**Дисциплина: Глубокое обучение (осенний семестр 2023-2024 уч. г.)**

**ПИ, лектор С.В. Макрушин**

**Теоретические вопросы. 1 вопрос в билете (1й вопрос – максимум 20 баллов).**

**Основы глубокого обучения**

1. Модель перцептрона. Проблема линейно неразделимых множеств и ее решение. Логика построения многослойных ИНС. Линейные слои (Linear Layers) в PyTorch.
2. Функции активации. Требования к функциям активации Популярные функции активации. Слои нелинейной активации (Non Linear Activations) в PyTorch.
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. Принципиальная логика основного цикла обучения нейронной сети в PyTorch.  Слои функций потерь (Loss Functions) в PyTorch.
9. Дифференцируемое программирование и реализация обратного распространения ошибки. Автоматическое дифференцирование в PyTorch. Пример и применение в обучении ИНС.
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. Принцип механизма Dropout. Слои регуляризации (Dropout Layers) в PyTorch.
15. Минбатчи – причина использования. Нормализация по мини-батчам. Слои нормализации (Normalization Layers) в PyTorch.
16. Многослойные сети. Граф потока вычислений. Класс nn.Module в PyTorch: назначение, основные поля и методы.

**Cверточные нейронные сети**

1. Специфика задач машинного обучения на изображениях. Принцип работы сверточных сетей. Преимущества сверточных сетей при решении этих задач.
2. Архитектура многослойной ИНС распознавания изображений на основе сверточных сетей. Сверточные слои (Convolution Layers) и сжимающие слои (Pooling Layers) в PyTorch.
3. Приемы для глубокого обучения на небольших наборах изображений.
4. Схема работы сверточной сети. Операции свертки, пулинга, общий вид сверточной сети для решения задачи классификации изображения.

**Рекуррентные нейронные сети**

1. Рекуррентная нейронная сеть, принципы ее обучения. Сложности применения рекуррентных нейронных сетей.
2. Модуль LSTM.
3. Модуль GRU.
4. Класс torch.nn.LSTM и torch.nn.GRU в PyTorch.
5. Продвинутые RNN-архитектуры: BiLSTM, улучшенные seq2seq модели машинного перевода на основе RNN.

**Механизм внимания и модуль Transformer и BERT**

1. Механизм Attention. Пример использования Attentinon.
2. Модуль Transformer: общая архитектура Transformer, работа self-attention и multi-head attention.
3. Модуль Transformer: общая архитектура Transformer, специфика работы декодера и его увязки с энкодером, позиционное кодирование в Transformer.
4. Модель BERT.

**Автоэнкодеры и генеративные модели**

1. Векторная интерпретация весов скрытого слоя многослойного перцептрона. Задача восстановления входного сигнала после сжатия в скрытом слое ИНС.
2. Автоэнкодеры. Линейный автоэнкодер и его аналог в статистических методах. Нелинейные и глубокие автоэнкодеры. Области значений латентного пространства. Примеры интерполяции и экстраполяции на многообразиях области значений латентных пространств автоэнкодеров.
3. Задачи дискриминативных и генеративных моделей, сравнение моделей. Классификация генеративных моделей. Прикладные задачи для генеративных моделей в области компьютерного зрения.
4. Архитектура GAN: описание общей архитектуры, модели обучения и архитектур глубоких моделей в GAN. Принцип подхода к обучению в GAN на примере обучения одномерной функции распределения.
5. Модель вариационного автоэнкодера (VAE): общие и специфические цели модели, общая архитектура модели и баейсовское моделирование. Функция потерь VAE и специфика получаемых скрытых представлений.
6. Модель вариационного автоэнкодера (VAE): баейсовское моделирование, функция ошибки на основе метода максимального правдоподобия, причины построения ELBO и метод построения этой оценки.
7. Denoising diffusion models: общий принцип работы, описание прямого процесса и основные принципы описания обратного процесса.

Оглавление

[Основы глубокого обучения 6](#_Toc154806533)

[1.Модель перцептрона. Проблема линейно неразделимых множеств и ее решение. Логика построения многослойных ИНС. Линейные слои (Linear Layers) в PyTorch. 6](#_Toc154806534)

[2. Функции активации. Требования к функциям активации Популярные функции активации. Слои нелинейной активации (Non Linear Activations) в PyTorch. 9](#_Toc154806535)

[3. Глубокое обучение. «Вторая весна искусственного интеллекта» и ее причины. 11](#_Toc154806536)

[4. Линейное отображение. Векторно-матричное дифференцирование. 14](#_Toc154806537)

[5. Проблема поиска градиента в общей логике обучения нейронной сети. Градиент функции многих переменных. Методы вычисления. 16](#_Toc154806538)

[6. Кросс-валидация. Выборки train, validation, test. Проблема переобучения. Ранняя остановка. 19](#_Toc154806539)

[7. Преобразование Softmax и функция потерь Cross Entropy loss. 21](#_Toc154806540)

[8. Механизм обратного распространения ошибки. Принципиальная логика основного цикла обучения нейронной сети в PyTorch. Слои функций потерь (Loss Functions) в PyTorch. 23](#_Toc154806541)

[9. Дифференцируемое программирование и реализация обратного распространения ошибки. Автоматическое дифференцирование в PyTorch. Пример и применение в обучении ИНС. 27](#_Toc154806542)

[10. Стохастический градиентный спуск. Батчи обучающей выборки. 31](#_Toc154806543)

[12. Проблема инициализации весов при обучении ИНС. Инициализация Ксавье. 35](#_Toc154806544)

[13. Гиперпараметры. Скорость обучения и размер батча. 37](#_Toc154806545)

[14. Переобучение модели и регуляризация. Dropout. Слои регуляризации (Dropout Layers) в PyTorch. 38](#_Toc154806546)

[15. Минбатчи – причина использования. Нормализация по мини-батчам. Слои нормализации (Normalization Layers) в PyTorch. 41](#_Toc154806547)

[Специфические задачи глубокого обучения 46](#_Toc154806548)

[17. Специфика задач машинного обучения на изображениях. Принцип работы сверточных сетей. Преимущества сверточных сетей при решении этих задач. 46](#_Toc154806549)

[18. Архитектура многослойной ИНС распознавания изображений на основе сверточных сетей. Сверточные слои (Convolution Layers) и сжимающие слои (Pooling Layers) в PyTorch. 50](#_Toc154806550)

[19. Приемы для глубокого обучения на небольших наборах изображений. 53](#_Toc154806551)

[20. Схема работы сверточной сети. Операции свертки, пулинга, общий вид сверточной сети для решения задачи классификации изображения. 54](#_Toc154806552)

[Рекуррентные нейронные сети 56](#_Toc154806553)

[21. Рекуррентная нейронная сеть, принципы ее обучения. Сложности применения рекуррентных нейронных сетей. 56](#_Toc154806554)

[22. Модуль LSTM. 58](#_Toc154806555)

[23. Модуль GRU. 61](#_Toc154806556)

[24. Класс torch.nn.LSTM и torch.nn.GRU в PyTorch. 62](#_Toc154806557)

[25. Продвинутые RNN-архитектуры: BiLSTM, улучшенные seq2seq модели машинного перевода на основе RNN. 65](#_Toc154806558)

[Механизм внимания и модуль Transformer и BERT 66](#_Toc154806559)

[26. Механизм Attention. Пример использования Attentinon. 66](#_Toc154806560)

[27. Модуль Transformer: общая архитектура Transformer, работа self-attention и multi-head attention. 69](#_Toc154806561)

[28. Модуль Transformer: общая архитектура Transformer, специфика работы декодера и его увязки с энкодером, позиционное кодирование в Transformer. 71](#_Toc154806562)

[29. Модель BERT. 73](#_Toc154806563)

[Автоэнкодеры и генеративные модели 75](#_Toc154806564)

[30. Векторная интерпретация весов скрытого слоя многослойного перцептрона. Задача восстановления входного сигнала после сжатия в скрытом слое ИНС. 75](#_Toc154806565)

[31. Автоэнкодеры. Линейный автоэнкодер и его аналог в статистических методах. Нелинейные и глубокие автоэнкодеры. Области значений латентного пространства. Примеры интерполяции и экстраполяции на многообразиях области значений латентных пространств автоэнкодеров. 76](#_Toc154806566)

[32. Задачи дискриминативных и генеративных моделей, сравнение моделей. Классификация генеративных моделей. Прикладные задачи для генеративных моделей в области компьютерного зрения. 78](#_Toc154806567)

[33. Архитектура GAN: описание общей архитектуры, модели обучения и архитектур глубоких моделей в GAN. Принцип подхода к обучению в GAN на примере обучения одномерной функции распределения. 79](#_Toc154806568)

[34. Модель вариационного автоэнкодера (VAE): общие и специфические цели модели, общая архитектура модели и баейсовское моделирование. Функция потерь VAE и специфика получаемых скрытых представлений. 81](#_Toc154806569)

[35. Модель вариационного автоэнкодера (VAE): баейсовское моделирование, функция ошибки на основе метода максимального правдоподобия, причины построения ELBO и метод построения этой оценки. 83](#_Toc154806570)

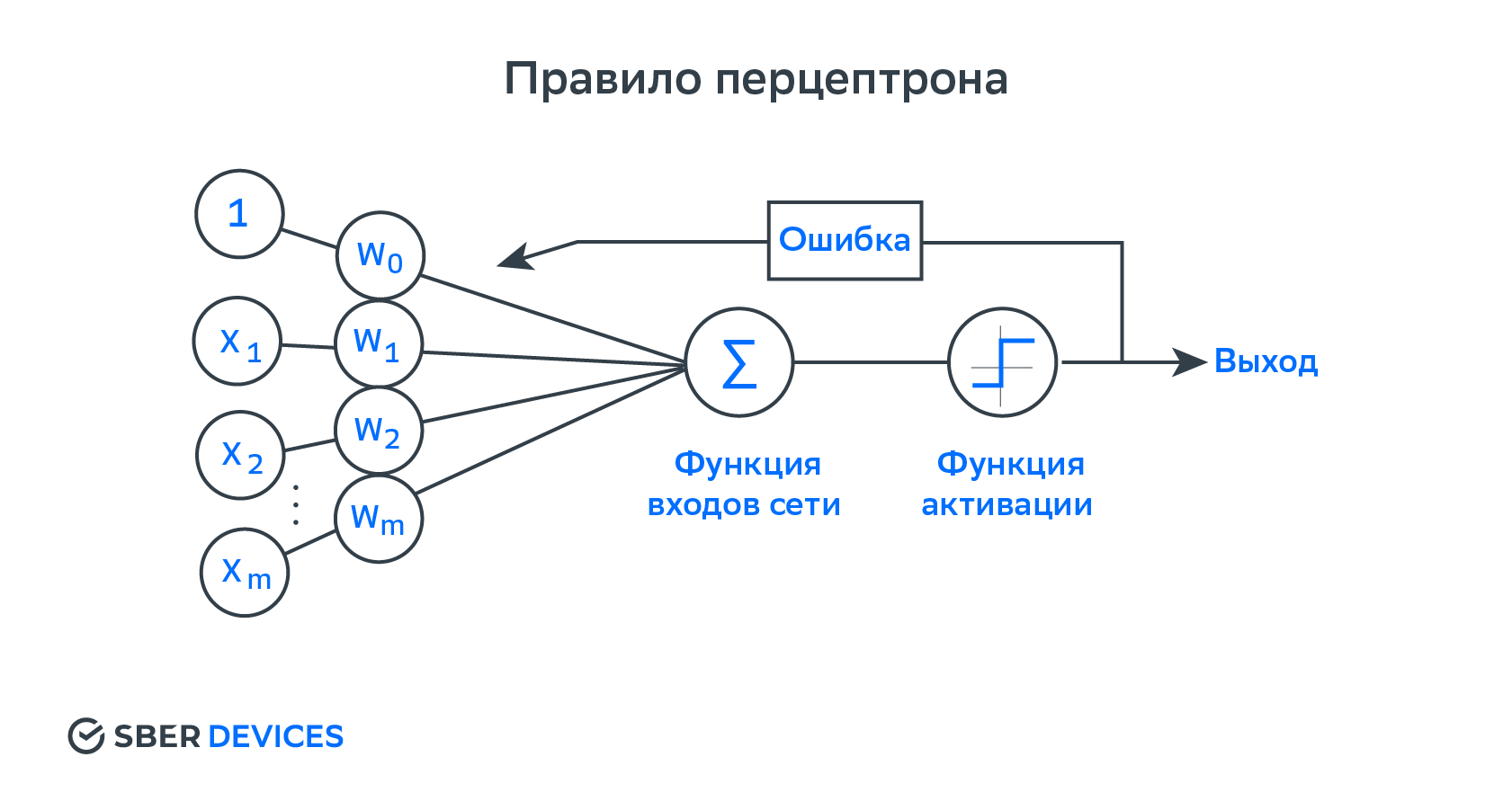
[36. Denoising diffusion models: общий принцип работы, описание прямого процесса и основные принципы описания обратного процесса. 84](#_Toc154806571)

Огромную благодарность выражаю Георгию Деменчуку и его команде студобороны(я просто хз, кто именно делал файлик с ответами на глубокое обучение(технологии анализа данных и машинного обучения)), часть вопросов я взял у них  
  
Уточнение по 30-36 вопросам и почему они были сделаны через чат(гпт4), потому что чат выдавал более читаемый и понятный текст чем, то что было в инете и на лекциях, я бы худо-бедно понял, но остальные не факт, поэтому взял с чата

Автор актуальной версии: Мельник Геннадий(vk: Bue Skim, tg: g1e1n1a1a1k1a1g1i1e1n1a1)

## Основы глубокого обучения

# 1.Модель перцептрона. Проблема линейно неразделимых множеств и ее решение. Логика построения многослойных ИНС. Линейные слои (Linear Layers) в PyTorch.



В 1958 Фрэнк Розенблатт изобретает однослойный перцептрон и демонстрирует его способность решать задачи классификации.

Основной инновацией Розенблатта была разработка алгоритма обучения перцептрона:

* изначально веса инициализируются случайным образом
* поочередно берется один обучающий пример, включающий набор входов 𝑥𝑖 и верное значение 𝑦
* для ошибочных предсказаний 𝑦̂ : веса увеличиваются, если 𝑦̂ =0 , а 𝑦=1
* веса уменьшаются, если 𝑦̂ =1, а 𝑦=0
* процедура повторяется до исчезновения ошибок

**Искусственная нейронная сеть (ИНС)** - математическая модель, а также её программное или аппаратное воплощение, разработанная под влиянием изучения организации и функционирования биологических нейронных сетей - сетей нервных клеток живого организма.

ИНС представляет собой систему (сеть) соединённых и взаимодействующих между собой простых вычислительных элементов (искусственных нейронов), которые, в общем случае, умеют:

* реагировать на входной сигнал, возвращая реакцию на него
* хранить некоторые параметры, обеспечивающие вариативность реакции на входной сигнал
* обучаться - определенным образом менять свои параметры в ходе выполнения операции обучения

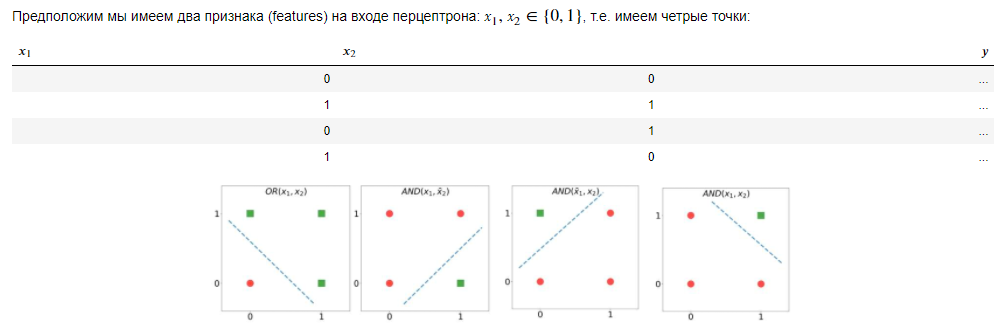
Каждый искусственный нейрон обычно достаточно просто устроен и является узлом в искусственной нейронной сети. В сети он имеет дело только с сигналами, которые он периодически получает, и сигналами, которые он периодически посылает другим нейронам. Несмотря на простоту устройства искусственные нейроны будучи соединёнными в достаточно большую сеть с управляемым взаимодействием способны решать сложные задачи, многие из которых, до использования ИНС решать не удавалось.

Для решения более сложных задач из перцептронов создается нейронная сеть.

