетях используются большие выборки данных. Найденное решение — стохастический градиентный спуск (Stochastic Gradient Descent — SGD).

При использовании **стохастического градиентного спуск (SGD)** – параметры обновляются для каждого обучающего примера. Это тратит меньше ресурсов, но процесс обновления становится более зашумлённым.

Преимущества SGD:

- Скорость. SGD быстрее, так как вычисляет градиенты по меньшему количеству данных.
- Шум в обновлениях. Стохастичность позволяет алгоритму «выбираться» из локальных минимумов, улучшая поиск глобального минимума.

Недостатки SGD:

- Шум. Параметры обновляются с высокой вариативностью — это делает процесс менее стабильным.
- Медленная сходимость. Стохастичность может затруднять приближение к оптимальному решению.

На практике чаще используют **мини-батч SGD,** где градиент вычисляется на небольших подмножествах данных (мини-батчах), а не на одном примере:

$\Theta_{t+1} - \eta \cdot \frac{1}{m} \sum_{i=1}^{m} \quad \nabla L(\Theta_t, x_i, y_i)$, где m — размер мини-батча.

Мини-батч градиентный спуск – компромисс между двумя предыдущими методами. Этот метод сочетает преимущества предыдущих двух, позволяя более эффективно масштабировать процесс обучения. Само подмножество называется батчем.

Преимущества мини-батча SGD:

- снижает уровень шума по сравнению с чистым SGD,
- позволяет эффективно использовать возможности современных процессоров и GPU.

Размер мини-батча (m) влияет на эффективность.

Маленький мини-батч (1–32):

- близок к SGD,
- имеет высокий уровень шума в обновлениях,
- хорошо подходит для задач с ограниченными вычислительными ресурсами.

Средний мини-батч (32–256):

- оптимальный вариант для большинства задач,
- хороший компромисс между точностью и скоростью вычислений.

Большой мини-батч (> 512):

- становится ближе к полному градиентному спуску,
- может терять способность избегать локальных минимумов из-за уменьшенного шума.

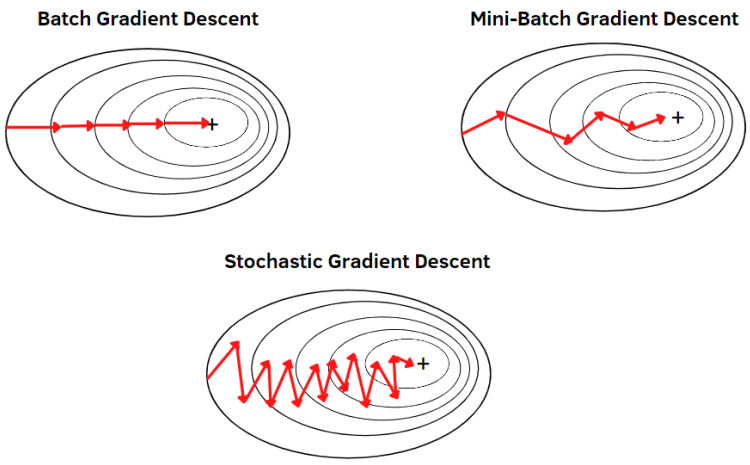
Рассмотрим на примере:

```
import numpy as np

# Функция потерь
def loss_fn(w, X, y):
    preds = np.dot(X, w)
    return np.mean((preds - y) ** 2)

# Градиент функции потерь
def gradient_fn(w, X, y):
    preds = np.dot(X, w)
    return np.dot(X.T, (preds - y)) / len(y)

# Данные
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5], [5, 6]])  # Матрица
признаков
y = np.array([5, 7, 9, 11, 13])  # Целевые значения
w = np.random.randn(2)  # Инициализация весов
learning_rate = 0.01
batch_size = 2  # Размер мини-батча
epochs = 100
```

```
# Mini-batch SGD
for epoch in range(epochs):
    # Перемешиваем данные
    indices = np.arange(len(X))
    np.random.shuffle(indices)
    X, y = X[indices], y[indices]

    # Обучение по мини-батчам
    for i in range(0, len(X), batch_size):
        X_batch = X[i:i + batch_size]
        y_batch = y[i:i + batch_size]

        # Вычисляем градиент и обновляем веса
        grad = gradient_fn(w, X_batch, y_batch)
        w = w - learning_rate * grad

    # Печатаем потери каждые 10 эпох
    if (epoch + 1) % 10 == 0:
        print(f"Эпоха {epoch + 1}, Потери: {loss_fn(w, X, y):.4f}")

print("Обученные веса:", w)
```

