# 

[**Основы глубокого обучения**](#_heading=h.yw3l4s8qw0r7) **3**

[**1. Модель перцептрона. Проблема линейно неразделимых множеств и ее решение. Логика построения многослойных ИНС.**](#_heading=h.456wwced7vsb) **3**

[2. Функции активации. Требования к функциям активации. Популярные функции активации.](#_heading=h.ao8qvx5i54nb) 6

[3. Глубокое обучение. «Вторая весна искусственного интеллекта» и ее причины.](#_heading=h.vo5a5dubrmvx) 7

[4. Линейное отображение. Векторно-матричное дифференцирование.](#_heading=h.mgczgu87r6ed) 10

[5. Проблема поиска градиента в общей логике обучения нейронной сети. Градиент функции многих переменных. Методы вычисления.](#_heading=h.g1bpgm3mncjt) 12

[6. Кросс-валидация. Выборки train, validation, test. Проблема переобучения. Ранняя остановка.](#_heading=h.wesedukjr7qv) 15

[**7. Преобразование Softmax и функция потерь Cross Entropy loss.**](#_heading=h.dnlzfv46m1eg) **16**

[**8. Механизм обратного распространения ошибки.**](#_heading=h.y5odtyx7wigd) **17**

[9. Дифференцируемое программирование и реализация обратного распространения ошибки.](#_heading=h.hh1bdnbnbrxz) 18

[10. Стохастический градиентный спуск. Батчи обучающей выборки.](#_heading=h.e6x5c2qzons5) 20

[11. Адаптивные методы градиентного спуска. Метод импульсов. Метод Нестерова.](#_heading=h.eqd40qwu0vsm) 21

[12. Проблема инициализации весов при обучении ИНС. Инициализация Ксавье.](#_heading=h.nfdayt8gomq1) 24

[13. Гиперпараметры. Скорость обучения и размер батча.](#_heading=h.v3t96xn90tpl) 26

[14. Переобучение модели и регуляризация. Dropout.](#_heading=h.cmqu1pyd6iks) 27

[15. Минбатчи – причина использования. Нормализация по мини-батчам.](#_heading=h.bt635ad6bz93) 29

[16. Многослойные сети. Граф потока вычислений.](#_heading=h.6vzj2uxtroqp) 30

[**Специфические задачи глубокого обучения**](#_heading=h.s2305y5d2s4t) **32**

[**17. Специфика задач машинного обучения на изображениях. Принцип работы сверточных сетей. Преимущества сверточных сетей при решении этих задач.**](#_heading=h.rg3j41zf9jp) **32**

[18. Архитектура многослойной ИНС распознавания изображений на основе сверточных сетей.](#_heading=h.y76pz2o8pbr) 36

[19. Приемы для глубокого обучения на небольших наборах изображений.](#_heading=h.2nvo6dofose5) 38

[20. Схема работ слоя сверточной сети. Пулинг. Гиперпараметры: padding, kernel size, stride, dilation.](#_heading=h.prv4ptqrqbdc) 39

[21. Задачи обработки текста: дистрибутивная семантика, матрица совместной встречаемости, представление слов в виде векторов малой размерности.](#_heading=h.1h732o4xxhq9) 41

[22. Word2vec: модель CBOW.](#_heading=h.j9l8xsxu3jpx) 44

[23. Word2vec: модель Skip-Gram.](#_heading=h.xlmd5w8hzuxz) 45

[24. Рекуррентная нейронная сеть, принципы ее обучения. Сложности применения рекуррентных нейронных сетей.](#_heading=h.l06jgrtjot9z) 46

[25. Модуль LSTM.](#_heading=h.tuy50d5x9tkh) 48

[26. Механизм Attention. Пример использования Attentinon.](#_heading=h.souhbxi9x1gf) 51

[27. Архитектура Transformer.](#_heading=h.nsvn84imkz1n) 53

[28. Модель BERT.](#_heading=h.u44z641v7b2) 54

[**PyTorch и задачи глубокого обучения**](#_heading=h.1yk80o773kxo) **56**

[**29. Класс Tensor. Операции, изменяющие размер тензора. Операции агрегации.**](#_heading=h.apt7oz88i1zc) **56**

[30. Принципиальная логика обучения нейронной сети.](#_heading=h.1fwv2n66rgik) 59

[31. Автоматическое дифференцирование в PyTorch. Пример и применение в обучении ИНС.](#_heading=h.of8o4qb3g9d7) 61

[32. Загрузка и преобразование данных. Классы Dataset, DataLoader, Transforms (и композиция трансформеров).](#_heading=h.hakwai67nobj) 63

