**Настя, Катя, Аня, Майя, Юра, Ярик, Дима, Андрей** *берем по 5 вопросов*

**ОГЛАВЛЕНИЕ**

[1. Модель перцептрона. Проблема линейно неразделимых множеств и ее решение. Логика построения многослойных ИНС.](#_xdecv9uw1gd2)

[2. Функции активации. Требования к функциям активации Популярные функции активации.](#_wa8ipr7gx2gj)

[3. Глубокое обучение. «Вторая весна искусственного интеллекта» и ее причины.](#_kxkyf4r1zlp2)

[4. Линейное отображение. Векторно-матричное дифференцирование.](#_d8i3ccb2ikik)

[5. Проблема поиска градиента в общей логике обучения нейронной сети. Градиент функции многих переменных. Методы вычисления.](#_7n68o7ss7zbb)

[6. Кросс-валидация. Выборки train, validation, test. Проблема переобучения. Ранняя остановка.](#_bncc34v5i7ve)

[7. Преобразование Softmax и функция потерь Cross Entropy loss.](#_w9eb1qjyju1)

[8. Механизм обратного распространения ошибки.](#_kre7tkw9dm25)

[9. Дифференцируемое программирование и реализация обратного распространения ошибки.](#_8y6r0zczs3t)

[10. Стохастический градиентный спуск. Батчи обучающей выборки.](#_kvapya6q2ntq)

[11. Адаптивные методы градиентного спуска. Метод импульсов. Метод Нестерова.](#_neamc5qhwd61)

[12. Проблема инициализации весов при обучении ИНС. Инициализация Ксавье.](#_7v3029k0t0ig)

[13. Гиперпараметры. Скорость обучения и размер батча.](#_tjljcqrt0hjo)

[14. Переобучение модели и регуляризация. Dropout.](#_jiakoxpcorgd)

[15. Минбатчи – причина использования. Нормализация по мини-батчам.](#_5shaez93hiaq)

[16. Многослойные сети. Граф потока вычислений.](#_o5v9vnv62pm7)

[17. Специфика задач машинного обучения на изображениях. Принцип работы сверточных сетей. Преимущества сверточных сетей при решении этих задач.](#_ody41rdnap40)

[18. Архитектура многослойной ИНС распознавания изображений на основе сверточных сетей.](#_yqknt9yj7o2t)

[19. Приемы для глубокого обучения на небольших наборах изображений.](#_idfo912cs870)

[20. Схема работ слоя сверточной сети. Пулинг. Гиперпараметры: padding, kernel size, stride, dilation.](#_1e7x11n0uyb)

[21. Задачи обработки текста: дистрибутивная семантика, матрица совместной встречаемости, представление слов в виде векторов малой размерности.](#_5uswhwv18uv8)

[22. Word2vec: модель CBOW.](#_xh8rp5sirjy6)

[23. Word2vec: модель Skip-Gram.](#_8lyw2nszslxm)

[24. Рекуррентная нейронная сеть, принципы ее обучения. Сложности применения рекуррентных нейронных сетей.](#_3ejcf8toot35)

[25. Модуль LSTM.](#_hgsznhijpdp9)

[26. Механизм Attention. Пример использования Attention.](#_x1biyos9tkhq)

[27. Архитектура Transformer.](#_62zujsf74go8)

[28. Модель BERT.](#_9ssq0ztrin0b)

[29. Класс Tensor. Операции, изменяющие размер тензора. Операции агрегации.](#_1fl0qd9nz02i)

[30. Принципиальная логика обучения нейронной сети.](#_ykxureqwngs7)

[31. Автоматическое дифференцирование в PyTorch. Пример и применение в обучении ИНС.](#_1ps03u9gp5h3)

[32. Загрузка и преобразование данных. Классы Dataset, DataLoader, Transforms (и композиция трансформеров).](#_v1zsqwgka1zx)

[33. Класс nn.Module. Назначение. Основные поля, методы.](#_a3obv2hsgvpk)

[34. Линейные слои (Linear Layers).](#_7qfg7qnwnu6t)

[35. Слои нелинейной активации (Non Linear Activations).](#_36550ywvw9xw)

[36. Слои нормализации (Normalization Layers).](#_lkn12j3n412c)

[37. Слои регуляризации (Dropout Layers).](#_r5m9vub88xia)

[38. Сверточные слои (Convolution Layers). Сжимающие слои (Pooling Layers).](#_i3mapvr8ryuz)

[39. Слои функций потерь (Loss Functions).](#_26s9kwio6sqp)

[40. Слои эмбеддингов nn.Embedding и их применение.](#_1n63h6eqv0b5)

[41. Класс torch.nn.LSTM и torch.nn.GRU.](#_1nzci3hj79ax)

## 1. Модель перцептрона. Проблема линейно неразделимых множеств и ее решение. Логика построения многослойных ИНС.

Модель перцептрона является одной из самых простых форм нейронных сетей. В идеальной ситуации, перцептрон может быть использован для разделения двух линейно разделимых классов данных. Однако, если классы данных не могут быть линейно разделены, то возникает проблема линейно неразделимых множеств.

Проблема линейно неразделимых множеств может быть решена путем использования многослойных нейронных сетей, таких как многослойные перцептроны (MLP) или глубокие нейронные сети.

Многослойный перцептрон (MLP) состоит из нескольких слоев нейронов, включая входной слой, скрытые слои и выходной слой. Каждый нейрон в слое связан с нейронами в соседних слоях. Нейроны в скрытых слоях и выходном слое обычно имеют нелинейную функцию активации, такую как сигмоидальная функция или функция ReLU (Rectified Linear Unit).

Идея построения многослойных нейронных сетей заключается в том, чтобы позволить нейронной сети учиться более сложным нелинейным отображениям данных. Каждый скрытый слой может изучать новые признаки или абстракции, на основе которых выходной слой может принять окончательное решение о классификации данных.

Обучение многослойных нейронных сетей, таких как MLP, обычно выполняется с использованием алгоритма обратного распространения ошибки (backpropagation). Этот алгоритм позволяет нейронной сети корректировать веса связей между нейронами во время обучения, минимизируя ошибку между предсказанными и фактическими значениями.

Таким образом, использование многослойных нейронных сетей позволяет решать проблему линейно неразделимых множеств и обрабатывать более сложные задачи классификации или регрессии.

## 2. Функции активации. Требования к функциям активации Популярные функции активации.

Функции активации являются неотъемлемой частью нейронных сетей и используются для введения нелинейности в выходные значения нейронов. Они применяются после выполнения взвешенной суммы входов нейрона для определения его активации или выхода.

**Требования к функциям активации:**

1. Нелинейность: Функции активации должны быть нелинейными, поскольку линейные функции активации приводят к эквивалентности многослойных сетей однослойным сетям.😭2. Дифференцируемость: Функции активации должны быть дифференцируемыми, чтобы градиентный спуск и обратное распространение ошибки могли быть применены для обучения нейронных сетей.

3. Монотонность: Желательно, чтобы функции активации были монотонными, то есть, чем больше вход, тем больше выход функции активации.

**Некоторые из популярных функций активации в нейронных сетях:**

1. Сигмоидная функция (Sigmoid): Функция, которая сжимает входное значение в диапазоне от 0 до 1. Она определена как f(x) = 1 / (1 + exp(-x)).

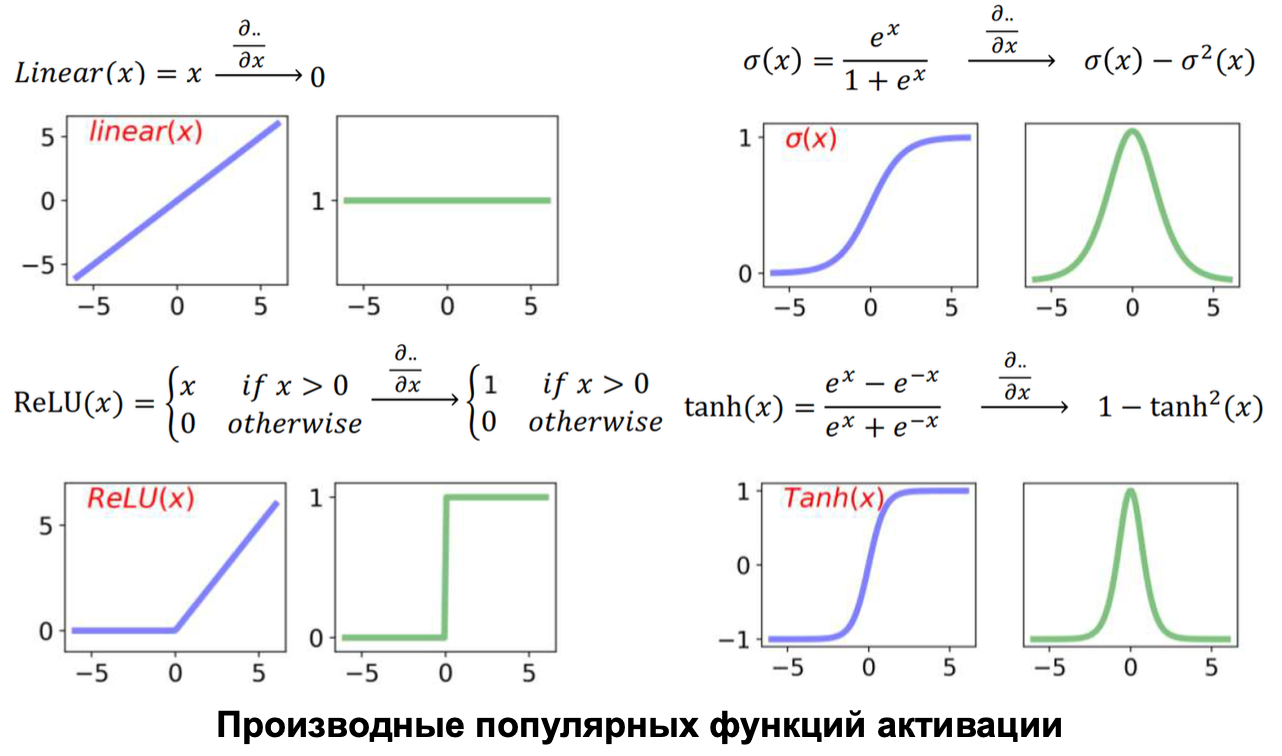
2. Гиперболический тангенс (Tanh): Похожа на сигмоидную функцию, но сжимает входное значение в диапазоне от -1 до 1. Она определена как f(x) = (exp(x) - exp(-x)) / (exp(x) + exp(-x)).

3. Функция ReLU (Rectified Linear Unit): Является простой нелинейной функцией, которая возвращает 0 для отрицательных значений и само значение для положительных значений. Она определена как f(x) = max(0, x).

4. Leaky ReLU: Похожа на функцию ReLU, но имеет небольшой наклон для отрицательных значений, чтобы избежать нулевого градиента. Она определена как f(x) = max(0.01x, x).

5. Гиперболический тангенс с усечением (Hard Tanh): Аналогично функции Tanh, но с усечением значений в диапазоне от -1 до 1. Она определена как f(x) = max(-1, min(1, x)).

6. Softmax: Часто используется в последнем слое нейронной сети для многоклассовой классификации. Преобразует входные значения в вероятности, сумма которых равна 1.



## 3. Глубокое обучение. «Вторая весна искусственного интеллекта» и ее причины.

Глубокое обучение — это подобласть машинного обучения, которая использует нейронные сети с несколькими слоями для извлечения высокоуровневых признаков из данных. Оно получило большую популярность и стало прорывом в области искусственного интеллекта благодаря нескольким факторам.

**"Вторая весна искусственного интеллекта"** - это термин, который относится к возрождению и прорыву глубокого обучения, произошедшему в последние годы.

**Этот прорыв обусловлен несколькими факторами:**

1. Увеличение доступности данных: С появлением Интернета и развитием цифровых технологий мы собираем и генерируем огромные объемы данных. Доступность больших наборов данных (Big Data) стала одним из ключевых факторов успеха глубокого обучения. Большие наборы данных позволяют обучать сложные модели глубокого обучения, которые способны извлекать более сложные и абстрактные признаки.

2. Вычислительная мощность: Графические процессоры (GPU) и специализированные аппаратные ускорители, такие как тензорные процессоры (TPU), стали доступны для обучения нейронных сетей. Эти вычислительные ресурсы позволяют эффективно выполнять операции с большими матрицами и ускоряют процесс обучения глубоких моделей.

3. Архитектурные инновации: В процессе развития глубокого обучения были предложены новые архитектуры нейронных сетей, такие как сверточные нейронные сети (CNN) для обработки изображений и рекуррентные нейронные сети (RNN) для обработки последовательностей. Также появились архитектуры, основанные на внимании (attention), такие как трансформеры (transformers), которые показали выдающиеся результаты в обработке естественного языка и машинном переводе.

4. Улучшение алгоритмов обучения: Появление новых методов оптимизации, таких как стохастический градиентный спуск (SGD) с моментом, адаптивные методы оптимизации (например, Adam), нормализация пакетов (batch normalization) и др., значительно улучшило процесс обучения глубоких моделей.

Все эти факторы вместе способствовали второй волне развития глубокого обучения и привели к его широкому применению в различных областях, таких как компьютерное зрение, обработка естественного языка, речевые технологии и многое другое.

## 4. Линейное отображение. Векторно-матричное дифференцирование.

Линейное отображение в математике относится к отображению между двумя векторными пространствами, которое обладает двумя основными свойствами: сохранением линейных комбинаций и сохранением скалярного умножения. Формально, линейное отображение L: V -> W между векторными пространствами V и W определено следующим образом:

1. L(u + v) = L(u) + L(v) для всех u, v из V (сохранение линейных комбинаций).

2. L(αu) = αL(u) для всех α из полей скаляров и всех u из V (сохранение скалярного умножения).

Здесь u и v - произвольные векторы из V, α - произвольный скаляр, а L(u) и L(v) - соответствующие образы этих векторов в W.

Векторно-матричное дифференцирование является техникой дифференцирования, применяемой для нахождения производных линейных отображений, представленных в виде матриц. Когда линейное отображение задается матрицей A, векторно-матричное дифференцирование позволяет находить производные этого отображения по отношению к входным векторам.

Пусть f(x) = Ax, где A - матрица размерности m x n, а x - вектор размерности n. Тогда производная отображения f(x) по вектору x, обозначаемая как df(x)/dx или ∂f/∂x, может быть найдена путем векторно-матричного дифференцирования.

Результатом векторно-матричного дифференцирования является матрица, называемая матрицей Якобиана или матрицей производных. Матрица Якобиана имеет размерность m x n и содержит частные производные элементов функции f(x) по элементам вектора x.

Векторно-матричное дифференцирование является важным инструментом в оптимизации, обратном распространении ошибки (backpropagation) в нейронных сетях и других областях, где требуется нахождение градиента или производных линейных отображений.

## 5. Проблема поиска градиента в общей логике обучения нейронной сети. Градиент функции многих переменных. Методы вычисления.

Проблема поиска градиента в общей логике обучения нейронной сети связана с тем, что нейронные сети обычно имеют большое количество параметров, и вычисление градиента функции с таким количеством переменных может быть вычислительно затратным и требовать большого объема памяти. Это может стать препятствием для эффективного обучения модели.

Градиент функции многих переменных представляет собой вектор, состоящий из частных производных функции по каждой переменной. Для функции f(x₁, x₂, ..., xn), где x₁, x₂, ..., xn - переменные, градиент функции обозначается как ∇f или grad(f) и определяется следующим образом:

∇f = (∂f/∂x₁, ∂f/∂x₂, ..., ∂f/∂xn)

Методы вычисления градиента функции многих переменных включают:

1. Аналитический метод: Если функция f(x) имеет аналитическую формулу, то градиент может быть вычислен аналитически путем нахождения частных производных функции по каждой переменной.

2. Численный метод: В численных методах градиент вычисляется приближенно путем численного дифференцирования. Наиболее распространенными численными методами являются метод конечных разностей и метод конечных разностей со сглаживанием (smoothed finite differences).

3. Автоматическое дифференцирование: Этот метод использует вычислительные графы и цепное правило дифференцирования для автоматического вычисления градиента. Автоматическое дифференцирование позволяет вычислить градиент функции, используя только значение функции и операции, применяемые для ее вычисления, без явного вычисления аналитической формулы или численного дифференцирования.

4. Градиентный спуск с обратным распространением ошибки (backpropagation): В контексте обучения нейронных сетей, градиентный спуск с обратным распространением ошибки является методом вычисления градиента для обновления весов нейронной сети. Он использует цепное правило дифференцирования для эффективного вычисления градиента на основе ошибки модели.

Эти методы позволяют эффективно вычислять градиент функции многих переменных и используются в различных задачах оптимизации и обучения нейронных сетей.

## 6. Кросс-валидация. Выборки train, validation, test. Проблема переобучения. Ранняя остановка.

### Кросс-валидация

Кросс-валидация (перекрестная проверка) – это метод оценки Моделей (Model) Машинного обучения (ML) путем обучения нескольких из них на подмножествах доступных входных данных и их оценки на другом дополнительном подмножестве. Такая проверка используется для обнаружения Переобучения (Overfitting), т.е. неспособности распознать паттерн.

Всегда необходимо проверять стабильность предсказывающего Алгоритма (Algorithm): нам нужна уверенность в том, что модель имеет представление о большинстве шаблонов в данных, что ее эффективность не падает от шумных данных.

#### Метод hold-out

Метод hold-out представляет из себя простое разделение на train и test

import numpy as np

from sklearn.model\_selection import train\_test\_split

X, y = np.arange(1000).reshape((500, 2)), np.arange(500)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y,

test\_size=0.2,

random\_state=42

)

#### Метод k-Fold

Метод k-Fold чаще всего имеют в виду, когда говорят о кросс-валидации. Он является обобщением метода hold-out и представляет из себя следующий алгоритм:

· Фиксируется некоторое целое число *k* (обычно от 5 до 10), меньшее числа семплов в датасете.

· Датасет разбивается на *k* одинаковых частей (в последней части может быть меньше семплов, чем в остальных). Эти части называются фолдами.

· Далее происходит *k* итераций, во время каждой из которых один фолд выступает в роли тестового множества, а объединение остальных — в роли тренировочного. Модель учится на *k – 1* фолде и тестируется на оставшемся.

· Финальный скор модели получается либо усреднением *k* получившихся тестовых результатов, либо измеряется на отложенном тестовом множестве, не участвовавшем в кросс-валидации.

**Sklearn: from** sklearn.model\_selection **import** KFold

### Выборки:

Обучающая выборка (training sample) — выборка, по которой производится настройка (оптимизация параметров) модели зависимости.

Если модель зависимости построена по обучающей выборке X^m, то оценка качества этой модели, сделанная по той же выборке X^m оказывается, как правило, оптимистически смещённой. Это нежелательное явление называют переобучением. На практике оно встречается очень часто. Хорошую эмпирическую оценку качества построенной модели даёт её проверка на независимых данных, которые не использовались для обучения.

Тестовая (или контрольная) выборка (test sample) — выборка, по которой оценивается качество построенной модели. Если обучающая и тестовая выборки независимы, то оценка, сделанная по тестовой выборке, является несмещённой.

Оценку качества, сделанную по тестовой выборке, можно применить для выбора наилучшей модели. Однако тогда она снова окажется оптимистически смещённой. Для получения немсещённой оценки выбранной модели приходится выделять третью выборку.

Проверочная выборка (validation sample) — выборка, по которой осуществляется выбор наилучшей модели из множества моделей, построенных по обучающей выборке.

### Переобучение:

Переобучение — это явление, когда обучаемая модель хорошо распознает примеры из обучающего множества, но при этом не распознает или плохо распознает любые другие примеры, не участвовавшие в процессе обучения (т.е. предъявляемые ей в процессе практического использования).

Переобучение — это результат чрезмерной подгонки параметров модели к зависимостям, содержащимся в обучающем множестве. Если происходит переобучение, то модель не приобретает способности к обобщению — возможности распространять обнаруженные на обучающем множестве зависимости и закономерности на новые данные.

Проблема переобучения в той или иной степени характерна для всех видов обучаемых моделей, но особенно остро она стоит для нейронных сетей. В процессе обучения производится подгонка весов нейронной сети таким образом, что сеть преобразовывала входы к желаемым выходам в соответствии с зависимостями, обнаруженными в обучающих данных.

Для того, чтобы избежать переобучения, можно использовать следующие несложные правила.

· Применение тестового множества. Тестовое множество формируется из обучающего набора данных случайным образом и его примеры подаются на вход сети между обучающими примерами, но корректировку весов сети на вызывают. Вначале обучения ошибка и на обучающем, и на тестовом множестве уменьшаются. Но начиная с какого-то ошибка на тестовом множестве начинает расти. Это сигнализирует о начале переобучения и необходимости принудительно остановить процесс обучения.

· Использование перекрёстной проверки. Все данные, на которых строится модель, разделяются на k блоков равного размера. При этом обучение производится на k-1 блоках, а тестирование — на k-м. Процедура повторяется k раз, при этом для тестирования каждый раз выбирается другой блок. В результате все блоки оказываются используемыми и как обучающие, и как тестирующие.

· Выбор конфигурации нейронной сети. Нужно выбирать конфигурацию сети (число слоёв и нейронов), чтобы количество параметров модели (весов) была в 2-3 раза меньше числа примеров обучающего множества. Если число параметров модели и обучающих примеров окажется соизмеримым, то сеть просто «запомнит» все комбинации вход-выход в примерах обучающего множества, и будет воспроизводить только их, а на новых данных допускать ошибки.

### Ранняя остановка

Ранняя остановка (early stopping) - это метод регуляризации в машинном обучении, который позволяет избежать переобучения модели.

Суть метода заключается в том, что обучение модели прерывается, когда ошибка на тестовых данных начинает увеличиваться. Это происходит потому, что в начале обучения модель обучается на более общих паттернах, которые лучше обобщают данные, а в конце обучения модель начинает запоминать шумы и нюансы в данных, что приводит к переобучению.

Ранняя остановка может быть реализована путем отслеживания ошибки на тестовых данных на каждой эпохе обучения. Если ошибка на тестовых данных начинает увеличиваться, обучение прерывается и модель сохраняется на последней эпохе, когда ошибка на тестовых данных была минимальной.

### Обобщенно и кратко

При разработке моделей машинного обучения очень важно не только обучить модель на данных обучающей выборки, но и оценить качество ее работы на независимых данных, которые отличаются от тех, на которых модель была обучена.

Для этого необходимо использовать тестовую выборку, которая должна быть независима от обучающей и содержать данные, которые модель еще не видела в процессе обучения. Оценка качества модели на тестовой выборке позволяет оценить ее способность обобщать данные и делать предсказания на новых данных.

Для избежания переобучения модели, когда она получает хороший результат на обучающей выборке, но плохой на новых данных, можно применять различные методы регуляризации. Один из таких методов - ранняя остановка, заключается в прерывании обучения модели, когда ошибка на тестовых данных начинает увеличиваться. Это происходит потому, что в начале обучения модель обучается на более общих паттернах, которые лучше обобщают данные, а в конце обучения модель начинает запоминать шумы и нюансы в данных, что приводит к переобучению. Таким образом, ранняя остановка позволяет сохранять модель на этапе, когда ее качество на тестовых данных было оптимальным и избежать переобучения.

Для выбора наилучшей модели можно также использовать перекрестную проверку. Данные разделяются на k блоков, из которых один блок используется для тестирования, а остальные для обучения. Процедура повторяется k раз, при этом для тестирования каждый раз выбирается другой блок. Этот метод позволяет получить более надежную оценку качества модели, так как каждый блок данных используется как для обучения, так и для тестирования.

Еще одним методом регуляризации является выбор конфигурации нейронной сети. Необходимо выбирать конфигурацию сети (число слоев и нейронов), чтобы количество параметров модели (весов) было в 2-3 раза меньше числа примеров обучающего множества. Если число параметров модели и обучающих примеров окажется соизмеримым, то сеть просто «запомнит» все комбинации вход-выход в примерах обучающего множества, и будет воспроизводить только их, а на новых данных допускать ошибки.

В целом, использование различных методов регуляризации и оценки качества модели на независимых данных позволяет избежать переобучения и получить более надежные результаты предсказаний.

## 7. Преобразование Softmax и функция потерь Cross Entropy loss.

### Softmax

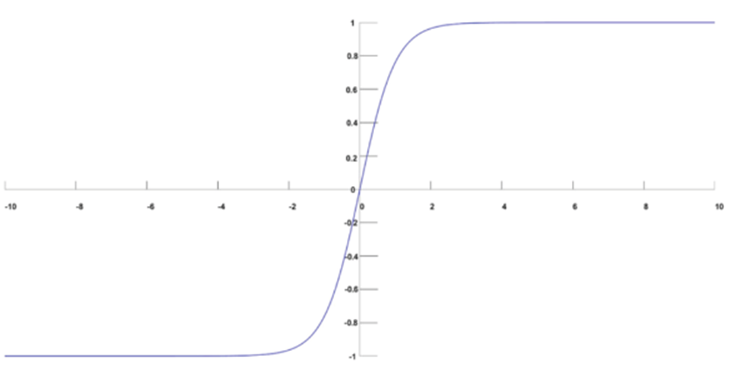
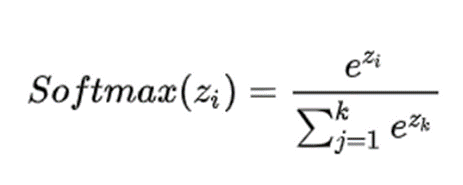


График функции

Софтмакс – функция, превращающая логиты (наборы чисел) в вероятности, причем сумма последних равна единице. Функция выводит в качестве результата вектор, представляющий распределения вероятностей списка потенциальных результатов.

Формула функции Softmax выглядит следующим образом:



где z\_i - это элемент входного вектора, а k - это общее число элементов в векторе.

График функции Softmax представляет собой гладкую кривую, начинающуюся от 0 и заканчивающуюся на 1, что соответствует сумме вероятностей всех элементов вектора. Кривая функции Softmax имеет свойство, что вероятность любого элемента вектора увеличивается, если значения других элементов уменьшаются, что позволяет использовать эту функцию для многоклассовой классификации.

Хотя функция Softmax имеет множество применений в машинном обучении, она также может иметь недостатки, такие как чувствительность к выбросам и несбалансированным данным, что может приводить к неверным вероятностным оценкам.