Когда использовать мини-батч SGD

- Большие данные: если данные не помещаются в память, их удобно разделить на батчи.
- Нейронные сети: большинство глубоких моделей обучаются с использованием мини-батчей.
- Оптимизация на GPU: векторные операции с мини-батчами эффективно используют аппаратное ускорение.



Akash / analyticsvidhya.com

RMSProp

RMSProp — это один из популярных алгоритмов оптимизации, который создали, чтобы улучшить стандартный стохастический градиентный спуск, особенно в задачах с нестационарными и переменными градиентами, как, например, в обучении нейронных сетей.

Значение RMSProp

RMSProp регулирует шаг обучения в зависимости от величины недавних градиентов для каждого параметра, чтобы не использовать один и тот же коэффициент шага для всех параметров модели, как в традиционном градиентном спуске. Если градиент для параметра велик, то шаг уменьшится, а если мал — шаг увеличится.

Такой подход позволяет ускорить сходимость.

Алгоритм основан на вычислении среднего квадрата градиента для каждого параметра — это скользящее среднее используется для корректировки скорости обучения.

Обновление параметров с использованием RMSProp выражается следующим образом:

$$v_t = \beta v_{t-1} \ + (1-\beta) {g_t}^2 v_t$$

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

где:

- v_t — среднее значение квадрата градиента на шаге t;
- β — коэффициент сглаживания (обычно около 0,9);
- g_t — градиент в момент времени t;
- η — коэффициент обучения (learning rate);
- ϵ — маленькая константа (например, $1e^{-8}$), которая добавляется, чтобы предотвратить деление на ноль.

Пояснение:

- v_t — это накопленное среднее квадратичное значение градиентов. Он помогает уменьшить влияние очень крупных градиентов, которые могут привести к нестабильности обучения.
- **Корректировка шага** — в следующем обновлении веса корректируются с учётом масштаба градиента, деля градиент на корень из среднего квадрата.
- β — этот параметр контролирует «память» алгоритма. Чем больше значение β, тем больше веса даются предыдущим значениям градиента. Обычно β выбирается около 0,9.

Преимущества RMSProp:

- Адаптивное изменение шага обучения. RMSProp изменяет шаг обучения на основе истории градиентов. Это позволяет справляться с ситуациями, когда градиенты имеют разные масштабы по направлениям параметров. Изменять шаг обучения полезно, когда градиенты колеблются или имеют большой диапазон.
- Устойчивость к нестабильности. Использование среднего квадрата градиента делает алгоритм более устойчивым, чем стандартный SGD, который может сильно колебаться, особенно в случае сложных и нестабильных функций потерь.
- Эффективность в задачах с нелинейными зависимостями. RMSProp хорошо работает в задачах с глубокими нейронными сетями, где градиенты могут быть очень большими или малыми на разных уровнях.

Рассмотрим пример:

```
import numpy as np

# Функция потерь
def loss_fn(w, X, y):
    preds = np.dot(X, w)
    return np.mean((preds - y)**2)

# Градиент функции потерь
def gradient_fn(w, X, y):
    preds = np.dot(X, w)
    return np.dot(X.T, (preds - y)) / len(y)

# Данные
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5], [5, 6]])  # Матрица признаков
y = np.array([5, 7, 9, 11, 13])                  # Целевые значения
w = np.random.randn(2)                           # Инициализация весов
learning_rate = 0.01
beta = 0.9                                  # Коэффициент для скользящего среднего
epsilon = 1e-8                              # Для предотвращения деления на ноль
epochs = 100

# Инициализация v (среднего квадрата градиентов)
v = np.zeros_like(w)

# Обучение с использованием RMSProp
for epoch in range(epochs):
    # Вычисляем градиент
    grad = gradient_fn(w, X, y)

    # Обновляем среднее квадрата градиентов
    v = beta * v + (1 - beta) * grad**2

    # Обновляем веса
    w -= learning_rate * grad / (np.sqrt(v) + epsilon)

    # Печатаем потери каждые 10 эпох
    if (epoch + 1) % 10 == 0:
        print(f"Эпоха {epoch + 1}, Потери: {loss_fn(w, X, y):.4f}")

print("Обученные веса:", w)
```