[33. Класс nn.Module. Назначение. Основные поля, методы.](#_heading=h.2wyr3g1pwvmw) 65

[34. Линейные слои (Linear Layers).](#_heading=h.bbf6vgdhvd0y) 68

[35. Слои нелинейной активации (Non Linear Activations).](#_heading=h.cwul2fvcsik4) 70

[36. Слои нормализации (Normalization Layers).](#_heading=h.sp3rrqftihid) 73

[37. Слои регуляризации (Dropout Layers).](#_heading=h.8srea9p8f0p) 75

[38. Сверточные слои (Convolution Layers). Сжимающие слои (Pooling Layers).](#_heading=h.oizi6p7pukz) 77

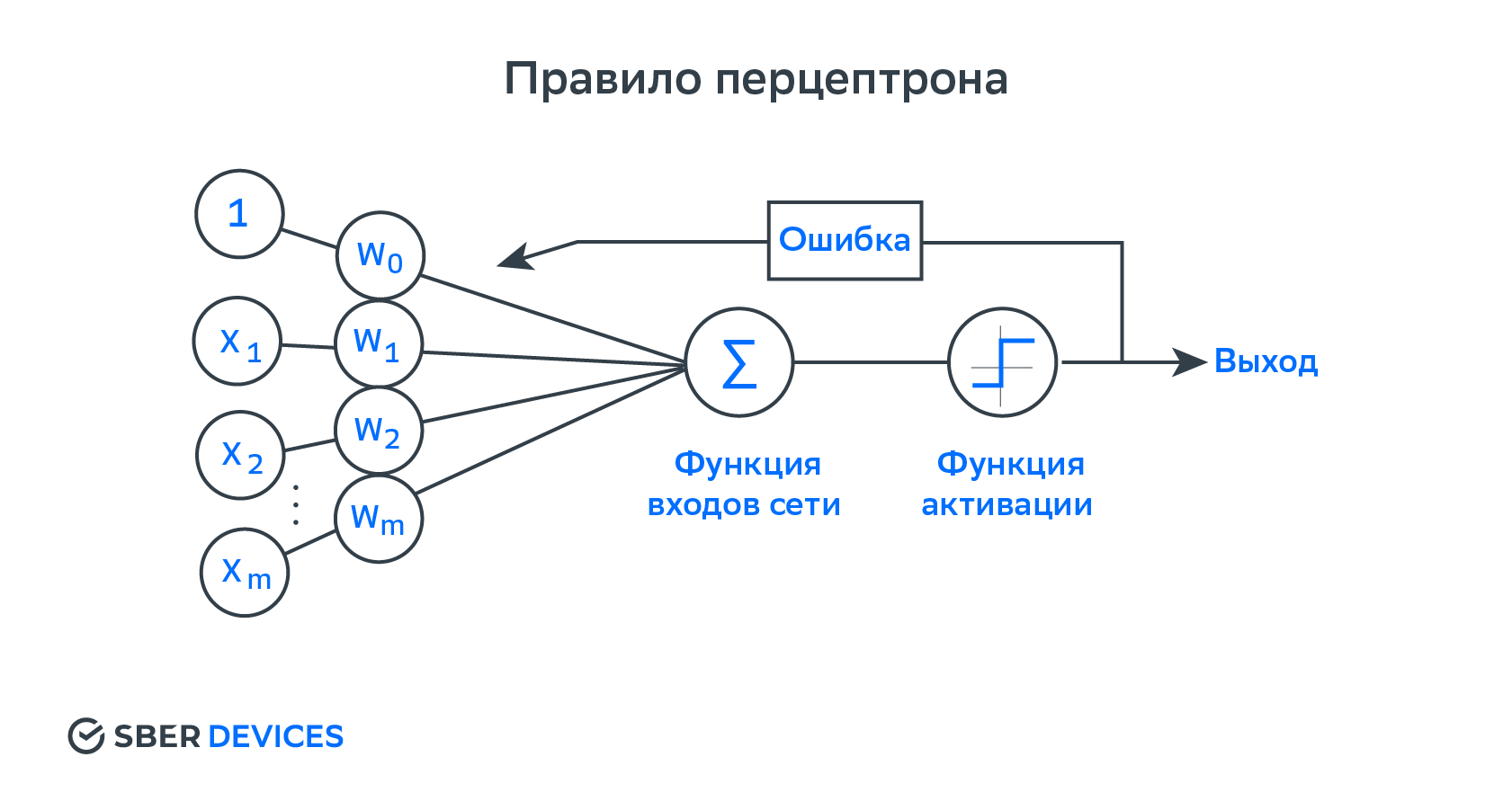
[39. Слои функций потерь (Loss Functions).](#_heading=h.40nus7iq564i) 80