### Cross entropy loss

Функция потерь перекрестной энтропии – это метрика, позволяющая оценить, насколько хорошо функционирует модель классификации в машинном обучении. Потеря (или ошибка) оценивается как число, находящееся между 0 и 1, где 0 – идеальная модель. Цель, как правило, заключается в том, чтобы максимально приблизить вашу модель к 0.

Функцию потерь перекрестной энтропии часто считают взаимозаменяемой с логистической ошибкой (или логистической функцией потерь и иногда рассматриваемой как функция потерь бинарной перекрестной энтропии), но это не всегда корректно.

Используя перекрестную энтропию, можно оценить ошибку (или разницу) между двумя вероятностными распределениями.

Например, в случае бинарной классификации перекрестная энтропия определяется как:



где:

p – предсказанная вероятность и

y – индикатор (0 или 1 в случае бинарной классификации)

Реализация:

import torch.nn as nn

criterion = nn.CrossEntropyLoss()

### Обобщенно и кратко

Softmax - функция, которая преобразует логиты в вероятности, где сумма вероятностей равна единице. Функция Softmax может использоваться для многоклассовой классификации. Она имеет свойство, что вероятность любого элемента вектора увеличивается, если значения других элементов уменьшаются. Однако, она может быть чувствительной к выбросам и несбалансированным данным, что приводит к неверным вероятностным оценкам.

Функция потерь перекрестной энтропии используется для оценки, насколько хорошо функционирует модель классификации в машинном обучении. Потеря оценивается как число, находящееся между 0 и 1, где 0 – идеальная модель. Она может быть использована для оценки ошибки между двумя вероятностными распределениями. В случае бинарной классификации, перекрестная энтропия определяется как p - предсказанная вероятность и y - индикатор (0 или 1 в случае бинарной классификации).

## 8. Механизм обратного распространения ошибки.

Механизм обратного распространения ошибки - это алгоритм, используемый в нейронных сетях для обучения модели с учителем.

Он работает путем вычисления градиента функции потерь по параметрам модели и последующего изменения этих параметров с целью минимизации потерь.

Процесс начинается с передачи входных данных через нейронную сеть для получения предсказаний. Затем вычисляется ошибка между предсказанными значениями и фактическими значениями.

Далее, ошибка обратно распространяется через сеть, и для каждого параметра вычисляется его вклад в общую ошибку. Это делается с помощью цепного правила дифференцирования.

И, наконец, параметры модели обновляются с использованием градиентного спуска, чтобы минимизировать ошибку.

Механизм обратного распространения ошибки является основой обучения нейронных сетей и имеет широкое применение в различных областях, включая компьютерное зрение, обработку естественного языка и другие.

## 9. Дифференцируемое программирование и реализация обратного распространения ошибки.

Дифференцируемое программирование – это подход к машинному обучению, который позволяет оптимизировать модели, используя градиенты функций ошибок. Он используется для поиска наилучших параметров модели, которые минимизируют ошибки на обучающих данных.

Дифференцируемое программирование основано на математической концепции дифференцирования, которая используется для вычисления производных функций. В машинном обучении дифференцирование используется для нахождения градиента функции ошибок – величины, которая показывает, как изменится значение функции ошибок при изменении параметров модели.

Одной из самых популярных техник дифференцируемого программирования является обратное распространение ошибки. Эта техника позволяет вычислить градиент функции ошибок по отношению к весам модели.

Обратное распространение ошибки работает путем прохода через модель в обратном направлении, начиная с выходных значений и заканчивая входными значениями. Во время этого процесса вычисляются градиенты функций активации и весов, используемых в модели.

В настоящее время существует множество библиотек и фреймворков, которые предоставляют реализацию обратного распространения ошибки для различных типов моделей. Эти инструменты делают дифференцируемое программирование доступным для широкого круга исследователей и разработчиков, которые хотят создавать более эффективные и точные модели машинного обучения.

## 10. Стохастический градиентный спуск. Батчи обучающей выборки.

### Стохастический градиентный спуск

Стохастический градиентный спуск (Stochastic Gradient Descent) - это алгоритм оптимизации, используемый для обучения машинного обучения. Он является одним из самых популярных алгоритмов обучения и широко применяется в различных задачах, таких как распознавание образов, классификация и регрессия.

Основная идея стохастического градиентного спуска заключается в том, чтобы на каждом шаге обновлять параметры модели, используя градиент функции потерь, вычисленный на небольшом подмножестве обучающих данных (батче). В отличие от обычного градиентного спуска, который вычисляет градиент на всей обучающей выборке, стохастический градиентный спуск может работать гораздо быстрее, так как он использует только часть данных на каждом шаге.

Другим важным преимуществом стохастического градиентного спуска является его способность работать с большими объемами данных. Вместо того, чтобы загружать все данные в память компьютера, как это делается при использовании обычного градиентного спуска, стохастический градиентный спуск может использовать данные порциями и обновлять параметры модели на каждом шаге.

Несмотря на свою эффективность, стохастический градиентный спуск имеет свои недостатки. В частности, из-за случайности выбора подмножества данных на каждом шаге, алгоритм может сходиться к локальному минимуму вместо глобального. Также возможен риск переобучения модели, особенно если размер батча слишком мал.

### Батчи обучающей выборки

Батчи обучающей выборки - это подмножества данных, которые используются для обновления параметров модели на каждом шаге стохастического градиентного спуска. В машинном обучении, обучающая выборка - это набор данных, который используется для тренировки модели. Она состоит из множества примеров, каждый из которых представляет собой пару входных и выходных значений. Батчи выборки - это небольшие группы примеров, которые передаются модели для обновления ее параметров.

Выбор размера батча является важным гиперпараметром при использовании стохастического градиентного спуска. Размер батча определяет, сколько примеров будет использоваться на каждом шаге для обновления параметров модели. Он может оказать значительное влияние на эффективность и скорость обучения модели.

Существует компромисс между размером батча и стабильностью обучения. Маленькие батчи могут привести к быстрой сходимости, так как они требуют меньшего количества вычислений, но могут также вызвать нестабильность градиента, что может привести к неэффективному обучению модели. Большие же батчи могут оказаться более стабильными, но могут также привести к более медленной сходимости, так как они требуют большего количества вычислений.

Выбор оптимального размера батча зависит от конкретной задачи и может потребовать некоторых экспериментов. Например, при обработке изображений и видео, рекомендуется использовать батчи размером от нескольких десятков до нескольких сотен элементов, так как они требуют большого количества вычислений, а использование слишком маленьких батчей может привести к нестабильности градиента. Однако, в целом, выбор оптимального размера батча зависит от конкретной задачи и может потребовать некоторых экспериментов.

### Обобщенно и кратко

Стохастический градиентный спуск (Stochastic Gradient Descent) является одним из самых популярных алгоритмов обучения машинного обучения. Он используется для обучения моделей в различных задачах, таких как распознавание образов, классификация и регрессия. Основная идея стохастического градиентного спуска заключается в том, чтобы на каждом шаге обновлять параметры модели, используя градиент функции потерь, вычисленный на небольшом подмножестве обучающих данных (батче). В отличие от обычного градиентного спуска, который вычисляет градиент на всей обучающей выборке, стохастический градиентный спуск может работать гораздо быстрее, так как он использует только часть данных на каждом шаге. Это также позволяет ему работать с большими объемами данных. Батчи обучающей выборки - это подмножества данных, которые используются для обновления параметров модели на каждом шаге стохастического градиентного спуска. Выбор размера батча является важным гиперпараметром при использовании стохастического градиентного спуска. Размер батча определяет, сколько примеров будет использоваться на каждом шаге для обновления параметров модели. Существует компромисс между размером батча и стабильностью обучения. Оптимальный размер батча зависит от конкретной задачи и может потребовать некоторых экспериментов.

## 11. Адаптивные методы градиентного спуска. Метод импульсов. Метод Нестерова.

Адаптивные методы градиентного спуска разработаны для оптимизации скорости обучения по отдельности для каждого параметра. Это особенно полезно при работе с многомерными и разреженными данными. Примеры включают:

* Adagrad: Изменяет скорость обучения на основе частоты параметра в данных. Параметры, которые встречаются часто, получают меньший шаг обучения, в то время как редкие параметры получают больший шаг. Это может быть полезно в задачах обработки естественного языка, где редкие слова необходимо обучать быстрее.
* Adadelta и RMSprop: Являются улучшенными версиями Adagrad, которые пытаются решить проблему затухающей скорости обучения. Они делают это, используя скользящее среднее квадратов градиентов вместо накопительной суммы.
* Adam: Сочетает идеи Momentum и RMSprop. Он поддерживает скользящее среднее как моментов, так и квадратов градиентов для обновления весов.

Метод импульса (Momentum) использует идею физического импульса и применяет ее к градиентному спуску. Это основано на идее, что если мы движемся в определенном направлении, то у нас есть больше шансов продолжить движение в этом же направлении. Это уменьшает колебания и ускоряет сходимость.

Метод Нестерова (Nesterov Accelerated Gradient, NAG) представляет собой модификацию метода импульса, где мы сначала делаем большой шаг в направлении предыдущего импульса (моментум), затем вычисляем градиент в новом "предвосхищенном" месте и делаем коррекцию. Это приводит к более устойчивым и быстрым обновлениям.

## 12. Проблема инициализации весов при обучении ИНС. Инициализация Ксавье.

Инициализация весов является важной стадией в обучении нейронных сетей. Причина в том, что начальные веса сети существенно влияют на то, к какому минимуму сходится процесс обучения. Если веса инициализированы неверно, нейронная сеть может столкнуться с проблемами затухания или взрыва градиентов, что замедляет процесс обучения и может привести к плохим результатам.

Затухание градиентов возникает, когда веса инициализируются слишком малыми значениями. В результате градиенты могут стать слишком малыми, что замедляет процесс обучения или даже делает его невозможным. Наоборот, взрыв градиентов происходит, когда веса инициализируются слишком большими значениями, и градиенты становятся слишком большими, что приводит к нестабильности в обучении.

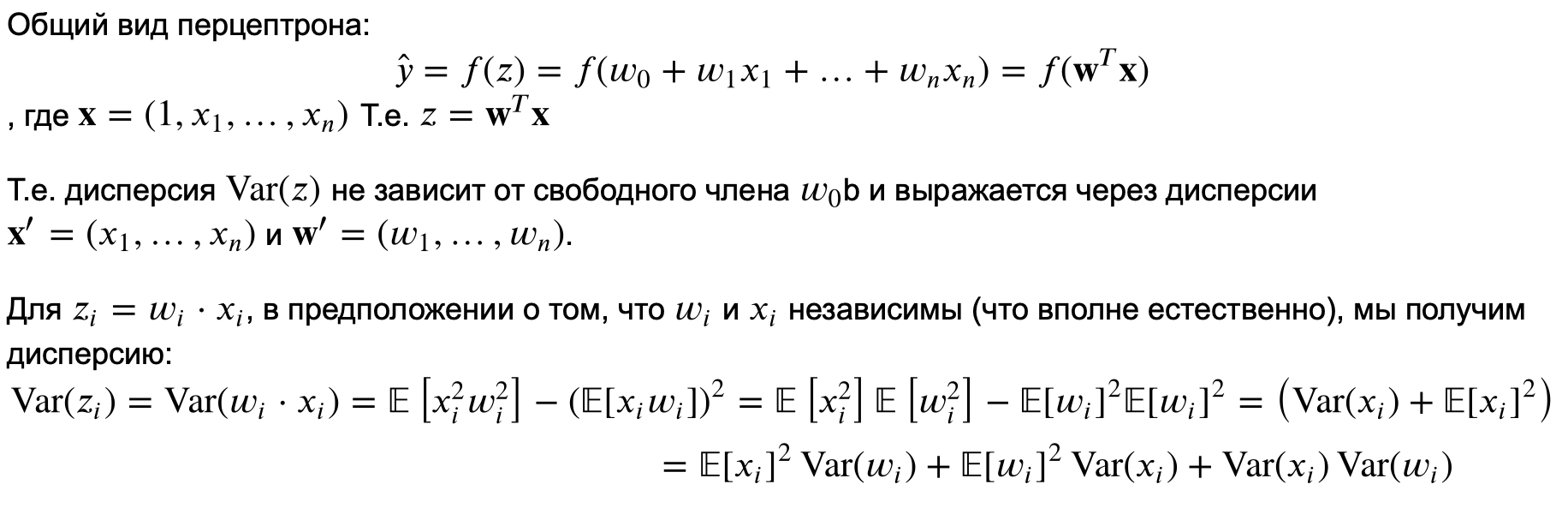
Инициализация Ксавье, также известная как инициализация Glorot, предлагает решение этой проблемы. Она выбирает веса из равномерного или нормального распределения, которое масштабировано с учетом числа входных и выходных нейронов в слое. Это обеспечивает равномерность вариации активаций и градиентов по слоям, что приводит к более эффективному обучению.

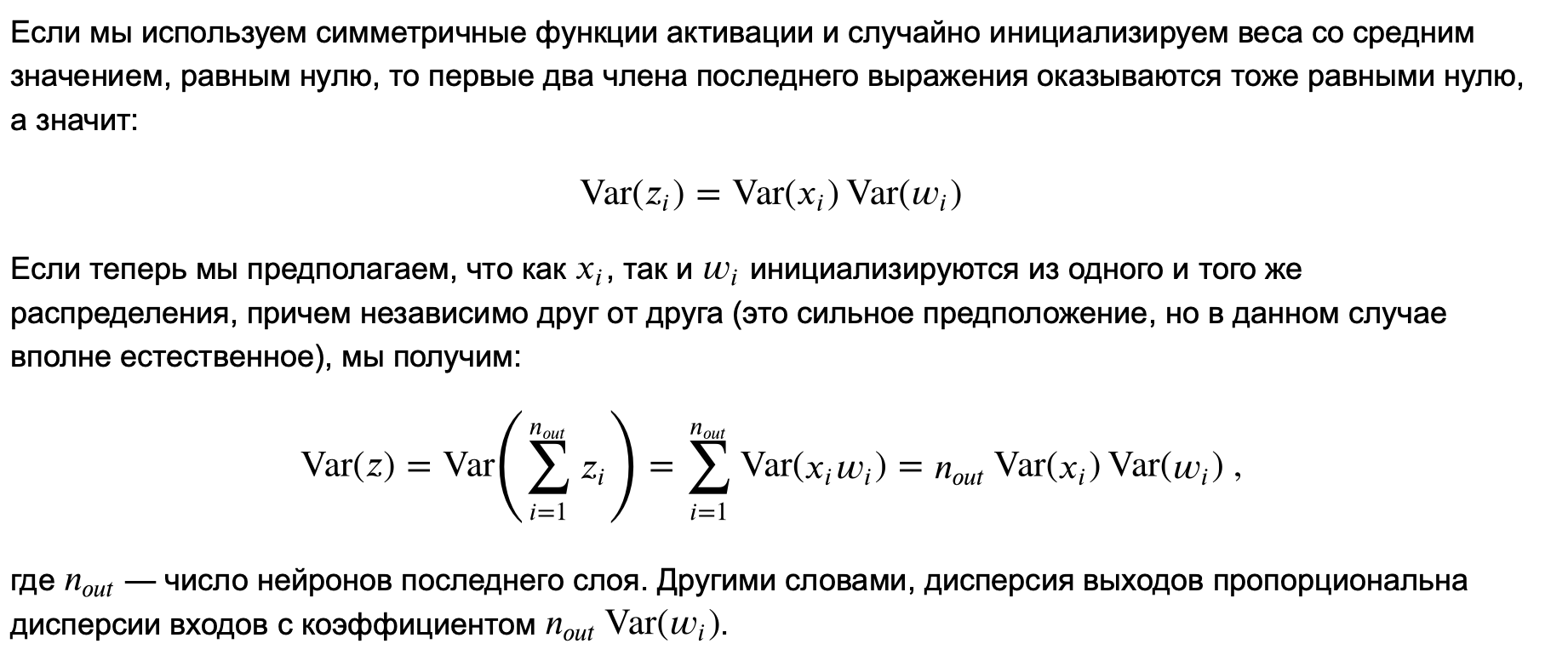
Пример реализации в PyTorch:

conv1 **=** torch.nn.Conv2d(**...**)

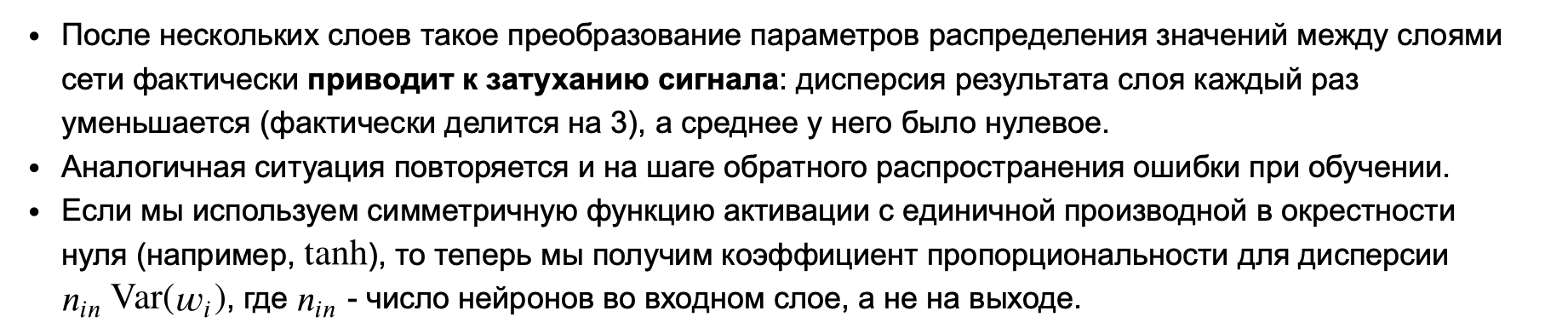
torch.nn.init.xavier\_uniform(conv1.weight)

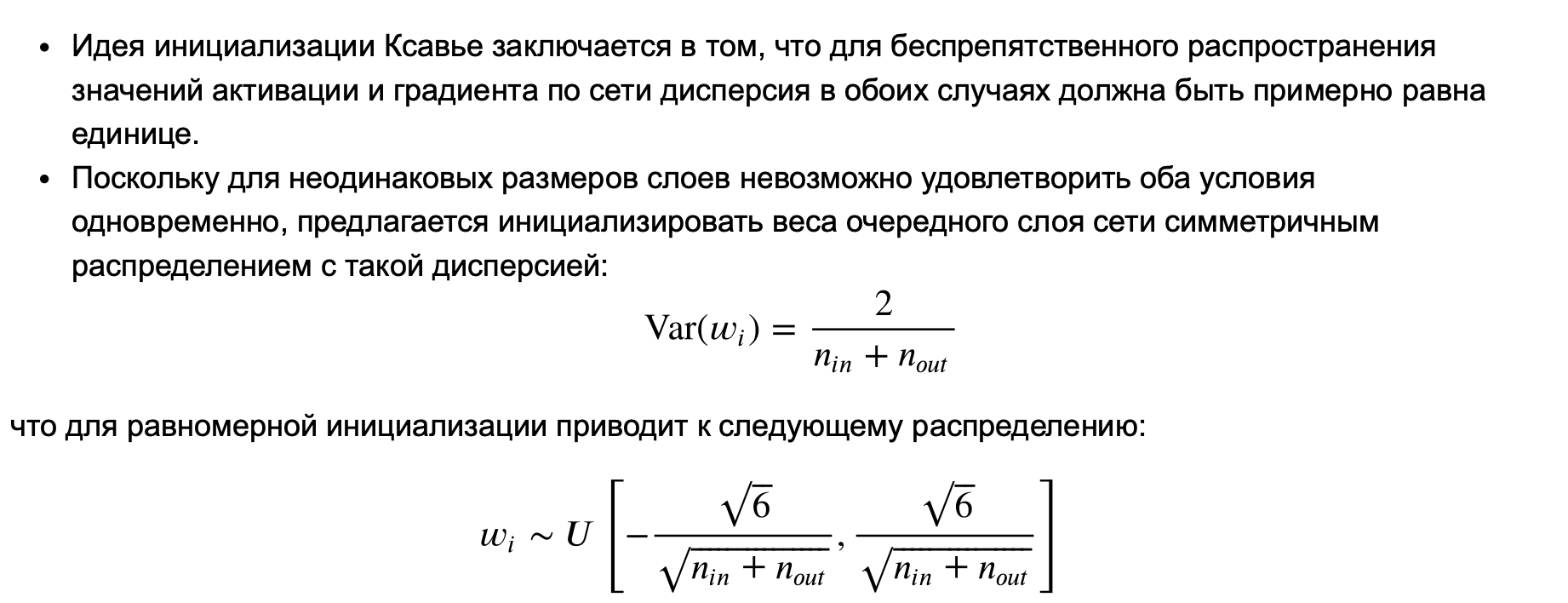
### Из лекции макрушина











## 13. Гиперпараметры. Скорость обучения и размер батча.

Гиперпараметры - это параметры, которые определяются до начала процесса обучения и не изменяются во время обучения. Они могут иметь значительное влияние на производительность и скорость обучения модели.

Скорость обучения является одним из наиболее важных гиперпараметров. Она определяет, насколько быстро модель обновляет свои веса в ответ на ошибку. Если скорость обучения слишком высока, обучение может стать нестабильным, и модель может пропустить оптимальное решение. Если скорость обучения слишком низкая, обучение может замедлиться, и модель может застрять в локальном минимуме.

Размер батча - это количество обучающих примеров, обрабатываемых одновременно. Большой размер батча может ускорить процесс обучения, но также может привести к менее точной оценке градиента. С другой стороны, маленький размер батча может замедлить обучение и увеличить вероятность застревания в локальном минимуме.

## 14. Переобучение модели и регуляризация. Dropout.

Переобучение (overfitting) - это проблема, при которой модель слишком хорошо обучается на обучающих данных и становится плохо обобщена на новых, неизвестных данных. Это может быть результатом слишком сложной модели или недостаточного количества данных для обучения.

Регуляризация - это техника, используемая для предотвращения переобучения, путем добавления штрафа к функции потерь, чтобы ограничить сложность модели. Это может включать L1-регуляризацию (лассо), L2-регуляризацию (гребневая регрессия) или оба вида.

Dropout - это техника регуляризации, используемая в нейронных сетях, которая случайным образом исключает (обнуляет) нейроны в слое при каждом прохождении вперед. Это заставляет сеть учиться на более робустных функциях и предотвращает переобучение.

### Из лекции макрушина

**Проблема переобучения модели**

модель, у которой слишком много свободных параметров, **плохо обобщается**: то есть слишком близко «облизывает» точки из тренировочного множества и в результате недостаточно хорошо предсказывает нужные значения в новых точках.

В современных нейронных сетях огромное число параметров (даже не самая сложная архитектура может содержать миллионы весов) **надо регуляризовать параметры**!

**Dropout**

**Регуляризация с помощью дропаута** (dropout regularization) - один из важнейших методов регуляризации нейронных сетей обеспечивший революцию глубокого обучения.

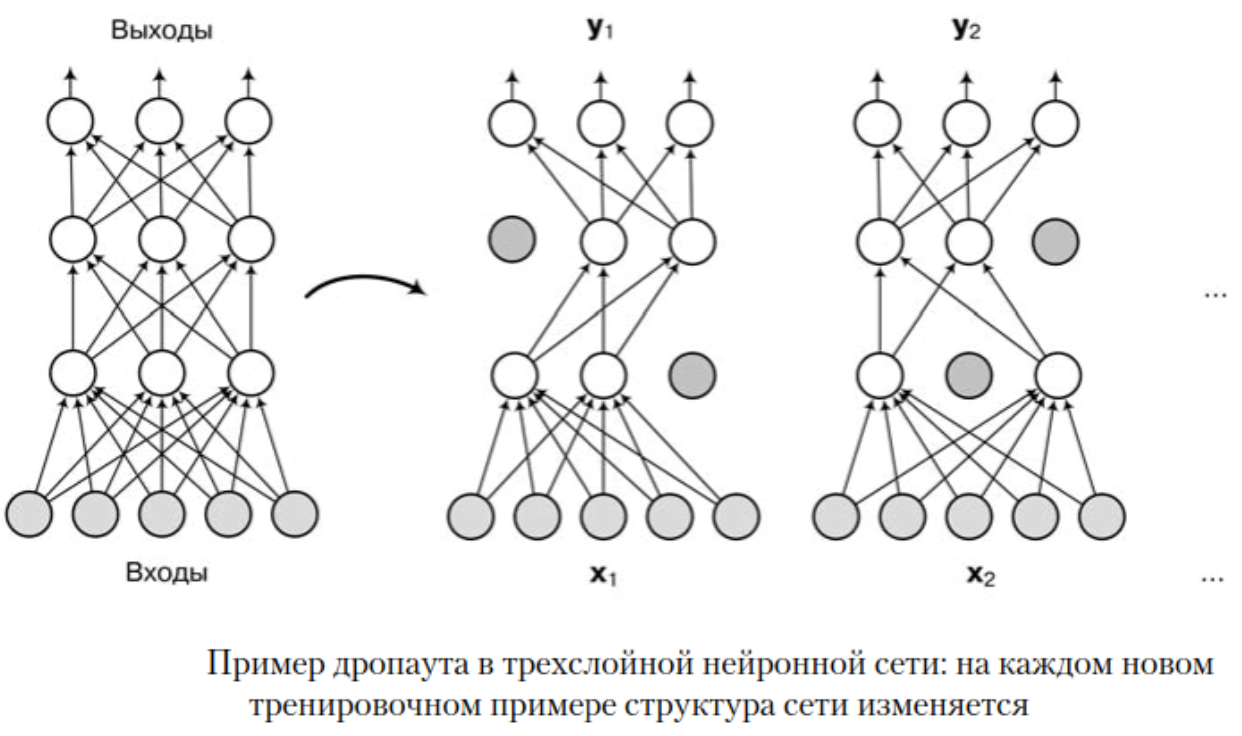
Идея метода (очень простая!):

* Для каждого нейрона (кроме самого последнего, выходного слоя) установим некоторую вероятность p, с которой он будет выброшен из сети.