RMSProp хорошо подходит для задач, в которых:

- градиенты сильно колеблются и имеют разный масштаб в разных направлениях;
- необходимо обучение нейронных сетей с нелинейными функциями активации и важно контролировать и уменьшать слишком большие градиенты;
- есть проблемы с нестабильностью градиентов, как это бывает при обучении глубоких нейронных сетей.

ADAM

Adam (Adaptive Moment Estimation) — один из самых популярных и эффективных алгоритмов оптимизации, которые используются для обучения нейронных сетей. Adam сочетает преимущества двух методов — **AdaGrad** (адаптивное обучение на основе истории градиентов) и **RMSProp** (адаптивное масштабирование градиентов) — и делает обучение более стабильным и эффективным.

Adam использует два скользящих средних:

- **первый момент (средний градиент)** оценивает направление и величину градиента (аналогично Momentum);
- **второй момент (средний квадрат градиента)** оценивает масштаб градиента (аналогично RMSProp).

Эти два момента помогают алгоритму адаптировать шаг обучения для каждого параметра, делая его быстрым на начальных этапах обучения и устойчивым к шуму в градиентах.

Формулы обновления Adam:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t,$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) {g_t}^2,$$

$$\widehat{m_t} = \frac{m_t}{1 - {\beta_1}^2},$$

$$\widehat{v_t} = \frac{v_t}{1 - {\beta_2}^t},$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\widehat{v_t}} + \epsilon} \widehat{m_t},$$

где:

- g_t — текущий градиент параметра θ;
- m_t — скользящее среднее градиента (первый момент);
- v_t — скользящее среднее квадрата градиента (второй момент);
- $\widehat{m_t}$— корректированное значение m_t;

- $\widehat{v_t}$ — корректированное значение v_t;
- η — шаг обучения (learning rate);
- ϵ — маленькое число, которое предотвращает деление на ноль (обычно 10^{-8});
- β_1 и β_2 — коэффициенты сглаживания (обычно β_1 = 0,9, β_2= 0,999).

Пояснение:

- **Коррекция смещения** — $\widehat{v_t}$ и $\widehat{m_t}$ учитывают, что на начальных итерациях m_t и v_t имеют сильное смещение к нулю.
- **Динамическое регулирование шага обучения** — используя $\sqrt{\widehat{v_t}}$, Adam автоматически регулирует шаг обучения для каждого параметра.

Преимущества Adam:

- Адаптивное обучение. Adam подстраивает шаг обучения для каждого параметра в зависимости от амплитуды его градиента.
- Быстрая сходимость. Adam быстро сходится в начальных этапах обучения благодаря использованию первого момента (Momentum).
- Устойчивость к шуму. Второй момент (средний квадрат градиента) помогает избежать перепробивания в шумных направлениях.
- Работает «из коробки». Значения гиперпараметров β_1, β_2, η и ϵ по умолчанию хорошо подходят для большинства задач.
- Универсальность. Adam работает с разреженными данными и глубокими нейронными сетями.

Недостатки Adam:

- Сходимость к неидеальным решениям. Adam может сходиться к локальным минимумам или седловым точкам, не являясь глобально оптимальным.
- Проблема с генерализацией. Модели, которые обучены с Adam, иногда хуже обобщаются на тестовых данных, чем модели, которые обучены SGD с импульсом (Momentum).
- Решение проблемы с долговременной памятью. Adam не всегда хорошо работает с задачами, в которых нужно учитывать долгосрочные зависимости.