* для получения нескольких результатов нейроны организуются в слой (layer) содержащий столько перцептронов, сколько требуется выходов
* выходы одного слоя могут использоваться в качестве входов следующего слоя - многослойные нейронные сети



Один перцептрон (с любой функцией активации) может научиться классифицировать только линейно разделимые множества объектов.



Марвин Мински публикует формальное доказательство ограниченности перцептрона и показывает, что он неспособен решать некоторые задачи (проблема «чётности» и «один в блоке»), связанные с инвариантностью представлений. В частности, один перцептрон не может реализовать функцию XOR (исключающее или).

Сам М. Мински показал, что XOR может быть реализован многослойной нейронной сетью из перцептронов. Нелинейная функция активации является критически важным элементом перцептрона, без нее линейная комбинация перцептронов позволяла бы строить только линейные разделяющие поверхности.

Линейные слои в PyTorch, это основные блоки для построения нейронных сетей. Они используются для преобразования входных данных с помощью линейного уравнения.

* **Создание линейного слоя**: **nn.Linear(in\_features, out\_features, bias=True/False)**, где **in\_features** - количество входных нейронов, **out\_features** - количество выходных нейронов.
* **Работа слоя**: Преобразует входные данные **x** в выходные **y** по формуле **y = xA^T + b**, где **A** - матрица весов, **b** - вектор смещения.
* **Пример использования**:

pythonCopy code

import torch.nn as nn import torch linear\_layer = nn.Linear(10, 5) # 10 входных, 5 выходных нейронов input\_data = torch.randn(1, 10) # случайные входные данные output\_data = linear\_layer(input\_data) # применение слоя

Линейные слои используются в многослойных перцептронах для классификации, регрессии и других задач машинного обучения. Веса и смещения слоя обновляются во время обучения для минимизации ошибки.

# 2. Функции активации. Требования к функциям активации Популярные функции активации. Слои нелинейной активации (Non Linear Activations) в PyTorch.

В искусственных нейронных сетях функция активации нейрона определяет выходной сигнал, который определяется входным сигналом или набором входных сигналов. В качестве аргумента принимает сигнал, получаемый на выходе входного сумматора.

Основные требования к ФА:

* функция должна быть монотонной (обычно монотонно не убывающая)
* иметь первую производную почти всюду (необходимо для обратного распространения ошибки при обучении нейронной сети)

Примеры ФА:

* Сигмоид (логистическая функция): 
* Гиперболический тангенс: 
* Единичная ступенчатая функция (функция Хевисайда): 
* 
* Изображение выглядит как текст

  Автоматически созданное описание

Слои нелинейной активации в PyTorch — это компоненты, которые добавляют нелинейность к моделям глубокого обучения, позволяя им учить более сложные закономерности в данных. В PyTorch есть несколько популярных функций нелинейной активации:

1. **ReLU (Rectified Linear Unit)**: Это одна из самых часто используемых функций активации. Она заменяет все отрицательные значения в выходных данных нейронов на ноль. Пример использования: **torch.nn.ReLU()**.
2. **Sigmoid**: Эта функция преобразует значения в диапазон от 0 до 1, что делает ее полезной для моделей, где необходимо предсказать вероятность. Пример использования: **torch.nn.Sigmoid()**.
3. **Tanh (Гиперболический тангенс)**: Преобразует данные так, чтобы они находились в диапазоне от -1 до 1. Это может быть полезно, если данные центрированы вокруг нуля. Пример использования: **torch.nn.Tanh()**.
4. **Leaky ReLU**: Это вариация ReLU, которая вместо того, чтобы устанавливать отрицательные значения в ноль, умножает их на небольшой коэффициент. Это помогает избежать проблемы "умирающих нейронов", когда нейроны перестают обучаться. Пример использования: **torch.nn.LeakyReLU()**.
5. **Softmax**: Часто используется в последнем слое многоклассовых классификационных моделей. Эта функция преобразует входные данные в распределение вероятностей по классам. Пример использования: **torch.nn.Softmax(dim=1)** для применения к выводам каждого образца.

Для использования этих функций активации в PyTorch, их обычно добавляют в модели как слои или применяют напрямую к данным в процессе определения модели.

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

Глубокое обучение — это особый раздел машинного обучения: новый подход к поиску представления данных, делающий упор на изучение последовательных слоев (или уровней) все более значимых представлений.

* Под глубиной в глубоком обучении не подразумевается многослойное представление данных.
* Количество слоев, на которые делится модель данных, называют глубиной модели.
* Другими подходящими названиями для этой области машинного обучения могли бы служить: многослойное обучение и иерархическое обучение.
* Под глубиной в глубоком обучении не подразумевается более глубокое понимание, достигаемое этим подходом.
* Современное глубокое обучение часто вовлекает в процесс десятки и даже сотни последовательных слоев представления.

Все они автоматически определяются под воздействием обучающих данных.

Другие подходы к машинному обучению ориентированы на изучении одного-двух слоев представления данных, по этой причине их иногда называют поверхностным обучением.

Основные причины быстрого взлета глубокого обучения:

* лучшая производительность во многих задачах
* упрощение решения проблем, за счет полной автоматизации важнейшего шаг в машинном обучении: конструирования признаков (раньше он выполнялся вручную)
* возможность эффективно использовать специализированное высокопроизводительное оборудование (GPU)

В целом, прогресс глубокого обучения объясняется тремя основными факторами:

* производительность оборудования
* доступность наборов данных и тестов
* алгоритмические достижения

Зима и вторая весна ИНС

До 1998 были предложены некоторые очень эффективные методы в области ИНС:

* Обратное распространение ошибки (backpropagation)
* Recurrent Long-Short Term Memory Networks (LSTM)
* Распознавание символов с помощью Convolutional Neural Networks (CNN)

Однако, в это время стали очень популярны некоторые альтернативные методы машинного обучения (в частности, SVM и т.д.)

* они обеспечивали аналогичное качество на тех же задачах
* не удавалось строить ИНС глубже нескольких слоев

В результате сообщество специалистов из области AI снова отвернулось от ИНС. В 1990е и 2000е годы альтернативные методы поверхностного обучения в большинстве задач превосходили методы, основанные на нейронных сетях.

Альтернативы нейронным сетям:

* Ядерные методы — это группа алгоритмов классификации, из которых наибольшую известность получил метод опорных векторов (Support Vector Machine, SVM).
* Деревья решений и случайные леса
* Градиентный бустинг



Начиная с 2010 года появляются важные успехи в области применения искусственных нейронных сетей. Выигрывается конкурс по классификации изображений с применением глубоких нейронных сетей, обучаемых на GPU: первый практический успех современного глубокого обучения. Соревнование ImageNet - крупномасштабное распознавание образов. Классификации цветных изображений с высоким разрешением на 1000 разных категорий после обучения по выборке, включающей в себя 1,4 миллиона изображений. Для начала 2010х годов - очень сложная задача машинного обучения. В 2011 году модель-победитель, основанная на классических подходах к распознаванию образов, показала точность лишь 74,3% . Достигается точности в 80+% — значительный прорыв методов глубокого обучения.

Какой ступени развития достигло глубокое обучение:

* классификация изображений на уровне человека;
* распознавание речи на уровне человека;
* распознавание рукописного текста на уровне человека;
* улучшение качества машинного перевода с одного языка на другой;
* улучшение качества машинного чтения текста вслух;
* появление цифровых помощников, таких как Google Now и Amazon Alexa;
* управление автомобилем на уровне человека;
* повышение точности целевой рекламы, используемой компаниями Google, Baidu и Bing;
* повышение релевантности поиска в интернете;
* появление возможности отвечать на вопросы, заданные вслух;
* игра в Го сильнее человека.

Кроме оборудования и данных, до конца 2000-х нам не хватало надежного способа обучения очень глубоких нейронных сетей, как результат:

* нейронные сети оставались очень неглубокими, имеющими один или два слоя представления;
* они не могли противостоять более совершенным поверхностным методам (методу опорных векторов и случайные леса). Основная проблема заключалась в распространении градиента через глубокие пакеты слоев. Сигнал обратной связи, используемый для обучения нейронных сетей, затухает по мере увеличения количества слоев.

В районе 2010 г. появились некоторые простые, но важных алгоритмические усовершенствования, позволившие улучшить распространение градиента:

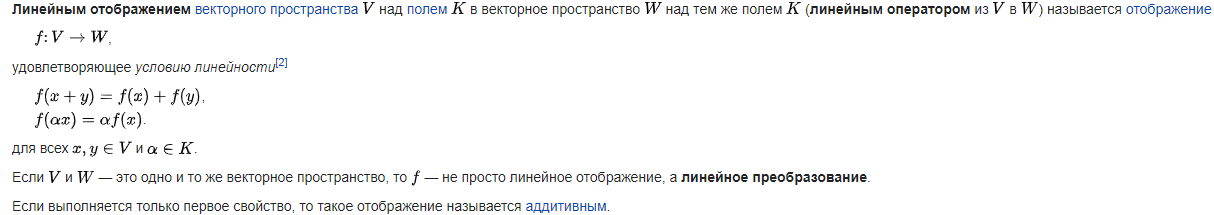
* улучшенные подходы к регуляризации
* улучшенные схемы инициализации весов
* улучшенные функции активации
* улучшенные схемы оптимизации (такие как RMSProp и Adam)

В последнее время были открыты еще более совершенные способы распространения градиента, с применением которых появилась возможность обучать с нуля модели с тысячами слоев в глубину. В частности:

* пакетная нормализация
* обходные связи
* отделимые свертки

# 4. Линейное отображение. Векторно-матричное дифференцирование.

Лине́йное отображе́ние — обобщение линейной числовой функции на случай более общего множества аргументов и значений.



Матрично-векторное дифференцирование.

Для вычисления большинства производных, которые возникают на практике, достаточно лишь небольшой таблицы стандартных производных и правил преобразования. Удобнее всего оказывается работать в терминах «дифференциала» — с ним можно не задумываться о промежуточных размерностях, а просто применять стандартные правила.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

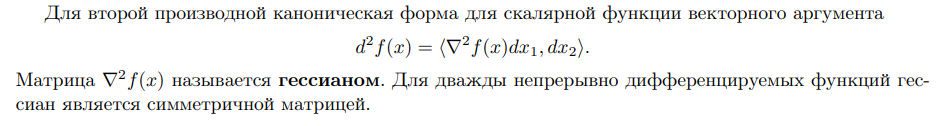
Автоматически созданное описание

Обычно, все возникающие на практике матрично-векторные функции составлены с помощью табличных функций и стандартных операций над ними. Благодаря универсальности приведённых правил дифференцировать сколь угодно сложные функции такого типа становится настолько же просто, как и дифференцировать одномерные функции.

Полученное в конце концов выражение нужно привести к одному из канонических видов:

Изображение выглядит как стол

Автоматически созданное описание



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

Градиент функции – это вектор, состоящий из частных производных функции по каждой из ее переменных. Он обычно используется для нахождения точек минимума (или максимума) функции, а также для получения гессиана функции.

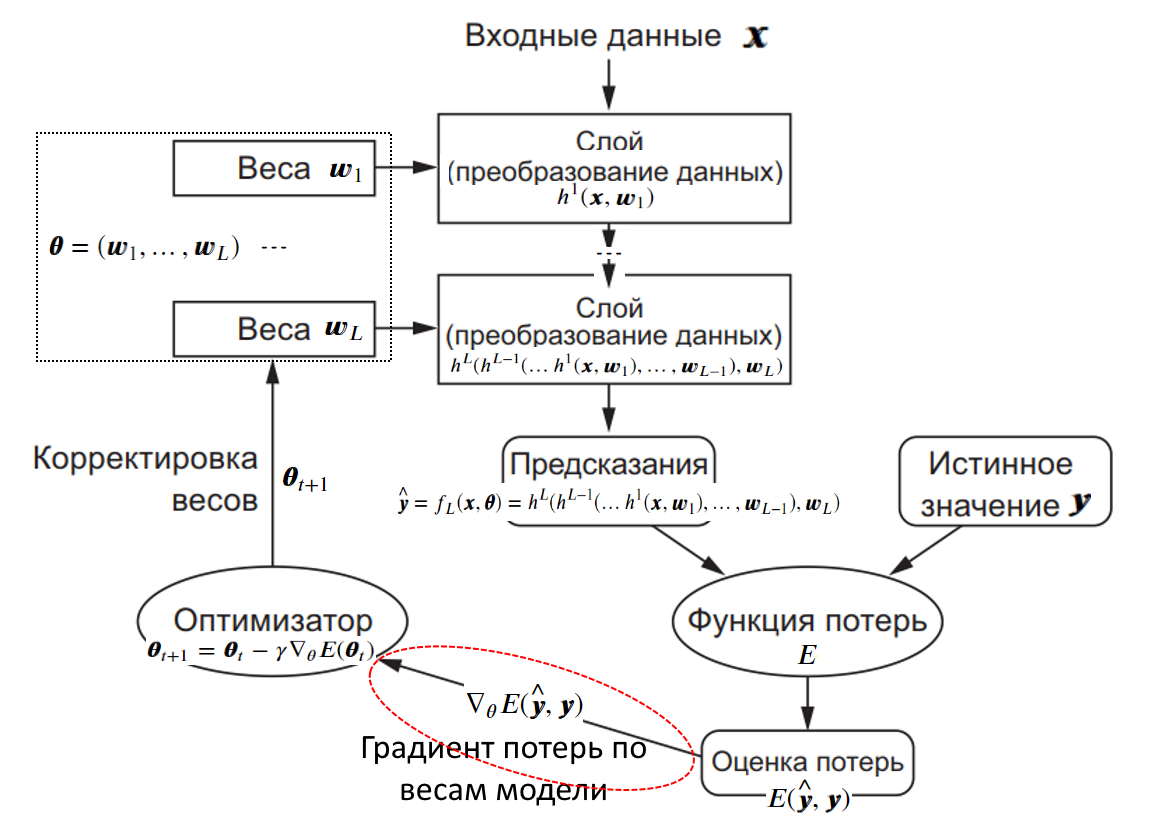
В нейронных сетях градиенты часто используются для обучения с помощью градиентного спуска. В этом случае градиенты используются для определения, в какую сторону нужно изменить веса нейронной сети, чтобы уменьшить значение функции потерь.

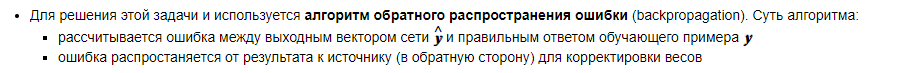
Проблема поиска градиента в общей логике обучения нейронной сети состоит в том, что нейронные сети обычно состоят из множества слоев, каждый из которых содержит множество нейронов. Это делает функцию потерь, которую мы хотим минимизировать, очень сложной и имеет множество локальных минимумов. Поэтому нам нужно использовать градиентный спуск или другие методы оптимизации, чтобы найти глобальный минимум функции потерь.

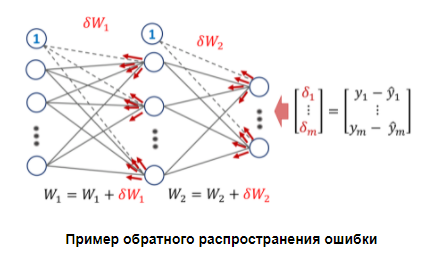
Градиент функции многих переменных – это вектор, состоящий из частных производных функции по каждой из ее переменных. Например, если у нас есть функция f(x, y) от двух переменных x и y, то градиент этой функции будет выглядеть так: *[∂f/∂x, ∂f/∂y]*

Градиент функции многих переменных используется в различных областях математики и инженерии, в том числе в нейронных сетях, для нахождения точек минимума (или максимума) функции, а также для получения гессиана функции.

Проблема: как найти градиент для нейронной сети: *∇𝜃𝐸(𝑓𝐿(𝑥𝑥,𝜃𝜃),𝑦𝑦)* ?

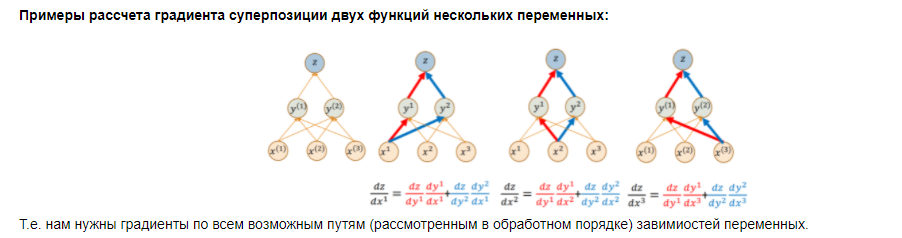






Изображение выглядит как текст

Автоматически созданное описание



Изображение выглядит как текст

Автоматически созданное описаниегр

Изображение выглядит как текст

Автоматически созданное описание

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

Кросс-валидация (перекрестная проверка) – это метод оценки Моделей (Model) Машинного обучения (Machine Learning) путем обучения нескольких из них на подмножествах доступных входных данных и их оценки на другом дополнительном подмножестве. Такая проверка используется для обнаружения Переобучения (Overfitting). При K-Fold проверке данные делятся на k подмножеств. Теперь удержание повторяется k раз, так что каждый из k подмножеств используется в качестве проверочного набора, а другие подмножества k-1 объединяются, чтобы сформировать обучающий набор. Ошибка усредняется по всем k испытаниям, чтобы получить обощенную эффективность нашей модели. Каждая точка данных попадает в набор для проверки ровно один раз и попадает в обучающий набор k-1 раз.