Алгоритм обучения меняется таким образом:

* на каждом новом тренировочном примере x мы сначала для каждого **разыгрываем вероятность p** и в зависимости от результата либо используем нейрон как обычно, **либо устанавливаем его выход всегда строго равным нулю** (вероятность этого события 1 - p).
* **Дальше все происходит без изменений**; ноль на выходе приводит к тому, что нейрон фактически выпадает из графа вычислений: и прямое вычисление, и обратное распространение градиента останавливаются на этом нейроне и дальше не идут.
* Для применения обученной сети **используются все нейроны** в конфигурации, которая была до применения дропаута, но **выход каждого нейрона умножается на вероятность p** (с которой нейрон оставляли при обучении)

Для очень широкого спектра архитектур и приложений замечательно подходит p = 1/2



Практика обучения нейронных сетей показывает, что дропаут действительно дает очень серьезные улучшения в качестве обученной модели.

Дропаут — это метод добиться **усреднения огромного чмсла моделей** (до 2^N возможных моделей, N — число нейронов, которые подвергаются дропауту). Он эквивалентен усреднению всех моделей, которые получались на каждом шаге случайным выбрасыванием отдельных нейронов.

Пример использования dropuot в pytorch:

**def** \_\_init\_\_(self, rnn\_type, input\_size, node\_fdim, hidden\_size, depth, dropout):

super(MPNEncoder, self).\_\_init\_\_()  
self.hidden\_size **=** hidden\_size  
self.input\_size **=** input\_size

self.depth **=** depth self.

W\_o **=** nn.Sequential(

nn.Linear(node\_fdim **+** hidden\_size, hidden\_size),

nn.ReLU(),

nn.Dropout(dropout)

)

**if** rnn\_type **==** 'GRU':

self.rnn **=** GRU(input\_size, hidden\_size, depth)

**elif** rnn\_type **==** 'LSTM':

self.rnn **=** LSTM(input\_size, hidden\_size, depth)

**else**:

**raise** ValueError('unsupported rnn cell type ' **+** rnn\_type)

## 15. Минбатчи – причина использования. Нормализация по мини-батчам.

Минибатчи используются в обучении моделей для оптимизации процесса обучения. Вместо того чтобы обрабатывать все обучающие примеры сразу или по одному, используется подмножество данных - минибатч. Есть несколько причин использования минибатчей:

* Эффективность вычислений: обработка небольших батчей позволяет параллелизовать вычисления, что приводит к более эффективному использованию ресурсов (например, графических процессоров).
* Улучшенная сходимость: обновления весов на основе минибатчей дают более точную оценку градиента, чем обновления на основе одного примера, что обычно приводит к более быстрой сходимости.
* Уменьшение переобучения: минибатчи добавляют некоторую степень шума в процесс обучения, что может помочь предотвратить переобучение.

Нормализация по мини-батчам является техникой масштабирования входов в слои нейронной сети. Вместо масштабирования всех входных данных сразу, она масштабирует каждый минибатч независимо. Это позволяет модели более эффективно справляться с проблемами внутренней ковариации (когда распределение входных данных меняется в процессе обучения), что обычно приводит к ускорению обучения и улучшению обобщающей способности модели.

### Из лекции макрушина

## 16. Многослойные сети. Граф потока вычислений.

Многослойные сети, также известные как многослойные перцептроны (MLP), состоят из нескольких слоев нейронов, каждый из которых подключен к нейронам следующего слоя. Это позволяет MLP обучаться и представлять более сложные функции, чем однослойные сети. Например, MLP способен представлять функции, которые отображают входные данные в нелинейные пространства признаков, что может быть полезно для задач, в которых линейная разделимость не является возможной.

Граф потока вычислений - это визуализация, которая отражает, как операции в нейронной сети соединяются для формирования конечного вывода. Он представляет собой направленный ациклический граф, где каждый узел представляет операцию (например, сложение, умножение, активацию), а ребра представляют тензоры, которые передаются между операциями. Это является основой для многих библиотек глубокого обучения, таких как TensorFlow и PyTorch, и позволяет эффективно вычислять градиенты с использованием метода обратного распространения ошибки.

## 17. Специфика задач машинного обучения на изображениях. Принцип работы сверточных сетей. Преимущества сверточных сетей при решении этих задач.

Задачи машинного обучения на изображениях, также известные как задачи компьютерного зрения, связаны с анализом и обработкой визуальной информации, представленной в виде изображений. Некоторые из таких задач включают классификацию изображений, обнаружение объектов, сегментацию изображений, распознавание образов и генерацию изображений.

Принцип работы сверточных сетей основан на использовании сверточных слоев, которые позволяют эффективно обрабатывать изображения. Сверточные слои представляют собой набор фильтров (ядер свертки), которые скользят по входному изображению, выполняя операцию свертки. Операция свертки состоит в поэлементном перемножении значений пикселей изображения с соответствующими значениями фильтра и суммировании результатов. Это позволяет выделять локальные особенности и закономерности на разных уровнях абстракции.

После сверточных слоев в сверточной сети обычно следуют слои подвыборки (pooling), которые позволяют уменьшить размерность представления, сохраняя наиболее важные характеристики. Затем полученные признаки передаются в полносвязные слои, которые выполняют классификацию или другую необходимую задачу.

Преимущества сверточных сетей при решении задач компьютерного зрения включают:

1. Локальность: сверточные сети способны захватывать локальные особенности и шаблоны в изображениях, что делает их эффективными для анализа пространственной структуры данных.
2. Параметры совместного использования: использование общих весов для фильтров свертки позволяет сети эффективно использовать параметры и обобщать свои знания на разные области изображений.
3. Инвариантность к пространственным трансформациям: сверточные сети устойчивы к небольшим изменениям в положении или масштабе объектов на изображении благодаря операции свертки.
4. Устойчивость к изменению масштаба: сверточные сети способны обнаруживать объекты на изображениях разного масштаба без необходимости явного изменения размера входных данных.
5. Иерархическое представление: сверточные сети создают иерархическое представление изображений, где более высокие уровни соответствуют более абстрактным концепциям, что облегчает классификацию и обнаружение объектов.
6. Автоматическое изучение признаков: сверточные сети способны автоматически извлекать и выделять важные признаки из изображений без необходимости вручную определять характеристики.

В целом, сверточные сети предоставляют мощный инструмент для обработки и анализа изображений, их эффективность и способность к изучению признаков делают их особенно полезными для задач компьютерного зрения.

## 18. Архитектура многослойной ИНС распознавания изображений на основе сверточных сетей.

Архитектура многослойной ИНС (нейронной сети) для распознавания изображений на основе сверточных сетей обычно состоит из нескольких типов слоев. Вот типичная архитектура такой сети:

1. Сверточные слои: Слои свертки выполняют операцию свертки на входном изображении с использованием нескольких фильтров. Каждый фильтр обнаруживает определенные признаки, такие как границы, текстуры или формы. Сверточные слои последовательно применяются для извлечения более абстрактных признаков на разных уровнях.
2. Слои подвыборки (pooling): Следующий тип слоев в архитектуре сверточной сети - это слои подвыборки. Они служат для уменьшения размерности пространства признаков, сохраняя важные характеристики. Обычно это достигается путем выбора максимального значения или усреднения значений в некотором окне.
3. Полносвязные слои: После сверточных и слоев подвыборки следуют полносвязные слои. Полносвязные слои соединяют каждый нейрон предыдущего слоя с каждым нейроном текущего слоя. Эти слои обрабатывают извлеченные признаки и выполняют классификацию или другие задачи, связанные с распознаванием изображений.
4. Выходной слой: В конце архитектуры сверточной сети обычно находится выходной слой, который принимает результаты от полносвязных слоев и генерирует соответствующий выход. Например, в задаче классификации изображений, выходной слой может состоять из нескольких нейронов, соответствующих различным классам, с использованием функции активации, такой как softmax.

В архитектуре сверточной сети могут присутствовать и другие слои или модификации, такие как слои нормализации, слои активации, слои снижения переобучения и т. д. Однако вышеперечисленные компоненты являются основными элементами архитектуры сверточных сетей для распознавания изображений.

## 19. Приемы для глубокого обучения на небольших наборах изображений.

Глубокое обучение требует больших объемов данных для эффективного обучения моделей. Однако, когда у нас есть ограниченный набор изображений, можно применить несколько приемов для повышения производительности и обобщающей способности модели. Вот несколько таких приемов для глубокого обучения на небольших наборах изображений:

1. Перенос обучения (Transfer Learning): Используйте предварительно обученные модели, предварительно обученные на больших наборах данных, и дообучайте их на своих небольших данных. Это позволяет модели использовать знания, полученные на других задачах, и может значительно улучшить производительность модели.
2. Аугментация данных (Data Augmentation): Применяйте различные преобразования к существующим изображениям для создания новых образцов данных. Например, можно применить случайные сдвиги, повороты, масштабирование, отражение и изменение яркости/контрастности. Это позволяет увеличить разнообразие данных и предотвратить переобучение модели.
3. Генерация синтетических данных: Если у вас есть достаточно знаний о задаче, можно создать синтетические изображения, которые имитируют реальные данные. Например, при обучении модели для распознавания лиц, можно создать синтетические изображения с различными освещением, позами и выражениями лица.
4. Регуляризация: Применяйте методы регуляризации, такие как L1 или L2 регуляризация, для контроля переобучения модели. Это поможет снизить сложность модели и улучшить ее обобщающую способность.
5. Постепенное обучение (Progressive Learning): Начните обучение модели на небольшом подмножестве данных и постепенно увеличивайте размер обучающей выборки. Это позволяет модели постепенно адаптироваться к новым данным и избегать переобучения на ранних этапах.
6. Ранняя остановка (Early Stopping): Остановите обучение модели, когда ее производительность на валидационной выборке перестает улучшаться. Это помогает предотвратить переобучение и сохранить лучшие обобщающие способности модели.
7. Использование малых моделей: Вместо использования глубоких моделей с большим числом параметров, рассмотрите использование более компактных архитектур, таких как MobileNet или SqueezeNet. Это позволяет снизить количество требуемых обучающих данных и ресурсов для обучения.

Комбинация этих приемов может помочь повысить производительность моделей глубокого обучения на небольших наборах изображений и сделать их более обобщающими.

## 20. Схема работ слоя сверточной сети. Пулинг. Гиперпараметры: padding, kernel size, stride, dilation.



Схема работы слоя сверточной сети:

1. Вход: Входной слой принимает изображение или выход предыдущего слоя. Изображение представлено в виде трехмерного тензора, где первое измерение соответствует высоте, второе измерение — ширине, а третье измерение — числу каналов изображения (например, RGB изображение имеет три канала).
2. Свертка: В слое свертки применяются фильтры (ядра свертки) к входному изображению. Каждый фильтр представлен трехмерным тензором, который "скользит" по входу и выполняет операцию свертки. Результатом является карты признаков, которые представляют собой трехмерные тензоры, где первые два измерения соответствуют высоте и ширине карты, а третье измерение — числу фильтров.
3. Функция активации: После операции свертки на карты признаков обычно применяется нелинейная функция активации, такая как ReLU (Rectified Linear Unit), для добавления нелинейности в сеть. Это помогает модели изучать более сложные и абстрактные представления данных.
4. Пулинг: Слой пулинга (или субдискретизации) следует после слоя свертки и служит для уменьшения размерности карты признаков. Наиболее распространенным типом пулинга является слой максимального подвыборки (max pooling), который выбирает максимальное значение в каждом окне пулинга. Это позволяет уменьшить количество параметров и сделать представления более инвариантными к небольшим сдвигам искомых признаков.
5. Выход: Выход слоя сверточной сети передается в следующий слой сверточной сети или полносвязный слой для дальнейшей обработки.

Гиперпараметры слоя сверточной сети:

1. Padding (заполнение): Параметр, определяющий, каким образом обрабатывать границы изображения при операции свертки. Значение "valid" означает отсутствие заполнения, а "same" означает заполнение, чтобы сохранить размеры входа.
2. Kernel Size (размер ядра свертки): Размер фильтра, используемого для операции свертки. Он задается в виде двухмерного целочисленного значения, например, (3, 3) для квадратного ядра 3x3.
3. Stride (шаг): Параметр, определяющий шаг, с которым фильтр перемещается по входной карте признаков. Значение 1 означает перемещение с шагом 1 пиксель, а большие значения увеличивают шаг.
4. Dilation (диляция): Параметр, определяющий интервал между элементами фильтра при операции свертки. Значение 1 означает, что фильтр применяется на каждый пиксель, а большие значения добавляют пропуски между элементами фильтра.

Комбинация этих гиперпараметров и их значений определяет размеры и характеристики операций свертки и пулинга в слоях сверточной сети, что влияет на ее способность извлекать признаки и классифицировать изображения.

## 21. Задачи обработки текста: дистрибутивная семантика, матрица совместной встречаемости, представление слов в виде векторов малой размерности.

Задачи обработки текста, такие как дистрибутивная семантика, матрица совместной встречаемости и представление слов в виде векторов малой размерности, являются ключевыми в области обработки естественного языка (Natural Language Processing, NLP). Рассмотрим каждую из этих задач подробнее:

1. Дистрибутивная семантика: Дистрибутивная семантика изучает связь между значением слова и его контекстом в тексте. Она основана на предположении, что слова, встречающиеся в похожих контекстах, имеют схожие значения. Для решения задачи дистрибутивной семантики используются методы анализа статистических связей между словами, такие как матрица совместной встречаемости.
2. Матрица совместной встречаемости: Матрица совместной встречаемости представляет собой таблицу, в которой каждый элемент указывает на количество раз, когда два слова встречаются в одном контексте. Путем анализа этой матрицы можно выявить семантические связи между словами. Например, если два слова часто встречаются в одном контексте, то вероятно, они имеют схожие значения. Матрица совместной встречаемости может быть использована для вычисления различных метрик, таких как коэффициент сходства, семантическая близость или факторы важности слов.
3. Представление слов в виде векторов малой размерности: Представление слов в виде векторов малой размерности (word embeddings) позволяет кодировать семантическую информацию о словах. Одним из популярных методов является Word2Vec, который обучает векторные представления слов на основе их контекста в большом корпусе текста. Word embeddings позволяют выполнять алгебраические операции с словами, такие как вычисление семантических расстояний или нахождение ближайших соседей по смыслу.

Эти задачи и методы имеют важное значение в NLP, позволяя моделям понимать и обрабатывать текстовые данные на более семантическом уровне. Они применяются в различных задачах, таких как определение тональности, машинный перевод, анализ сентиментов и многих других.

## 22. Word2vec: модель CBOW.

Идея word2vec состоит в том, чтобы напрямую обучать вектора слов малой размерности

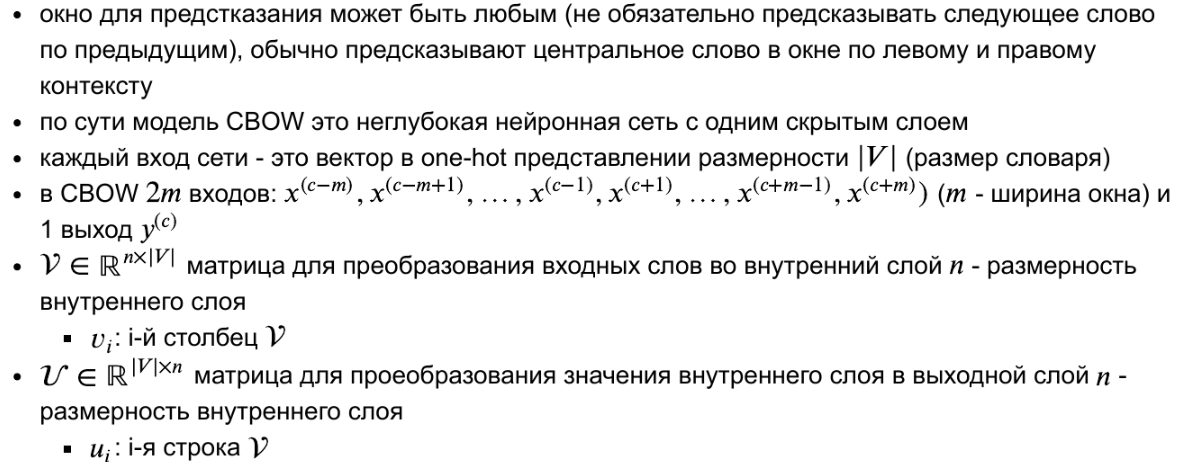
* A neural probabilistic language model (Bengio et al., 2003)
* word2vec (Mikolov et al. 2013) A recent, even simpler and faster model
* Faster and can easily incorporate a new sentence/document or add a word to the vocabulary

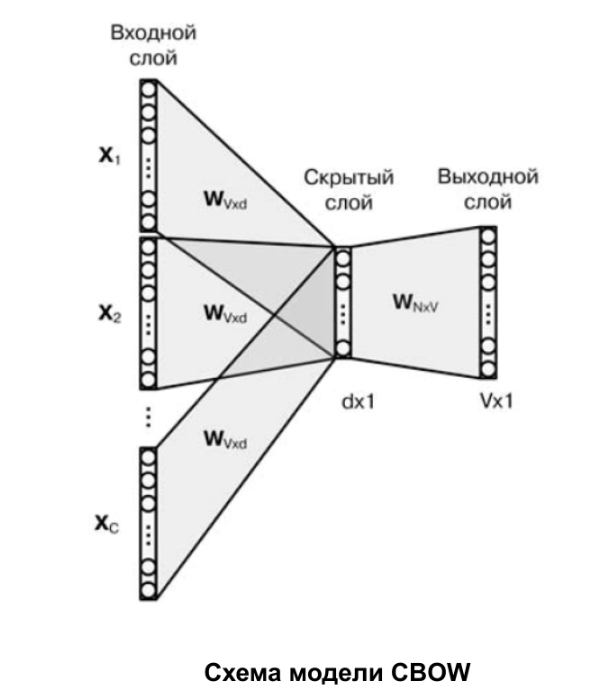
Для word2vec разработаны два основных алгоритма обучения:

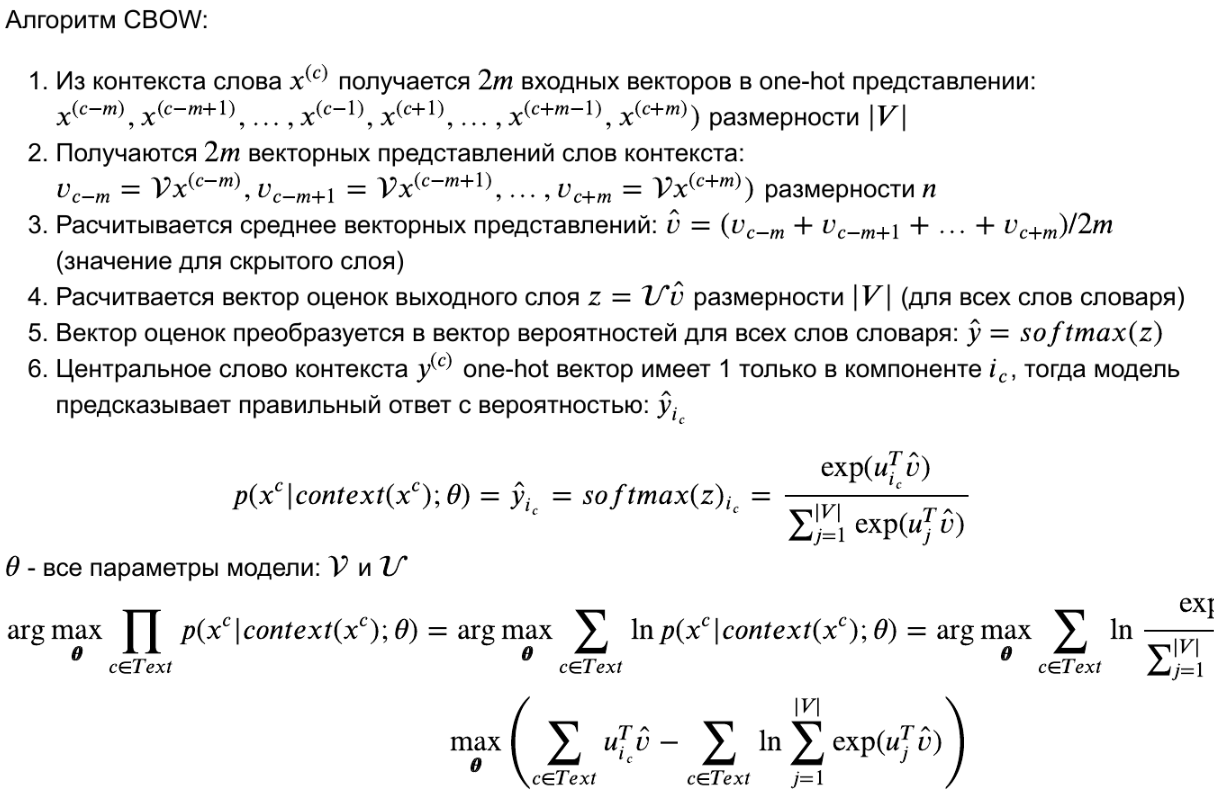
* CBoW (англ. Continuous Bag of Words, «непрерывный мешок со словами», bag — мультимножество) предсказывает текущее слово, исходя из окружающего его контекста.
* Skip-gram - использует текущее слово, чтобы предугадывать окружающие его слова.

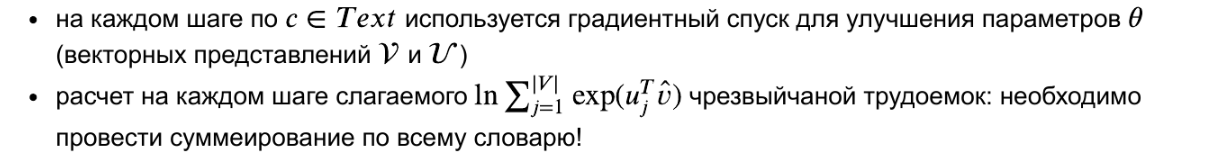
Модель CBOW

Модель CBOW учится как можно лучше по заданному контексту слова восстановить само слово.









Пример реализации на всякий случай:

class CBOW(nn.Module):

def \_\_init\_\_(self, vocab\_size, embd\_size, context\_size, hidden\_size):

super(CBOW, self).\_\_init\_\_()

self.embeddings = nn.Embedding(vocab\_size, embd\_size)

self.linear1 = nn.Linear(2\*context\_size\*embd\_size, hidden\_size)

self.linear2 = nn.Linear(hidden\_size, vocab\_size)

def forward(self, inputs):

embedded = self.embeddings(inputs).view((1, -1))

hid = F.relu(self.linear1(embedded))

out = self.linear2(hid)

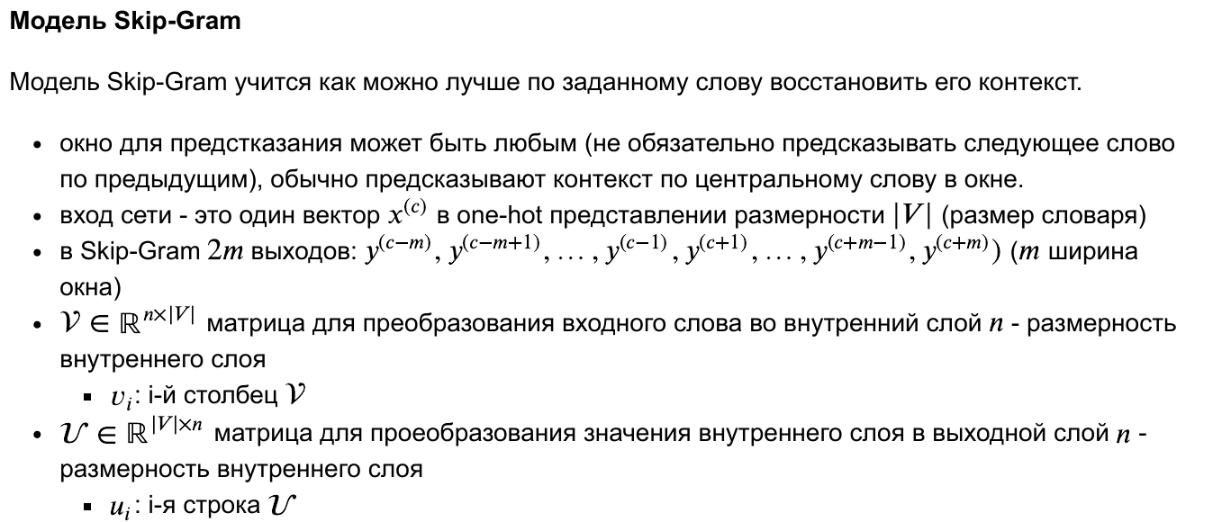
log\_probs = F.log\_softmax(out)