Рассмотрим пример:

```
import numpy as np

# Функция потерь
def loss_fn(w, X, y):
    preds = np.dot(X, w)
    return np.mean((preds - y)**2)

# Градиент функции потерь
```

```
def gradient_fn(w, X, y):
    preds = np.dot(X, w)
    return np.dot(X.T, (preds - y)) / len(y)

# Данные
X = np.array([[1, 2], [2, 3], [3, 4], [4, 5], [5, 6]])  # Матрица
признаков
y = np.array([5, 7, 9, 11, 13]) # Целевые значения
w = np.random.randn(2)# Инициализация весов
learning_rate = 0.01
beta1, beta2 = 0.9, 0.999
epsilon = 1e-8
epochs = 100

# Инициализация моментов
m = np.zeros_like(w)
v = np.zeros_like(w)

# Обучение с использованием Adam
for epoch in range(epochs):
    # Вычисляем градиент
    grad = gradient_fn(w, X, y)

    # Обновляем моменты
    m = beta1 * m + (1 - beta1) * grad
    v = beta2 * v + (1 - beta2) * (grad**2)

    # Коррекция смещения
    m_hat = m / (1 - beta1**(epoch + 1))
    v_hat = v / (1 - beta2**(epoch + 1))

    # Обновляем веса
    w -= learning_rate * m_hat / (np.sqrt(v_hat) + epsilon)

    # Печатаем потери каждые 10 эпох
    if (epoch + 1) % 10 == 0:
        print(f"Эпоха {epoch + 1}, Потери: {loss_fn(w, X, y):.4f}")

print("Обученные веса:", w)
```

Обучение нейронной сети

Решим задачу распознавания цифр с помощью PyTorch.

1. Устанавливаем и импортируем необходимые библиотеки:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torchvision import transforms
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
```

2. Проверяем версию библиотек:

```
print(f"Using PyTorch version: {torch.__version__}")
```

3. Проверяем, есть ли доступ к GPU:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Device: {device}")
```

4. Скачиваем данные и делим их на тренировочную и валидационную выборки:

```
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,), (0.5,))])

train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
transform=transform, download=True)

test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
transform=transform, download=True)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

5. Проверяем, какие данные хранятся в датасете:

```
examples = iter(train_loader)
example_data, example_targets = next(examples)
fig, axes = plt.subplots(1, 5, figsize=(10, 5))

for i in range(5):
      axes[i].imshow(example_data[i][0], cmap="gray")
      axes[i].set_title(f"Label: {example_targets[i].item()}")
      axes[i].axis('off') plt.show()
```

Предобработка данных

1. Преобразуем изображение в вектор:

```
x_train_flat = x_train.view(-1, 28 * 28).float()
print(x_train.shape, x_train_flat.shape)

x_val_flat = x_val.view(-1, 28 * 28).float()
print(x_val.shape, x_val_flat.shape)
```

2. Нормализуем данные и кодируем таргет:

```
# Центрируем и нормализуем данные

x_train_float = (x_train_flat / 255.0) - 0.5  # Приводим к диапазону [-0.5, 0.5]
x_val_float = (x_val_flat / 255.0) - 0.5
print(x_train_float.shape, x_val_float.shape)
# Преобразуем метки в one-hot-формат

y_train_oh = F.one_hot(torch.tensor(y_train), num_classes=10)
y_val_oh = F.one_hot(torch.tensor(y_val), num_classes=10)


print(y_train_oh.shape)
print(y_train_oh[:5], y_train[:5])
```

Определение архитектуры сети