Набор обучающих данных — это набор примеров, используемых в процессе обучения, который используется для соответствия параметрам (например, весам), например, классификатора.

Набор данных проверки — это набор данных примеров, используемых для настройки гиперпараметров. Базовый процесс использования набора данных проверки для выбора модели (как часть набора данных для обучения, набора данных проверки и набора тестовых данных): Поскольку наша цель - найти сеть, имеющую наилучшую производительность на новых данных, самый простой подход к сравнению различных сетей - оценить функцию ошибок с использованием данных, которые не зависят от данных, используемых для обучения. Различные сети обучаются путем минимизации соответствующей функции ошибок, определенной для набора обучающих данных. Затем производительность сетей сравнивается путем оценки функции ошибок с использованием независимого набора проверки, и выбирается сеть, имеющая наименьшую ошибку по сравнению с набором проверки. Такой подход называется методом удержания. Поскольку эта процедура сама по себе может привести к некоторому переоснащению набора для проверки, производительность выбранной сети должна быть подтверждена путем измерения ее производительности на третьем независимом наборе данных, называемом тестовым набором.

Набор тестовых данных — это набор данных, который не зависит от набора данных для обучения, но имеет то же распределение вероятностей, что и набор данных для обучения, то есть это набор примеров, используемых только для оценки производительности (т. Е. Обобщения) полностью определенного классификатора

Проблема с обучением нейронных сетей заключается в выборе количества обучающих эпох для использования. Слишком много эпох может привести к переобучению учебного набора данных, в то время как слишком мало может привести к модели недостаточного соответствия. Ранняя остановка – это метод, который позволяет указать произвольно большое количество периодов обучения и прекратить обучение, как только производительность модели перестает улучшаться в наборе проверенных данных.

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

Softmax используется для задач классификации нескольких классов. Это форма логистической регрессии, которая нормализует входное значение в вектор значений, который следует распределению вероятности, общая сумма которого равна 1.

Выходные значения находятся в диапазоне [0,1], что хорошо, потому что мы можем чтобы избежать двоичной классификации и учесть как можно больше классов или измерений в нашей модели нейронной сети.

При выполнении мультиклассовой классификации с использованием Softmax Regression у нас есть ограничение: наша модель предсказывает только один класс среди "с" классов. Для наших данных это означает, что модель предсказывает, что на изображении будет только одна из цифр (от 0 до 9).

Мы интерпретировали результат логистической модели как вероятность. Точно так же мы хотим интерпретировать выходные данные модели многоклассовой классификации как распределение вероятностей. Итак, мы хотим, чтобы наша модель выводила вектор размера "c" с каждым значением в векторе, представляющим вероятность каждого класса.

Другими словами, "с" значение в векторе представляет вероятность того, что наше предсказание будет "с" классом. Поскольку все они являются вероятностями, их сумма будет равна 1.

Изображение выглядит как текст

Автоматически созданное описание

Кросс-энтропия (или логарифмическая функция потерь – log loss): Кросс-энтропия измеряет расхождение между двумя вероятностными распределениями. Если кросс-энтропия велика, это означает, что разница между двумя распределениями велика, а если кросс-энтропия мала, то распределения похожи друг на друга.

Изображение выглядит как текст

Автоматически созданное описание

Для регрессии softmax мы используем потерю кросс-энтропии.

# 8. Механизм обратного распространения ошибки. Принципиальная логика основного цикла обучения нейронной сети в PyTorch. Слои функций потерь (Loss Functions) в PyTorch.

Этот алгоритм определяет два “потока” в сети. Входные сигналы двигаются в прямом направлении, в результате чего мы получаем выходной сигнал, из которого мы получаем значение ошибки. Величина ошибки двигается в обратном направлении, в результате происходит корректировка весовых коэффициентов связей сети.

Для выходного слоя корректировка весов интуитивна понятна, но для скрытых слоев долгое время не было известно алгоритма. Веса скрытого нейрона должны изменяться прямо пропорционально ошибке тех нейронов, с которыми данный нейрон связан. Вот почему обратное распространение этих ошибок через сеть позволяет корректно настраивать веса связей между всеми слоями. В этом случае величина функции ошибки уменьшается и сеть обучается.

В качестве метода минимизации ошибки используется метод градиентного спуска, суть этого метода сводится к поиску минимума (или максимума) функции за счет движения вдоль вектора градиента (мы ищем минимум функции ошибки). Для поиска минимума движение должно быть осуществляться в направлении антиградиента.

Алгоритм распространения ошибки сводится к следующим этапам:

* прямое распространение сигнала по сети, вычисления состояния нейронов;
* вычисление значения ошибки δ для выходного слоя;
* обратное распространение: последовательно от конца к началу для всех скрытых слоев вычисляем δ по формуле 2.13;
* обновление весов сети на вычисленную ранее δ ошибки.

Изображение выглядит как текст

Автоматически созданное описание

Принципиальная логика основного цикла обучения нейронной сети в PyTorch включает несколько ключевых шагов:

1. **Инициализация нейронной сети**: Сначала вы создаете архитектуру нейронной сети, определяя слои и функции активации.
2. **Выбор функции потерь и оптимизатора**: Выберите функцию потерь (например, кросс-энтропию для задач классификации) и оптимизатор (например, SGD или Adam), которые будут использоваться для обучения сети.
3. **Цикл обучения**:
   * **Прямое распространение (Forward Pass)**: На этом этапе данные проходят через нейронную сеть, и на выходе получается прогноз.
   * **Вычисление функции потерь**: Оценивается, насколько предсказания сети отличаются от истинных значений.
   * **Обратное распространение (Backward Pass)**: Вычисляются градиенты функции потерь по отношению к весам сети.
   * **Оптимизация**: Оптимизатор обновляет веса сети в направлении уменьшения потерь.
4. **Валидация**: После каждой эпохи (полного прохода через обучающий набор данных) обычно проводится валидация на отдельном наборе данных, чтобы проверить, как сеть обобщает на новые данные.
5. **Повторение**: Эти шаги повторяются на протяжении многих эпох до тех пор, пока сеть не будет достаточно обучена.

Функция потерь (loss function) является основным компонентом любой модели машинного обучения. Она используется для измерения того, насколько хорошо модель предсказывает ожидаемые результаты. Чем меньше значение функции потерь, тем лучше модель справляется с предсказанием.

Существует множество различных функций потерь, каждая из которых подходит для решения определенной задачи. Например, если вы используете модель для классификации, то часто используют функцию потерь "categorical cross-entropy". Если вы используете модель для регрессии, то часто используют "mean squared error".

В некоторых случаях, модели используют составные функции потерь, называемые слоями функций потерь. Это означает, что модель использует несколько различных функций потерь для решения различных аспектов задачи. Например, в задаче классификации изображения может использоваться сочетание "categorical cross-entropy" и "mean squared error" функции потерь. Первая из них используется для измерения точности классификации, а вторая - для измерения качества детекции объекта на изображении.

Использование слоев функций потерь может помочь модели сосредоточиться на решении различных аспектов задачи и достичь лучшей качества предсказания. Однако, необходимо быть осторожным, так как использование неподходящей комбинации функций потерь может привести к переобучению модели.

**class torch.nn.MSELoss(size\_average=None, reduce=None, reduction='mean')**

Создает критерий,который измеряет среднюю квадратичную ошибку (квадратичная норма L2)между каждым элементом на входе Х and target У .

Восстановленные (т.е. с reduction установленным на 'none' ) потери можно описать как:

Изображение выглядит как текст

Автоматически созданное описание

where N размер партии. Если reduction не равно 'none' (по умолчанию 'mean' ), то:

Изображение выглядит как текст

Автоматически созданное описание

x and y тензоры произвольной формы с общим числом элементов n каждый.

Операция среднего значения по-прежнему работает над всеми элементами и делит на n .Деление на n можно избежать, если установить reduction = 'sum' .

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

# 9. Дифференцируемое программирование и реализация обратного распространения ошибки. Автоматическое дифференцирование в PyTorch. Пример и применение в обучении ИНС.

Дифференцируемое программирование (differentiable programming) – парадигма программирования, при которой программа (функция расчета значения) может быть продифференцирована в любой точке, обычно с помощью автоматического дифференцирования.

Это свойство позволяет использовать к программе методы оптимизации, основанные на расчете градиента, обычно - методы градиентного спуска.

Дифференцируемое программирование используется в:

* глубоком обучении
* глубоком обучении комбинированном с физическими моделями в робототехнике
* специализированных методах трассировки лучей
* обработке изображений

Большинство фреймоврков для дифференцируемого программирования использует граф потока вычислений определяющий выполнение программы и ее структуры данных.

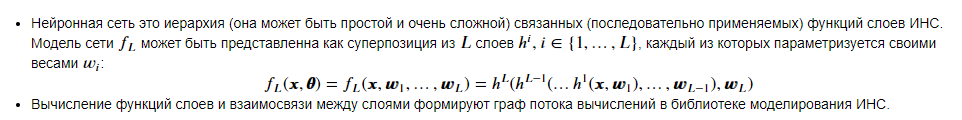
Как реализовать алгоритм обратного распространения ошибки удобно для использования в задачах моделирования ИНС?

Основная абстракция TensorFlow, PyTorch и других аналогичных библиотеках - граф потока вычислений.

Рассматриваемые библиотеки обычно:

* задают граф потока вычислений (формирует объект отложенных вычислений)
* запускают процедуру выполнения отложенных вычислений и получает результаты вычислений (в т.ч. ошибку модели).

Возможность в явном виде работать с графом потока вычислений дает большое преимущество для автоматического решения задачи обратного распространения ошибки, являющейся составляющей задачей обучения модели ИНС.



Изображение выглядит как текст

Автоматически созданное описание

PyTorch предоставляет мощный инструмент для автоматического дифференцирования, который может быть использован для обучения нейронных сетей. Это достигается с помощью создания графа вычислений, в котором каждый узел представляет собой операцию, а ребра представляют тензоры, которые передаются между операциями. При вычислении градиента используется обратное распространение ошибки (backpropagation).

Чтобы использовать автоматическое дифференцирование в PyTorch, необходимо создать объект torch.Tensor, связать его с графом вычислений и затем использовать функцию backward() для вычисления градиента.

Пример:

import torch

# Create tensors

x = torch.tensor([2.0], requires\_grad=True)

y = x\*\*2 + 2\*x + 1

# Compute gradients

y.backward()

# Print gradients

print(x.grad) # Output: tensor([4.])

В этом примере создается тензор x с одним элементом, который имеет параметр requires\_grad равным True. Этот параметр говорит PyTorch, что мы хотим вычислять градиенты по этому тензору. Затем мы создаем новый тензор y, который является результатом операции x\*\*2 + 2\*x + 1. Эта операция создает граф вычислений, в котором тензор x является источником, а тензор y является следующим узлом.

Затем мы вызываем функцию y.backward(), чтобы вычислить градиенты по тензору x. Это делается с использованием алгоритма обратного распространения ошибки. После этого мы можем получить градиент по тензору x с помощью свойства x.grad. В данном случае вывод будет tensor([4.])

В обучении ИНС автоматическое дифференцирование используется для вычисления градиентов по параметрам модели при обратном распространении ошибки. Это позволяет обновлять параметры модели в направлении антиградиента, что помогает уменьшить значение функции потерь.

Пример использования автоматического дифференцирования в обучении ИНС:

import torch

# Define a model

model = torch.nn.Linear(2, 1)

# Define loss function

loss\_fn = torch.nn.MSELoss()

# Define optimizer

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Generate some data

X = torch.randn(100, 2)

Y = X @ torch.tensor([[2.0], [3.0]]) + torch.randn(100, 1)

# Training loop

for i in range(100):

# Forward pass

Y\_pred = model(X)

loss = loss\_fn(Y\_pred, Y)

# Backward pass and optimization

optimizer.zero\_grad()

loss.backward()

optimizer.step()

В этом примере мы создаем линейную модель с двумя входами и одним выходом, затем определяем функцию потерь как среднеквадратичную ошибку и оптимизатор как стохастический градиентный спуск. Затем мы генерируем некоторые данные и запускаем цикл обучения, который состоит из шагов прямого и обратного распространения. На каждом шаге мы делаем прямой проход через модель, чтобы получить предсказания, вычисляем значение функции потерь и делаем обратный проход через модель с помощью функции loss.backward(). Это вычисляет градиенты по параметрам модели и записывает их в .grad атрибуты соответствующих тензоров. Затем мы вызываем optimizer.step(), чтобы обновить параметры модели с использованием вычисленных градиентов. И затем обнуляем градиенты optimizer.zero\_grad() перед каждым новым проходом.

Это один из способов использования автоматического дифференцирования в PyTorch для обучения нейронных сетей. Этот метод позволяет упростить процесс обучения, избавив от необходимости ручного вычисления градиентов.

# 10. Стохастический градиентный спуск. Батчи обучающей выборки.

Существует версия градиентного спуска — стохастический градиентный спуск. Стохастический = случайный.

Пакетный градиентный спуск вычисляет один шаг, используя сразу весь набор данных, тогда как стохастический за шаг использует только 1 элемент. Можно два этих варианта скрестить, получив мини-пакетный (mini-batch) спуск, который за раз обрабатывает, к примеру, 100 элементов, а не все или один.

Два этих варианта ведут себя похоже, но не одинаково. Пакетный спуск действительно следует в направлении наискорейшего спуска, тогда как стохастический, используя только один элемент из обучающей выборки, не может верно вычислить градиент для всей выборки.

О разнице, плюсах и минусах данных двух подходов написано достаточно много, вкратце подведу итог:

* Пакетный спуск хорош для строго выпуклых функций, потому что уверенно стремится к минимуму глобальному или локальному.
* Стохастический в свою очередь лучше работает на функциях с большим количеством локальных минимумов — каждый шаг есть шанс, что очередное значение «выбьет» из локальной ямы и конечное решение будет более оптимальным, нежели для пакетного спуска.
* Стохастический вычисляется быстрее — на каждом шаге нужны не все элементы из выборки. Вся выборка целиком может не влезть в память. Но требуется больше шагов.
* Для стохастического легко добавить новые элементы во время работы («онлайн» обучение).
* В случае mini-batch, можно также векторизовать код, что значительно ускорит его выполнение.

Также у градиентного спуска существует множество модификаций — momentum, наискорейшего спуска, усреднение, Adagrad, AdaDelta, RMSProp и другие.

Метод градиентного спуска - метод нахождения локального экстремума (минимума или максимума) функции с помощью движения вдоль градиента. В нашем случае шаг метода градиентного спуска выглядит следующим образом:



Выполнение на каждом шаге градиентного спуска суммирование по всем (𝑥,𝑦)∈𝐷 обычно слишком неэффективно

Для выпуклых функций задача локальной оптимизации - найти локальный минимум (максимум) автоматически превращается в задачу глобальной оптимизации - найти точку, в которой достигается наименьшее (наибольшее) значение функции, то есть самую низкую (высокую) точку среди всех.

Оптимизировать веса одного перцептрона - выпуклая задача, но для большой нейронной сети целевая функция не является выпуклой.

У нейронных сетей функция ошибки может задавать очень сложный ландшафт с огромным числом локальных максимумов и минимумов. Это свойство необходимо для обеспечения выразительности нейронных сетей, позволяющей им решать так много разных задач

Градиентный спуск с изменяемой скоростью обучения

Параметр метода градиентного спуска: скорость обучения 𝜂 показывает, насколько сильно мы сдвигаем параметры в сторону градиента на очередном шаге.

Скорость обучения — это чрезвычайно важный параметр.

Если она будет слишком большой:

* алгоритм станет "прыгать" по практически случайным точкам пространства и не попадет в минимум, потому что все время будет его "перепрыгивать"

Если она будет слишком маленькой:

* обучение станет гораздо медленнее
* алгоритм рискует успокоиться и сойтись в первом же локальном минимуме, который скорее всего не окажется самым лучшим.