[40. Слои эмбеддингов nn.Embedding и их применение.](#_heading=h.zgr53cqnu5ld) 82

[41. Класс torch.nn.LSTM и torch.nn.GRU.](#_heading=h.uehlclru8wf5) 84

## Основы глубокого обучения

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



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

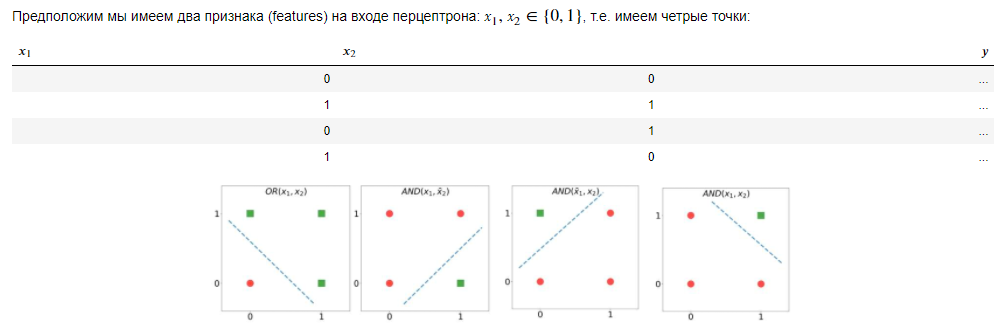
Каждый искусственный нейрон обычно достаточно просто устроен и является узлом в искусственной нейронной сети. В сети он имеет дело только с сигналами, которые он периодически получает, и сигналами, которые он периодически посылает другим нейронам. Несмотря на простоту устройства искусственные нейроны 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.

### 

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

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

### 

### 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)

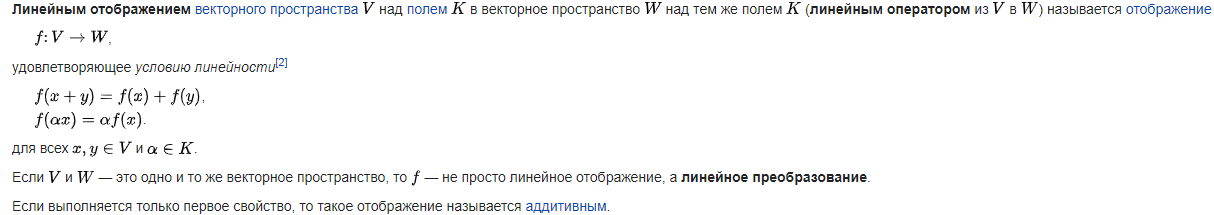
В последнее время были открыты еще более совершенные способы распространения градиента, с применением которых появилась возможность обучать с нуля модели с тысячами слоев в глубину. В частности:

* 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

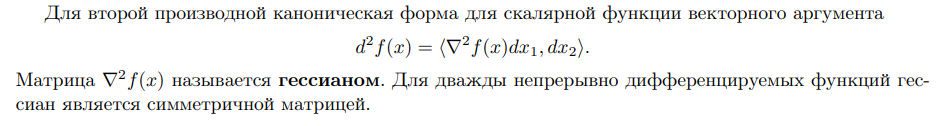
Автоматически созданное описание

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.