```
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()

    self.flatten = nn.Flatten()
    self.fc1 = nn.Linear(28 * 28, 256)  # Входные данные → 256 нейронов
    self.fc2 = nn.Linear(256, 128)  # 256 → 128 нейронов
    self.fc3 = nn.Linear(128, 10)  # 128 → 10 нейронов (количество классов)
    self.relu = nn.ReLU()  # Активация ReLU
    self.softmax = nn.Softmax(dim=1)  # Активация softmax для выхода

    def forward(self, x):
        x = self.flatten(x)  # Преобразуем входное изображение в одномерный вектор
        x = self.relu(self.fc1(x))  # Пропускаем через первый слой и ReLU
        x = self.relu(self.fc2(x))  # Пропускаем через второй слой и ReLU
        x = self.softmax(self.fc3(x))  # Пропускаем через выходной слой и применяем softmax
        return x
```

Ключевые элементы

1. **nn.Module –** базовый класс для всех моделей в PyTorch. Любая нейронная сеть или её часть (например, слой) является наследником nn.Module.

Зачем он нужен

- Организует слои и вычисления в одной структуре.
- Автоматически отслеживает параметры модели и их обновления.

- Позволяет легко сохранять и загружать модели.
- Предоставляет методы, такие как .train() и .eval(), для переключения режимов работы (обучение или оценка).

Как использовать

- Наследуем ваш класс от nn.Module.
- Определяем слои внутри конструктора __init__.
- Реализуем прямой проход (forward pass) в методе forward.

2. **nn.Flatten()** преобразует входное изображение 28 × 28 в вектор размерности 784 для подачи в полносвязный слой. Полносвязные слои (nn.Linear) принимают только одномерные данные. Flatten превращает двумерные и многомерные тензоры в формат, подходящий для дальнейших вычислений.
3. **nn.Linear() –** полносвязный слой (или слой с линейной зависимостью), в котором каждый вход связан с каждым выходом.

Выполняет линейное преобразование входных данных:$y = xW^T + b$, где:

- W^T — транспонированная матрица весов;
- b — вектор смещения;
- x — входной вектор.

Полносвязный слой используется для извлечения признаков или создания конечных предсказаний в классификационных задачах.

Параметры:

- in_features — размер входного вектора;
- out_features — размер выходного вектора;
- bias — добавлять смещение или нет (по умолчанию True).

4. **ReLU()**

Ускоряет обучение, оставляя положительные значения без изменений и обнуляя отрицательные. Добавляет нелинейность в модель, что позволяет сети лучше приближать сложные функции и устраняет проблему исчезающего градиента, которая часто возникает при использовании сигмоидных активаций.

5. **Softmax(dim=1)**

Преобразует выход сети в вероятности классов. Softmax — функция активации, которая преобразует сырые выходы модели в вероятности принадлежности классу. Она используется на выходном слое моделей для задач классификации, чтобы интерпретировать выход как распределение вероятностей между классами.

Для каждого элемента вычисляется по формуле: $f_i(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$, где x_i — значение на выходе сети для класса i.

Dim указывает размерность, вдоль которой выполняется softmax. Обычно dim = 1, так как входной тензор имеет форму (N, C), где N — размер батча, а C — количество классов.

```
model = SimpleNN().to(device)
print(model)
```

to(device): отправляем модель на устройство (CPU или GPU).

Определение функции потерь и оптимизатора

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())
```

nn.CrossEntropyLoss(): функция потерь для задач классификации, объединяет softmax и вычисление отрицательной логарифмической вероятности.

optim.Adam: адаптивный оптимизатор, который автоматически настраивает скорость обучения для каждого параметра. Далее в курсе мы разберём его работу подробнее.

Цикл обучения

```
for epoch in range(num_epochs):
    model.train()  # Переводим модель в режим обучения
    train_loss, correct = 0, 0
    for X_batch, y_batch in train_loader:  # Итерируемся по мини-батчам
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)

    # Forward pass
    outputs = model(X_batch)  # Прогон через сеть
    loss = criterion(outputs, y_batch)  # Вычисление ошибки

    # Backward pass
    optimizer.zero_grad()  # Обнуляем градиенты
    loss.backward()  # Считаем градиенты
    optimizer.step()  # Обновляем параметры модели

    train_loss += loss.item()  # Суммируем потери
    _, preds = torch.max(outputs, 1)  # Находим предсказания
    correct += (preds == y_batch).sum().item()  # Считаем количество верных
предсказаний

    train_loss /= len(train_loader)  # Средняя потеря на батч
    train_acc = correct / len(train_loader.dataset)  # Точность на обучении
    train_loss_history.append(train_loss)
    train_acc_history.append(train_acc)
```

Ключевые шаги

1. **Model.train()** — перевод модели в режим обучения.
2. **Прямой проход**:
    - прогон входных данных через сеть,
    - вычисление потерь с помощью функции `criterion`.