Простейшая стратегия управления скоростью обучения:

* сначала скорость обучения должна быть большой: это позволяет быстрее прийти в правильную область пространства поиска
* затем скорость обучения должна быть маленькой: это позволяет детально исследовать окрестности точки минимума и в итоге попасть в нее (затухание)

Адаптивные методы градиентного спуска

Замедление обучения с фиксированными параметрами никак не учитывает характеристики оптимизируемой функции. Адаптивные методы градиентного спуска меняют параметры ГС в зависимости от результатов взаимодействия с оптимизируемой функцией.

Пример:

В местах, где поверхность функции больше наклонена в одном измерении, чем в другом, обычный стохастический градиентный спуск может вести себя некорректно. Например, если минимум находится в сильно вытянутом «овраге»:

* шаг градиентного спуска будет направлен от одной близкой и крутой стенки этого оврага к другой
* когда точка попадет на другую стенку, градиент станет направлен в противоположную сторону
* т.е. в процессе оптимизации текущее решение будет "прыгать" между стенками оврага, но к общему минимуму будет продвигаться очень медленно

В этой ситуации могут помочь адаптивные методы градиентного спуска.

Метод импульсов

Идея метода импульс (momentum): точка не просто подчиняется правилам градиентного спуска, а подчиняется законам механики, в первую очередь имеет инерцию, т.е.:

* у точки есть скорость
* положение точки на следующем шаге определяется ее текущим положением и скоростью
* ускорение (скорость изменения скорости) определяется величиной градиента
* на каждом шаге пересчитывается как положение точки, так и ее скорость

Т.е. у точки, которая спускается по поверхности, появляется импульс, она движется по инерции и стремится этот импульс сохранить (отсюда и название метода).

Формальная запись метода импульсов:

Изображение выглядит как текст, антенна

Автоматически созданное описание

Изображение выглядит как стол

Автоматически созданное описание

Метод Нестерова

Изображение выглядит как текст

Автоматически созданное описание

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

Обучение сети — сложная задача оптимизации в пространстве очень высокой размерности, которая фактически решается методами локального поиска. Для таких задач один из ключевых вопросов: где начинать этот локальный поиск?

Качество начального приближения принципиально влияет на получаемые в результате локальные оптимумы.

Хорошая инициализация весов может позволить нам обучать глубокие сети:

* лучше (в смысле метрик качества)
* быстрее (в смысле числа требующихся обновлений весов, т.е. числа итераций, т.е. времени обучения)

Первая идея, которая привела к большим успехам в этом направлении: предобучение без учителя (unsupervised pretraining):

* отдельные слои глубокой сети последовательно обучаются без учителя
* затем веса полученных слоев считаются начальным приближением и дообучаются уже на размеченном наборе данных
* сначала основным инструментом для предобучения без учителя в стали так называемые ограниченные машины Больцмана
* затем для этого стали использоваться автокодировщики

Основные приниципы испльзующиеся при предобучении:

* Предобучение слоев происходит последовательно, от нижних к верхним.
* позволяет избежать проблемы затухающих градиентов
* существенно уменьшает объем вычислений на каждом этапе

Предобучение протекает без учителя, то есть без учета имеющихся размеченных данных. Это часто позволяет существенно расширить обучающую выборку (например, собрать миллионы изображений из интернета без описания намного проще, чем собрать даже тысячу правильно размеченных изображений).

В результате предобучения получается модель, которую затем нужно дообучить на размеченных данных. Модели, обученные таким образом, в конечном счете стабильно сходятся к существенно лучшим решениям, чем при случайной инициализации.

Но сейчас на практике предварительное послойное обучение проводится редко, т.к. был найдены более простой и хорошо мотивированный способ инициализации весов, позволяющий существенно ускорить обучение и улучшить качество, его часто называют инициализацией Ксавье (Xavier initialization).

Инициализация Ксавье





Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

# 13. Гиперпараметры. Скорость обучения и размер батча.

**Гиперпараметры** - это настройки, которые необходимо выбрать перед обучением нейронной сети. Они оказывают влияние на процесс обучения и качество обученной модели. Два важных гиперпараметра - это скорость обучения (learning rate) и размер батча (batch size).

Некоторые примеры гиперпараметров в нейронных сетях:

1. Число слоев: глубина сети, может влиять на сложность модели и ее способность обобщать на новые данные
2. Число нейронов в слое: число нейронов в каждом слое сети, может влиять на качество модели и ее способность запоминать функцию
3. Функция активации: функция, которая используется для вычисления выхода нейрона в каждом слое, некоторые примеры функций: ReLU, sigmoid, tanh, etc.
4. Размер батча: это количество примеров, которые используются для обновления весов в каждой итерации обучения.
5. Скорость обучения: это темп, с которым модель обновляет веса в зависимости от градиента. Если скорость обучения слишком высока, то модель может проскочить минимум функции, и наоборот

**Скорость обучения** определяет, насколько быстро модель будет обучаться. Если скорость слишком высока, то модель может перескочить экстремум функции потерь и не найти оптимальное решение. Если скорость слишком низкая, то обучение может занять слишком много времени. Оптимальная скорость обучения зависит от конкретной задачи и модели, и ее нужно подбирать экспериментально.

**Размер батча** определяет, сколько примеров из выборки используются для обновления весов модели на протяжении одной итерации обучения. Обычно используется значение от 32 до 512. Больший размер батча обычно позволяет ускорить обучение, так как уменьшает число обновлений весов, но может привести к ухудшению качества модели, так как увеличивает разброс градиента. Маленький размер батча, с другой стороны, может улучшить качество модели, но может уменьшить скорость обучения. Как и скорость обучения, размер батча также зависит от конкретной задачи и модели, и его также нужно подбирать экспериментально.

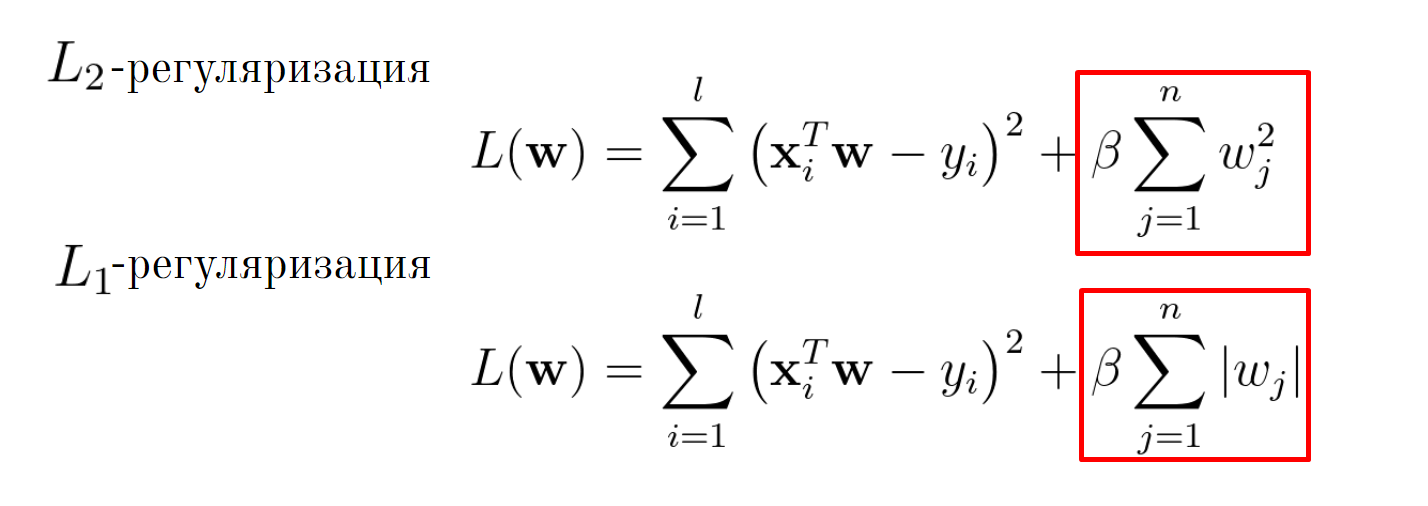
Важно отметить, что **нет оптимальных значений скорости обучения и размера батча**, которые бы работали во всех случаях. Эти гиперпараметры требуется подбирать отдельно для каждой задачи и модели. Обычно это делается с помощью методов типа сетки поиска (grid search) или случайного поиска (random search). В любом случае, важно провести несколько экспериментов с различными значениями этих гиперпараметров, чтобы понять, как они влияют на процесс обучения и качество модели.

# 14. Переобучение модели и регуляризация. Dropout. Слои регуляризации (Dropout Layers) в PyTorch.

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!

Weight decay

Так как существует бесконечное количество весов, дающих одинаковые предсказания, а для оптимального обучения веса модели не должны быть большими по модулю, то можно изменить функцию потерь так, чтобы штрафовать за большие значения. Наиболее часто используются L1 и L2 регуляризации.



Изображение выглядит как текст

Автоматически созданное описание

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

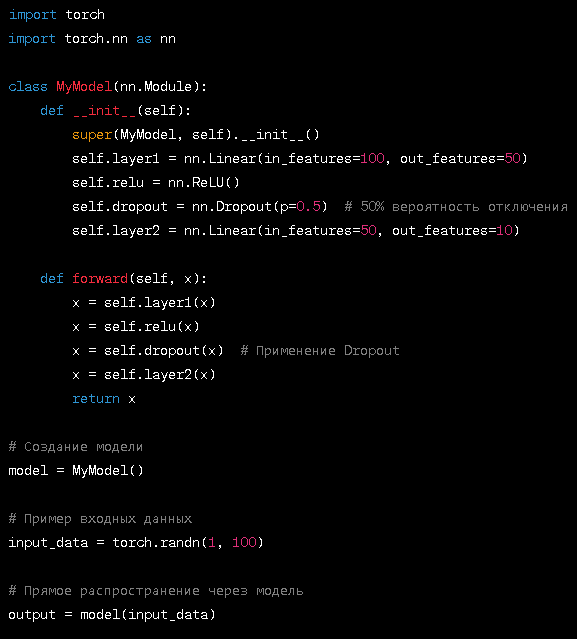
Практика обучения нейронных сетей показывает, что дропаут действительно дает очень серьезные улучшения в качестве обученной модели

Дропаут — это метод добиться усреднения огромного числа моделей. Он эквивалентен усреднению всех моделей, которые получались на каждом шаге случайным выбрасыванием отдельных нейронов.

#ChatGPT

слои регуляризации, такие как Dropout, используются для предотвращения переобучения сети путем случайного отключения некоторых нейронов во время обучения. Это помогает сделать модель более устойчивой к изменениям во входных данных.

Основной класс для Dropout в PyTorch - это **torch.nn.Dropout**. Его использование довольно простое. Вы создаете экземпляр **Dropout**, указывая вероятность отключения каждого нейрона (обычно от 0.2 до 0.5), и применяете этот слой к данным в вашей сети. Вот пример:



Важно помнить, что Dropout должен быть активен только во время обучения. Во время оценки модели (валидации или тестирования) его следует отключать. Это делается автоматически, если вы используете **model.eval()**. Во время обучения необходимо переключить модель обратно в режим обучения с помощью **model.train()**.

# 15. Минбатчи – причина использования. Нормализация по мини-батчам. Слои нормализации (Normalization Layers) в PyTorch.

Проблема внутреннего сдвига переменных (internal covariance shift) при глубоком обучении:

Если на очередном шаге градиентного спуска меняются веса одного из первых (нижних) слоев

* изменяются распределения активаций выходов этого слоя
* всем последующим слоям надо адаптироваться к новому распределению входных данных

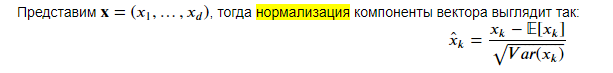
Сопутствующая проблема: "насыщение" функций активации

Часто в нейронных сетях используются сигмоидальные функции активации ( 𝑓(𝑥)=𝜎(𝑥) , 𝑓(𝑥)=tanh(𝑥) ), одной из особенностей которых является "насыщение" значений функций активации:

* когда входы получают большие по модулю значения производная 𝑓′(𝑥) быстро стремится к 0
* отрицательное следствие: при близких к 0 производных обратное распространение ошибки очень сильно затухает на этих градиентах
* потенциальное решение: замена функций активации на 𝑅𝑒𝐿𝑈 , но это не единственный и не всегда подходящий способ

Наиболее распространенный метод решения: нормализация данных. Нормализация входов часто помогает, и прежде чем обучать нейронную сеть, ее желательно делать.

Введем слой нормализации.



Среднее и дисперсию в формуле хотелось нужно вычислять по всему датасету, но это совершенно невозможно вычислительно, поэтому применим очередное упрощение: будем рассчитывать эти величины по текущему мини-батчу. Данный подход называется нормализацией по мини-батчам.

Недостатки нормализации по мини-батчам:

Если используется сигмоидная функция активации (например: 𝑓(𝑥)=𝜎(𝑥) , 𝑓(𝑥)=tanh(𝑥) ), то после нормализации ее аргумента нелинейность по сути пропадает.

Для исправления этого недостатка, слой нормализации должен иметь возможность "настроиться" как тождественная функция. Т.е. при некоторых комбинациях параметров он должен работать как 𝑓(𝑥)=𝑥 .

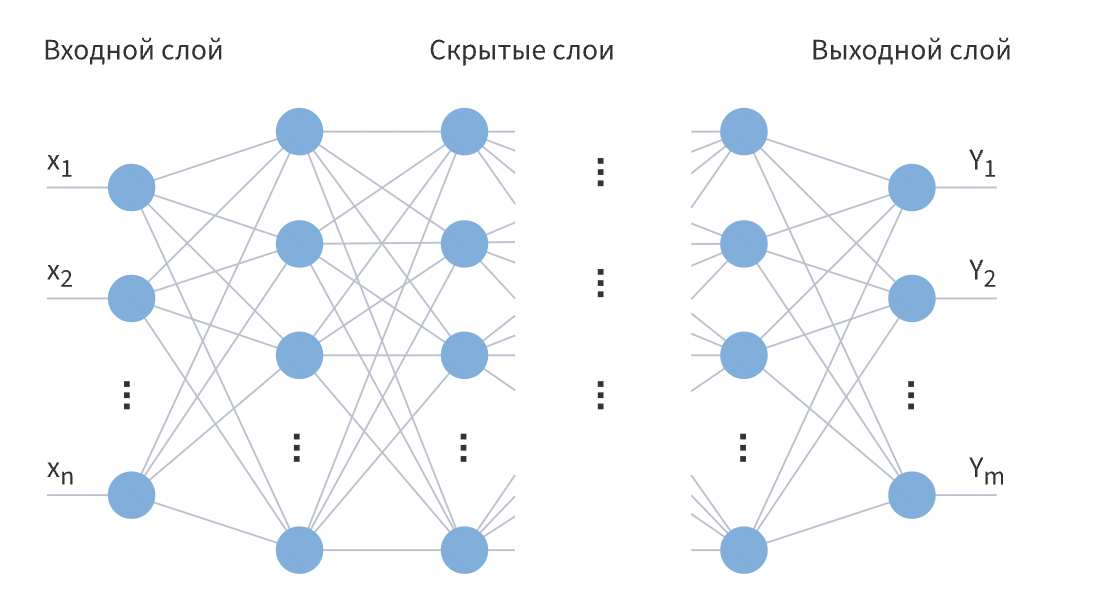
#Chatgpt

Слои нормализации в PyTorch играют важную роль в обучении нейронных сетей, помогая ускорить обучение и повышать его стабильность. Наиболее популярные слои нормализации в PyTorch:

1. **Batch Normalization (torch.nn.BatchNorm1d, BatchNorm2d, BatchNorm3d)**: Этот слой нормализует выходные данные предыдущего слоя по мини-пакетам, т.е. для каждого признака в мини-пакете он вычитает среднее значение и делит на стандартное отклонение. Это помогает уменьшить внутреннюю ковариационную сдвиг (Internal Covariate Shift).
2. **Layer Normalization (torch.nn.LayerNorm)**: Нормализация на уровне слоя применяется к каждому элементу в мини-пакете и нормализует статистику по всем признакам внутри каждого образца. Это особенно полезно для рекуррентных нейронных сетей (RNN).
3. **Instance Normalization (torch.nn.InstanceNorm1d, InstanceNorm2d, InstanceNorm3d)**: Применяется в основном в задачах компьютерного зрения, таких как стилизация изображений. В отличие от пакетной нормализации, инстансная нормализация нормализует каждый канал отдельно для каждого образца в пакете.
4. **Group Normalization (torch.nn.GroupNorm)**: Групповая нормализация разбивает каналы на группы и нормализует эти группы по отдельности. Это полезно, когда размер мини-пакета мал, и пакетная нормализация неэффективна.
5. **Spectral Normalization (torch.nn.utils.spectral\_norm)**: Спектральная нормализация применяется к весам слоёв, чтобы контролировать их липшицеву норму. Это часто используется в генеративных состязательных сетях (GAN).