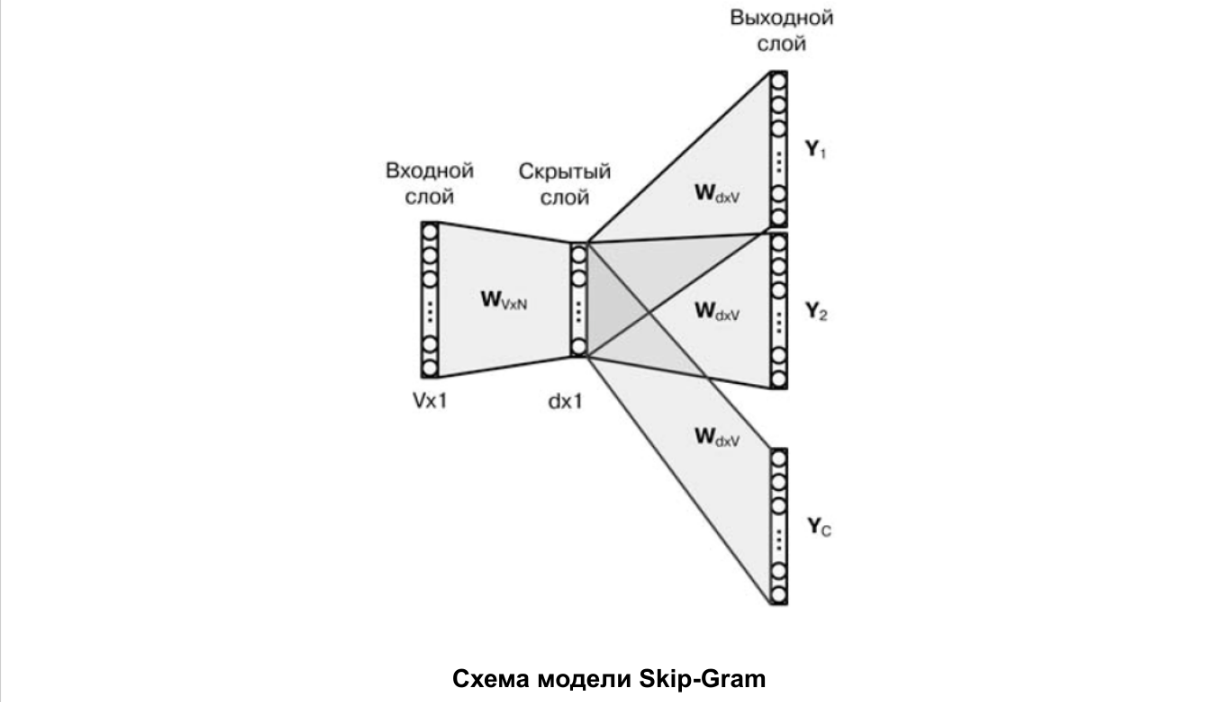
return log\_probs

## 23. Word2vec: модель Skip-Gram.

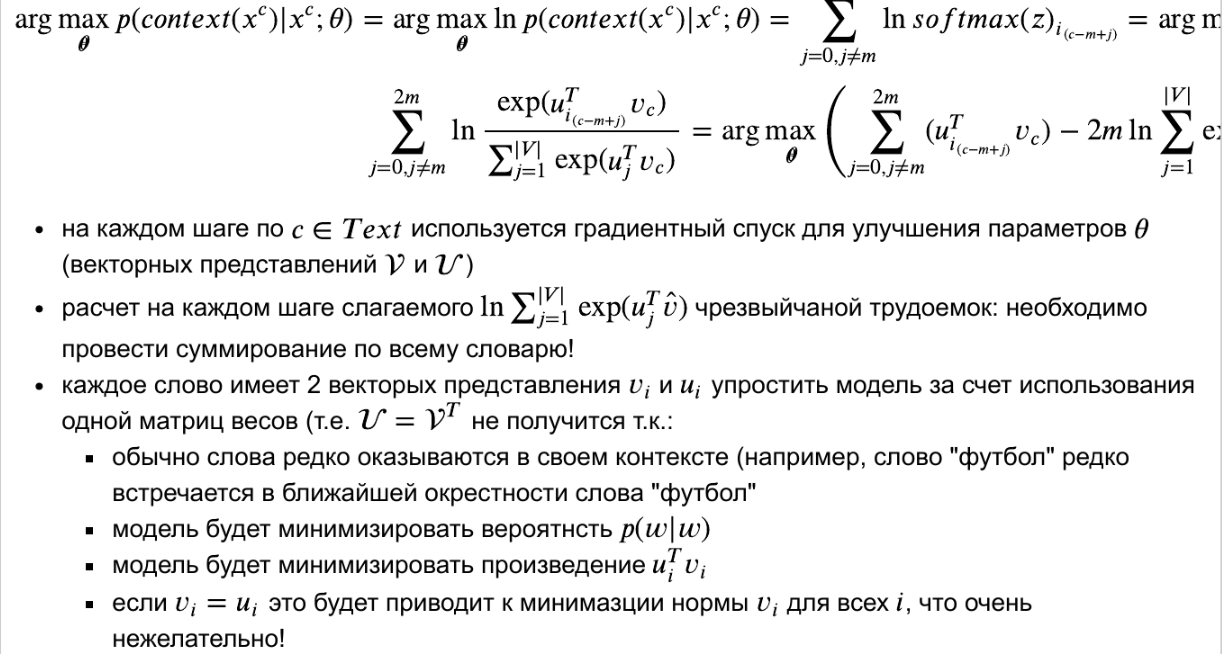
Идея word2vec состоит в том, чтобы напрямую обучать вектора слов малой размерности

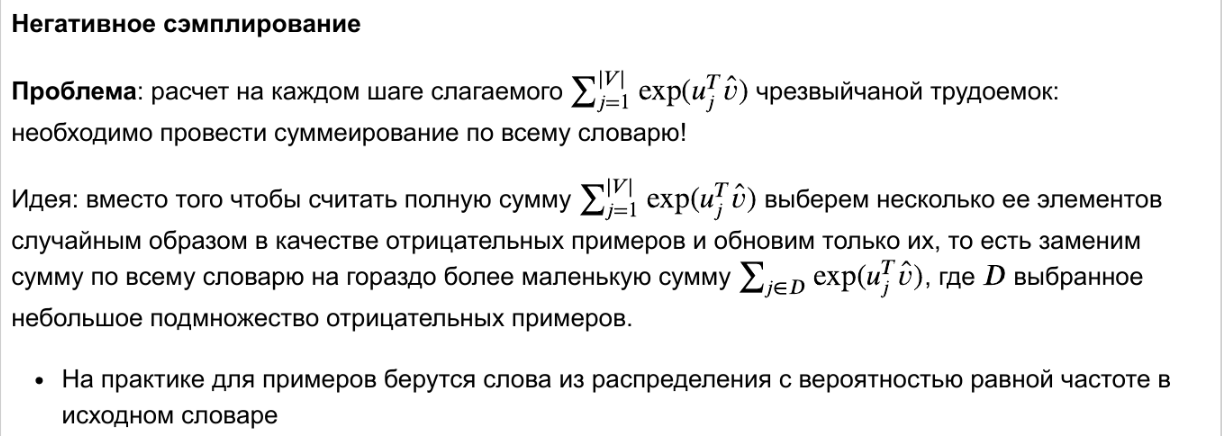
Для word2vec разработаны два основных алгоритма обучения:

* CBoW (англ. Continuous Bag of Words, «непрерывный мешок со словами», bag — мультимножество) предсказывает текущее слово, исходя из окружающего его контекста.
* Skip-gram - использует текущее слово, чтобы предугадывать окружающие его слова.









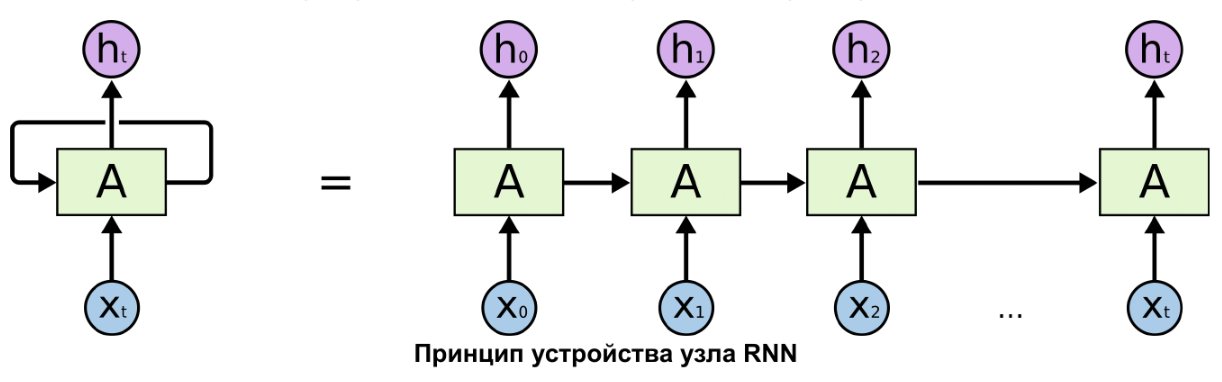
## 24. Рекуррентная нейронная сеть, принципы ее обучения. Сложности применения рекуррентных нейронных сетей.

Рекуррентная нейронная сеть (RNN) - это класс искусственных нейронных сетей, в которых узел может получать входы не только от других узлов и текущих входных данных но и выходы узлов, полученные при рассмотрении предыдущих входных данных последовательности.

* обмен вектором внутреннего состояния, полученного на предыдущем шаге, позволяет использовать информацию о предыдущих шагах, которые сеть уже обработала
* при рассмотрении всей последовательности веса каждого узла одни и те же при рассмотрении всех входных данных последовательности
* Такая архитектура сети позволяет обрабатывать серии событий во времени или последовательные пространственные цепочки произвольной размерности

Трудность рекуррентной сети:

* если учитывать каждый шаг времени, то становится необходимым для каждого шага времени (последовательности) создавать свой слой нейронов, что создает серьезные вычислительные сложности
* многослойные реализации вычислительно неустойчивы: в них как правило либо исчезают либо зашкаливают веса
* если ограничить расчет фиксированным временным окном, то полученные модели не будут отражать долгосрочных трендов



Распространение ошибок в узле RNN

𝑎𝑡=𝑏+𝑊𝑠𝑡−1+𝑈𝑥𝑡

𝑠𝑡=𝑓(𝑎𝑡)

𝑜𝑡=𝑐+𝑉𝑠𝑡

𝑦𝑡=ℎ(𝑜𝑡)

Где:

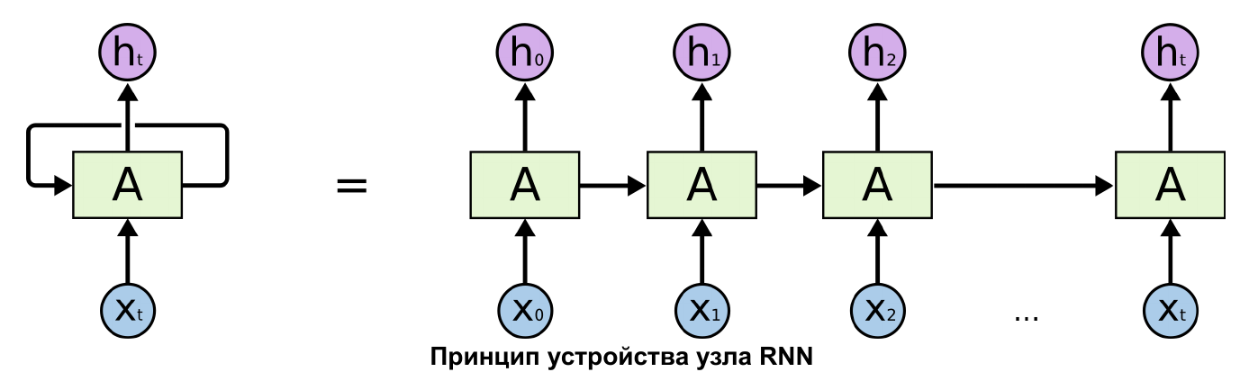
𝑓 — это нелинейность рекуррентной сети (обычно 𝜎 , tanh или 𝑅𝑒𝐿𝑈 )

ℎ — функция, с помощью которой получается ответ (например, 𝑠𝑜𝑓𝑡𝑚𝑎𝑥 )



Распространение ошибки в архитектуре RNN

* В прямой нейронной сети ошибка на конкретном нейроне вычисляется как функция от ошибок нейронов, которые используют его выходное значение формируется ациклический графы вычислений:
* В архитектуре RNN нейрон принимает в качестве входа результат вычисления в нем самом (через вектор состояния)
* Важно понимать, что при этом петли в графе вычислений не образуется
* Вычисления, которые делает рекуррентная сеть, можно развернуть обратно до начала обрабатываемой последовательности
* Можно сказать, что на каждом шаге обрабатываемой последовательности сеть создает копии самой себя



И на каждом последовательности мы фактически обучаем глубокую нейронную сеть, в которой столько слоев, сколько элементов в последовательности на данный момент мы уже видели

Рекуррентная сеть — разворачиваться вдоль элементов последовательности 1… в очень-очень многоуровневую обычную сеть, в которой одни и те же веса переиспользуются на каждом уровне.

* Для хранения весов достаточно одной матрицы
* Градиенты по весам не затухают до нуля сразу же (как это бывает в обычных глубоких сетях)
* Если матрица весов меняет норму вектора градиента при проходе через один «слой» обратного распространения, то при проходе через T слоев эта норма изменяется экспоненциально (т.к. веса матрицы одни и те же) это приводит:
  + к "взрыву градиентов" (exploding gradients), если матрица заметно увеличивает норму вектора градиента
  + к экспоненциальному затуханию градиентов (Vanishing gradients), если матрица заметно уменьшает норму вектора градиента

пример реализации:

Модель:

torch.nn.RNN(\*args, \*\*kwargs)

class Model(nn.Module):

def \_\_init\_\_(self, input\_size, output\_size, hidden\_dim, n\_layers):

super(Model, self).\_\_init\_\_()

self.hidden\_dim = hidden\_dim

self.n\_layers = n\_layers

# RNN Layer

self.rnn = nn.RNN(input\_size, hidden\_dim, n\_layers, batch\_first=True)

self.fc = nn.Linear(hidden\_dim, output\_size)

def forward(self, x):

batch\_size = x.size(0)

hidden = self.init\_hidden(batch\_size)

out, hidden = self.rnn(x, hidden)

out = out.contiguous().view(-1, self.hidden\_dim)

out = self.fc(out)

return out, hidden

def init\_hidden(self, batch\_size):

hidden = torch.zeros(self.n\_layers, batch\_size, self.hidden\_dim).to(device)

return hidden

# Создаем инстанс модели с установленными гиперпараметрами

model = Model(input\_size=dict\_size, output\_size=dict\_size, hidden\_dim=12, n\_layers=1)

model = model.to(device)

n\_epochs = 100

lr=0.01

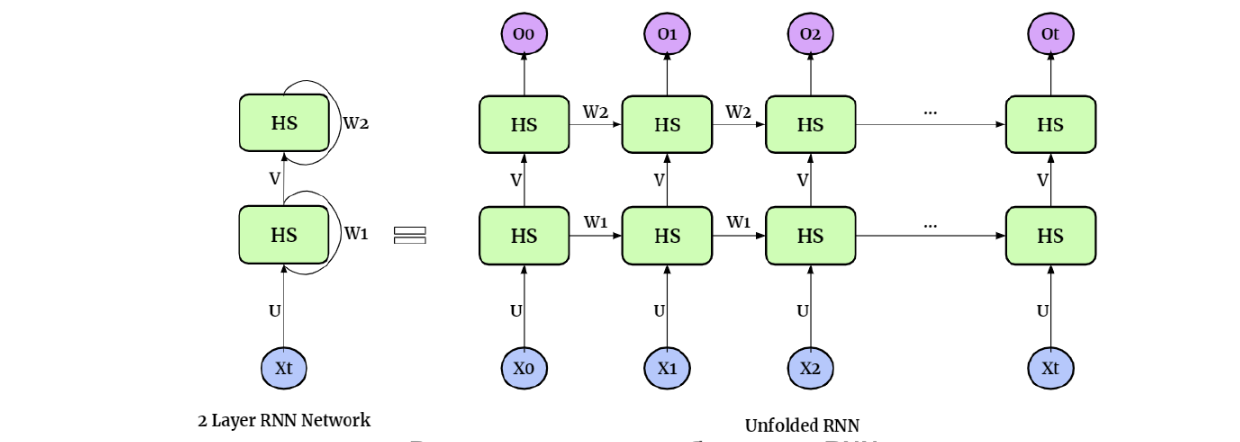
criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=lr)

Построение многослойных нейронных сетей на базе архитектуры RNN

Рассмотрим всю рекуррентную сеть как слой и используем ее выходы как входы для следующего рекуррентного слоя.

Мотивация: каждый слой действует в своем собственном «масштабе времени», примерно как каждый слой сверточной сети действует в своем масштабе, на свой размер окна входов.



Двунаправленные рекуррентные сети (bidirectional RNN): для входной последовательности запустим RNN (обычно с разными весами) два раза: один слой будет читать последовательность слева направо, а другой — справа налево

* Матрицы весов для двух направлений абсолютно независимы и между ними нет взаимодействия
* Ограничение: данный подход возможен только для последовательностей, которые даны сразу целиком (например для предложений естественного языка).
* Мотивация в том, чтобы получить состояние, отражающее контекст и слева, и справа для каждого элемента последовательности (например для отнесения слова к части речи т.к. важно анализировать все предложение, и слева, и справа от слова)
* Вместо классической рекуррентной сети из трех матриц, может использоваться любая другая конструкция, например LSTM или GRU.

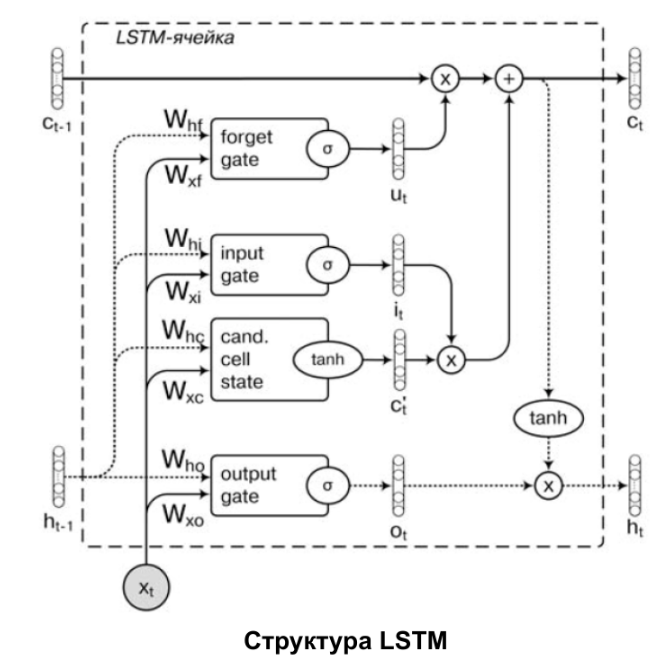
## 25. Модуль LSTM.

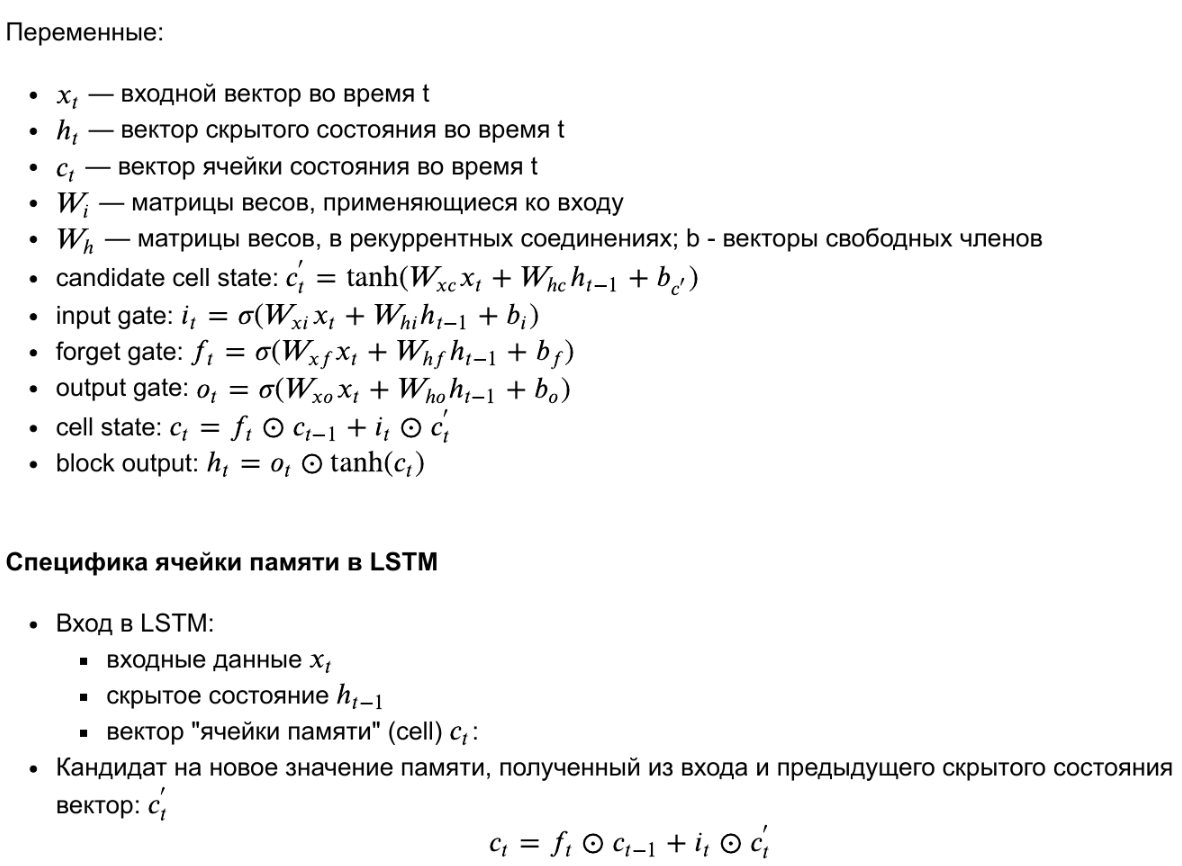
LSTM (Long Short-Term Memory)

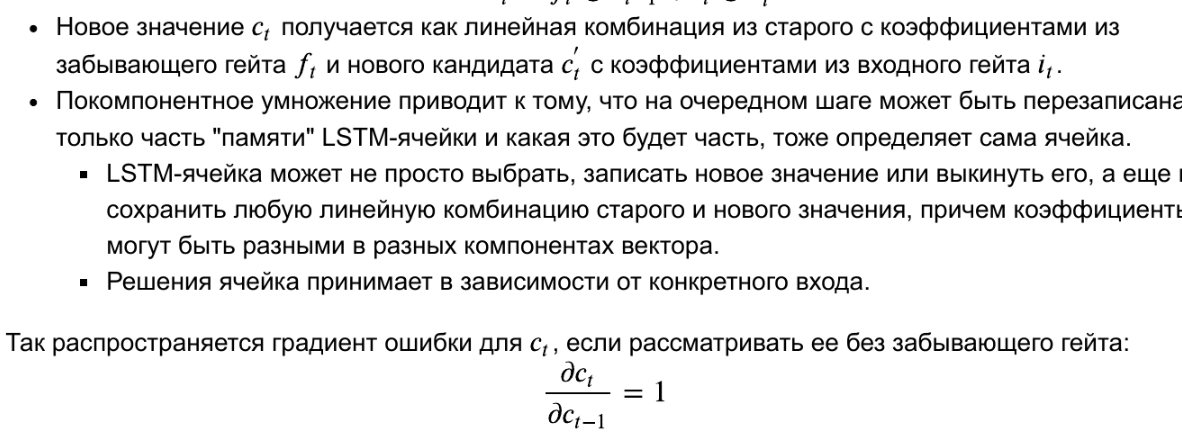
* обычные рекуррентные сети очень плохо справляются с ситуациями, когда нужно что-то «запомнить» надолго: влияние скрытого состояния или входа с шага t на последующие состояния рекуррентной сети экспоненциально затухает
* LSTM хорошо приспособлена к обучению на задачах классификации, обработки и прогнозирования временных рядов в случаях, когда важные события разделены временными лагами с неопределённой продолжительностью и границами вместо одного-единственного числа, на которое влияют все последующие состояния, используется специального вида ячейка моделирующая "долгую память"
* LSTM моделирует процессы записи и чтения из этой "ячейки памяти"
* у ячейки не один набор весов, как у обычного нейрона, а сразу несколько

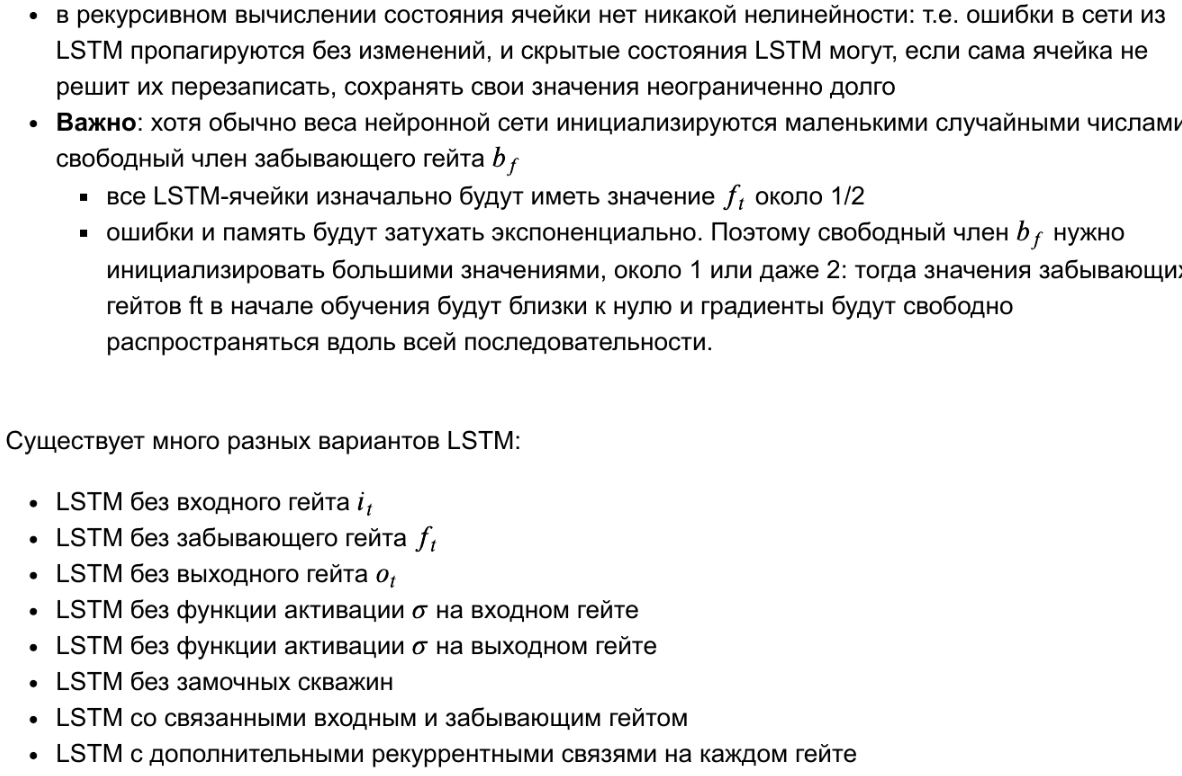
В LSTM есть три основных вида узлов, которые называются гейтами:

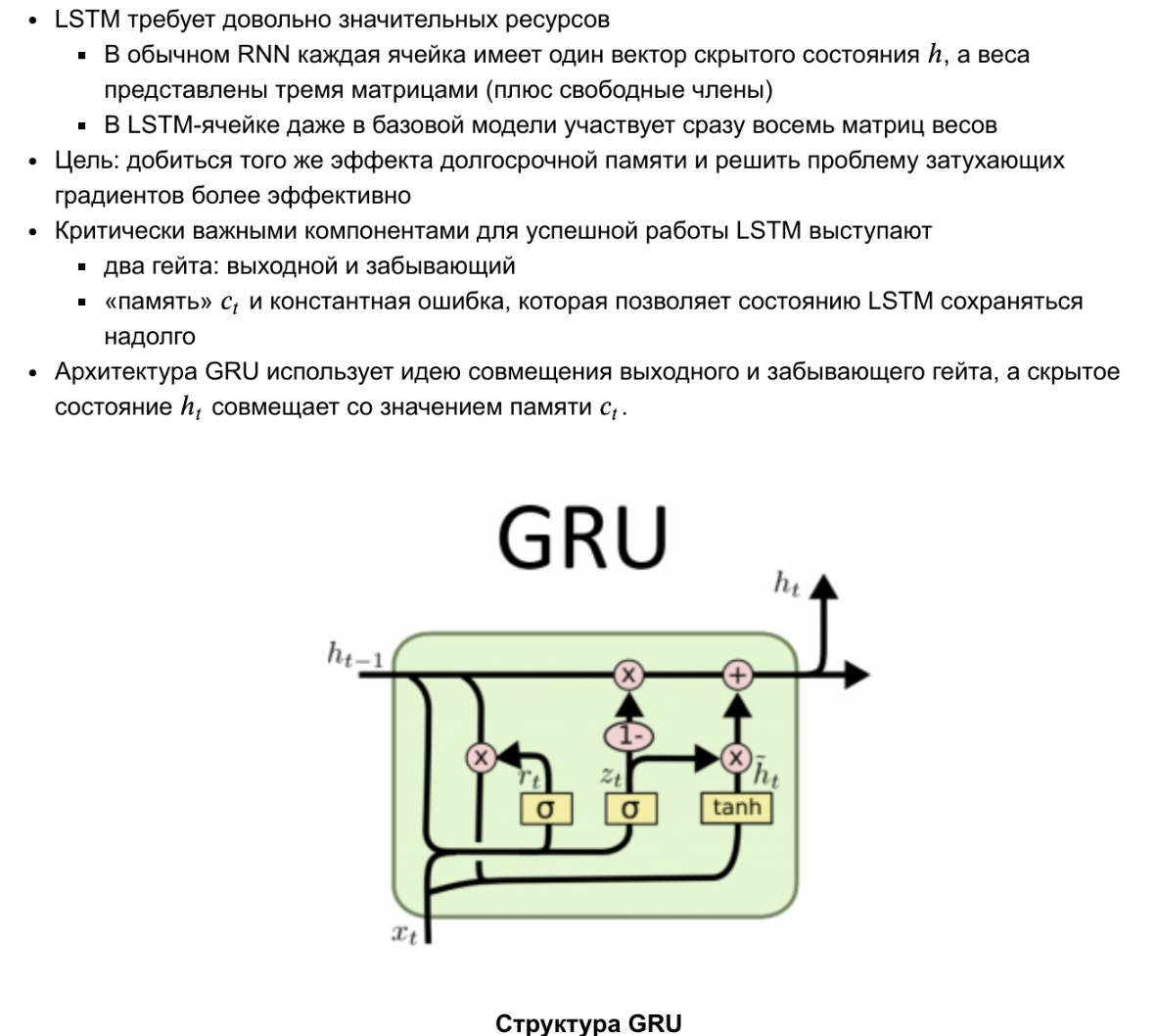
* входной (input gate)
* забывающий (forget gate)
* выходной (output gate)
* рекуррентная ячейка со скрытым состоянием

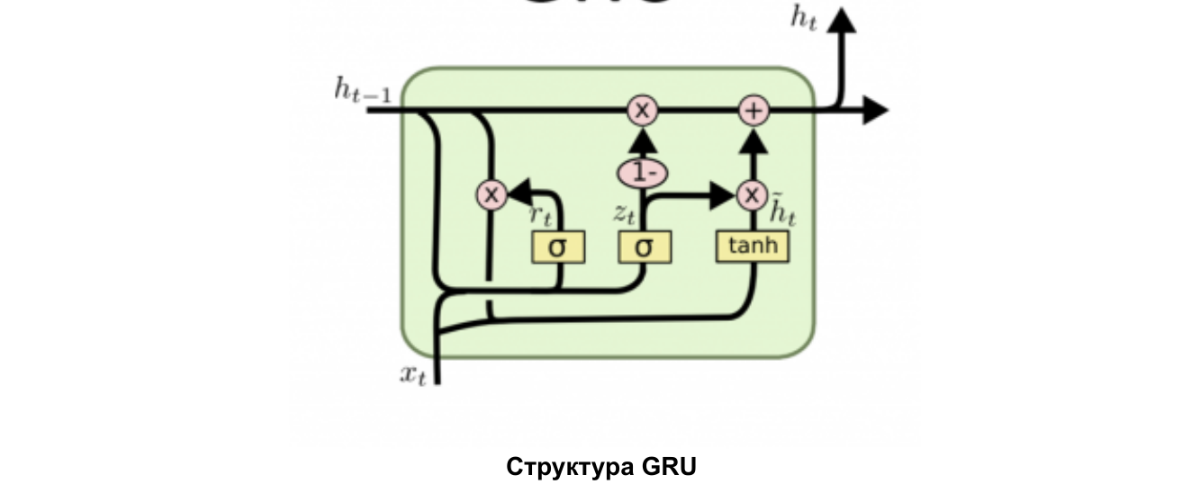












Модель:

torch.nn.LSTM(\*args, \*\*kwargs)

пример использования:  
class RNNModule(nn.Module):

def \_\_init\_\_(self, n\_vocab, seq\_size, embedding\_size, lstm\_size):

super(RNNModule, self).\_\_init\_\_()

self.seq\_size = seq\_size

self.lstm\_size = lstm\_size

self.embedding = nn.Embedding(n\_vocab, embedding\_size)

self.lstm = nn.LSTM(embedding\_size, lstm\_size, batch\_first=True)

self.dense = nn.Linear(lstm\_size, n\_vocab)

def forward(self, x, prev\_state):

embed = self.embedding(x)

output, state = self.lstm(embed, prev\_state)

logits = self.dense(output)

return logits, state

def zero\_state(self, batch\_size):

return (torch.zeros(1, batch\_size, self.lstm\_size), torch.zeros(1, batch\_size, self.lstm\_size))

## 26. Механизм Attention. Пример использования Attention.

После успеха этой методики в машинном переводе последовали ее внедрения в:

* других задачах обработки естественного языка
* задачах генерации описания изображения (в применении к CNN)
* в порождающих состязательных сетях (GAN)

Специфика:

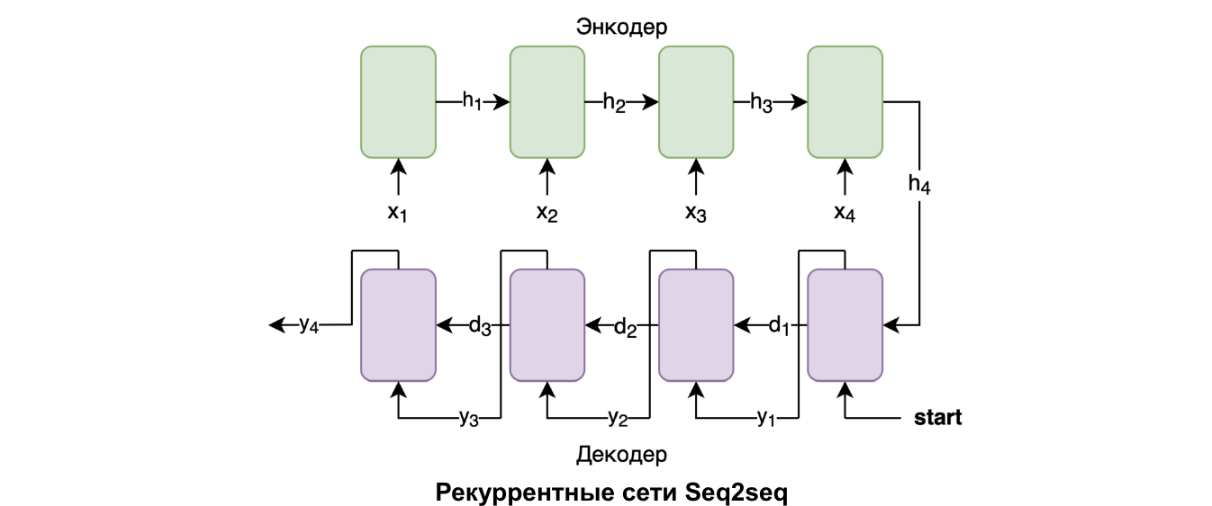
* Для энкодера и декодера используются различные веса сети (RNN модуля)
* Скрытое состояние энкодера на последнем шаге обработки предложения (последовательности) является ключевым т.к. по сути кодирует все предложение. Затем декодер использует именно это состояние для того, чтобы сгенерировать перевод предложения на другом языке.
* Результат сэмплинга декодера на предыдущем шаге передается на следующий блок декодера в добавок к внутреннему состоянию RNN декодера.

Проблемы:

* Расстояние между местом кодирования слова и местом его декодирования большое.
* Последовательность слов в исходном и целевом предложении часто должна быть разная.
* Часто количество слов в исходном и целевом предложении различается.

Базовая архитектура Seq2seq

Seq2seq состоит из двух рекуррентных нейронных сетей (RNN): Энкодера и Декодера.



Энкодер - принимает последовательность (например: предложение на языке A) и сжимает его в вектор скрытого состояния.

Декодер - выдает слово на языке B, принимает последнее скрытое состояние энкодера (для первого слова) / предыдущее скрытое состояние декодера (последующие слова) и предыдущее предсказанное слово.

Применение механизма внимания для Seq2seq: базовый принцип

Использование механизма внимания позволяет решать задачу нахождения закономерности между словами находящимися на большом расстоянии друг от друга. LSTM, GRU и аналогичные блоки используются для улучшения передачи информации на большое количество итераций по сравнению с базовыми RNN, несмотря на это сохраняется проблема: влияние предыдущих состояний на текущее уменьшается экспоненциально от расстояния между словами

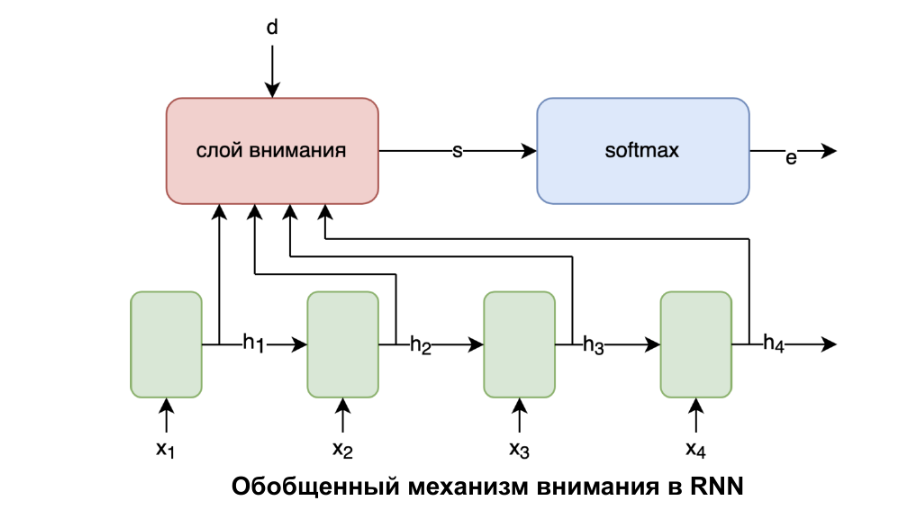
В классическом применении RNN результатом является только последнее скрытое

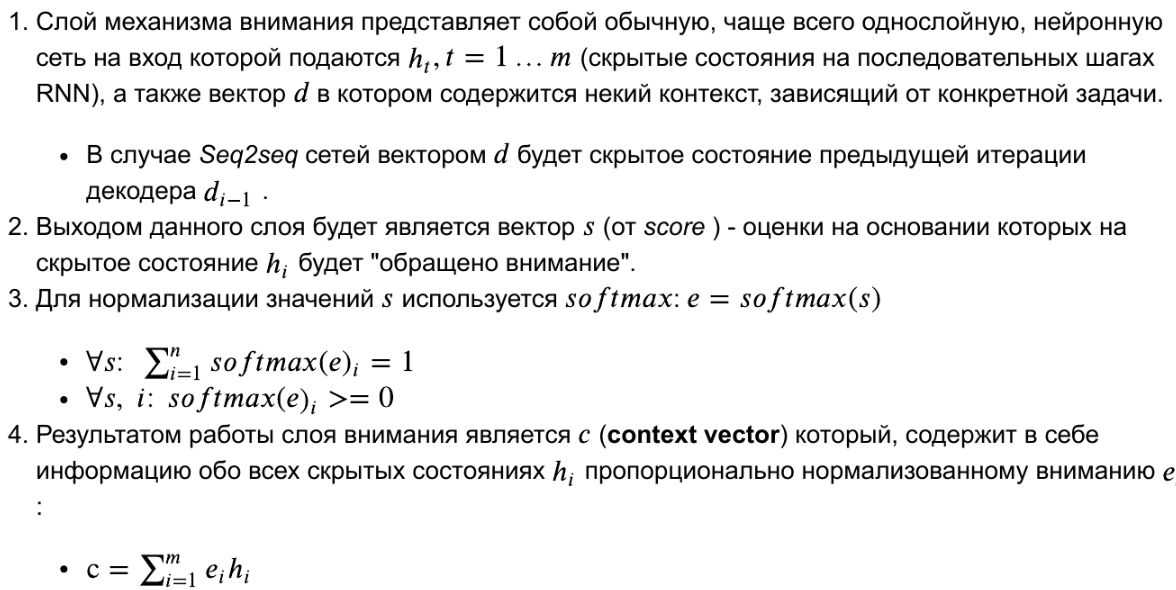
состояние ℎ , где - длина последовательности входных данных.

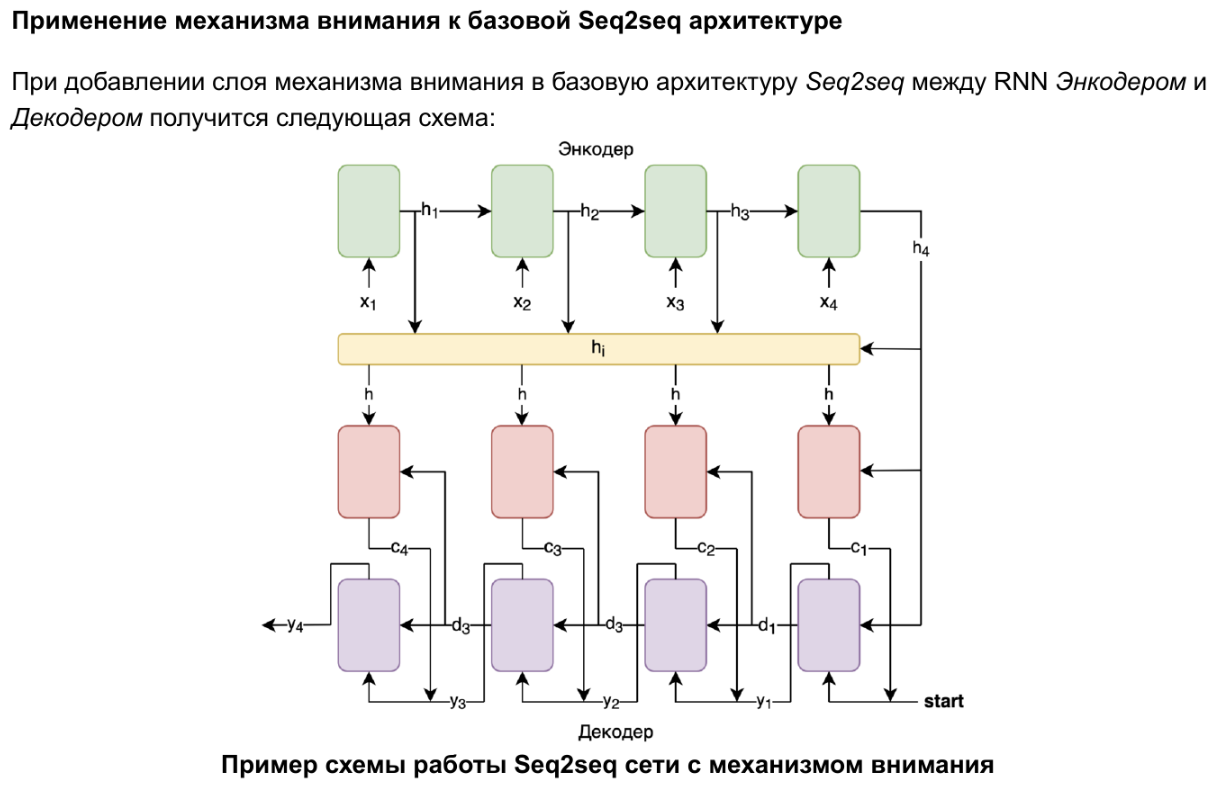
Механизм внимания улучшает этот показатель до линейного.

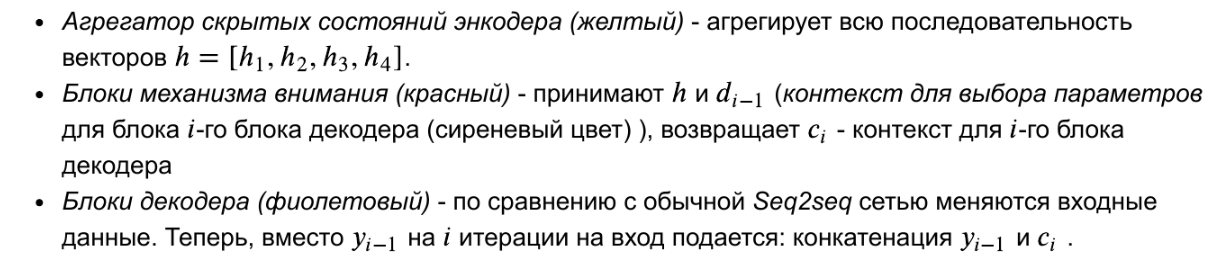
Использование механизма внимания позволяет использовать информацию, полученную не только из последнего скрытого состояния, но и из любого скрытого состояния ℎ для любого

t ∈ 1…m .









При помощи механизма внимания достигается "фокусирование" декодера на определенных скрытых состояниях энкодера, на любом из шагов энкодера.

В случаях машинного перевода эта возможность помогает декодеру предсказывать на какие скрытые состояния при исходных определенных словах на языке A необходимо обратить больше внимания при генерации очередного слова перевода на язык B.

При этом расстояние (в шагах последовательности) между генерируемым словом и

влияющими на него словами из исходной последовательности никак не влияют на

возможность использование скрытых состояний, сформированных под их воздействием.

Для других задач

Задача Image Captioning : на вход подается изображение, а на выходе создается текстовое

предложение, описывающее визуальное содержание картинки.

Классическая нейростевая модель для решения задачи Image Captioning состоит из: сверточной нейронной сети (CNN) для извлечения визуальных характеристик из картинки и рекуррентной нейронной сети (RNN) для перевода результата работы сверточной сети в текст.

Сверточный модуль внимания (сonvolutional block attention module) — модуль внимания для сверточных нейросетей. Применяется для задач детектирования объектов на изображениях и классификации с входными данными больших размерностей.

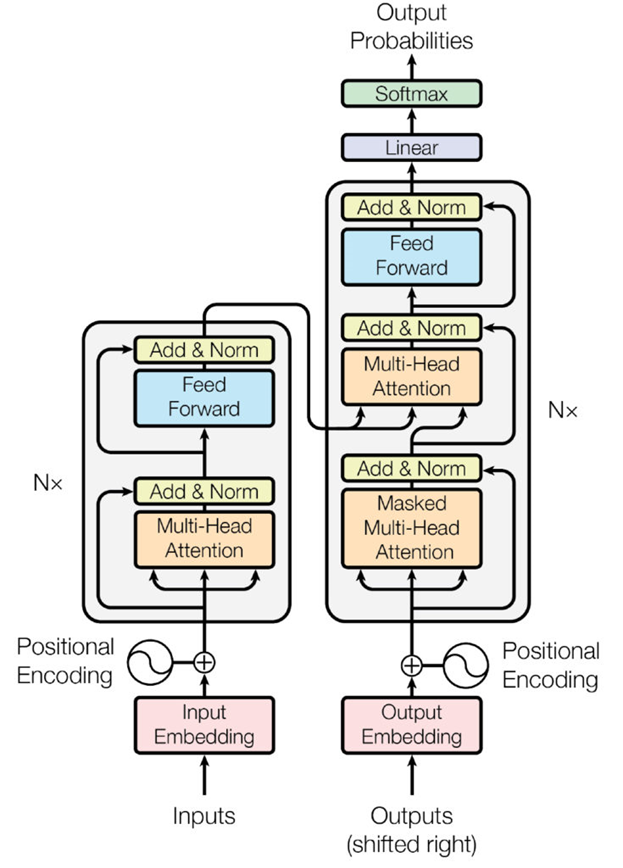
## 27. Архитектура Transformer.

### Трансформер

Трансформеры — относительно новый тип нейросетей, направленный на решение последовательностей с легкой обработкой дальнодействующих зависимостей. На сегодня это самая продвинутая техника в области обработки естественной речи (NLP).

С их помощью можно переводить текст, писать стихи и статьи и даже генерировать компьютерный код.В отличие от рекуррентных нейронных сетей (RNN), трансформеры не обрабатывают последовательности по порядку. Например, если исходные данные — текст, то им не нужно обрабатывать конец предложения после обработки начала. Благодаря этому такую нейросеть можно распараллелить и обучить значительно быстрее.

Основными компонентами трансформеров являются энкодер и декодер.



Энкодер преобразовывает входящую информацию (например, текст) и конвертирует ее в вектор (набор чисел). Декодер, в свою очередь, расшифровывает ее в виде новой последовательности (например, ответ на вопрос) слов на другом языке — смотря для каких целей создавалась нейросеть.

Трансформеры изначально позиционировались как нейросеть для обработки и понимания естественного языка. За четыре года с момента их появления они обрели популярность и появились во множестве сервисов, используемых ежедневно миллионами людей.

Одним из самых простых примеров является языковая модель BERT компании Google, разработанная в 2018 году.

Другой пример популярной языковой модели на базе трансформеров — это GPT (Generative Pre-trained Transformer) компании OpenAI.

Другие инновации, лежащие в основе трансформеров, сводятся к трем основным концепциям:

· позиционные энкодеры (Positional Encodings);

· внимание (Attention);

· самовнимание (Self-Attention).

### Позиционные энкодеры

Начнем с первого — позиционных энкодеров. Допустим, необходимо перевести текст с английского на русский. Стандартные модели RNN «понимают» порядок слов и обрабатывают их последовательно. Однако это затрудняет распараллеливание процесса.

Позиционные кодировщики позволяют преодолеть этот барьер. Идея состоит в том, чтобы взять все слова во входной последовательности — в данном случае английское предложение — и добавить к каждому номер в его порядке.

### Внимание

Внимание помогает избегать таких ситуаций как например перевод фраз с одного языка на другой. Его механизм позволяет текстовой модели «смотреть» на каждое слово в исходном предложении при принятии решения о том, как их переводить. Узнать модели, на какие слова «обращать внимание» на каждом шаге, помогают тренировочные данные. Наблюдая за тысячами английских и французских предложений, алгоритм узнает взаимозависимые типы слов. Он учится учитывать пол, множественность и другие правила грамматики.

### Самовнимание

Последняя часть трансформеров — это поворот внимания, называемый «самовниманием».

Если внимание помогает выравнивать слова при переводе с одного языка на другой, то самовнимание позволяет модели понимать смысл и закономерности языка.

### Обобщенно и кратко

Трансформеры – это новый тип нейросетей, который позволяет решать последовательности с легкой обработкой дальнодействующих зависимостей. Они являются самой продвинутой техникой в области обработки естественного языка (NLP), позволяя переводить текст, писать стихи и статьи, генерировать компьютерный код и многое другое. Трансформеры не обрабатывают последовательности по порядку, благодаря чему их можно распараллелить и обучить значительно быстрее, чем рекуррентные нейронные сети (RNN).

Основными компонентами трансформеров являются энкодер и декодер. Энкодер преобразовывает входящую информацию (например, текст) и конвертирует ее в вектор. Декодер расшифровывает конвертированный вектор в виде новой последовательности (например, ответ на вопрос) слов на другом языке, смотря для каких целей создавалась нейросеть.

Трансформеры позиционируются как нейросеть для обработки и понимания естественного языка. Они обрели популярность и появились во множестве сервисов, используемых ежедневно миллионами людей. Языковая модель BERT компании Google и GPT (Generative Pre-trained Transformer) компании OpenAI являются популярными примерами языковых моделей на базе трансформеров.

Основные концепции, лежащие в основе трансформеров, сводятся к позиционным энкодерам, вниманию и самовниманию. Позиционные энкодеры позволяют обрабатывать слова во входной последовательности и добавлять к каждому слову номер в его порядке. Внимание помогает модели «смотреть» на каждое слово в исходном предложении при принятии решения о том, как их переводить. Самовнимание позволяет модели понимать смысл и закономерности языка.

## 28. Модель BERT.

**BERT (Bidirectional Encoder Representations from Transformers)** — это языковая модель, разработанная Google в 2018 году. Она является одной из самых продвинутых языковых моделей на базе трансформеров и используется для обработки естественного языка. BERT обучается на огромном объеме текстовых данных, что позволяет ему понимать контекст и смысл слов в предложении.

BERT используется для таких задач, как анализ тональности текста, ответы на вопросы, генерация текстов и многое другое. Он может использоваться как самостоятельно, так и в сочетании с другими моделями машинного обучения.

BERT обучается на огромных корпусах текстов, что позволяет ему понимать контекст и смысл слов в предложении. Он использует методы self-attention, что позволяет ему лучше понимать зависимости между словами в предложении.

BERT имеет две основные модификации: BERT-base и BERT-large. BERT-large имеет большую мощность, но требует большего количества вычислительных ресурсов.

BERT стал очень популярной моделью, используемой во многих сервисах, таких как поисковые системы, машинный перевод и другие. Он также является одной из основных моделей, используемых в обработке естественного языка и машинном обучении.

Архитектура BERT состоит из двух частей: энкодера и декодера. Энкодер преобразовывает предложение в вектор фиксированной длины, а декодер использует полученный вектор для выполнения задачи, например, предсказания следующего слова в предложении. BERT обучается на огромном количестве данных, что позволяет ему изучать контекст и смысл слов в предложении.