3. **Обратное распространение**:
    - обнуление градиентов,
    - расчёт новых градиентов через `loss.backward()`,
    - обновление параметров с помощью `optimizer.step()`.
4. **Подсчёт точности**:
    - находим метки с максимальной вероятностью через `torch.max`,
    - сравниваем предсказания с истинными метками.

Оценка на тестовых данных

```
model.eval()  # Переводим модель в режим оценки
val_loss, correct = 0, 0
with torch.no_grad():  # Отключаем вычисление градиентов для ускорения
      for X_batch, y_batch in test_loader:
      X_batch, y_batch = X_batch.to(device), y_batch.to(device)
      outputs = model(X_batch)
      loss = criterion(outputs, y_batch)
      val_loss += loss.item()
      _, preds = torch.max(outputs, 1)
      correct += (preds == y_batch).sum().item() # Сравнение предсказанных
классов с истинными метками

val_loss /= len(test_loader)
val_acc = correct / len(test_loader.dataset)
val_loss_history.append(val_loss)
```

Ключевые моменты:

1. **Model.eval()**. Переводит модель в режим оценки (evaluation mode). В этом режиме отключаются определённые механизмы, которые нужны только во время обучения, такие как:
    - **Dropout** — используется для регуляризации во время обучения. В режиме оценки все нейроны остаются активными. Во время обучения механизм Dropout случайно отключает часть нейронов, чтобы предотвратить переобучение. В режиме оценки необходимо использовать всю мощность сети без случайных отключений.
    - **Batch Normalization** — использует усреднённые значения и дисперсию, накопленные в процессе обучения, вместо вычисления на основе текущего батча.
2. **Torch.no_grad()** — контекстный менеджер, который отключает автоматическое вычисление градиентов в PyTorch. Во время обучения PyTorch хранит граф вычислений, чтобы позже использовать его для обратного распространения (backpropagation). В режиме оценки этот граф не нужен, так как мы не обновляем параметры. Torch.no_grad() отключает построение графа, освобождая память. За счёт исключения операций, связанных с вычислением градиентов, оценка модели становится быстрее.
3. **Подсчёт точности** аналогичен этапу обучения. После прямого прохода через модель мы получаем сырые выходы. Для классификации выходы преобразуются в

предсказания классов с помощью torch.max(outputs, dim=1) и выбирается индекс с максимальным значением вероятности (класс с наивысшей вероятностью).

4. **Сравнение предсказаний с истинными метками**

correct += (preds == labels).sum().item():

- preds == labels — возвращает тензор булевых значений (True/False);
- .sum() — считает количество True (правильных предсказаний);
- .item() — преобразует это значение в скаляр Python.

Сравнение этапов обучения и оценки

Этап	**Обучение (model.train())**	**Оценка (model.eval())**
Dropout **BatchNorm**	Активен Вычисляет по батчу	Отключён Использует накопленные значения
Градиенты **Цель**	Считаются Обновление параметров	Не считаются (torch.no_grad()) Оценка качества модели

Графики обучения

```
plt.figure(figsize=(12, 5))
plt.plot(train_acc_history, label='Train Accuracy')
plt.plot(val_acc_history, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Accuracy over Epochs')
plt.legend()
plt.show()

plt.figure(figsize=(12, 5))
plt.plot(train_loss_history, label='Train Loss')
plt.plot(val_loss_history, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss over Epochs')
plt.legend()
plt.show()
```