**16. Многослойные сети. Граф потока вычислений. Класс nn.Module в PyTorch: назначение, основные поля и методы.**

Многослойными называются нейронные сети, в которых нейроны сгруппированы в слои. При этом каждый нейрон предыдущего слоя связан со всеми нейронами следующего слоя, а внутри слоёв связи между нейронами отсутствуют.



Слои нумеруются слева направо. Первый слой называют входным или распределительным. Его нейроны принимают элементы вектора признаков и распределяют их по нейронам следующего слоя. При этом обработка данных во входном слое не производится.

Последний слой называется выходным. На выходах его нейронов формируется результат работы сети — элементы выходного вектора.

Между входным и выходным слоем располагаются один или несколько промежуточных или скрытых слоёв. Скрытыми они называются потому, что их входы и выходы неизвестны для внешних по отношению к нейронной сети программам и пользователю.

Для обучения многослойных нейронных сетей используется обучение с учителем. Наиболее популярным алгоритмом обучения для них является алгоритм обратного распространения ошибки и его разновидности.

Основная абстракция TensorFlow, PyTorch и других аналогичных библиотеках - граф потока вычислений.

Под глубоким обучением как правило понимают обучение функции, представляющей собой композицию множества нелинейных преобразований. Такая сложная функция ещё называется потоком или графом вычислений. Фреймворк глубокого обучения должен уметь делать всего три вещи:

1. Определять граф вычислений;
2. Дифференцировать граф вычислений;
3. Вычислять его.

Рассматриваемые библиотеки обычно:

* задают граф потока вычислений (формирует объект отложенных вычислений)
* запускают процедуру выполнения отложенных вычислений и получает результаты вычислений (в т.ч. ошибку модели). Возможность в явном виде работать с графом потока вычислений дает большое преимущество для автоматического решения задачи обратного распространения ошибки, являющейся составляющей задачей обучения модели ИНС.
* Нейронная сеть это иерархия (она может быть простой и очень сложной) связанных (последовательно применяемых) функций слоев ИНС.
* Вычисление функций слоев и взаимосвязи между слоями формируют граф потока вычислений в библиотеке моделирования ИНС.

Класс **nn.Module** в PyTorch является базовым классом для всех нейронных сетей и обеспечивает большую часть необходимой функциональности для создания и управления моделями. Вот основные аспекты этого класса:

**Назначение**

1. **Организация компонентов сети**: **nn.Module** позволяет объединять различные слои и функции активации в единую сетевую структуру.
2. **Управление параметрами**: Автоматически управляет параметрами сети (весами, смещениями), что облегчает их оптимизацию и сохранение.
3. **Поддержка GPU**: Поддерживает автоматическое переключение между использованием CPU и GPU.

**Основные поля**

1. **Параметры сети**: Все параметры (веса, смещения) модулей, являющиеся его подклассами, автоматически регистрируются в списке параметров.
2. **Подмодули**: Можно добавлять дочерние модули, которые также являются экземплярами **nn.Module**.

**Основные методы**

1. **forward**: Основной метод, который определяет прямой проход (forward pass) через модель. Этот метод должен быть переопределен при создании нового класса, наследующего от **nn.Module**.
2. **\_\_init\_\_**: Конструктор класса, где определяются слои и другие компоненты модели.
3. **to(device)**: Перемещает параметры и буферы модели на указанный устройство (**cpu** или **cuda**).
4. **state\_dict()**: Возвращает словарь, содержащий все параметры модели.
5. **load\_state\_dict(state\_dict)**: Загружает параметры из словаря в модель.
6. **zero\_grad()**: Обнуляет градиенты параметров, что необходимо перед выполнением обратного распространения ошибки (backpropagation).
7. **eval()**: Переводит модель в режим оценки (evaluation mode), который отключает слои, ведущие себя по-разному во время обучения и тестирования (например, Dropout).
8. **train()**: Переключает модель в режим обучения.

Специфические задачи глубокого обучения

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

**Сверточные нейронные сети** (convolutional neural networks, CNN) — класс архитектур ИНС, основная идея которых состоит в том, чтобы переиспользовать одни и те же части нейронной сети для работы с разными локальными участками входов.

* Идея сверточных сетей основана на результатах исследований зрительной коры головного мозга
* Изначально основным приложением сверточных нейронных сетей является обработка изображений.
* Обработка изображений до сих пор является одним из основных приложений для сверточных сетей, однако существует множество примеров применения сверточных нейронных сетей в самых различных областях.

Существует разделение функциональности различных зон визуальной коры. Различные зоны выделяют: локальные признаки небольших участков считанного с сетчатки изображения; локальные признаки, слегка обобщая; цвет, текстуры объектов; геометрические фигуры и очертания объектов; модуляция посредством нашего внимания; занимается распознаванием движений; обобщаются данные обо всей картинке; распознавание сложных объектов;

Данная функциональная специализация хорошо соответствует логике глубоких ИНС: более высокие уровни нужны для того, чтобы выделять более общие признаки, соответствующие абстрактным свойствам входа, а на нижних уровнях признаки более конкретные.

Иерархия среди зон не очень строгая: есть масса прямых связей, когда, например, нейроны из зоны V1 подаются на вход не только в зону V2, но и напрямую в зону V5. Анлогичных подходы используются и в сверточных архитектурах ИНС. Считается, что внимание, которое зарождается в зоне V4, потом переходит обратно к V2 и V1

Специфика задач МО на изображениях:

* Данные в изображениях имеют пространственную структуру, закодированную в виде структуры массива, хранящего информацию о пикселях
* В большинстве случаев аффинные (и многие более сложные) преобразования не меняют смысл изображения (при этом массив входных данных меняется кардинально)
* Двухмерные изображения часто имеют несколько каналов (фактически изображение представляется как тензор: Высота \* Ширина \* Количество\_каналов)
* Объем данных хранящийся даже в одной небольшой картинке очень велик (например RGB изображения 256\*256 пикселей хранит ~200 тыс. значений)

Нам известно, что нейронные сети хороши в распознавании изображений. Причём хорошая точность достигается и обычными сетями прямого распространения, однако, когда речь заходит про обработку изображений с большим числом пикселей, то число параметров для нейронной сети многократно увеличивается. Причём настолько, что время, затрачиваемое на их обучение, становится невообразимо большим.

Так, если требуется работать с цветными изображениями размером 64х64, то для каждого нейрона первого слоя полносвязной сети потребуется 64·64·3 = 12288 параметров, а если сеть должна распознавать изображения 1000х1000, то входных параметров будет уже 3 млн! А помимо входного слоя есть и другие слои, на которых, зачастую, число нейронов превышает количество нейронов на входном слое, из-за чего 3 млн запросто превращаются в триллионы! Такое количество параметров просто невозможно рассчитать быстро ввиду недостаточно больших вычислительных мощностей компьютеров.

Главной особенностью свёрточных сетей является то, что они работают именно с изображениями, а потому можно выделить особенности, свойственные именно им. Многослойные персептроны работают с векторами, а потому для них нет никакой разницы, находятся ли какие-то точки рядом или на противоположных концах, так как все точки равнозначны и считаются совершенно одинаковым образом. Изображения же обладают локальной связностью. Например, если речь идёт об изображениях человеческих лиц, то вполне логично ожидать, что точки основных частей лица будут рядом, а не разрозненно располагаться на изображении. Поэтому требовалось найти более эффективные алгоритмы для работы с изображениями и ими оказались свёрточные сети.

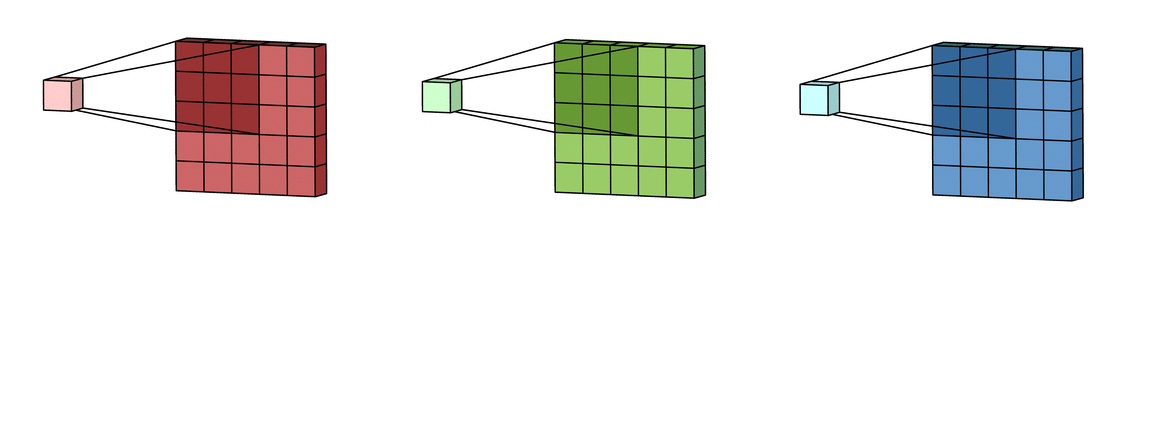
Свёрточные нейронные сети состоят из базовых блоков, благодаря чему их можно собирать как конструктор, добавляя слой за слоем и получая всё более мощные архитектуры. Основными блоками свёрточных нейронных сетей являются свёрточные слои, слои подвыборки (пулинга), слои активации и полносвязные слои.

**Слой свёртки**, как можно догадаться по названию типа нейронной сети, является самым главным слоем сети. Его основное назначение – выделить признаки на входном изображении и сформировать карту признаков. Карта признаков – это всего лишь очередной тензор (массив матриц), в котором каждый канал отвечает за какой-нибудь выделенный признак.

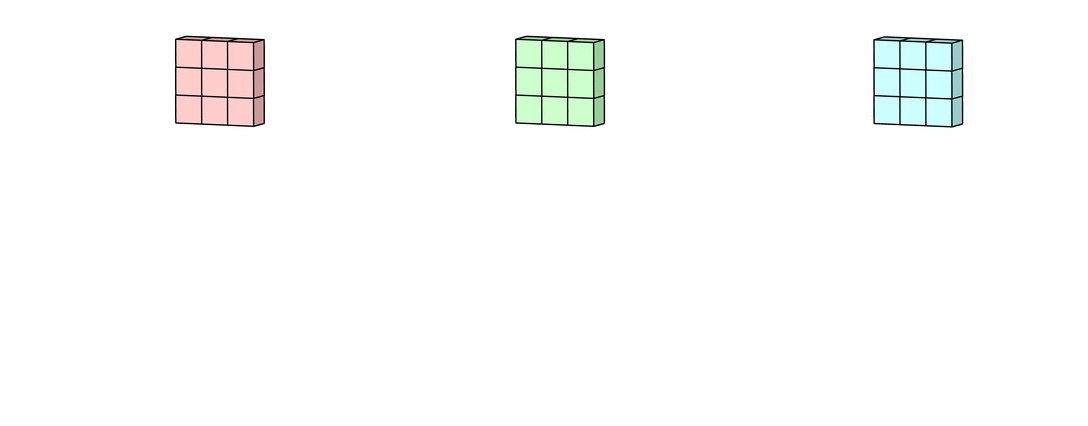
Для того, чтобы слой мог выделять признаки, в нём имеются так называемые фильтры (или ядра). Ядра — это всего лишь набор тензоров. Эти тензоры имеют один и тот же размер, а их количество определяет глубину выходного 3D массива. При этом глубина самих фильтров совпадает с количеством каналов входного изображения.

Для того, чтобы сформировать карту признаков из входного изображения, производится операция свёртки входного тензора с каждым из фильтров. Свёртка – это операция вычисления нового значения выбранного пикселя, учитывающая значения окружающих его пикселей. Алгоритм получения результата свёртки можно описать так:

Фильтр накладывается на левую верхнюю часть изображения и производится покомпонентное умножение значений фильтра и значений изображения, после чего фильтр перемещается дальше по изображению до тех пор, пока аналогичным образом не будут обработаны все его участки.



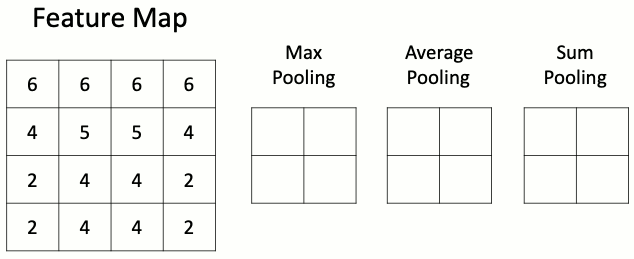
Затем числа полученных матриц суммируются в единую матрицу — результат применения фильтра.



После этого к каждому значению матрицы добавляется одинаковое число – значение смещения данного фильтра. Полученная матрица составляет один канал выходной карты признаков.

Слой подвыборки (пулинга)

Данный слой позволяет уменьшить пространство признаков, сохраняя наиболее важную информацию. Существует несколько разных версий слоя пулинга, среди которых максимальный пулинг, средний пулинг и пулинг суммы. Наиболее часто используется именно слой макспулинга.



Слой активации

Данный слой представляет из себя некоторую функцию, которая применяется к каждому числу входного изображения. Наиболее часто используются такие функции активации, как ReLU, Sigmoid, Tanh, LeakyReLU. Обычно активационный слой ставится сразу после слоя свёртки, из-за чего некоторые библиотеки даже встраивают ReLU функцию прямо в свёрточный слой.

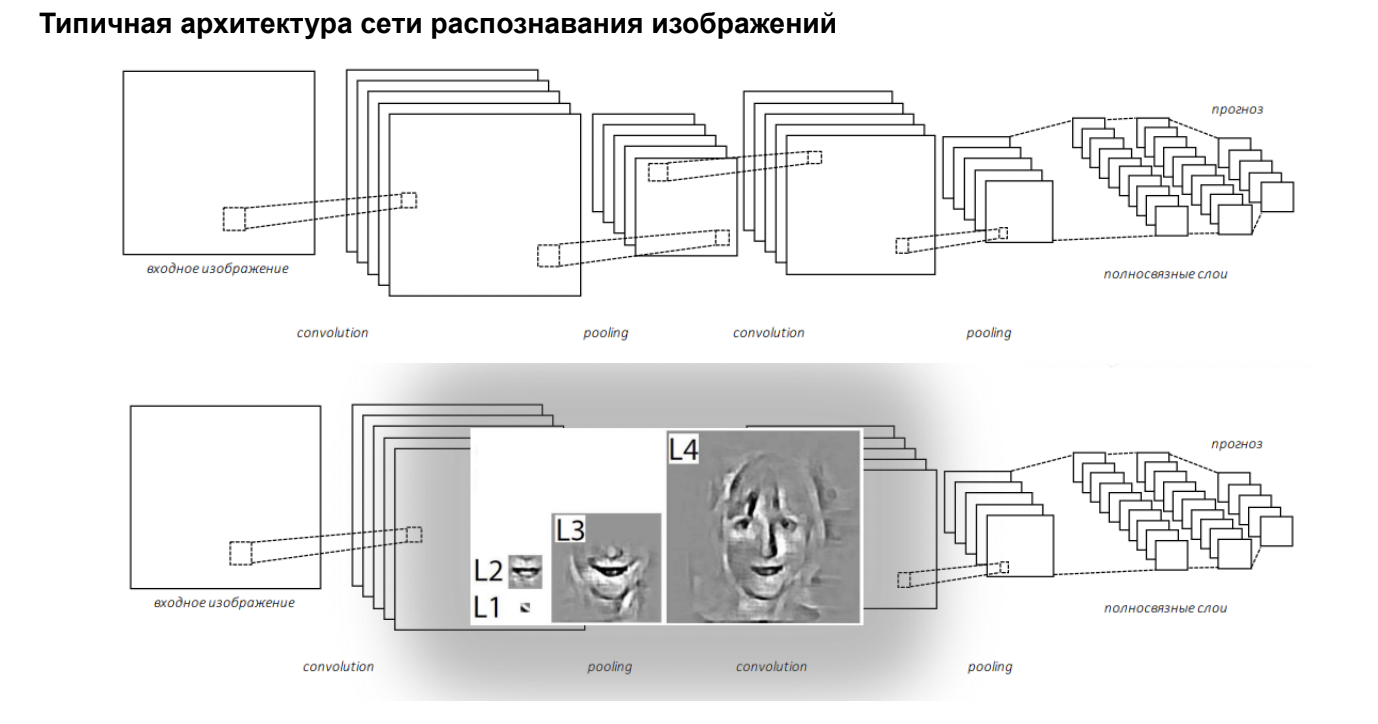
Полносвязный слой

Данный слой содержит матрицу весовых коэффициентов и вектор смещений и ничем не отличается от такого же слоя в обыкновенной полносвязной сети. Единственным гиперпараметром слоя является количество выходных значений. При этом результатом применения слоя является вектор или тензор, у которого матрицы в каждом канале имеют размер 1х1

Обучение свёрточной сети

Как и полносвязная нейронная сеть, свёрточная сеть обучается с помощью алгоритма обратного распространения ошибки. Сначала выполняется прямое распространение от первого слоя к последнему, после чего вычисляется ошибка на выходном слое и распространяется обратно. При этом на каждом слое вычисляются градиенты обучаемых параметров, которые в конце обратного распространения используются для обновления весов с помощью градиентного спуска.