При подаче текста на вход сети сначала выполняется его токенизация. Токенами служат слова, доступные в словаре, или их составные части — если слово отсутствует в словаре, оно разбивается на части, которые в словаре присутствуют (см. рис. 1). Словарь является составляющей модели — так, в BERT-Base используется словарь около 30,000 слов. В самой нейронной сети токены кодируются своими [векторными представлениями (англ. embeddings), а именно, соединяются представления самого токена (предобученные), номера его предложения, а также позиции токена внутри своего предложения. Входные данные поступают на вход и обрабатываются сетью параллельно, а не последовательно, но информация о взаимном расположении слов в исходном предложении сохраняется, будучи включённой в позиционную часть эмбеддинга соответствующего токена.

Выходной слой основной сети имеет следующий вид: поле, отвечающее за ответ в задаче предсказания следующего предложения, а также токены в количестве, равном входному. Обратное преобразование токенов в вероятностное распределение слов осуществляется полносвязным слоем с количеством нейронов, равным числу токенов в исходном словаре.

BERT обучается одновременно на двух задачах — предсказания следующего предложения (англ. next sentence prediction) и генерации пропущенного токена (англ. masked language modeling). На вход BERT подаются токенизированные пары предложений, в которых некоторые токены скрыты. Таким образом, благодаря маскированию токенов, сеть обучается глубокому двунаправленному представлению языка, учится понимать контекст предложения. Задача же предсказания следующего предложения есть задача бинарной классификации — является ли второе предложение продолжением первого. Благодаря ей сеть можно обучить различать наличие связи между предложениями в тексте.

Интерпретация этапа предобучения — обучение модели языку.

Этот этап обучения зависит от задачи, и выход сети, полученной на этапе предобучения, может использоваться как вход для решаемой задачи. Так, например, если решаем задачу построения вопросно-ответной системы, можем использовать в качестве ответа последовательность токенов, следующую за разделителем предложений. В общем случае дообучаем модель на данных, специфичных задаче: знание языка уже получено на этапе предобучения, необходима лишь коррекция сети.

Интерпретация этапа fine-tuning — обучение решению конкретной задачи при уже имеющейся общей модели языка.

Гиперпараметрами модели являются

H— размерность скрытого пространства кодировщика,

L— количество слоёв-кодировщиков,

A— количество голов в механизме внимания

### Обобщенно и кратко

BERT, или Bidirectional Encoder Representations from Transformers, - это модель обработки естественного языка, которая стала все более популярной в последние годы. Он используется для широкого спектра задач, включая анализ тональности текста, ответы на вопросы, генерацию текста и многое другое. Архитектура BERT состоит из двух основных компонентов: кодировщика и декодировщика.

Одной из ключевых особенностей BERT является его способность учиться на огромных объемах данных, что позволяет ему понимать контекст и смысл слов в предложении. При обработке текста BERT сначала токенизирует его, разбивая его на отдельные слова или подслова. Затем эти токены кодируются в векторные представления.

BERT обучается на двух задачах одновременно: предсказания следующего предложения и генерации пропущенного слова. Это позволяет ему учиться глубокому двунаправленному представлению языка и понимать контекст предложения.

Интерпретация этапа предварительного обучения заключается в задаче моделирования языка, где BERT обучается понимать общую структуру языка. Заключительным этапом является дообучение, где модель обучается решению конкретных задач, используя свои предварительно обученные знания языка.

Что касается гиперпараметров модели, они включают в себя размерность скрытого пространства кодировщика, количество слоев кодировщика и количество голов в механизме внимания.

В целом, BERT - это мощный инструмент для обработки естественного языка, который революционизировал эту область, обеспечивая передовые результаты на различных задачах. Его способность учиться и понимать язык на глубоком уровне сделала его популярным выбором для многих приложений, включая поисковые системы, чат-ботов и многое другое.

## 29. Класс Tensor. Операции, изменяющие размер тензора. Операции агрегации.

### Класс Tensor

Класс Tensor - это основной класс в библиотеке PyTorch, который представляет многомерный массив. Tensor может быть создан из списка или массива numpy.

Tensor - это не просто массив, это объект, который содержит в себе данные и информацию о своей форме (размерности). Это позволяет выполнять на нем различные операции, например, изменять его форму или вычислять среднее значение.

Класс Tensor в PyTorch поддерживает множество операций, которые позволяют выполнять различные вычисления с тензорами, такие как сложение, вычитание, умножение, деление, экспоненту и логарифм. Также в PyTorch есть операции сравнения, которые позволяют сравнивать тензоры поэлементно.

### Операции изменения размера тензора

PyTorch предоставляет множество операций, которые позволяют изменять форму тензора. Они могут быть полезны в случае, когда тензор нужно привести к нужной размерности или сделать его подходящим для выполнения операции.

Некоторые из операций изменения размера тензора:

- torch.Tensor.view(\*shape) - метод, который изменяет форму тензора без изменения его данных. Он возвращает новый тензор с указанными размерами. Это полезно, когда нужно изменить форму тензора, но при этом сохранить его данные.

- torch.Tensor.reshape(\*shape) - метод, который изменяет форму тензора, сохраняя его данные. Он возвращает новый тензор с указанными размерами. Этот метод поддерживает более широкий диапазон форм, чем view , но может потребовать копирования данных.

- torch.Tensor.permute(\*dims) - метод, который переставляет размерности тензора. Он возвращает новый тензор с переставленными размерностями. Это полезно, когда нужно изменить порядок размерностей в тензоре.

- torch.Tensor.squeeze(dim=None) - метод, который удаляет размерность из тензора, если она имеет размер 1. Если аргумент dim указан, то он удаляет только размерность dim . Это полезно, когда нужно убрать не нужную размерность из тензора.

- torch.Tensor.unsqueeze(dim) - метод, который добавляет размерность к тензору на указанную позицию. Это полезно, когда нужно добавить размерность к тензору для выполнения операции или для соответствия форме другого тензора.

### Операции агрегации

PyTorch также предоставляет множество операций агрегации, которые позволяют получить одно значение из тензора. Они могут быть полезны в случае, когда нужно вычислить сумму значений тензора или найти максимальное значение.

Некоторые из операций агрегации:

- torch.Tensor.mean(dim=None) - метод, который вычисляет среднее значение тензора по указанным размерностям. Это полезно, когда нужно вычислить среднее значение пикселей в изображении или среднее значение предсказаний модели на тестовом наборе данных.

- torch.Tensor.sum(dim=None) - метод, который вычисляет сумму значений тензора по указанным размерностям. Это полезно, когда нужно вычислить общее количество объектов в выборке или сумму всех элементов в матрице.

- torch.Tensor.max(dim=None) - метод, который вычисляет максимальное значение тензора по указанным размерностям. Это полезно, когда нужно найти максимальное значение в массиве или выбрать наибольшее предсказание модели.

- torch.Tensor.min(dim=None) - метод, который вычисляет минимальное значение тензора по указанным размерностям. Это полезно, когда нужно найти минимальное значение в массиве или выбрать наименьшее предсказание модели.

### Обобщенно и кратко

Класс Tensor является основным классом в библиотеке PyTorch, который представляет многомерный массив. Он позволяет выполнять на нем различные операции, такие как сложение, вычитание, умножение, деление, экспоненту и логарифм, а также операции сравнения.

PyTorch предоставляет множество операций, которые позволяют изменять форму тензора, такие как методы view , reshape , permute , squeeze и unsqueeze . Они позволяют изменять форму тензора без изменения его данных, переставлять размерности тензора, удалять размерность из тензора или добавлять размерность к тензору.

PyTorch также предоставляет множество операций агрегации, которые позволяют получить одно значение из тензора, такие как методы mean, sum, max и min . Они могут быть полезны в случае, когда нужно вычислить среднее значение, сумму значений, максимальное или минимальное значение тензора по указанным размерностям.

## 30. Принципиальная логика обучения нейронной сети.

Обучение нейронной сети на PyTorch состоит из нескольких шагов, требующих настройки различных параметров. Однако, несмотря на это, сам процесс обучения нейронной сети на PyTorch является относительно простым.

Первым шагом является подготовка данных. Необходимо разделить данные на тренировочный и тестовый наборы и преобразовать их в тензоры PyTorch. Тензоры - это многомерные массивы, которые используются для хранения и обработки данных в нейронных сетях.

Вторым шагом является определение модели. Необходимо создать модель нейронной сети, определить количество скрытых слоев, количество нейронов в каждом слое и функцию активации. Функция активации - это нелинейная функция, которая активирует нейрон при получении определенного входного сигнала.

Третьим шагом является определение функции потерь. Необходимо выбрать функцию потерь, которая будет минимизироваться во время обучения. Функция потерь измеряет разницу между предсказанными и фактическими значениями и используется для определения того, насколько хорошо нейронная сеть предсказывает выходные данные.

Четвертым шагом является определение оптимизатора. Необходимо выбрать оптимизатор, который будет обновлять веса нейронной сети в процессе обучения. Оптимизаторы - это алгоритмы, которые используются для настройки весов нейронов в процессе обучения.

Пятый шаг - это обучение модели. Необходимо передать тренировочные данные в модель и выполнить обратное распространение ошибки, чтобы обновить веса нейронной сети. Обратное распространение ошибки - это процесс, который используется для настройки весов нейронов в соответствии с выбранной функцией потерь.

В процессе обучения модели необходимо следить за параметрами обучения, такими как скорость обучения и количество эпох. Скорость обучения определяет, насколько быстро веса нейронной сети будут обновляться в процессе обучения. Слишком низкая скорость обучения может привести к долгому процессу обучения, а слишком высокая - к нестабильности в обучении. Количество эпох - это количество раз, которое тренировочный набор данных будет передан в модель.

Шестой и последний шаг - это оценка модели. После обучения модели необходимо оценить ее производительность на тестовых данных. При оценке модели можно использовать различные метрики, такие как точность, полноту и F1-меру.

### Обобщенно

Обучение нейронной сети на PyTorch состоит из нескольких этапов: подготовка данных, определение модели, выбор функции потерь и оптимизатора, обучение модели и оценка ее производительности на тестовых данных. В процессе обучения модели необходимо следить за параметрами обучения, такими как скорость обучения и количество эпох. После обучения модели можно использовать различные метрики, такие как точность, полнота и F1-мера, для оценки ее производительности.

## 31. Автоматическое дифференцирование в PyTorch. Пример и применение в обучении ИНС.

### Автоматическое дифференцирование

Автоматическое дифференцирование - это процесс вычисления производных функции во время вычисления значения этой функции. Это значит, что при использовании автоматического дифференцирования в PyTorch можно не задумываться о том, как вычислять градиенты функций вручную. PyTorch автоматически вычисляет градиенты функции, что является необходимым для обучения нейронных сетей.

При обучении нейронных сетей мы хотим минимизировать ошибку нашей модели. Для этого мы используем градиентный спуск, который требует вычисления градиентов функции ошибки по параметрам нашей модели. С помощью автоматического дифференцирования в PyTorch мы можем легко вычислить эти градиенты и использовать их для обновления параметров нашей модели в процессе обучения.

В PyTorch автоматическое дифференцирование реализовано через классы torch.Tensor и torch.autograd.Function. Когда мы создаем тензор в PyTorch, мы можем указать, что нам нужно вычислять градиенты для этого тензора, установив параметр requires\_grad=True. Затем, когда мы вычисляем функцию, содержащую этот тензор, PyTorch автоматически создает граф вычислений и вычисляет градиенты функции относительно этого тензора.

Одним из преимуществ автоматического дифференцирования в PyTorch является то, что оно позволяет вычислять градиенты для сложных функций, включая функции, которые содержат другие функции, которые в свою очередь содержат другие функции, и так далее. Это делает процесс обучения более эффективным и удобным для разработчиков.

Пример использования автоматического дифференцирования в PyTorch:

import torch

# Создание тензоров

x = torch.tensor([5.], requires\_grad=True)

y = torch.tensor([7.], requires\_grad=True)

# Вычисление функции

f = 3\*x\*\*2 + 2\*y

# Вычисление градиентов

f.backward()

# Вывод градиентов

print(x.grad) # Выводит тензор 30.0

print(y.grad) # Выводит тензор 2.0

В этом примере мы создаем два тензора x и y, которые являются переменными, для которых мы хотим вычислить градиенты. Затем мы вычисляем функцию f с использованием этих переменных. После этого мы вызываем метод backward(), который автоматически вычисляет градиенты функции f по переменным x и y. В конце мы выводим градиенты переменных x и y.

### Применение автоматического дифференцирования в обучении ИНС

В обучении ИНС мы хотим минимизировать ошибку нашей модели. Для этого мы используем градиентный спуск, который требует вычисления градиентов функции ошибки по параметрам нашей модели. С помощью автоматического дифференцирования в PyTorch мы можем легко вычислить эти градиенты и использовать их для обновления параметров нашей модели в процессе обучения.

В заключение, автоматическое дифференцирование в PyTorch очень полезно для обучения нейронных сетей, так как оно позволяет вычислять градиенты функций автоматически. Это делает процесс обучения более эффективным и удобным для разработчиков.

## 32. Загрузка и преобразование данных. Классы Dataset, DataLoader, Transforms (и композиция трансформеров).

Загрузка и преобразование данных являются важными шагами при работе с анализом данных и машинным обучением. В PyTorch для этих целей используются классы Dataset, DataLoader и Transforms, а также возможность композиции трансформеров.

**Класс Dataset** предоставляет интерфейс для доступа к данным и их преобразования. Он представляет собой абстрактный класс, от которого можно наследоваться для создания пользовательских наборов данных. Класс Dataset требует реализации двух методов: \_\_len\_\_() для получения размера набора данных и \_\_getitem\_\_(index) для получения элемента данных по индексу.

**Класс DataLoader** предоставляет удобный способ загрузки данных из Dataset и предоставляет функциональность по работе с пакетами (batching), перемешиванию данных (shuffling), параллельной загрузке и другими возможностями. Он позволяет эффективно итерироваться по данным во время обучения модели.

**Класс Transforms** предоставляет механизмы для преобразования данных. Он представляет собой набор предопределенных преобразований данных, таких как изменение размера изображений, нормализация, аугментация и т.д. Класс Transforms может применяться непосредственно к данным или использоваться вместе с Dataset и DataLoader.

**Композиция трансформеров** позволяет применять несколько преобразований последовательно к данным. В PyTorch это достигается с помощью класса transforms.Compose, который позволяет объединять несколько трансформеров в цепочку.

## 33. Класс nn.Module. Назначение. Основные поля, методы.

Класс nn.Module является одним из ключевых компонентов библиотеки PyTorch, используемой для создания и обучения нейронных сетей. Он предоставляет базовый функционал для определения и управления моделями глубокого обучения.

**Назначение класса nn.Module** заключается в обеспечении модульности и гибкости при построении нейронных сетей. Он представляет собой базовый класс, от которого наследуются все пользовательские модели, создаваемые в PyTorch. Класс nn.Module обеспечивает удобный способ определения структуры модели, хранения обучаемых параметров и прямого/обратного распространения сигнала в сети.

**Основные поля класса nn.Module:**

1. self.training: булевое поле, указывающее на режим работы модели (обучение или оценка). Влияет на поведение некоторых слоев, таких как Dropout или Batch Normalization.
2. self.\_parameters: словарь, содержащий все обучаемые параметры модели, такие как веса и смещения слоев. Они автоматически оптимизируются в процессе обучения.
3. self.\_buffers: словарь, содержащий необучаемые параметры модели, которые могут изменяться, но не оптимизируются автоматически, например, средние и дисперсии в Batch Normalization.
4. self.\_modules: словарь, содержащий все подмодули модели. Это позволяет иерархически организовывать модель и управлять ее частями.

**Основные методы класса nn.Module:**

1. \_\_init\_\_(): конструктор класса, инициализирующий модель. Здесь определяются все слои и параметры модели.
2. forward(input): метод, определяющий прямой проход модели. В этом методе определяется порядок применения слоев и преобразований для получения выходного значения на основе входных данных.
3. parameters(): метод, возвращающий итератор по всем обучаемым параметрам модели. Это позволяет передать параметры оптимизатору для обновления во время обучения.
4. zero\_grad(): метод, обнуляющий градиенты всех обучаемых параметров модели. Используется перед вычислением градиентов в процессе обратного распространения ошибки.
5. to(device): метод, перемещающий модель и все ее параметры на указанное устройство (например, GPU). Это позволяет ускорить вычисления при наличии поддерживаемого аппаратного обеспечения.

## 34. Линейные слои (Linear Layers).

Линейные слои являются одними из основных составляющих нейронных сетей. Они представляют собой линейное преобразование входных данных, которое выполняется путем умножения входных значений на матрицу весов и последующего добавления смещений.

В контексте нейронных сетей, линейный слой может быть представлен как функция:

**f(x) = Wx + b,**

где x - входные данные, W - матрица весов, b - вектор смещений, и f(x) - выходные значения слоя.

**Ключевая особенность** линейных слоев состоит в их способности выучивать нелинейные зависимости между данными. Хотя сам слой выполняет только линейное преобразование, в сочетании с нелинейной функцией активации, такой как ReLU (Rectified Linear Unit), линейные слои могут моделировать сложные нелинейные отображения.

В многослойной нейронной сети, линейные слои часто объединяются с нелинейной функцией активации, такой как ReLU или Sigmoid, для создания более мощных моделей, способных изучать сложные данные.

**При обучении нейронной сети**, веса и смещения линейного слоя обновляются в процессе обратного распространения ошибки с использованием градиентного спуска или других оптимизационных алгоритмов. Это позволяет модели подстраиваться под данные и находить оптимальные веса для выполнения поставленной задачи.

Линейные слои **широко применяются** в различных архитектурах нейронных сетей, включая перцептроны, сверточные нейронные сети (CNN) и рекуррентные нейронные сети (RNN). Они являются ключевыми строительными блоками для обработки и анализа данных в задачах машинного обучения, включая классификацию, регрессию, сегментацию и многие другие.

**Пример использования линейного слоя в PyTorch:**

import torch

import torch.nn as nn

# Создание линейного слоя

linear\_layer = nn.Linear(in\_features=10, out\_features=5)

# Пример применения линейного слоя

input\_data = torch.randn(32, 10) # входные данные размером 32x10

output = linear\_layer(input\_data) # применение линейного слоя

print(output.shape) # вывод размерности выходных данных

В этом примере мы создаем линейный слой с помощью nn.Linear, указывая количество входных (in\_features) и выходных (out\_features) признаков. Затем мы применяем линейный слой к некоторым входным данным (input\_data) и получаем выходные значения (output). Размерность выходных данных можно узнать с помощью свойства shape.

## 35. Слои нелинейной активации (Non Linear Activations).

Слои нелинейной активации являются важным компонентом нейронных сетей, позволяющим моделировать сложные нелинейные зависимости в данных. Они добавляют нелинейность в выходные значения слоя и позволяют нейронной сети изучать и аппроксимировать более сложные функции.

В отличие от линейных слоев, которые выполняют только линейное преобразование входных данных, слои нелинейной активации применяют нелинейную функцию к выходу линейного слоя. Это позволяет модели изучать и приближать нелинейные отображения между входными и выходными данными.

Некоторые из наиболее распространенных функций активации включают в себя:

1. ReLU (Rectified Linear Unit): Функция активации ReLU определяется как f(x) = max(0, x). Она преобразует все отрицательные значения в ноль, сохраняя положительные значения без изменений. ReLU широко используется в нейронных сетях из-за своей простоты и эффективности.
2. Sigmoid: Функция активации сигмоид (Sigmoid) определяется как f(x) = 1 / (1 + exp(-x)). Она ограничивает выходное значение в диапазоне от 0 до 1 и широко применяется в задачах бинарной классификации.
3. Tanh (гиперболический тангенс): Функция активации тангенс гиперболический (Tanh) определяется как f(x) = (exp(x) - exp(-x)) / (exp(x) + exp(-x)). Она ограничивает выходное значение в диапазоне от -1 до 1 и используется в различных задачах.
4. LeakyReLU: Функция активации LeakyReLU определяется как f(x) = max(αx, x), где α - маленькое положительное число (обычно около 0.01). Она вводит небольшой отрицательный наклон для отрицательных значений, чтобы предотвратить проблему "мертвых нейронов" (dead neurons) в нейронной сети.

Выбор конкретной функции активации зависит от задачи и свойств данных. В некоторых случаях может быть полезно применять различные функции активации в разных слоях сети.

Пример использования функции активации ReLU в PyTorch:

import torch

import torch.nn as nn

# Создание слоя с функцией активации ReLU

relu\_layer = nn.ReLU()

# Пример применения функции активации

input\_data = torch.randn(32, 10) # входные данные размером 32x10

output = relu\_layer(input\_data) # применение функции активации

print(output.shape) # вывод размерности выходных данных

В этом примере мы создаем слой с функцией активации ReLU с помощью nn.ReLU(). Затем мы применяем эту функцию активации к некоторым входным данным (input\_data) и получаем выходные значения (output). Размерность выходных данных можно узнать с помощью свойства shape.

## 36. Слои нормализации (Normalization Layers).

В глубоком обучении, нормализация данных является важной частью процесса обучения нейронных сетей. Она помогает уменьшить влияние различных масштабов значений признаков на процесс обучения, ускорить сходимость модели и повысить ее точность. В PyTorch, существуют различные слои нормализации, которые можно использовать для этой цели.

Процесс нормализации данных в PyTorch осуществляется путем применения определенных математических операций к выходным данным слоя. В зависимости от того, на каком уровне происходит нормализация, выделяют три типа слоев нормализации: Batch Normalization (BN), Layer Normalization (LN) и Instance Normalization (IN).

### Batch Normalization

Batch Normalization - это метод нормализации данных, который используется для ускорения обучения нейронных сетей. Он применяется к выходным данным слоя и нормализует их по среднему значению и стандартному отклонению батча. Таким образом, BN работает на уровне батча, а не на уровне отдельных примеров. Использование BN слоя позволяет достичь более быстрой сходимости модели и существенно сократить время обучения.

Пример использования BN в PyTorch:

import torch.nn as nn

class NeuralNetwork(nn.Module):

def \_\_init\_\_(self):

super(NeuralNetwork, self).\_\_init\_\_()

self.fc1 = nn.Linear(10, 20)

self.bn1 = nn.BatchNorm1d(20)

self.fc2 = nn.Linear(20, 2)

def forward(self, x):

x = self.fc1(x)

x = self.bn1(x)

x = self.fc2(x)

return x

### Layer Normalization

Layer Normalization - это метод нормализации данных, который применяется на уровне слоя. Он нормализует выходные данные слоя по среднему значению и стандартному отклонению слоя. В отличие от BN, который работает на уровне батча, LN нормализует данные на уровне слоя. Таким образом, LN позволяет устранить нежелательное влияние различных батчей на процесс обучения модели.

Пример использования LN в PyTorch:

import torch.nn as nn

class NeuralNetwork(nn.Module):

def \_\_init\_\_(self):

super(NeuralNetwork, self).\_\_init\_\_()

self.fc1 = nn.Linear(10, 20)

self.ln1 = nn.LayerNorm(20)

self.fc2 = nn.Linear(20, 2)

def forward(self, x):

x = self.fc1(x)

x = self.ln1(x)

x = self.fc2(x)

return x

### Instance Normalization

Instance Normalization - это метод нормализации данных, который применяется на уровне каждого примера в батче. Он нормализует выходные данные по среднему значению и стандартному отклонению примера. Этот метод нормализации используется для обработки изображений, где каждый пиксель рассматривается как отдельный пример. Использование IN позволяет добиться лучшей производительности модели на задачах компьютерного зрения.

Пример использования IN в PyTorch:

import torch.nn as nn

class NeuralNetwork(nn.Module):

def \_\_init\_\_(self):

super(NeuralNetwork, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 64, kernel\_size=3)

self.in1 = nn.InstanceNorm2d(64)

self.conv2 = nn.Conv2d(64, 128, kernel\_size=3)

self.in2 = nn.InstanceNorm2d(128)

def forward(self, x):

x = self.conv1(x)

x = self.in1(x)

x = self.conv2(x)

x = self.in2(x)

return x

С помощью этих слоев нормализации данных в PyTorch можно добиться лучшей производительности нейронной сети и улучшить ее обучение. Однако, при выборе метода нормализации данных необходимо учитывать особенности конкретной задачи и размера датасета.

### Обобщенно и кратко

В глубоком обучении, нормализация данных является важной частью процесса обучения нейронных сетей. Она помогает уменьшить влияние различных масштабов значений признаков на процесс обучения, ускорить сходимость модели и повысить ее точность. В PyTorch, существуют три типа слоев нормализации: Batch Normalization (BN), Layer Normalization (LN) и Instance Normalization (IN).

Batch Normalization - это метод нормализации данных, который используется для ускорения обучения нейронных сетей. Он применяется к выходным данным слоя и нормализует их по среднему значению и стандартному отклонению батча.

Layer Normalization - это метод нормализации данных, который применяется на уровне слоя. Он нормализует выходные данные слоя по среднему значению и стандартному отклонению слоя.

Instance Normalization - это метод нормализации данных, который применяется на уровне каждого примера в батче. Он нормализует выходные данные по среднему значению и стандартному отклонению примера.

Применение этих слоев нормализации данных в PyTorch позволяет добиться лучшей производительности и точности нейронной сети при обучении. Однако, при выборе метода нормализации данных необходимо учитывать особенности конкретной задачи и размера датасета.

## 37. Слои регуляризации (Dropout Layers).

Слои регуляризации, такие как слои Dropout, являются важными инструментами в глубоком обучении для борьбы с переобучением моделей. Переобучение возникает, когда модель слишком точно запоминает обучающие данные и не может обобщать на новые данные.

Слой Dropout работает путем случайного обнуления (отключения) некоторых выходных признаков во время обучения. Это делается независимо для каждого образца входных данных. На каждой итерации обучения каждый признак отключается с определенной вероятностью p, или оставляется включенным с вероятностью (1-p). Таким образом, каждый раз, когда передается пакет данных через слой Dropout, некоторые признаки случайным образом отключаются, и модель обучается на подмножестве признаков.

Преимущество слоя Dropout заключается в том, что он предотвращает переобучение и снижает взаимозависимость между нейронами в сети. Он заставляет модель стать более устойчивой и генерализировать лучше на новых данных.

В pytorch dropout-регуляризация используется с помощью torch.nn.Dropout(p=0.5, inplace=False), где p ([*float*](https://docs.python.org/3/library/functions.html#float)) – вероятность того, что элемент будет обнулен. По умолчанию: 0,5.