Тестирование модели

```
model.eval() # Перевод модели в режим оценки:
```

```
all_preds, all_labels = [], [] #списки для хранения предсказаний
with torch.no_grad(): #отключение вычисления градиентов
      for X_batch, y_batch in test_loader: #перебор батчей из test_loader
      X_batch = X_batch.to(device) #перенос данных на устройство
      outputs = model(X_batch) #получение предсказаний
      _, preds = torch.max(outputs, 1) # определение предсказанных классов
      all_preds.extend(preds.cpu().numpy()) #сохранение предсказаний
      all_labels.extend(y_batch.numpy()) #сохранение меток

accuracy = accuracy_score(all_labels, all_preds)
print(f"Test Accuracy: {accuracy:.4f}")
```

Улучшение обучения

Batch-нормализация

В начале обучения с помощью backpropagation веса и распределения могут меняться. Иногда это приводит к проблемам, например медленной сходимости. Backpropagation может нарушать начальную настройку весов, ухудшая обучение.

Чтобы решить эту проблему, используется метод **Batch Normalization** (нормализация по батчам). Он заключается в нормализации выходных данных нейрона таким образом, чтобы у них было нулевое среднее значение и фиксированная дисперсия. Это позволяет поддерживать быструю скорость сходимости даже в процессе обучения.

Как работает нормализация

Рассмотрим выход нейрона h_i. Для нормализации нейрона его нужно:

- центрировать: вычесть среднее значение выходных данных нейрона;
- нормализовать: поделить на стандартное отклонение (или квадратный корень из дисперсии).

  $h_i = \gamma_i \frac{h_i - \mu_i}{\sqrt{\sigma_i^2}} + \beta_i h_i$

  Эти статистики — среднее значение и дисперсия — вычисляются на основе данных текущего батча.

Какие проблемы могут возникнуть?

В процессе обучения нейросеть изменяется, особенно в начале, когда веса случайные и сеть активно обучается. В это время статистики (среднее и дисперсия) могут значительно меняться, поэтому их нужно пересчитывать на основе последних данных, а не на старых.

Обычно для нормализации используется батч размером 32 или 64 объекта. Этого достаточно, чтобы статистики были адекватными и обучение шло эффективно. Но во время экспериментов можно брать батчи и больше — до 256 или 512.

Что делать на этапе тестирования

Представим случай, когда сеть используется для предсказаний: например, нужно оценить среднее и дисперсию на одной картинке, но данных для этого нет. В таком случае используется **экспоненциальное сглаживание** для оценки этих статистик.

Сначала среднее и дисперсия обновляются на основе предыдущих значений с весами, которые уменьшаются экспоненциально. Благодаря этому мы накапливаем данные с предыдущих шагов, чтобы использовать их для более точной оценки статистик в процессе тестирования.

Как работать с параметрами γ и β

Для более гибкой настройки активаций нейросети нужны параметры масштабирования и смещенияγ и β, которые позволяют нейросети:

- масштабировать выход нейрона, увеличивая или уменьшая дисперсию;
- смещать активацию, добавляя дополнительную степень свободы.

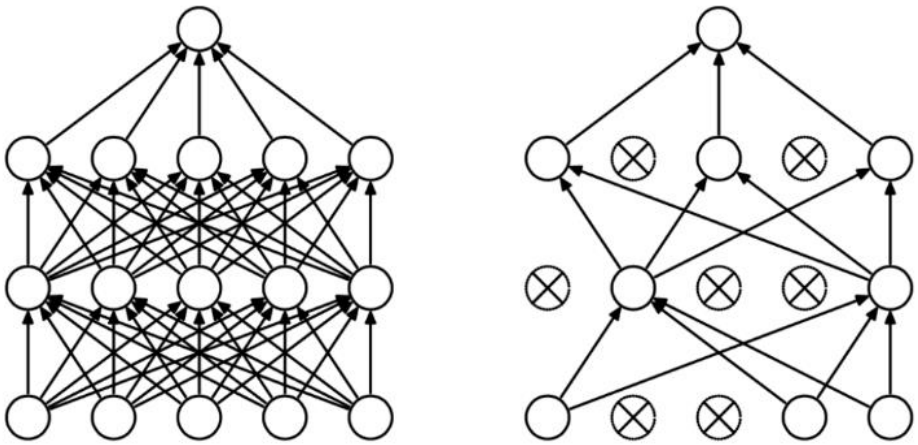
Обучение этих коэффициентов происходит так же, как и обучение весов — через градиентный спуск. Коэффициенты могут быть легко дифференцированы, и по ним можно также считать градиенты для оптимизации.