# 18. Архитектура многослойной ИНС распознавания изображений на основе сверточных сетей. Сверточные слои (Convolution Layers) и сжимающие слои (Pooling Layers) в PyTorch.

****

CNN использует тот факт, что близлежащие пиксели более тесно связаны, чем отдаленные. Мы анализируем влияние близлежащих пикселей с помощью чего-то, называемого фильтром/ядром, и перемещаем его по изображению сверху слева вниз справа. Для каждой точки на изображении значение рассчитывается на основе фильтра с помощью операции свертки.

CNN имеют входной уровень, выходной и скрытые слои. Скрытые слои обычно состоят из сверточных слоев, слоев активации, слоев пулинга и полносвязных слоев.

**Основная архитектура CNN.**

**Свёртка (Convolution)** Основная цель шага свертки - извлечь элементы из входного изображения. Сверточный слой - это всегда первый шаг в CNN. Его основное назначение – выделить признаки на входном изображении и сформировать карту признаков. Карта признаков – это всего лишь очередной тензор (массив матриц), в котором каждый канал отвечает за какой-нибудь выделенный признак. Для того, чтобы слой мог выделять признаки, в нём имеются так называемые фильтры (или ядра). Эти тензоры имеют один и тот же размер, а их количество определяет глубину выходного 3D массива. При этом глубина самих фильтров совпадает с количеством каналов входного изображения.

Операцию свёртки можно представить следующим алгоритмом:

1. Скользящее окно, называемое фильтром, с размером (n,n) двигается по входному признаку. Количество движений определяется заданным количеством фильтров.
2. Каждый полученный шаблон имеет форму (n,n,d), где d — глубина входного признака.
3. Каждый шаблон умножается на своё ядро свёртки, в результате, формируется выходная карта признаков. Полученная выходная карта признаков имеет форму (h,w,N), где h и w — длина и ширина, полученные в результате отсечения, а N — количество фильтров.

**Активация** Применение функции активации на картах признаков нужно, чтобы увеличить нелинейность в сети, то есть чтобы нельзя было заменить всю сеть на одно линейное преобразование.

**Pooling** Модель должна уметь распознавать информацию в разных положениях объекта и видах. Объединение постепенно уменьшает размер входного представления. Это позволяет обнаруживать объекты на изображении независимо от того, где они находятся. Объединение помогает уменьшить количество необходимых параметров и количество необходимых вычислений. Это также помогает контролировать переобучение.

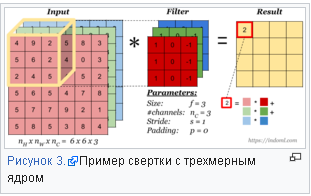
**Flattening** Вы сглаживаете объединенную карту элементов в последовательный столбец чисел (длинный вектор). Это позволяет информации стать входными данными для полносвязных слоёв.

**Полносвязный слой (Fully connected layer)** Основная цель полносвязных слоёв - объединить наши функции в большее количество атрибутов. Они будут предсказывать классы с большей точностью. На этом этапе ошибка вычисляется, а затем обратно реплируется. Веса и детекторы признаков настраиваются, чтобы помочь оптимизировать производительность модели. Затем процесс происходит снова и снова и снова. Именно так наша сеть обучается данным.

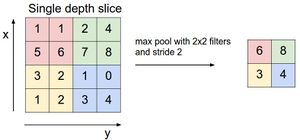
**Сверточный слой**

Сверточный слой нейронной сети представляет из себя применение операции свертки к выходам с предыдущего слоя, где веса ядра свертки являются обучаемыми параметрами. Еще один обучаемый вес используется в качестве константного сдвига (англ. *bias*). При этом есть несколько важных деталей:

* В одном сверточном слое может быть несколько сверток. В этом случае для каждой свертки на выходе получится своё изображение. Например, если вход имел размерность w×h, а в слое было n сверток с ядром размерности kx×ky, то выход будет иметь размерность ;
* Ядра свертки могут быть трёхмерными. Свертка трехмерного входа с трехмерным ядром происходит аналогично, просто скалярное произведение считается еще и по всем слоям изображения. Например, для усреднения информации о цветах исходного изображения, на первом слое можно использовать свертку размерности 3×w×h. На выходе такого слоя будет уже одно изображение (вместо трёх);
* Можно заметить, что применение операции свертки уменьшает изображение. Также пиксели, которые находятся на границе изображения участвуют в меньшем количестве сверток, чем внутренние. В связи с этим в сверточных слоях используется дополнение изображения (англ. *padding*). Выходы с предыдущего слоя дополняются пикселями так, чтобы после свертки сохранился размер изображения. Такие свертки называют *одинаковыми* (англ. *same convolution*), а свертки без дополнения изображения называются *правильными* (англ. *valid convolution*). Среди способов, которыми можно заполнить новые пиксели, можно выделить следующие:
  + *zero shift*: 00[ABC]00;
  + *border extension*: AA[ABC]CC;
  + *mirror shift*: BA[ABC]CB;
  + *cyclic shift*: BC[ABC]AB.
* Еще одним параметром сверточного слоя является *сдвиг* (англ. *stride*). Хоть обычно свертка применяется подряд для каждого пикселя, иногда используется сдвиг, отличный от единицы — скалярное произведение считается не со всеми возможными положениями ядра, а только с положениями, кратными некоторому сдвигу s. Тогда, если если вход имел размерность w×h а ядро свертки имело размерность kx×ky и использовался сдвиг s, то выход будет иметь размерность 



Пулинговый слой

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Maxpool.jpeg)

[Рисунок 4.](https://www.slideshare.net/YUNGKUEICHEN/convolutional-neural-network-cnn-image-recognition) Пример операции пулинга с функцией максимума

Пулинговый слой призван снижать размерность изображения. Исходное изображение делится на блоки размером w×h�×ℎ и для каждого блока вычисляется некоторая функция. Чаще всего используется функция максимума (англ. *max pooling*) или (взвешенного) среднего (англ. *(weighted) average pooling*). Обучаемых параметров у этого слоя нет. Основные цели пулингового слоя:

* уменьшение изображения, чтобы последующие свертки оперировали над большей областью исходного изображения;
* увеличение инвариантности выхода сети по отношению к малому переносу входа;
* ускорение вычислений.

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

**Расширение данных**

Причиной переобучения является недостаточное количество образцов для обучения модели, способной обобщать новые данные. Имея бесконечный объем данных, можно было бы получить модель, учитывающую все аспекты распределения данных: эффект переобучения никогда не наступил бы. Прием расширения данных реализует подход создания дополнительных обучающих данных из имеющихся путем трансформации образцов множеством случайных преобразований, дающих правдоподобные изображения. Цель состоит в том, чтобы на этапе обучения модель никогда не увидела одно и то же изображение дважды.

Это поможет модели выявить больше особенностей данных и достичь лучшей степени обобщения.

**Использование предварительно обученной глубокой нейронной сети и её дообучение**

Типичным и эффективным подходом к глубокому обучению на небольших наборах изображений является использование предварительно обученной сети. Предварительно обученная сеть — это сохраненная сеть, прежде обученная на большом наборе данных, обычно в рамках масштабной задачи классификации изображений. Если этот исходный набор данных достаточно велик и достаточно обобщен, тогда пространственная иерархия признаков, изученных сетью, может эффективно выступать в роли обобщенной модели видимого мира и быть полезной во многих разных задачах распознавания образов, даже если эти новые задачи будут связаны с совершенно иными классами, отличными от классов в оригинальной задаче.

Например, можно обучить сеть на изображениях из ImageNet (где подавляющее большинство классов — животные и бытовые предметы) и затем использовать эту обученную сеть для идентификации чего-то иного, например предметов мебели на изображениях. Такое можно осуществить путём заморозки последних слоев, выделяющих признаки, и добавлением новой части модели (например, полносвязного классификатора).

Такая переносимость изученных признаков между разными задачами — главное преимущество глубокого обучения перед многими более старыми приемами поверхностного обучения, которое делает глубокое обучение очень эффективным инструментом для решения задач с малым объемом данных.

# 20. Схема работы сверточной сети. Операции свертки, пулинга, общий вид сверточной сети для решения задачи классификации изображения.

**Слой свёртки**. Его назначение – выделить признаки на входном изображении и сформировать карту признаков. Карта признаков – это массив матриц, в котором каждый канал отвечает за какой-нибудь выделенный признак. Для того, чтобы слой мог выделять признаки, в нём имеются фильтры. Эти тензоры имеют один и тот же размер, а их количество определяет глубину выходного массива. При этом глубина самих фильтров совпадает с количеством каналов входного изображения.

Свёртка – это операция вычисления нового значения выбранного пикселя, учитывающая значения окружающих его пикселей. Алгоритм получения результата свёртки:

1. Скользящее окно, называемое фильтром, с размером (n,n) двигается по входному признаку. Количество движений определяется заданным количеством фильтров. Каждый полученный шаблон имеет форму (n,n,d), где d — глубина входного признака.
2. Каждый шаблон умножается на своё ядро свёртки, в результате, формируется выходная карта признаков. Полученная выходная карта признаков имеет форму (h,w,N), где h и w — длина и ширина, полученные в результате отсечения, а N — количество фильтров.

Пулинг.

Данный слой позволяет уменьшить пространство признаков, сохраняя наиболее важную информацию. Существует несколько разных версий слоя пулинга, среди которых максимальный пулинг и средний пулинг. Наиболее часто используется именно слой макспулинга.

Max pooling - Он выводит максимальное значение в каждом окне. Часто он использует фильтр размером 2x2 и шагом 2.

Average pooling выводит среднее значение в каждом окне.

Данные гиперпараметры определяют порядок применения свертки.

**padding (Эффект границ)**

Каждый раз, когда к входному изображению применяется операция свертки, результирующее выходное изображение сжимается. Иногда для извлечения некоторых низкоуровневых объектов необходимо сохранить исходный размер изображения. Также пиксели в углах используются меньше, поэтому большой объем информации по краям используется неэффективно. Поэтому к данным добавляют границы.

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.

Изображение выглядит как седзи, кроссворд, с плиткой, коллекция картинок

Автоматически созданное описание

**kernel size**

Размер ядра (фильтра) свертки. Часто используют 3\*3 и 5\*5.

**stride**

Шаг свертки, то есть количество пикселей, на которые ядро должно перемещаться как по вертикали, так и по горизонтали во время свертки. Часто используется stride=1.

**dilation**

Контролирует расстояние между точками ядра. При dilation=1 промежутка между точками нет.

**Общий вид сверточной сети для решения задачи классификации изображений:**

* Входное изображение подается на сверточные слои, где извлекаются различные признаки.
* Следующие за ними слои пулинга снижают размерность этих признаков.
* После нескольких таких повторений, извлеченные признаки подаются на полносвязные слои, которые выполняют задачу классификации на основе этих признаков.

## Рекуррентные нейронные сети

# 21. 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

Изображение выглядит как текст, часы

Автоматически созданное описание

Трудность рекуррентной сети:

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

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…𝑇 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

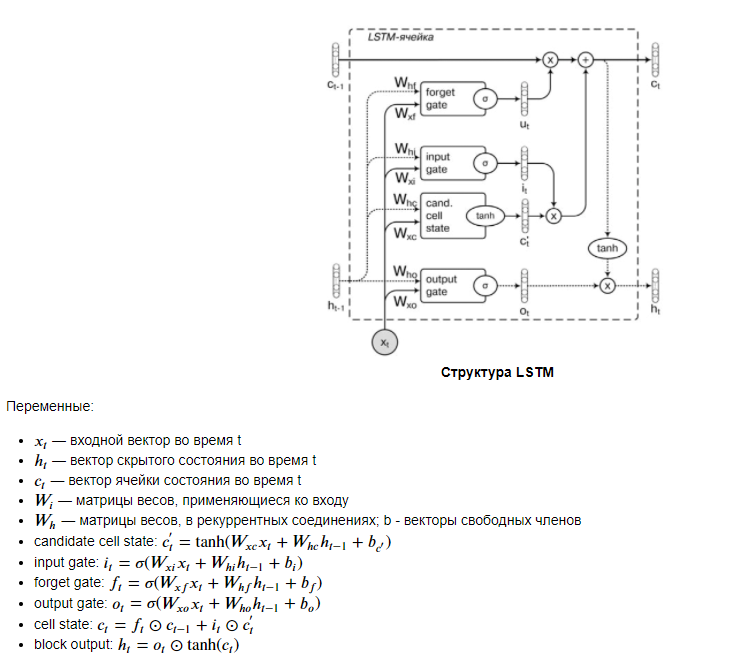
# 22. Модуль LSTM.

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



Изображение выглядит как текст

Автоматически созданное описание

Существует много разных вариантов LSTM:

* LSTM без входного гейта 𝑖𝑡
* LSTM без забывающего гейта 𝑓𝑡
* LSTM без выходного гейта 𝑜𝑡
* LSTM без функции активации 𝜎 на входном гейте
* LSTM без функции активации 𝜎 на выходном гейте
* LSTM без замочных скважин
* LSTM со связанными входным и забывающим гейтом
* LSTM с дополнительными рекуррентными связями на каждом гейте

LSTM требует довольно значительных ресурсов

* В обычном RNN каждая ячейка имеет один вектор скрытого состояния ℎ , а веса представлены тремя матрицами (плюс свободные члены)
* В LSTM-ячейке даже в базовой модели участвует сразу восемь матриц весов

Критически важными компонентами для успешной работы LSTM выступают

* два гейта: выходной и забывающий
* «память» 𝑐𝑡 и константная ошибка, которая позволяет состоянию LSTM сохраняться надолго

# 23. Модуль GRU.

#ChatGpt, и некоторые вопросы сверху я переписал без использования chatgpt, но забыл убрать пометку об использовании

GRU, или блок рекуррентных элементов с затворами (Gated Recurrent Unit), представляет собой одну из разновидностей рекуррентных нейронных сетей (RNN). Основное отличие GRU от стандартных RNN заключается в том, что он использует так называемые затворы для контроля потока информации. Это помогает решить проблему исчезающего градиента, с которой часто сталкиваются стандартные RNN при обучении на длинных последовательностях данных.

GRU имеет два основных затвора:

1. **Затвор обновления (Update Gate)**: Определяет, насколько новая информация будет влиять на состояние ячейки. По сути, он решает, какая часть прошлого состояния должна сохраниться и какую новую информацию следует добавить.
2. **Затвор сброса (Reset Gate)**: Помогает модели решить, насколько важна предыдущая информация для текущего состояния. Если затвор сброса близок к нулю, это означает, что прошлая информация игнорируется, что полезно для моделирования временных разрывов в данных.

В целом, GRU используется во многих задачах обработки естественного языка (NLP) и последовательностных данных, таких как прогнозирование временных рядов, распознавание речи, машинный перевод и многое другое. Он предлагает баланс между сложностью и эффективностью обучения по сравнению с другими вариантами рекуррентных нейронных сетей, такими как LSTM (Long Short-Term Memory).

# 24. Класс torch.nn.LSTM и torch.nn.GRU в PyTorch.

LSTM (Long Short-Term Memory) и GRU (Gated Recurrent Unit) являются двумя типами рекуррентных слоев, которые широко используются в нейронных сетях для обработки последовательностей данных.

LSTM - это тип рекуррентного слоя, который использует контролируемые забывания (forget gates) и входные записи (input gates) для регулирования информации, которая проходит через ячейку состояния. Это позволяет LSTM эффективно запоминать и забывать данные на длительные промежутки времени.

GRU - это более простой вариант рекуррентного слоя, который также использует контролируемые забывания, но не имеет входные записи. Это делает его более простым в реализации и может дать хорошие результаты в задачах с небольшим количеством данных.

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

Применяет многослойную РНК с долговременной кратковременной памятью (LSTM)к входной последовательности.