пример использования дропаута при построении модели:

### Из лекции макрушина

**Проблема переобучения модели**

модель, у которой слишком много свободных параметров, **плохо обобщается**: то есть слишком близко «облизывает» точки из тренировочного множества и в результате недостаточно хорошо предсказывает нужные значения в новых точках.

В современных нейронных сетях огромное число параметров (даже не самая сложная архитектура может содержать миллионы весов) **надо регуляризовать параметры**!

**Dropout**

**Регуляризация с помощью дропаута** (dropout regularization) - один из важнейших методов регуляризации нейронных сетей обеспечивший революцию глубокого обучения.

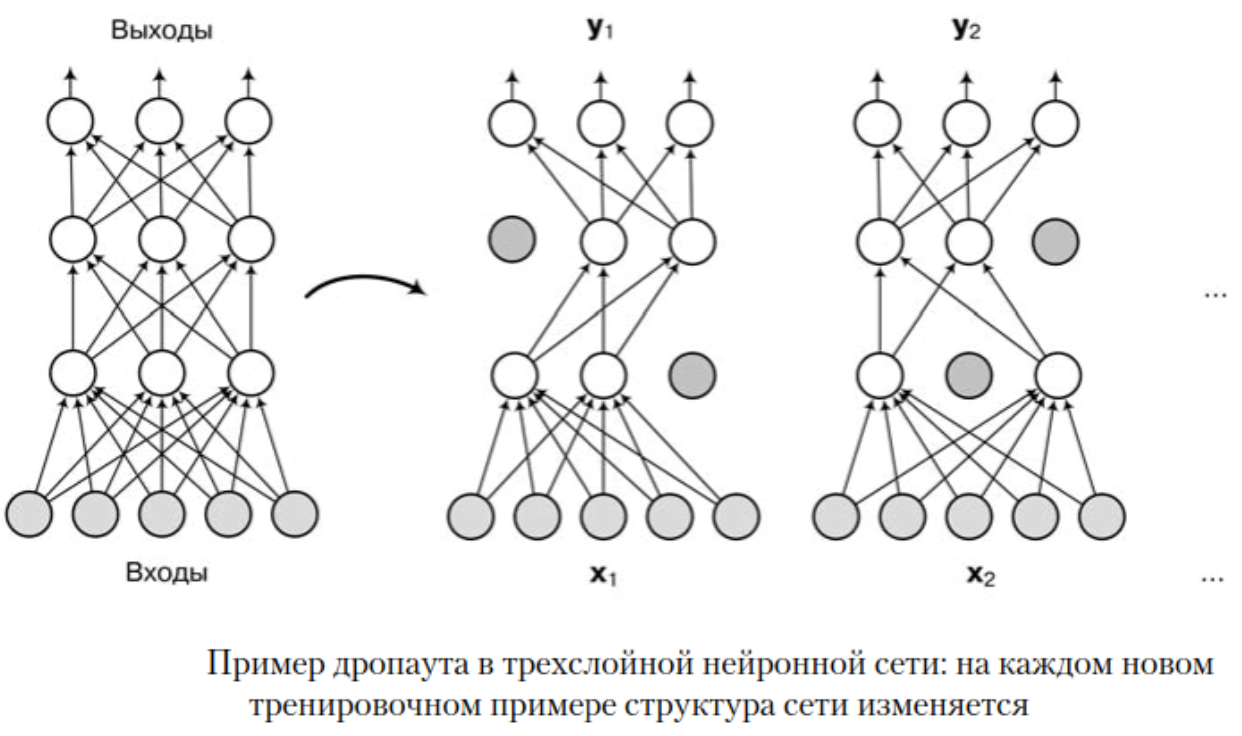
Идея метода (очень простая!):

* Для каждого нейрона (кроме самого последнего, выходного слоя) установим некоторую вероятность p, с которой он будет выброшен из сети.

Алгоритм обучения меняется таким образом:

* на каждом новом тренировочном примере x мы сначала для каждого **разыгрываем вероятность p** и в зависимости от результата либо используем нейрон как обычно, **либо устанавливаем его выход всегда строго равным нулю** (вероятность этого события 1 - p).
* **Дальше все происходит без изменений**; ноль на выходе приводит к тому, что нейрон фактически выпадает из графа вычислений: и прямое вычисление, и обратное распространение градиента останавливаются на этом нейроне и дальше не идут.
* Для применения обученной сети **используются все нейроны** в конфигурации, которая была до применения дропаута, но **выход каждого нейрона умножается на вероятность p** (с которой нейрон оставляли при обучении)

Для очень широкого спектра архитектур и приложений замечательно подходит p = 1/2



Практика обучения нейронных сетей показывает, что дропаут действительно дает очень серьезные улучшения в качестве обученной модели.

Дропаут — это метод добиться **усреднения огромного чмсла моделей** (до 2^N возможных моделей, N — число нейронов, которые подвергаются дропауту). Он эквивалентен усреднению всех моделей, которые получались на каждом шаге случайным выбрасыванием отдельных нейронов.

Пример использования dropuot в pytorch:

**def** \_\_init\_\_(self, rnn\_type, input\_size, node\_fdim, hidden\_size, depth, dropout):

super(MPNEncoder, self).\_\_init\_\_()  
self.hidden\_size **=** hidden\_size  
self.input\_size **=** input\_size

self.depth **=** depth self.

W\_o **=** nn.Sequential(

nn.Linear(node\_fdim **+** hidden\_size, hidden\_size),

nn.ReLU(),

nn.Dropout(dropout)

)

**if** rnn\_type **==** 'GRU':

self.rnn **=** GRU(input\_size, hidden\_size, depth)

**elif** rnn\_type **==** 'LSTM':

self.rnn **=** LSTM(input\_size, hidden\_size, depth)

**else**:

**raise** ValueError('unsupported rnn cell type ' **+** rnn\_type)

## 38. Сверточные слои (Convolution Layers). Сжимающие слои (Pooling Layers).

### Обобщенно и кратко

Сверточные слои и слои сжатия (пулинга) являются ключевыми компонентами сверточных нейронных сетей (Convolutional Neural Networks, CNN) и широко используются в обработке изображений и анализе последовательностей.

Сверточные слои выполняют операцию свертки между входными данными и фильтрами (ядрами свертки), чтобы извлечь локальные пространственные шаблоны. Каждый фильтр перемещается по входным данным, применяя свертку, и результаты суммируются для создания карты признаков. Это позволяет нейронной сети обнаруживать различные уровни абстракции в данных, начиная от простых функций, таких как границы и углы, до более сложных шаблонов.

Слой сжатия (пулинга) используется для уменьшения размерности карты признаков, сохраняя важные информационные характеристики. Самая распространенная операция пулинга - это операция субдискретизации (subsampling), которая уменьшает размер карты признаков, выбирая наиболее значимые значения в заданной области.

Сверточные слои и слои сжатия вместе позволяют моделям CNN автоматически извлекать иерархические признаки из входных данных, что делает их особенно эффективными для задач компьютерного зрения и обработки изображений.

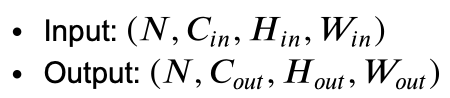
### Реализация в питхурче:

**torch.nn.Conv2d**(in\_channels, out\_channels, kernel\_size, stride=1, padding=0,

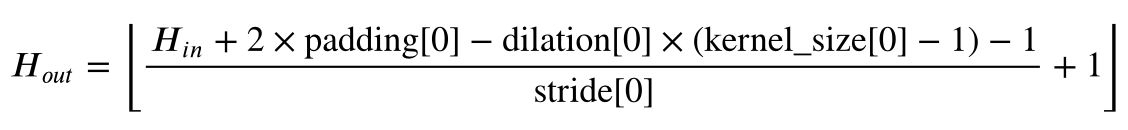
dilation=1, groups=1, bias=True, padding\_mode='zeros')

* in\_channels (int) – Number of channels in the input image ( )
* out\_channels (int) – Number of channels produced by the convolution ( )
* kernel\_size (int or tuple) – Size of the convolving kernel

Shape:



torch.nn.MaxPool2d(kernel\_size, stride=None, padding=0, dilation=1, return\_indices=False, ceil\_mode=False) - Applies a 2D max pooling over an input signal composed of several input planes

* kernel\_size – the size of the window to take a max over
* stride – the stride of the window. **Default value is kernel\_size**
* padding – implicit zero padding to be added on both sides
* dilation – a parameter that controls the stride of elements in the window
* return\_indices – if True, will return the max indices along with the outputs. Useful for torch.nn.MaxUnpool2d later
* ceil\_mode – when True, will use ceil instead of floor to compute the output shape 

пример:

ex1 **=** example\_data

print('ex1:', ex1.shape)

conv1 **=** nn.Conv2d(1, 10, kernel\_size**=**5)

ex2 **=** conv1(ex1)

print('ex2:', ex2.shape)

ex3 **=** F.max\_pool2d(ex2, 2)

print('ex3:', ex3.shape)

conv2 **=** nn.Conv2d(10, 20, kernel\_size**=**5)

ex4 **=** conv2(ex3)

print('ex4:', ex4.shape)  
ex5 **=** F.max\_pool2d(ex4, 2)

print('ex5:', ex5.shape)

output:

ex1: torch.Size([1000, 1, 28, 28])

ex2: torch.Size([1000, 10, 24, 24])

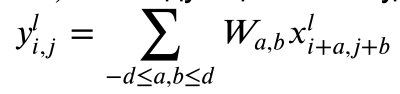
ex3: torch.Size([1000, 10, 12, 12])

ex4: torch.Size([1000, 20, 8, 8])

ex5: torch.Size([1000, 20, 4, 4])

### // Из лекции макрушина (взять если нужно налить в ответ еще воды или разобраться в теме)

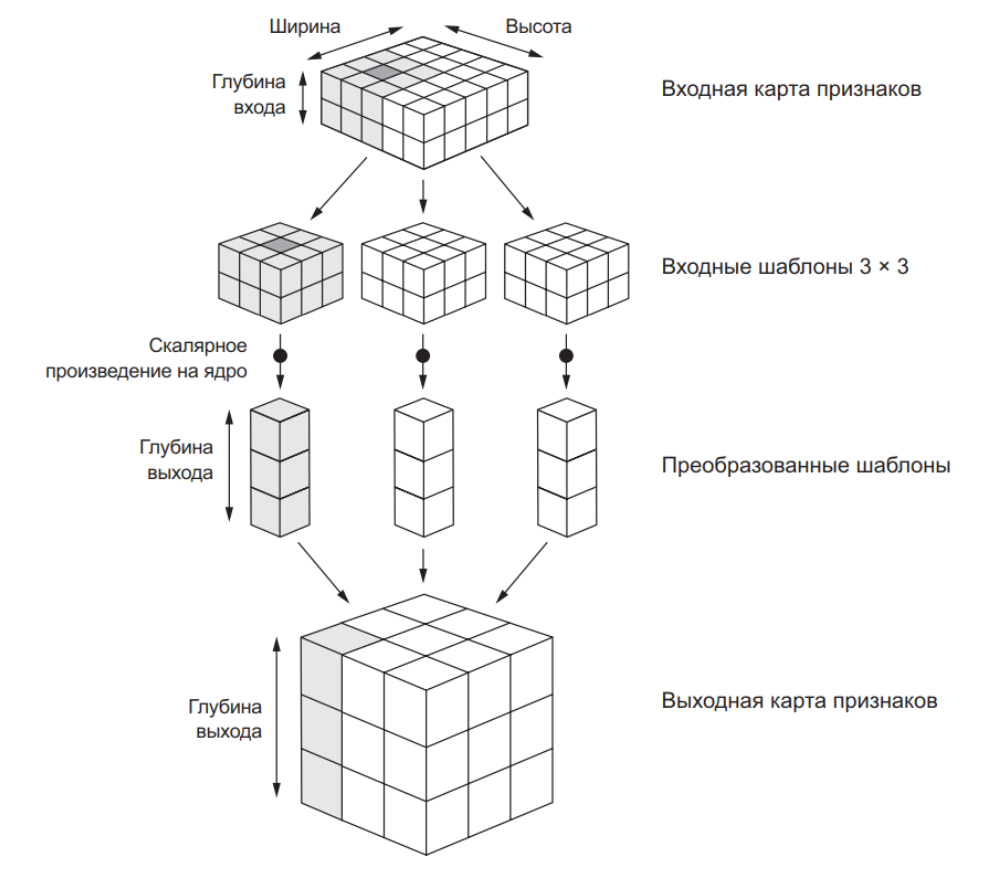
Сверточные слои:

* изучают **локальные шаблоны** (в случае с изображениями — шаблоны в **небольших двумерных окнах** во входных данных)
* изучают шаблоны **инвариантные в отношениии переноса** (шаблон будет с одинаковым качеством выявлен как в центре изображения, так и в левом нижнем углу, для полносвязной сети эти две задачи выглядели бы как совершенно разные задачи)
* Сверточный слой выделяет признаки в каждом окне небольшой нейронной сетью. В каждом окне выделяются одни и те же признаки, т.е. используется одна и та же маленькая нейронная сеть. Например, если окно 3 \* 3 пикселя, то входов у нее будет всего 9 и для каждого окна, в котором она будет применена, будет получен отдельный выход.
* Свертка — линейное преобразование входных данных особого вида. Если x1 — карта признаков в слое под номером l, то результат двумерной свертки с ядром размера 2d +1 и матрицей весов W размера (2d + 1) x (2d + 1) на следующем слое будет таким:, где - результат свертки на уровне l, а — ее вход, то есть выход всего предыдущего слоя.
* Почти всегда после свертки в нейронной сети следует нелинейность, которую можно записать так: . В качестве функции может использоваться любая функция активации, в глубоких сетях обычно предпочитают функцию ReLU, но и классические функции сигмоиды и tanh тоже встречаются.



**Применение одного слоя свертки к одному слою входных данных**

1. Свертка работает **методом скользящего окна**: она двигает окно с размером ядра (например 3 \* 3) по трехмерной входной карте признаков, останавливается в каждой возможной позиции и извлекает трехмерный шаблон окружающих признаков (с формой *[ высота\_окна, ширина\_окна, глубина\_входа ]* ).
2. Затем каждый **трехмерный шаблон преобразуется** (путем умножения тензора на матрицу весов, получаемую в ходе обучения, которая называется ядром свертки) в одномерный вектор с формой *[выходная глубина]*.
3. Затем все эти векторы **собираются в трехмерную выходную карту** с формой *[высота\_окна, ширина\_окна, выходная\_глубина]* \* Каждое пространственное местоположение в выходной карте признаков соответствует тому же местоположению во входной карте признаков (например, правый нижний угол выхода содержит информацию о правом нижнем углу входа).
4. Размерность тензора сверточных весов определяется **размерами ядра свертки и числом каналов как на входе, так и на выходе** (число каналов на входе и на выходе может не совпадать!). Таким обазом для двухмерного изображения тензор ядра свертки четырехмерный соследующими размерностями: *[высота\_ядра, ширина\_ядра, входные\_каналы, выходные\_каналы]*.



**Изменение размеров карты признаков**

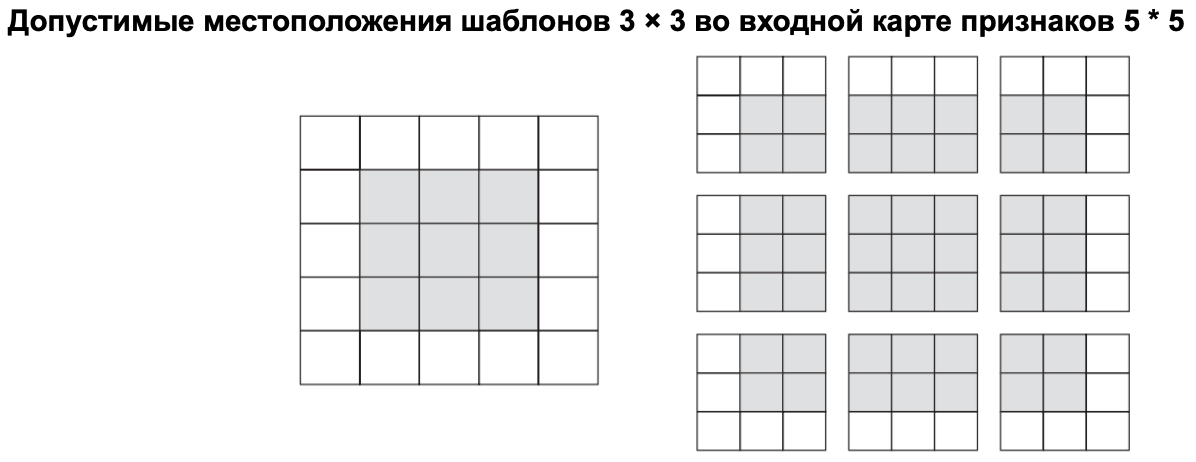
Выходные ширина и высота карты признаков могут отличаться от входных из-за:

* эффекта границ, который, при необходимости может устраняться дополнением входной карты признаков;
* использования шага свертки. Изменения размеров карты признаков обычно не является проблемой, зачастую уменьшение размеров карты является даже желательным эффектом.

**Эффект границ**

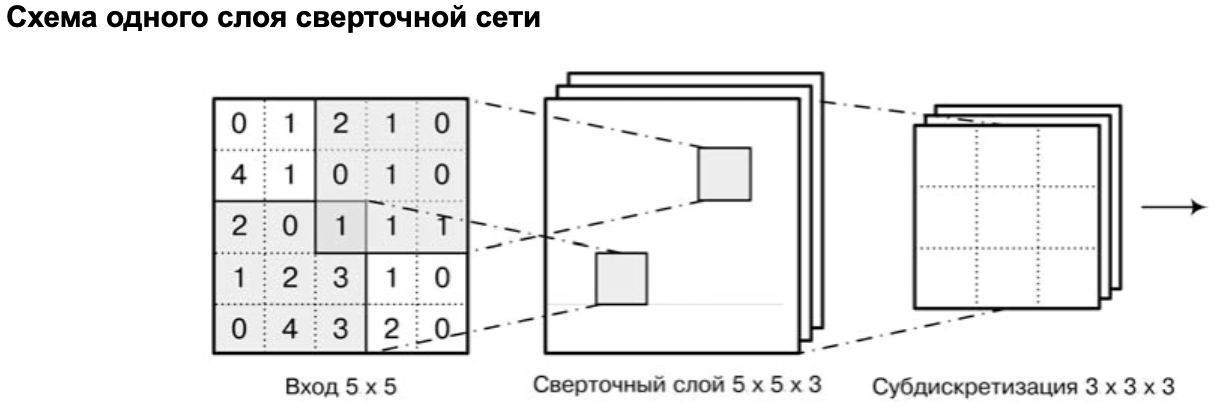
Рассмотрим карту признаков 5 × 5 (всего 25 клеток). Существует всего 9 клеток, в которых может находиться центр окна 3 × 3, образующих сетку 3 × 3. Следовательно, карта выходных признаков будет иметь размер 3 × 3. Она получилась немного сжатой: ровно на две клетки вдоль каждого измерения.

Чтобы получить выходную карту признаков с теми же пространственными размерами, что и входная карта, можно использовать дополнение (padding). Дополнение заключается в добавлении соответствующего количества строк и столбцов с каждой стороны входной карты признаков, чтобы можно было поместить центр окна свертки в каждую входную клетку.



В слое Conv2D дополнение настраивается аргумента *padding* :

* значенияе *"valid"* (по умолчанию) - отсутствие дополнения (будут использоваться только допустимые местоположения окна),
* значенияе *"same"* - карта дополняется так, чтобы выходная карта признаков имела ту же ширину и высоту, что и входная».



## 39. Слои функций потерь (Loss Functions).

Слои функций потерь играют важную роль в обучении нейронных сетей, так как они определяют, насколько хорошо модель справляется с поставленной задачей и позволяют оптимизировать параметры модели в процессе обучения.

Функция потерь (или функция ошибки) измеряет расхождение между предсказаниями модели и истинными значениями целевой переменной. Цель обучения нейронной сети - минимизировать значение функции потерь.

В зависимости от типа задачи (классификация, регрессия, сегментация и т. д.) и свойств данных, применяются различные функции потерь. Некоторые распространенные функции потерь включают среднеквадратичную ошибку (MSE) для регрессии, перекрестную энтропию (cross-entropy) для классификации и дискриминативную функцию потерь (discriminative loss) для сегментации.

Выбор подходящей функции потерь зависит от конкретной задачи и требует тщательного анализа данных и целей модели. Наиболее часто используемые функции потерь:

1. **Среднеквадратичная ошибка (MSE)**: эта функция потерь измеряет среднеквадратичное отклонение между истинными и предсказанными значениями. Это стандартная функция потерь для задач регрессии. В контексте нейронных сетей, MSE воздействует на модель таким образом, чтобы она уменьшала разницу между предсказаниями и истинными значениями

import torch

import torch.nn as nn

# инициализация функции потерь

criterion = nn.MSELoss()

# предсказанные значения (предположим, что мы получили их от нашей модели)

predictions = torch.randn(3, 5, requires\_grad=True)

# истинные значения

targets = torch.randn(3, 5)

# вычисление функции потерь

loss = criterion(predictions, targets)

# обратное распространение ошибки

loss.backward()

1. **Перекрестная энтропия (Cross Entropy)**: эта функция потерь обычно используется в задачах классификации. Перекрестная энтропия измеряет разницу между двумя вероятностными распределениями: распределением, предсказанным моделью, и истинным распределением меток. В контексте классификации модель обучается минимизировать перекрестную энтропию, что делает предсказанные вероятности ближе к истинным меткам.

# инициализация функции потерь

criterion = nn.CrossEntropyLoss()

# предсказанные значения (предположим, что мы получили их от нашей модели)

predictions = torch.randn(3, 5, requires\_grad=True)

# истинные значения (индексы классов)

targets = torch.empty(3, dtype=torch.long).random\_(5)

# вычисление функции потерь

loss = criterion(predictions, targets)

# обратное распространение ошибки

loss.backward()

1. **Бинарная кросс-энтропия (Binary Cross Entropy)**: это специализированная версия перекрестной энтропии, которая применяется к бинарным классификационным задачам. Она измеряет расхождение между истинными метками (которые могут быть либо 0, либо 1) и предсказанными вероятностями модели.

# инициализация функции потерь

criterion = nn.BCELoss()

# предсказанные значения (предположим, что мы получили их от нашей модели)

predictions = torch.sigmoid(torch.randn(3, 5, requires\_grad=True))

# истинные значения

targets = torch.empty(3, 5).random\_(2)

# вычисление функции потерь

loss = criterion(predictions, targets)

# обратное распространение ошибки

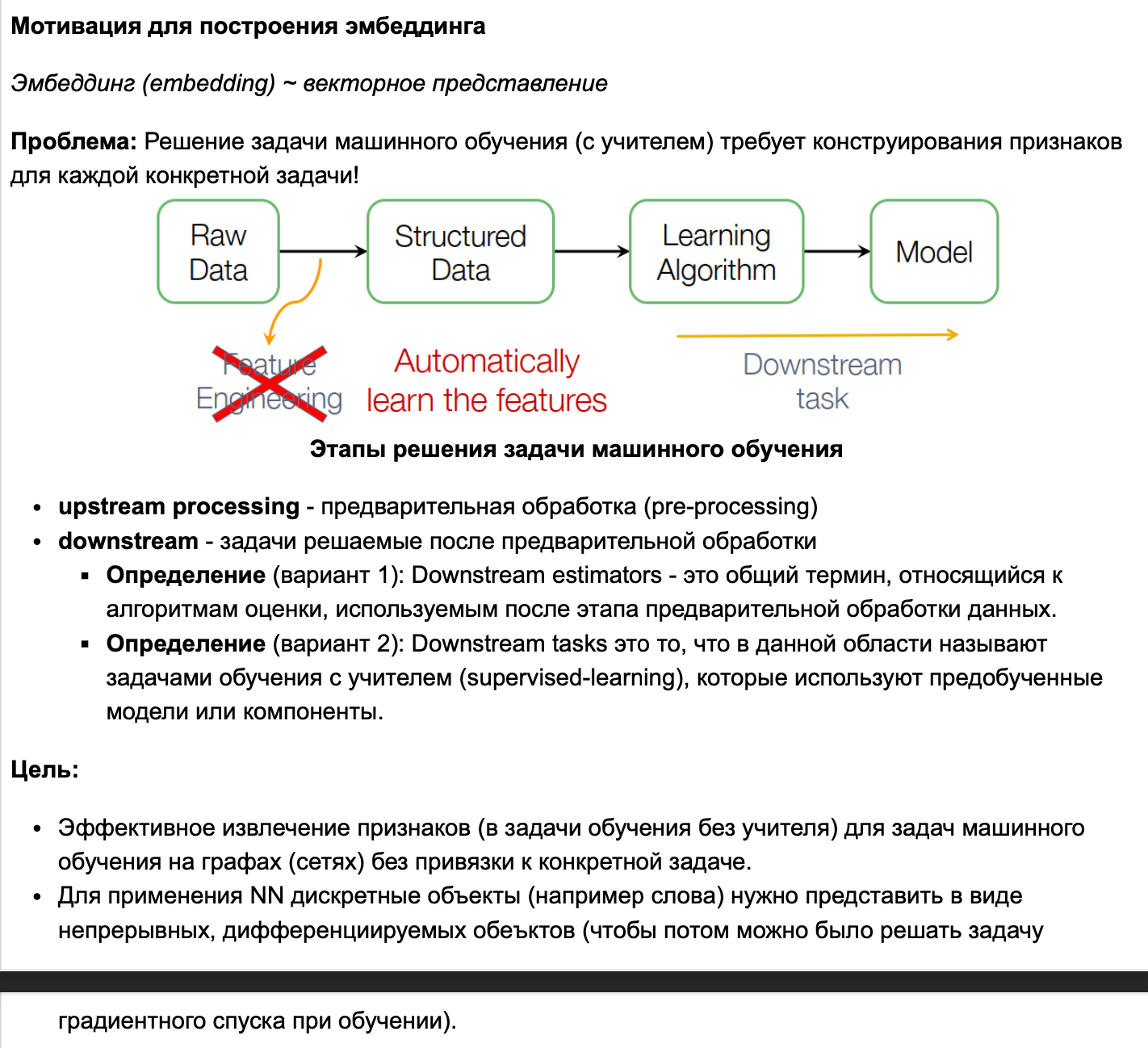
loss.backward()

## 40. Слои эмбеддингов nn.Embedding и их применение.

Слой эмбеддингов nn.Embedding является инструментом для обработки категориальных (дискретных) данных в нейронных сетях. Он позволяет представить категориальные признаки в виде непрерывных векторных представлений, называемых эмбеддингами.