Batch Normalization решает проблему с изменением статистик в процессе обучения и улучшает сходимость нейросети. Метод основан на нормализации выходов нейронов и помогает избежать увеличения дисперсии, которое может замедлить обучение. Использование Batch Normalization особенно важно на больших данных, когда изменения сети в процессе обучения могут быть значительными.

Dropout-регуляризация

Регуляризация нейросетей нужна, чтобы предотвратить переобучение (overfitting). В современных нейросетях количество параметров может быть большим, и если сеть сложная, она может быстро «запомнить» тренировочные данные. Это приведёт к идеальному решению для этих данных, но плохой обобщающей способности. Поэтому нужно замедлить процесс сходимости, чтобы нейросеть могла лучше обобщать и показывать хорошее качество на неизвестных данных (например, на тестовом наборе).

Dropout — это техника регуляризации, которая случайным образом «выключает» (зануляет) нейроны в процессе обучения. На каждом шаге обучения с вероятностью P нейрон остаётся активным, а с вероятностью 1−P — «выключается» (становится равным нулю). Обычно значение P берут от 0,2 до 0,5. Это помогает создать множество подсетей, каждая из которых обучается на разных подмножествах сети. Это делает модель более устойчивой и менее склонной к переобучению.



Классическая нейросеть и нейросеть после использования Dropout. Boopathy Prabadevi, Natarajan Deepa, Ravi Raj Gulati, Latha Bhaskaran Krithika, Rajagopal Sivakumar / researchgate.net

В процессе обучения нейроны случайным образом «выключаются», и сеть обучается с меньшим количеством активных нейронов. Это приводит к тому, что веса, которые связаны с выключенными нейронами, не участвуют в процессе обучения, из-за чего статистика модели может изменяться. Когда сеть используется для предсказаний на тестовых данных, нужно компенсировать эту случайную потерю нейронов.

Чтобы привести веса к ожидаемому значению после Dropout, нужно **нормировать** веса при тестировании. Если нейрон был включён с вероятностью P во время обучения, то его вес нужно умножить на P. Это компенсирует эффект, что на этапе обучения этот нейрон был «выключен» с вероятностью 1 − P1.

Dropout создаёт эффект, который похож на ансамбль моделей. Каждый раз, когда сеть «выключает» нейроны, она фактически обучается на меньшем подмножестве данных. Это можно интерпретировать как обучение нескольких подсетей, которые решают одну и ту же задачу. Когда сеть тестируется, она использует все нейроны, но с нормированными весами, что позволяет объединить решения этих подсетей.

- **Dropout** помогает предотвратить переобучение, создавая случайные подсети и позволяя сети обучаться более универсально.
- Во время тестирования нужно нормировать веса, чтобы компенсировать эффект случайных выключений нейронов во время обучения.
- Нормирование весов приближает эффект Dropout к обучению ансамбля нейросетей — это улучшает обобщающие способности модели.

Заключение

Нейронные сети — это сложные, но гибкие модели, которые позволяют решать широкий спектр задач машинного обучения. Понимание их основных элементов, таких как нейроны, веса, смещения и функции активации, является фундаментом для успешной работы с ними. Библиотека PyTorch предоставляет удобные инструменты для создания и настройки архитектуры нейронных сетей, что значительно упрощает процесс разработки моделей.

Выбор подходящих функций активации и функций потерь, а также использование эффективных методов оптимизации, таких как Adam или SGD, напрямую влияет на качество обучения модели. Процесс обучения, основанный на обратном распространении ошибки, позволяет постепенно улучшать параметры сети, минимизируя ошибку на обучающих данных.

После изучения этих тем вы сможете уверенно создавать и обучать нейронные сети, адаптируя их под конкретные задачи. Это станет важным шагом на пути к углубленному изучению машинного обучения и искусственного интеллекта.