**Parameters**

* input\_size — количество ожидаемых функций во входных данных x
* hidden\_size — количество объектов в скрытом состоянии h
* num\_layers — количество повторяющихся слоев. Например, установка num\_layers=2 будет означать объединение двух LSTM вместе для формирования stacked LSTM , при этом второй LSTM принимает выходные данные первого LSTM и вычисляет окончательные результаты. По умолчанию: 1
* смещение — если False , то слой не использует веса смещения b\_ih и b\_hh . По умолчанию: True
* batch\_first — если True , то тензоры ввода и вывода предоставляются как (batch, seq, feature). По умолчанию: False
* отсев — если он не равен нулю, вводит слой Dropout на выходах каждого слоя LSTM, кроме последнего слоя, с вероятностью отсева, равной dropout . По умолчанию: 0
* двунаправленный — если True , становится двунаправленным LSTM. По умолчанию: False
* proj\_size — если > 0 , будет использоваться LSTM с проекциями соответствующего размера. По умолчанию: 0

Если выполняются следующие условия: 1) cudnn включен, 2) входные данные находятся на GPU 3) входные данные имеют тип torch.float16 4) используется графический процессор V100, 5) входные данные не в формате PackedSequence , постоянный алгоритм может быть выбран для повышения производительности.



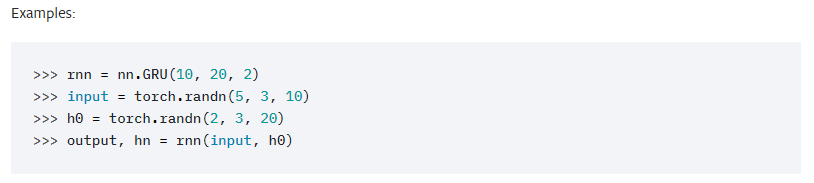
**class torch.nn.GRU(\*args, \*\*kwargs)**

Применяет многослойный рекуррентный блок (GRU)RNN к входной последовательности.

**Parameters**

* input\_size — количество ожидаемых функций во входных данных x
* hidden\_size — количество объектов в скрытом состоянии h
* num\_layers — количество повторяющихся слоев. Например, установка num\_layers=2 будет означать объединение двух GRU вместе для формирования stacked GRU GRU , при этом второй GRU принимает выходные данные первого GRU и вычисляет окончательные результаты. По умолчанию: 1
* смещение — если False , то слой не использует веса смещения b\_ih и b\_hh . По умолчанию: True
* batch\_first — если True , то тензоры ввода и вывода предоставляются как (batch, seq, feature). По умолчанию: False
* отсев – если он не равен нулю, вводит слой Dropout на выходах каждого уровня GRU, кроме последнего слоя, с вероятностью отсева, равной dropout . По умолчанию: 0
* двунаправленный — если True , становится двунаправленным GRU. По умолчанию: False

Если выполняются следующие условия: 1) cudnn включен, 2) входные данные находятся на GPU 3) входные данные имеют тип torch.float16 4) используется графический процессор V100, 5) входные данные не в формате PackedSequence , постоянный алгоритм может быть выбран для повышения производительности.



Torch LSTM и torch GRU - это два разных типа рекуррентных нейронных сетей (RNN), которые используются в глубоком обучении. Обе эти сети основаны на архитектурах Long Short-Term Memory (LSTM) и Gated Recurrent Unit (GRU) соответственно.

Torch LSTM - это тип RNN, который предназначен для запоминания долгосрочных зависимостей в данных. Он состоит из ряда ячеек памяти, которые соединены друг с другом и могут хранить информацию в течение длительных периодов времени. Torch LSTM способен изучать сложные шаблоны в данных и может использоваться для таких задач, как языковое моделирование, машинный перевод и создание подписей к изображениям.

Torch GRU - это тип RNN, который предназначен для захвата краткосрочных зависимостей в данных. Он состоит из серии вентилей, которые управляют потоком информации между ячейками памяти. Torch GRU способен изучать сложные закономерности в данных и может использоваться для таких задач, как распознавание речи, прогнозирование временных рядов и обработка естественного языка.

Как torch LSTM, так и torch GRU являются мощными инструментами для глубокого обучения и могут использоваться для решения самых разных задач. Torch LSTM лучше подходит для задач, требующих долговременной памяти, в то время как torch GRU лучше подходит для задач, требующих кратковременной памяти. В зависимости от задачи тот или иной вариант может оказаться более подходящим.

# 25. Продвинутые RNN-архитектуры: BiLSTM, улучшенные seq2seq модели машинного перевода на основе RNN.

**BiLSTM (Двунаправленная долгая краткосрочная память)**

* **Основная идея**: BiLSTM состоит из двух LSTM-слоёв, работающих в противоположных направлениях. Один слой обрабатывает данные от начала к концу, а другой — наоборот. Это позволяет улавливать зависимости как в прошлом, так и в будущем контексте.
* **Применение**: Это особенно полезно в задачах, где контекст имеет значение для понимания каждого элемента в последовательности, например, в машинном переводе, распознавании речи и анализе настроений.

**Улучшенные Seq2Seq модели на основе RNN**

* **Seq2Seq (Sequence-to-Sequence)**: Это модели, которые преобразуют последовательность входных данных в последовательность выходных данных. Они идеально подходят для машинного перевода, где необходимо перевести предложение с одного языка на другой.
* **Улучшения**:
  + **Внимание (Attention Mechanism)**: Механизм внимания позволяет модели фокусироваться на определенных частях входной последовательности при генерации каждого слова выходной последовательности, улучшая качество перевода.
  + **Энкодер-декодер с множественными слоями**: Использование множественных слоев в энкодере и декодере улучшает способность модели обрабатывать сложные языковые структуры.
  + **Резидуальное обучение (Residual Learning)**: Добавление резидуальных соединений между слоями может помочь в обучении глубоких сетей, улучшая производительность и ускоряя обучение.

## Механизм внимания и модуль Transformer и BERT

# 26. Механизм Attention. Пример использования Attentinon.

Механизм внимания (attention). Изначально механизм внимания был представлен в контексте рекуррентных Seq2seq сетей для "обращения внимания" блоков декодеров на скрытые состояния RNN для любой итерации энкодера, а не только последней.

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 состянию 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.

Приемы, направленные на преодоление проблемы :

* Для языков, в которых последовательность слов в эквивалентных передложениях примерно одинакова, может быть полезно декодеру гененрировать предложение в обратном порядке. Это дает возможность иметь небольшую последовательность преобразований ("небольшое расстояние" ) от энкодинга слов (последних слов в исходном предложении) до места их декодирования.
* Может быть полезно скрытое состояние энкодера на последнем шаге передавать на вход каждому блоку декодера, чтобы каждый декодер имел прямой доступ к энкодингу всего предложения (этот подход тоже сокращает сложность передачи информации об исходном предложение "сквозь" последовательность в декодере)

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 ℎ𝑚 , где 𝑚 - 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 ℎ𝑡 для любого 𝑡∈1…𝑚 .

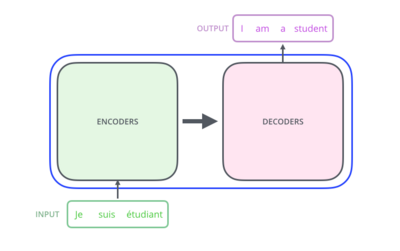


При помощи механизма внимания достигается "фокусирование" декодера на определенных скрытых состояниях энкодера, на любом из шагов энкодера.

# 27. Модуль Transformer: общая архитектура Transformer, работа self-attention и multi-head attention.

#микс материалов прошлых лет и chatgpt, ну и инета

Архитектура трансформера

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:TransformerSimpleArchitecture.png)

Архитектура трансформера

Устройство трансформера состоит из кодирующего и декодирующего компонентов. На вход принимается некая последовательность, создается ее векторное представление (англ. *embedding*), прибавляется вектор позиционного кодирования, после чего набор элементов без учета порядка в последовательности поступает в кодирующий компонент (параллельная обработка), а затем декодирующий компонент получает на вход часть этой последовательности и выход кодирующего. В результате получается новая выходная последовательность.

Внутри кодирующего и декодирующего компонента нет рекуррентности. Кодирующий компонент состоит из кодировщиков, которые повторяются несколько раз, аналогично устроен декодирующий компонент. Трансформер — это поставленные друг за другом модели внимания, которые позволяют исходную последовательность векторов перевести в новую последовательность векторов, которые кодируют информацию о контексте каждого элемента. Трансформер-кодировщик переводит исходные векторы в скрытые, которые правильно сохраняют в себе информацию о контексте каждого элемента. Далее трансформер-декодировщик декодирует результат кодировщика в новую последовательность, которая состоит из эмбедингов элементов выходного языка. После по эмбедингам генерируются сами итоговые элементы с помощью вероятностной языковой модели.

Рассматриваем задачу машинного перевода Seq2seq. Новая архитектура для решения этой задачи, которая не является ни RNN, ни CNN.

**Self-Attention**

Механизм self-attention позволяет модели оценивать взаимосвязь между всеми словами во входной последовательности. Это достигается путём создания трёх векторов для каждого слова: ключа (key), значения (value) и запроса (query). Self-attention вычисляет взаимодействие каждого слова с каждым, что позволяет модели улавливать контекст и зависимости в данных.

Отличается применением слоя Multi-head attention.

Multi-head attention - слой, который дает возможность каждому входному вектору (эмбеддингу слова) взаимодействовать с любыми другими словами предложения через attention mechanism, вместо:

* последовательной передачи hidden state, как в RNN
* свертки эмбеддингов соседних слов, как в CNN

Изображение выглядит как текст

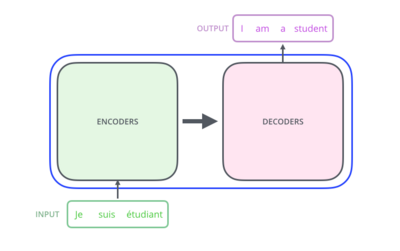
Автоматически созданное описание



# 28. Модуль Transformer: общая архитектура Transformer, специфика работы декодера и его увязки с энкодером, позиционное кодирование в Transformer.

#микс материалов прошлых лет и chatgpt, ну и инета

Архитектура трансформера

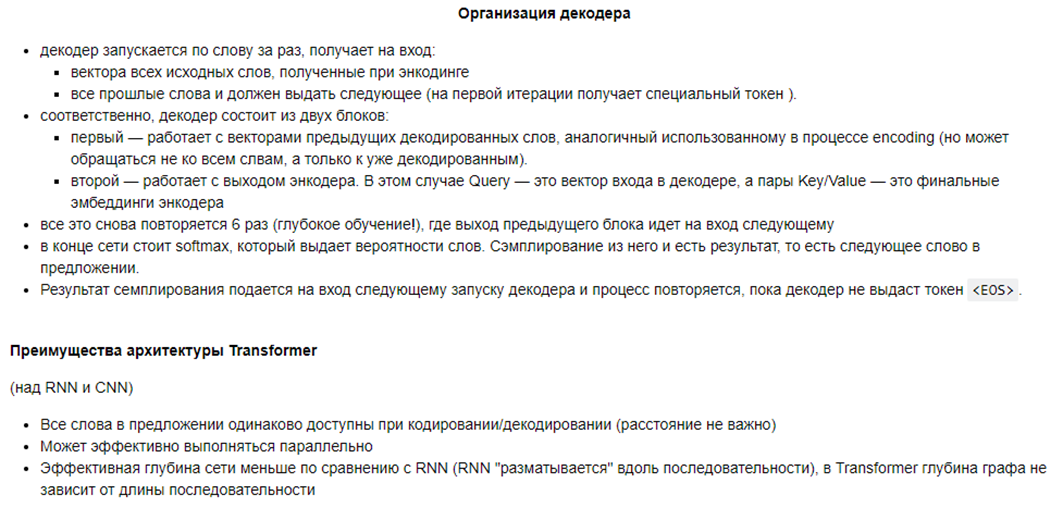
[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:TransformerSimpleArchitecture.png)

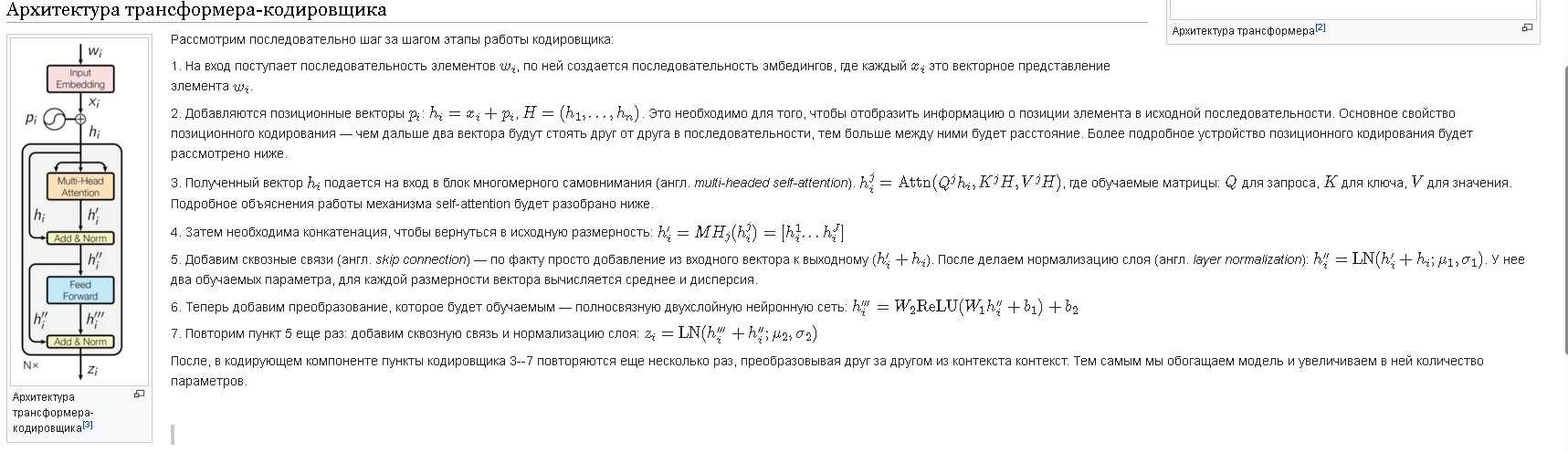
Архитектура трансформера

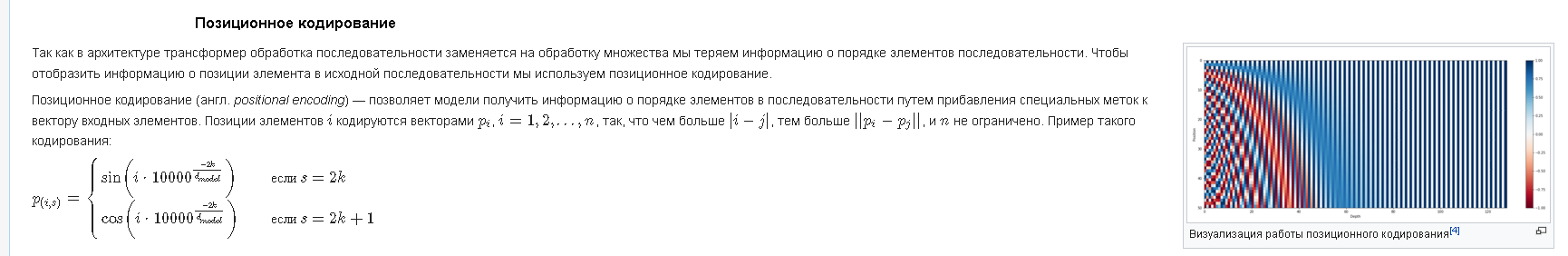
Устройство трансформера состоит из кодирующего и декодирующего компонентов. На вход принимается некая последовательность, создается ее векторное представление (англ. *embedding*), прибавляется вектор позиционного кодирования, после чего набор элементов без учета порядка в последовательности поступает в кодирующий компонент (параллельная обработка), а затем декодирующий компонент получает на вход часть этой последовательности и выход кодирующего. В результате получается новая выходная последовательность.

Внутри кодирующего и декодирующего компонента нет рекуррентности. Кодирующий компонент состоит из кодировщиков, которые повторяются несколько раз, аналогично устроен декодирующий компонент. Трансформер — это поставленные друг за другом модели внимания, которые позволяют исходную последовательность векторов перевести в новую последовательность векторов, которые кодируют информацию о контексте каждого элемента. Трансформер-кодировщик переводит исходные векторы в скрытые, которые правильно сохраняют в себе информацию о контексте каждого элемента. Далее трансформер-декодировщик декодирует результат кодировщика в новую последовательность, которая состоит из эмбедингов элементов выходного языка. После по эмбедингам генерируются сами итоговые элементы с помощью вероятностной языковой модели.

Рассматриваем задачу машинного перевода Seq2seq. Новая архитектура для решения этой задачи, которая не является ни RNN, ни CNN.