Слой nn.Embedding принимает индексы категориальных признаков и возвращает соответствующие эмбеддинги для каждого индекса. Эти эмбеддинги обычно инициализируются случайно, а затем обучаются вместе с остальными параметрами модели в процессе обратного распространения ошибки.

Применение слоя эмбеддингов особенно полезно в задачах, где важна семантическая связь между категориями, например, в обработке естественного языка (Natural Language Processing, NLP). С помощью эмбеддингов можно представить слова, символы или другие дискретные единицы текста в виде плотных векторов, учитывая их семантическую схожесть и контекстуальные отношения.



**// Перед непосредственно эмбеддингами я бы рассказала про векторное представление и one-hot кодировку**

**Векторное представление слов** - это способ кодирования слов или фраз в машинно-читаемые векторы чисел. Это необходимо, потому что, хотя слова имеют значение для нас, машины не могут напрямую обрабатывать текстовые данные. Вместо этого нам нужно конвертировать эти текстовые данные в числовые форматы, которые машины могут обрабатывать.

**One-hot представление** - это метод кодирования категориальных данных в бинарный вектор. Для каждого уникального слова в словаре создается вектор, длиной равный общему количеству уникальных слов в словаре. Все значения в векторе устанавливаются в 0, за исключением одного, который устанавливается в 1, чтобы обозначить наличие этого слова.

Пример: представим, что у нас есть словарь из трех слов: {'cat', 'dog', 'bird'}. One-hot представление для слова 'cat' будет [1, 0, 0], для слова 'dog' будет [0, 1, 0], и для слова 'bird' будет [0, 0, 1].

Проблема с one-hot представлением состоит в том, что оно не учитывает семантическую близость слов и может быть неэффективным для больших словарей, поскольку большинство элементов в векторах будут равны нулю (это называется "разреженностью" данных).

**Слой эмбеддинга (Embedding layer)**, такой как **nn.Embedding** в PyTorch, используется для преобразования one-hot векторов в плотные векторы фиксированной длины. Эти плотные векторы обучаются вместе с другими параметрами в нейронной сети. В результате слова с похожими значениями будут иметь более близкие векторы, что улучшает обработку естественного языка для многих задач, таких как классификация текстов, генерация текстов, анализ тональности и многое другое.

torch.nn.Embedding(*num\_embeddings*, *embedding\_dim*, *padding\_idx=None*, *max\_norm=None*, *norm\_type=2.0*, *scale\_grad\_by\_freq=False*, *sparse=False*, *\_weight=None*, *\_freeze=False*, *device=None*, *dtype=None*)

Parameters

* num\_embeddings ( [int](https://docs.python.org/3/library/functions.html#int) ) — размер словаря вложений
* embedding\_dim ( [int](https://docs.python.org/3/library/functions.html#int) ) — размер каждого вектора встраивания
* padding\_idx ( [int](https://docs.python.org/3/library/functions.html#int) , необязательный ) — если указано, дополняет вывод вектором внедрения в padding\_idx (инициализируется нулями) всякий раз, когда встречается индекс.
* max\_norm ( [float](https://docs.python.org/3/library/functions.html#float) , необязательный ) — если указано, каждый вектор вложения с нормой больше max\_norm перенормируется, чтобы иметь норму max\_norm .
* norm\_type ( [float](https://docs.python.org/3/library/functions.html#float) , необязательный ) – p p-нормы для вычисления для опции max\_norm . По умолчанию 2 .
* scale\_grad\_by\_freq ( логическое значение , необязательный ) — если задано, это будет масштабировать градиенты на величину, обратную частоте слов в мини-пакете. По умолчанию False .
* разреженный ( [bool](https://docs.python.org/3/library/functions.html#bool) , необязательный ) — если True , градиент относительно матрицы weight будет разреженным тензором. Дополнительные сведения о разреженных градиентах см. в разделе «Примечания».

Variables

~ Embedding.weight ( [Tensor](https://runebook.dev/ru/docs/pytorch/tensors#torch.Tensor) ) — обучаемые веса модуля формы (num\_embeddings, embedding\_dim), инициализированные из N(0,1)

Shape:

* Input:(∗) , IntTensor или LongTensor произвольной формы, содержащие индексы для извлечения
* Output:(∗, Н) , где \* - входная форма, аН=embedding\_dim

Вот пример использования слоя nn.Embedding:

**>>>** *# an Embedding module containing 10 tensors of size 3*

**>>> embedding** **=** **nn.Embedding(**10**,** 3**)**

**>>>** *# a batch of 2 samples of 4 indices each*

**>>>** input **=** **torch.LongTensor([[**1**,** 2**,** 4**,** 5**],** **[**4**,** 3**,** 2**,** 9**]])**

**>>> embedding(**input**)**

tensor([[[-0.0251, -1.6902, 0.7172],

[-0.6431, 0.0748, 0.6969],

[ 1.4970, 1.3448, -0.9685],

[-0.3677, -2.7265, -0.1685]],

[[ 1.4970, 1.3448, -0.9685],

[ 0.4362, -0.4004, 0.9400],

[-0.6431, 0.0748, 0.6969],

[ 0.9124, -2.3616, 1.1151]]])

В этом примере каждое уникальное слово в словаре представлено вектором размерностью 5. Эмбеддинги инициализируются случайными числами и обновляются в процессе обучения сети с использованием методов оптимизации, таких как стохастический градиентный спуск.

## 41. Класс torch.nn.LSTM и torch.nn.GRU.

Классы torch.nn.LSTM и torch.nn.GRU являются реализациями рекуррентных нейронных сетей (Recurrent Neural Networks, RNN) в фреймворке PyTorch.

LSTM (Long Short-Term Memory) и GRU (Gated Recurrent Unit) - это две популярные архитектуры RNN, которые преодолевают проблему затухания и взрыва градиентов, характерную для простых RNN. Они способны моделировать долгосрочные зависимости в последовательных данных и позволяют передавать информацию через длительные временные интервалы.

Классы torch.nn.LSTM и torch.nn.GRU предоставляют удобные интерфейсы для создания и обучения моделей LSTM и GRU. Они принимают входные данные с последовательным форматом (например, временные ряды, текстовые данные) и обрабатывают их с учетом внутренних состояний и входного потока.

LSTM и GRU нашли широкое применение в различных областях, включая обработку естественного языка, машинный перевод, распознавание речи и генерацию текста. Они предоставляют мощный инструментарий для работы с последовательными данными и моделирования контекстуальных зависимостей.

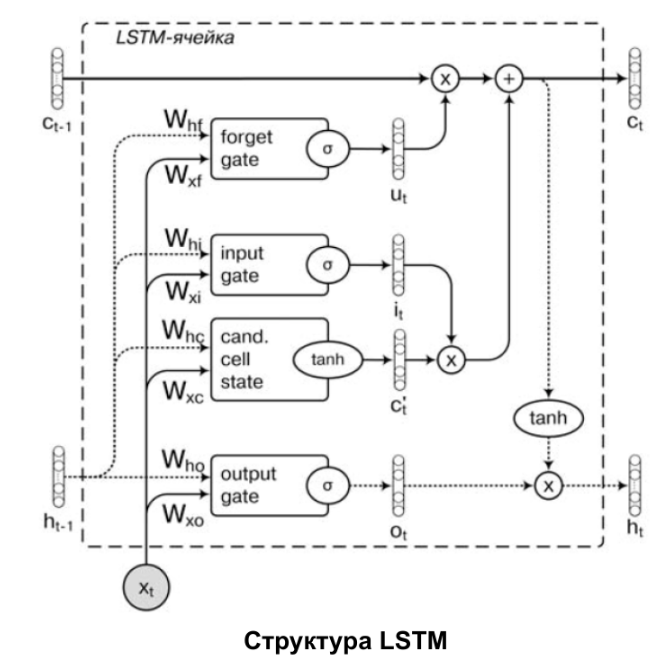
// (см вопрос 25, там полностью лекция макрушина):

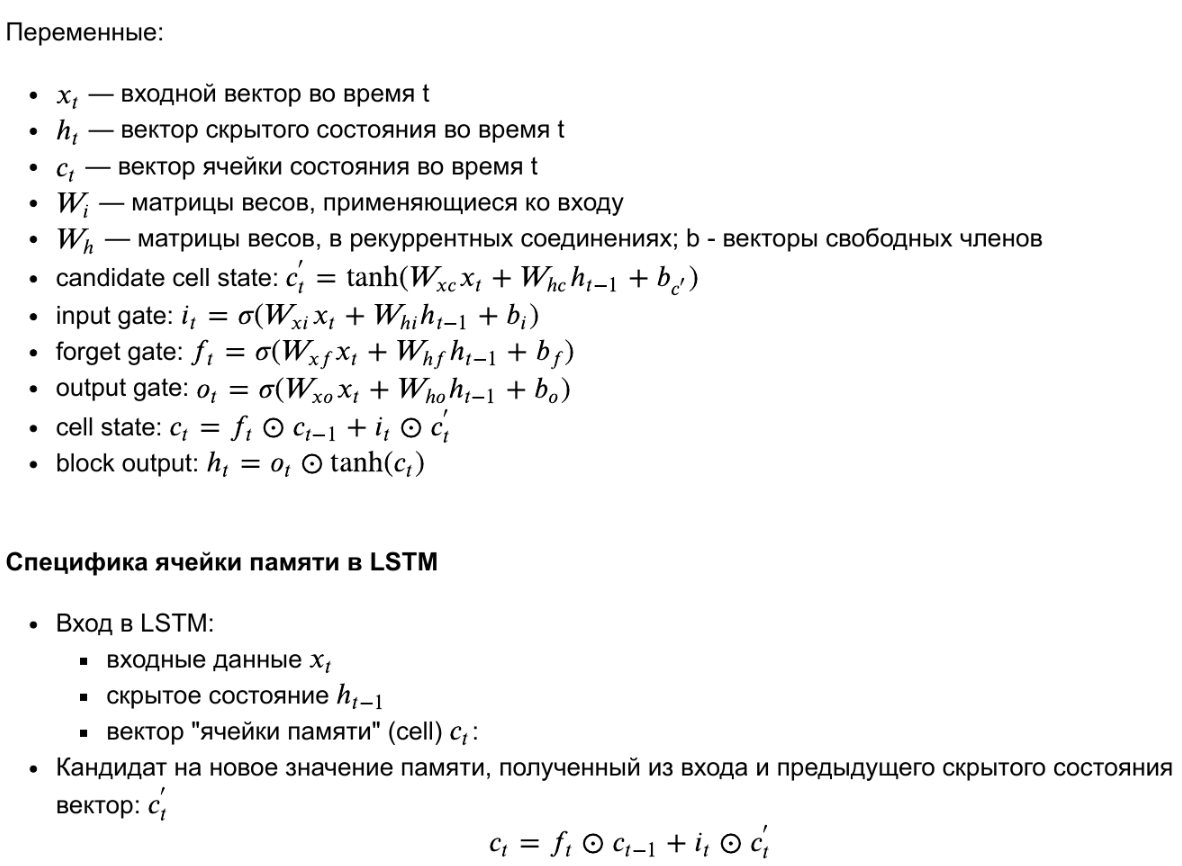
LSTM (Long Short-Term Memory)

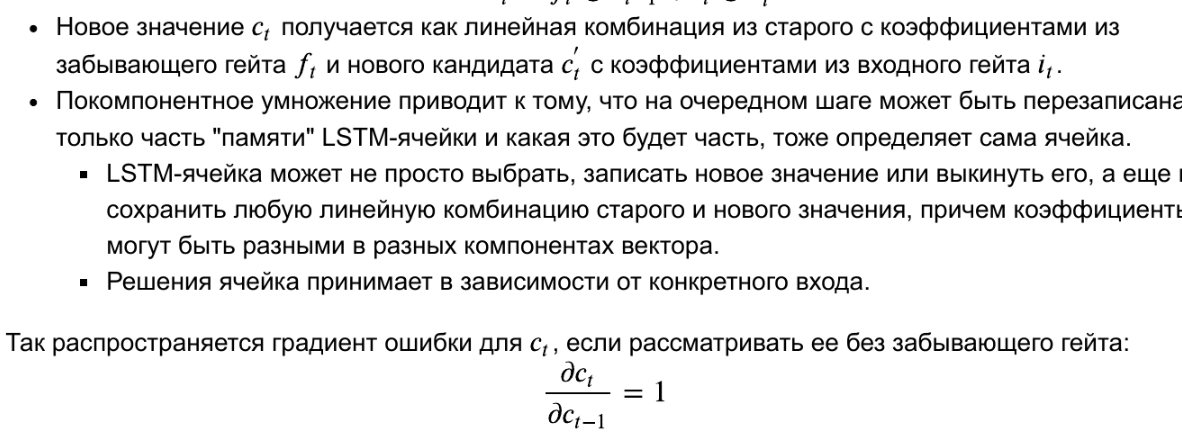
* обычные рекуррентные сети очень плохо справляются с ситуациями, когда нужно что-то «запомнить» надолго: влияние скрытого состояния или входа с шага t на последующие состояния рекуррентной сети экспоненциально затухает
* LSTM хорошо приспособлена к обучению на задачах классификации, обработки и прогнозирования временных рядов в случаях, когда важные события разделены временными лагами с неопределённой продолжительностью и границами вместо одного-единственного числа, на которое влияют все последующие состояния, используется специального вида ячейка моделирующая "долгую память"
* LSTM моделирует процессы записи и чтения из этой "ячейки памяти"
* у ячейки не один набор весов, как у обычного нейрона, а сразу несколько

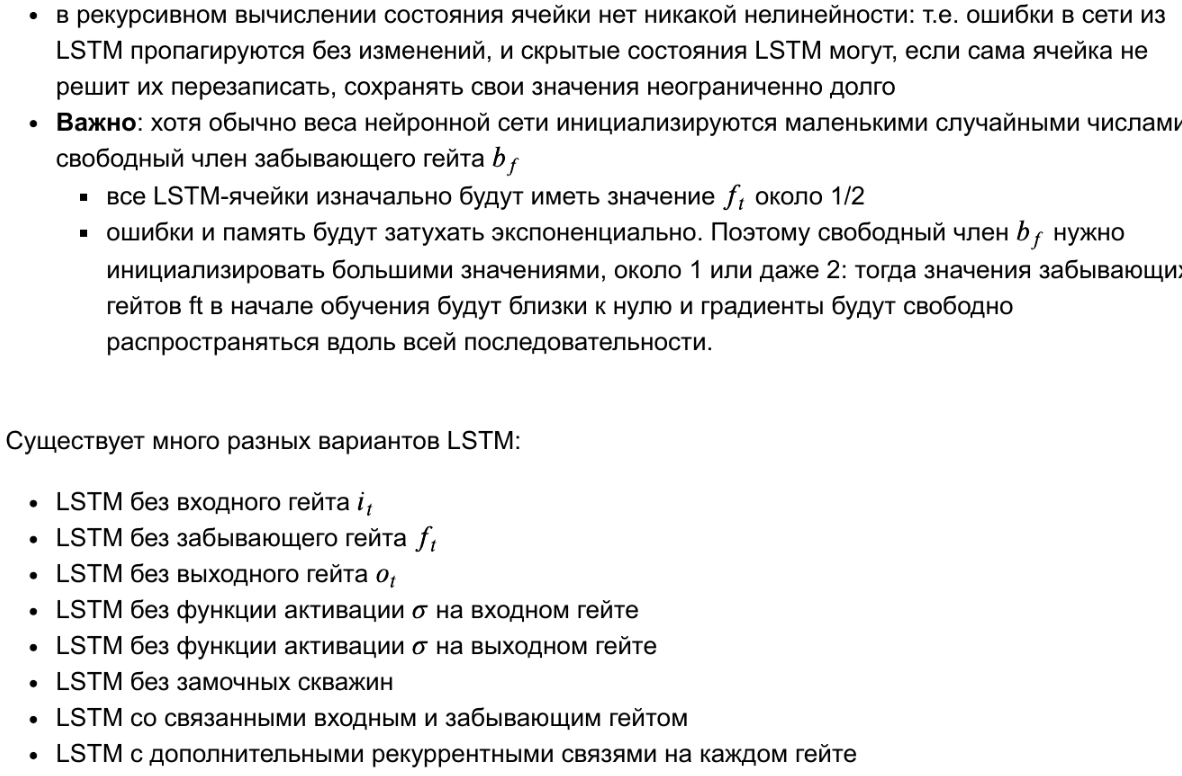
В LSTM есть три основных вида узлов, которые называются гейтами:

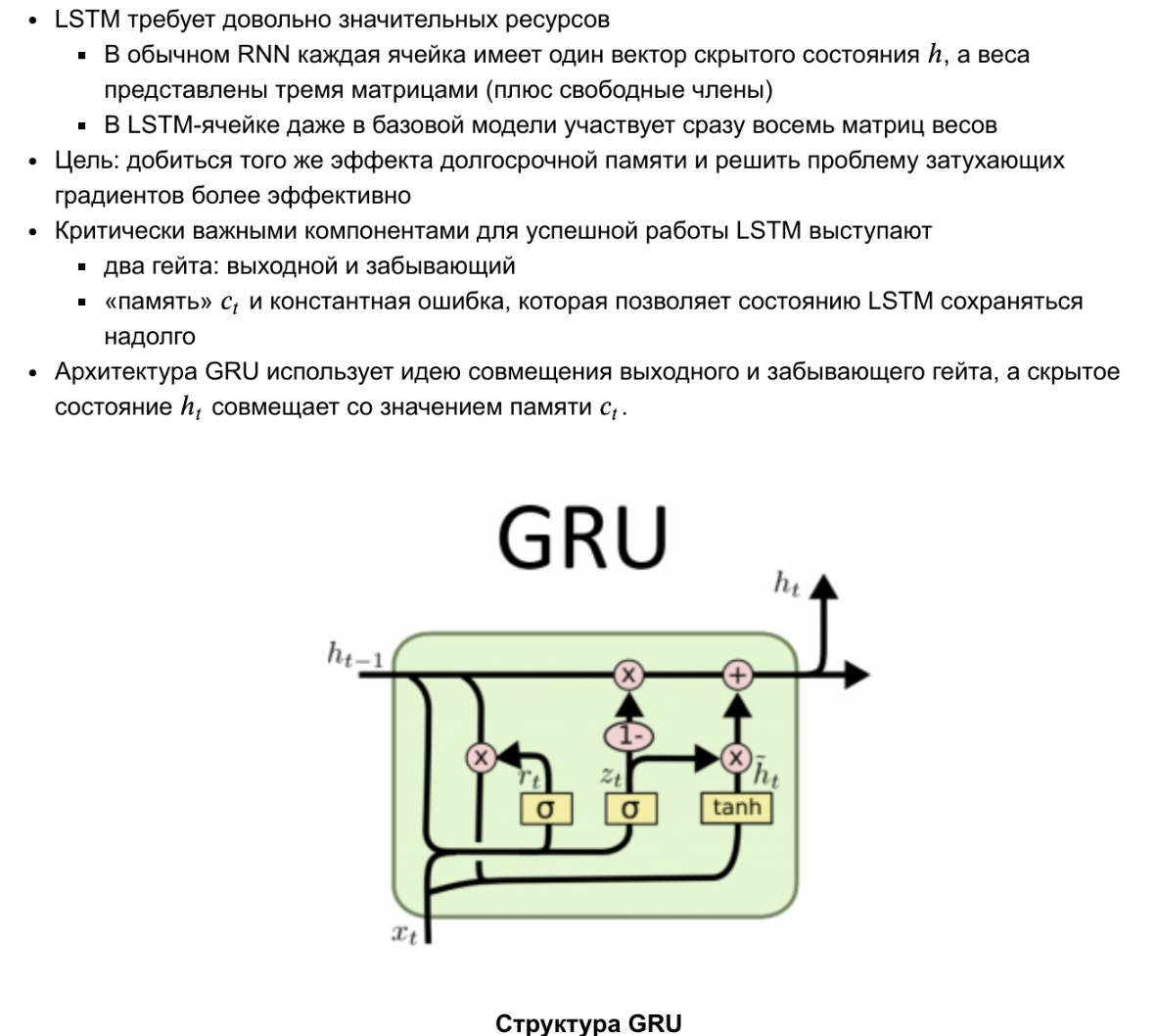
* входной (input gate)
* забывающий (forget gate)
* выходной (output gate)
* рекуррентная ячейка со скрытым состоянием

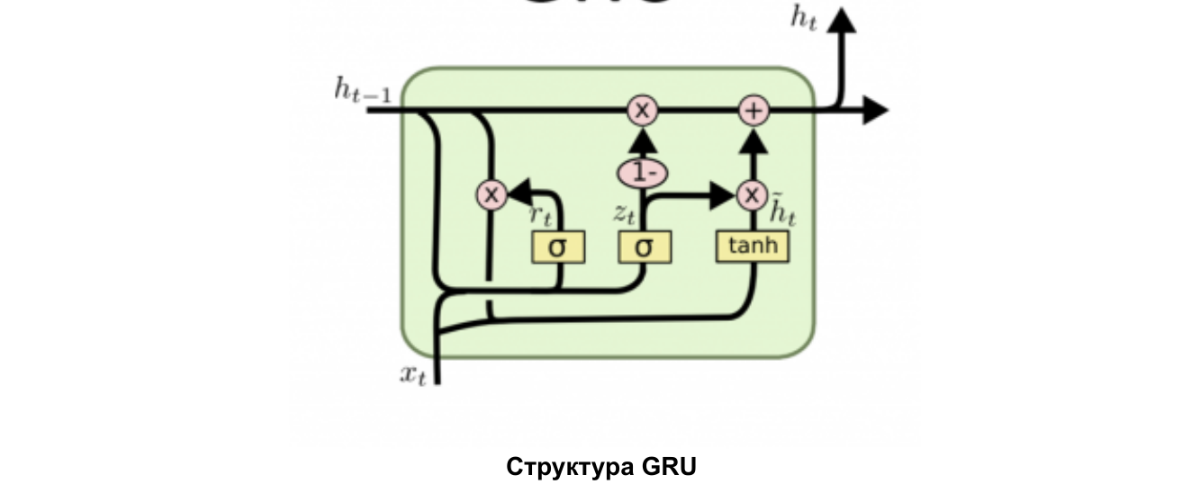












Модель:

torch.nn.LSTM(\*args, \*\*kwargs)

пример использования:  
class RNNModule(nn.Module):

def \_\_init\_\_(self, n\_vocab, seq\_size, embedding\_size, lstm\_size):

super(RNNModule, self).\_\_init\_\_()

self.seq\_size = seq\_size

self.lstm\_size = lstm\_size

self.embedding = nn.Embedding(n\_vocab, embedding\_size)

self.lstm = nn.LSTM(embedding\_size, lstm\_size, batch\_first=True)

self.dense = nn.Linear(lstm\_size, n\_vocab)

def forward(self, x, prev\_state):

embed = self.embedding(x)

output, state = self.lstm(embed, prev\_state)

logits = self.dense(output)

return logits, state

def zero\_state(self, batch\_size):

return (torch.zeros(1, batch\_size, self.lstm\_size), torch.zeros(1, batch\_size, self.lstm\_size))

**Основы глубокого обучения**

1. Модель перцептрона. Проблема линейно неразделимых множеств и ее решение. Логика построения многослойных ИНС.

2. Функции активации. Требования к функциям активации Популярные функции активации.

3. Глубокое обучение. «Вторая весна искусственного интеллекта» и ее причины.

4. Линейное отображение. Векторно-матричное дифференцирование.

5. Проблема поиска градиента в общей логике обучения нейронной сети. Градиент функции многих переменных. Методы вычисления.

6. Кросс-валидация. Выборки train, validation, test. Проблема переобучения. Ранняя остановка.

7. Преобразование Softmax и функция потерь Cross Entropy loss.

8. Механизм обратного распространения ошибки.

9. Дифференцируемое программирование и реализация обратного распространения ошибки.

10. Стохастический градиентный спуск. Батчи обучающей выборки.

11. Адаптивные методы градиентного спуска. Метод импульсов. Метод Нестерова.

12. Проблема инициализации весов при обучении ИНС. Инициализация Ксавье.

13. Гиперпараметры. Скорость обучения и размер батча.

14. Переобучение модели и регуляризация. Dropout.

15. Минбатчи – причина использования. Нормализация по мини-батчам.

16. Многослойные сети. Граф потока вычислений.

**Специфические задачи глубокого обучения**

17. Специфика задач машинного обучения на изображениях. Принцип работы сверточных сетей. Преимущества сверточных сетей при решении этих задач.

18. Архитектура многослойной ИНС распознавания изображений на основе сверточных сетей.

19. Приемы для глубокого обучения на небольших наборах изображений.

20. Схема работ слоя сверточной сети. Пулинг. Гиперпараметры: padding, kernel size, stride, dilation.

21. Задачи обработки текста: дистрибутивная семантика, матрица совместной встречаемости, представление слов в виде векторов малой размерности.

22. Word2vec: модель CBOW.

23. Word2vec: модель Skip-Gram.

24. Рекуррентная нейронная сеть, принципы ее обучения. Сложности применения рекуррентных нейронных сетей.

25. Модуль LSTM.

26. Механизм Attention. Пример использования Attentinon.

27. Архитектура Transformer.

28. Модель BERT.

**PyTorch и задачи глубокого обучения**

29. Класс Tensor. Операции, изменяющие размер тензора. Операции агрегации.

30. Принципиальная логика обучения нейронной сети.

31. Автоматическое дифференцирование в PyTorch. Пример и применение в обучении ИНС.

32. Загрузка и преобразование данных. Классы Dataset, DataLoader, Transforms (и композиция трансформеров).

33. Класс nn.Module. Назначение. Основные поля, методы.

34. Линейные слои (Linear Layers).

35. Слои нелинейной активации (Non Linear Activations).

36. Слои нормализации (Normalization Layers).

37. Слои регуляризации (Dropout Layers).

38. Сверточные слои (Convolution Layers). Сжимающие слои (Pooling Layers).

39. Слои функций потерь (Loss Functions).

40. Слои эмбеддингов nn.Embedding и их применение.

41. Класс torch.nn.LSTM и torch.nn.GRU.