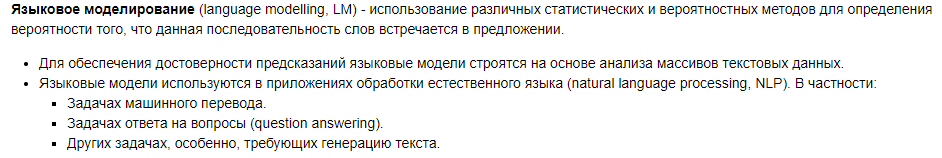




# 29. Модель BERT.

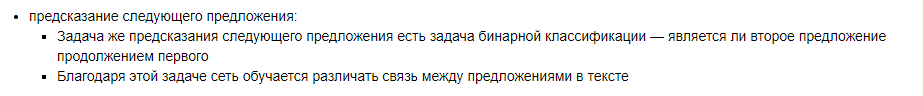
Bidirectional Encoder Representations from Transformers (BERT) - языковая модель, основанная на архитектуре трансформер, предназначенная для предобучения языковых представлений с целью их последующего применения в широком спектре задач обработки естественного языка. BERT был создан и опубликован в 2018 году Якобом Девлином и его коллегами из Google.

* BERT является автокодировщиком
* BERT использует трансформер-архитектуру
* В каждом слое кодировщика применяется двустороннее внимание, что позволяет учитывать контекст с обеих сторон от рассматриваемого токена

 Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

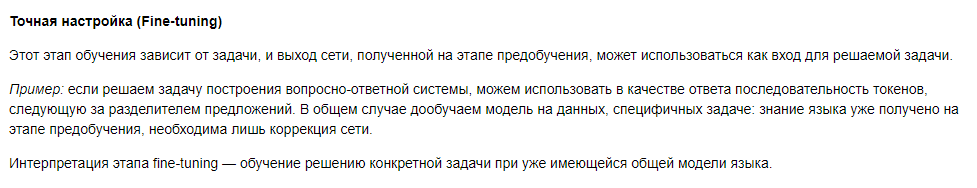
Автоматически созданное описание 

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание



## Автоэнкодеры и генеративные модели

# 30. Векторная интерпретация весов скрытого слоя многослойного перцептрона. Задача восстановления входного сигнала после сжатия в скрытом слое ИНС.

Многослойный перцептрон (МП) - это тип прямой нейронной сети, состоящей из нескольких слоев. В каждом слое присутствуют нейроны, которые принимают взвешенные суммы входных сигналов и пропускают их через нелинейную функцию активации.

1. **Сжатие входного сигнала в скрытом слое**: В контексте задачи восстановления входного сигнала, скрытый слой ИНС часто используется для сжатия информации. Это означает, что входные данные представляются в более компактной форме. Например, если входной слой имеет 100 нейронов, а скрытый слой - только 10, то сеть пытается закодировать информацию с 100 измерений в 10.
2. **Векторная интерпретация весов**: Веса в скрытом слое можно интерпретировать как векторы в пространстве входных данных. Каждый нейрон скрытого слоя связан со всеми входными нейронами, и веса этих связей формируют вектор, который представляет, как каждый нейрон скрытого слоя реагирует на различные аспекты входных данных. Эти векторы могут быть поняты как представление ключевых характеристик входных данных.
3. **Восстановление входного сигнала**: Задача восстановления заключается в том, чтобы с помощью выходного слоя сети воссоздать исходный входной сигнал на основе сжатого представления, полученного в скрытом слое. Это требует, чтобы сеть не только сжимала данные, но и училась эффективно восстанавливать исходные данные из сжатого представления.
4. **Обучение**: В процессе обучения сеть настраивает веса таким образом, чтобы минимизировать ошибку между восстановленным и исходным сигналом. Таким образом, веса скрытого слоя адаптируются для эффективного кодирования важных особенностей входных данных.

# 31. Автоэнкодеры. Линейный автоэнкодер и его аналог в статистических методах. Нелинейные и глубокие автоэнкодеры. Области значений латентного пространства. Примеры интерполяции и экстраполяции на многообразиях области значений латентных пространств автоэнкодеров.

Автоэнкодеры — это тип нейронных сетей, используемых для задач необучаемого (или полуобучаемого) обучения. Они предназначены для автоматического изучения эффективных представлений (кодировок) для наборов данных, обычно в целях снижения размерности или генерации данных.

основные компоненты и принципы работы автоэнкодеров:

1. **Кодировщик (Encoder)**: Эта часть автоэнкодера преобразует входные данные в более компактное, сжатое представление, которое называется латентным пространством или латентными переменными. В процессе кодирования данные с высокой размерностью сжимаются в данные с меньшей размерностью.
2. **Латентное пространство (Latent Space)**: Это представление данных в сжатом виде. Оно служит «мостом» между кодировщиком и декодировщиком. Латентное пространство захватывает ключевые характеристики и структуры исходных данных.
3. **Декодировщик (Decoder)**: Декодировщик выполняет обратное преобразование латентного представления обратно в представление, похожее на исходные данные. Цель декодировщика — восстановить данные до их первоначального состояния (или как можно ближе к нему).
4. **Линейный автоэнкодер и его аналог в статистических методах**: Линейный автоэнкодер – это особый вид нейронной сети, предназначенный для уменьшения размерности данных. В линейном автоэнкодере и кодировщик, и декодировщик состоят из линейных преобразований (умножение на матрицу весов). Это схоже с методом главных компонент (PCA) в статистике, который также используется для снижения размерности и основан на линейных преобразованиях.
5. **Нелинейные и глубокие автоэнкодеры**: Нелинейные автоэнкодеры отличаются от линейных использованием нелинейных преобразований (например, сигмоиды, ReLU). Это позволяет им моделировать более сложные структуры в данных. Глубокие автоэнкодеры содержат много слоев как в кодировщике, так и в декодировщике, что позволяет им извлекать более абстрактные представления данных на разных уровнях.
6. **Области значений латентного пространства**: Латентное пространство в контексте автоэнкодеров – это средний слой сети, куда кодировщик сжимает входные данные. Величина и структура этого пространства существенно влияют на способность автоэнкодера изучать эффективные представления данных. В идеале, это пространство должно отражать основные факторы вариативности данных.
7. **Примеры интерполяции и экстраполяции на многообразиях области значений латентных пространств автоэнкодеров**: Интерполяция в латентном пространстве автоэнкодера подразумевает выбор двух точек в этом пространстве и вычисление промежуточных точек между ними. Экстраполяция – это продолжение этого пути за пределы исходных точек. Эти процессы могут демонстрировать, как автоэнкодер изучает непрерывные и плавные представления данных, позволяя, например, плавно переходить от одного изображения к другому или от одного стиля представления к другому.

# 32. Задачи дискриминативных и генеративных моделей, сравнение моделей. Классификация генеративных моделей. Прикладные задачи для генеративных моделей в области компьютерного зрения.

1. **Задачи дискриминативных моделей**: Дискриминативные модели в машинном обучении используются для предсказания метки или категории объекта на основе его характеристик. Примеры включают классификацию изображений, определение спама в электронной почте и диагностику болезней по медицинским данным.
2. **Задачи генеративных моделей**: Генеративные модели обучаются создавать данные, похожие на те, что они видели во время обучения. Примеры задач включают создание реалистичных изображений, текстов, голоса, видео и музыки. Они также используются в задачах увеличения объема данных (data augmentation) и создании реалистичных сценариев для тренировок в виртуальной реальности.
3. **Сравнение моделей**: Главное отличие между дискриминативными и генеративными моделями заключается в их целях. Дискриминативные модели нацелены на различение между разными категориями данных, в то время как генеративные модели стремятся создавать новые данные, похожие на те, что им были даны на обучение. Дискриминативные модели обычно более эффективны в задачах классификации и предсказаний, тогда как генеративные модели лучше подходят для задач, связанных с созданием нового контента.
4. **Классификация генеративных моделей**: Генеративные модели можно классифицировать на основе их подхода к обучению и созданию данных. Некоторые популярные типы включают Генеративно-состязательные сети (GAN), Вариационные автокодировщики (VAE) и Генеративные рекурсивные сети (RNN). Каждый тип имеет свои уникальные свойства и применения в зависимости от типа задачи и характера данных.
5. **Прикладные задачи для генеративных моделей в области компьютерного зрения**: В компьютерном зрении генеративные модели используются для широкого спектра задач. Это включает генерацию реалистичных изображений и видео, создание 3D-моделей из 2D-изображений, стилизацию изображений, улучшение качества изображений и видео, и даже создание виртуальных сред для обучения искусственного интеллекта.

# 33. Архитектура GAN: описание общей архитектуры, модели обучения и архитектур глубоких моделей в GAN. Принцип подхода к обучению в GAN на примере обучения одномерной функции распределения.

Архитектура GAN, или Generative Adversarial Networks, включает две основные сети: генеративную сеть (генератор) и дискриминативную сеть (дискриминатор).

1. **Общая архитектура**:
   * Генератор создает данные, пытаясь имитировать настоящие данные.
   * Дискриминатор оценивает данные, пытаясь отличить поддельные данные от настоящих.
   * Эти две сети обучаются одновременно в процессе, напоминающем игру, где каждая сеть пытается обмануть другую.
2. **Модели обучения**:
   * Во время обучения генератор получает случайный шум и преобразует его в данные.
   * Дискриминатор затем оценивает, являются ли эти данные реальными или сгенерированными.
   * Целью генератора является обмануть дискриминатор, заставляя его принять поддельные данные за настоящие.
   * Целью дискриминатора является правильное различение поддельных и настоящих данных.
3. **Архитектуры глубоких моделей в GAN**:
   * Генераторы и дискриминаторы могут быть построены с использованием различных архитектур глубокого обучения, таких как сверточные нейронные сети (CNN) для изображений или рекуррентные нейронные сети (RNN) для последовательных данных.
   * Выбор архитектуры зависит от типа данных и задачи.
4. **Принцип подхода к обучению в GAN на примере одномерной функции распределения**:
   * Предположим, у нас есть одномерная функция распределения, которую мы хотим научиться генерировать.
   * Генератор начинает с создания данных, основанных на случайном шуме, и пытается адаптировать свои выходные данные, чтобы они походили на данные из реального распределения.
   * Дискриминатор оценивает, насколько хорошо данные генератора соответствуют реальным данным.
   * Обе сети обучаются одновременно: генератор улучшает свою способность имитировать реальные данные, а дискриминатор становится лучше в различении настоящих и поддельных данных.
   * В итоге, при успешном обучении, генератор сможет точно воспроизводить одномерное распределение, а дискриминатор будет с трудом различать реальные и сгенерированные данные.

# 34. Модель вариационного автоэнкодера (VAE): общие и специфические цели модели, общая архитектура модели и баейсовское моделирование. Функция потерь VAE и специфика получаемых скрытых представлений.

1. **Общие и специфические цели модели VAE**:
   * **Общие цели**: Основная цель VAE - генерировать новые данные, которые похожи на данные из обучающего набора. Это достигается путем изучения способа представления входных данных в сжатом виде (скрытые представления) и последующего их восстановления.
   * **Специфические цели**: VAE находит широкое применение в задачах, связанных с синтезом изображений, стилизацией, обучением без учителя, сжатием данных, и даже в усовершенствовании других моделей глубокого обучения за счет предобучения.
2. **Общая архитектура модели VAE**:
   * VAE состоит из двух основных компонентов: кодировщика (encoder) и декодировщика (decoder).
   * **Кодировщик** преобразует входные данные в распределение в скрытом пространстве (обычно гауссовское), определяя такие параметры, как среднее значение и стандартное отклонение.
   * **Декодировщик** затем использует выборки из этого распределения для генерации данных, максимально приближенных к исходным.
3. **Байесовское моделирование в VAE**:
   * VAE использует принципы байесовской статистики для моделирования. Он предполагает, что данные генерируются из некоторого скрытого (латентного) распределения, и пытается аппроксимировать это распределение.
   * Процесс генерации данных в VAE можно рассматривать как сэмплирование из апостериорного распределения скрытых переменных, что является ключевым аспектом байесовского подхода.
4. **Функция потерь VAE**:
   * Функция потерь в VAE состоит из двух частей: реконструктивной потери и потери KL-дивергенции.
   * **Реконструктивная потеря** измеряет, насколько хорошо декодировщик восстанавливает входные данные из скрытых представлений.
   * **Потеря KL-дивергенции** измеряет, насколько распределение скрытых представлений отклоняется от стандартного нормального распределения, что способствует регуляризации модели.
5. **Специфика получаемых скрытых представлений**:
   * Скрытые представления в VAE не являются точными значениями, а представляют собой параметры распределения (например, среднее и дисперсия для гауссовского распределения).
   * Это дает модели возможность генерировать разнообразные выходные данные, даже если на вход подается одно и то же изображение, благодаря случайности, вносимой в процесс выборки из распределения скрытых переменных.

# 35. Модель вариационного автоэнкодера (VAE): баейсовское моделирование, функция ошибки на основе метода максимального правдоподобия, причины построения ELBO и метод построения этой оценки.

1. **Байесовское моделирование в контексте VAE**: Вариационные автоэнкодеры используют принципы байесовского моделирования для оценки сложных распределений данных. Основная идея заключается в том, чтобы аппроксимировать истинное распределение данных с помощью более простого распределения, параметры которого можно оптимизировать. В контексте VAE это означает использование вариационного вывода для приближения апостериорного распределения скрытых переменных, данными наблюдения.
2. **Функция ошибки на основе метода максимального правдоподобия**: Функция ошибки в VAE, также известная как функция потерь, обычно включает два компонента: один, связанный с реконструкцией данных (часто используя максимальное правдоподобие), и другой, связанный с регуляризацией распределения скрытых переменных. Часть, связанная с максимальным правдоподобием, обычно измеряет, насколько хорошо модель восстанавливает входные данные, что соответствует уменьшению разницы между исходными данными и их реконструкцией.
3. **Причины построения ELBO (Evidence Lower BOund)**: ELBO является ключевым компонентом вариационных автоэнкодеров. Основная цель ELBO - предоставить нижнюю границу для логарифма маргинального правдоподобия данных. Эта граница используется, потому что прямой расчет маргинального правдоподобия часто бывает вычислительно неосуществим. ELBO помогает сбалансировать между точностью реконструкции данных и регуляризацией распределения скрытых переменных.
4. **Метод построения оценки ELBO**: Оценка ELBO строится путем введения вариационного распределения, которое приближает истинное апостериорное распределение скрытых переменных. Затем оптимизируется разница между этим вариационным распределением и апостериорным распределением посредством минимизации дивергенции Кульбака-Лейблера. Это приводит к увеличению ELBO, что, в свою очередь, улучшает приближение к истинному распределению данных.

# 36. Denoising diffusion models: общий принцип работы, описание прямого процесса и основные принципы описания обратного процесса.

Денойзинговые диффузионные модели (denoising diffusion models) представляют собой класс генеративных моделей, использующих идеи из статистической физики для создания новых данных (например, изображений). Рассмотрим их принципы работы, прямой процесс и обратный процесс.

**Общий Принцип Работы**

1. **Исходные данные:** Диффузионные модели начинают работу с исходного распределения данных (например, набора изображений).
2. **Прямой процесс (диффузия):** Это пошаговое добавление шума к исходным данным, что превращает данные в случайный шум.
3. **Обратный процесс (денойзинг):** На этом этапе модель учится постепенно удалять шум, восстанавливая из шумового состояния исходные данные.

**Прямой Процесс (Диффузия)**

1. **Шум:** На каждом шаге к данным добавляется гауссовский шум. Этот процесс повторяется много раз, постепенно увеличивая степень шума.
2. **Пошаговость:** Прямой процесс состоит из множества маленьких шагов. Чем больше шагов, тем более плавно данные переходят от исходных к полностью зашумленным.
3. **Контроль шума:** В каждый момент времени известно, насколько шумовое состояние отличается от исходных данных, что позволяет точно контролировать процесс.

**Основные Принципы Описания Обратного Процесса (Денойзинг)**

1. **Обучение на данных:** Модель обучается восстанавливать исходные данные из зашумленных версий, учась на разных степенях шума.
2. **Постепенное уменьшение шума:** Обратный процесс начинается с полностью зашумленных данных и постепенно уменьшает шум, восстанавливая исходное изображение.
3. **Итеративность:** Как и в прямом процессе, обратный процесс также итеративный, состоящий из множества шагов.
4. **Предсказание шума:** Вместо того чтобы напрямую восстанавливать исходные данные, модель часто учится предсказывать, какой шум был добавлен, и затем удалять его.
5. **Использование условной генерации:** Модель может быть обучена генерировать данные, удовлетворяющие определенным условиям, что делает её мощным инструментом для создания целенаправленных изображений или данных.