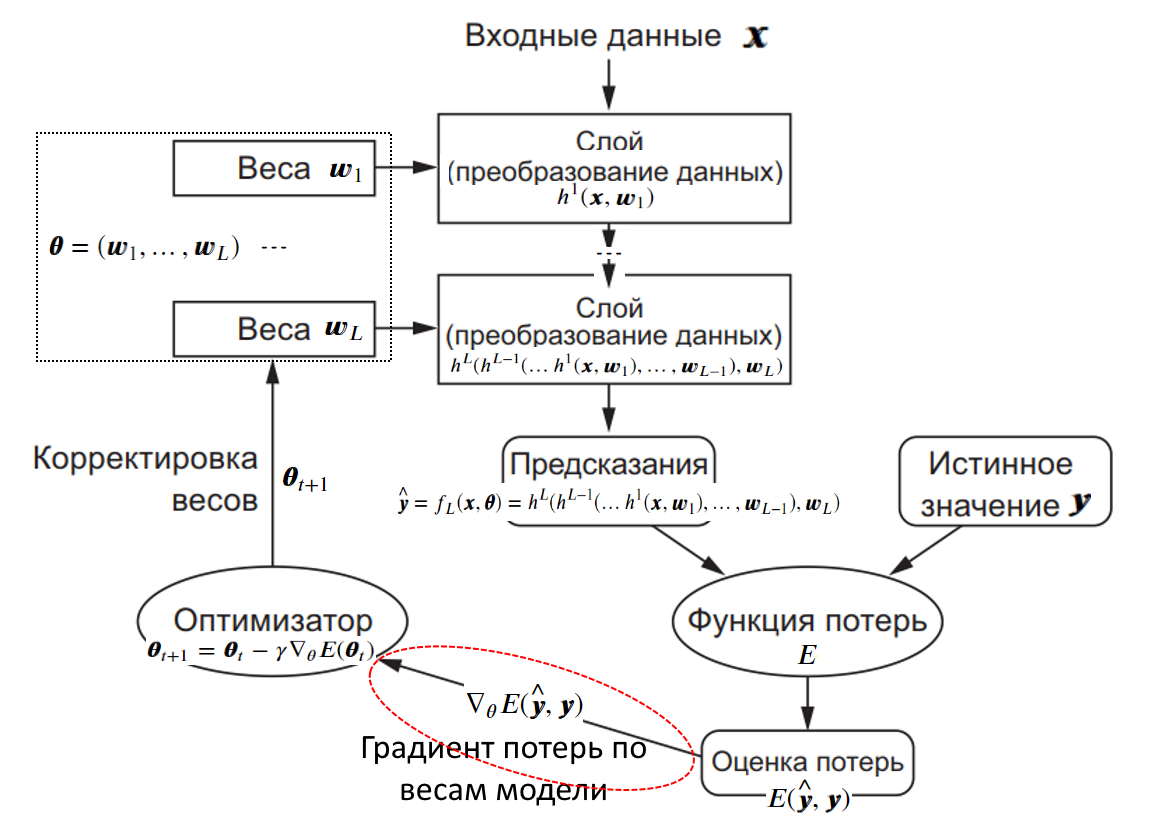
В нейронных сетях градиенты часто используются для обучения с помощью градиентного спуска. В этом случае градиенты используются для определения, в какую сторону нужно изменить веса нейронной сети, чтобы уменьшить значение функции потерь.

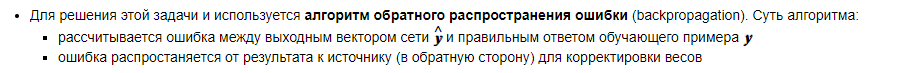
Проблема поиска градиента в общей логике обучения нейронной сети состоит в том, что нейронные сети обычно состоят из множества слоев, каждый из которых содержит множество нейронов. Это делает функцию потерь, которую мы хотим минимизировать, очень сложной и имеет множество локальных минимумов. Поэтому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

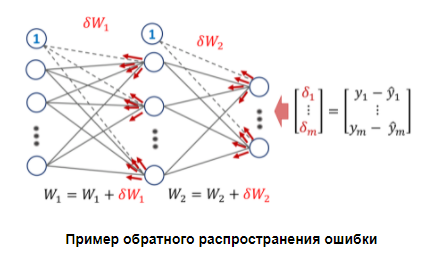
Градиент функции многих переменных – это вектор, состоящий из частных производных функции по каждой из ее переменных. Например, если у нас есть функция f(x, y) от двух переменных x и y, то градиент этой функции будет выглядеть так: *[∂f/∂x, ∂f/∂y]*

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)* ?

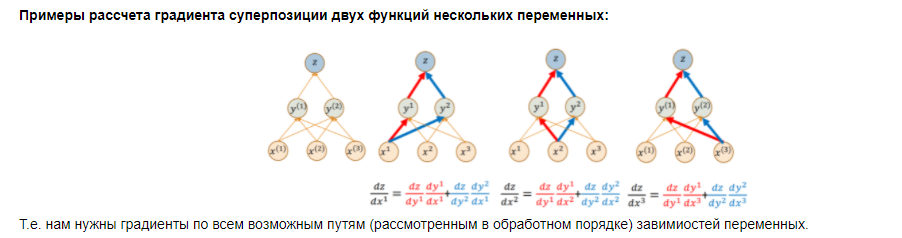






Изображение выглядит как текст

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.

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

### 

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

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.

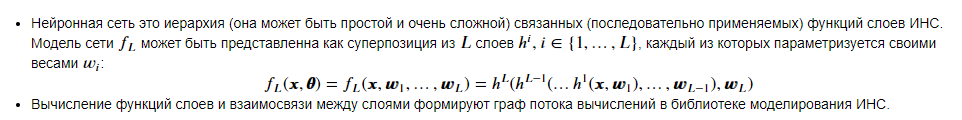
Как реализовать алгоритм обратного распространения ошибки удобно для использования в задачах моделирования ИНС?

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

### 

### 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.

### 

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

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.

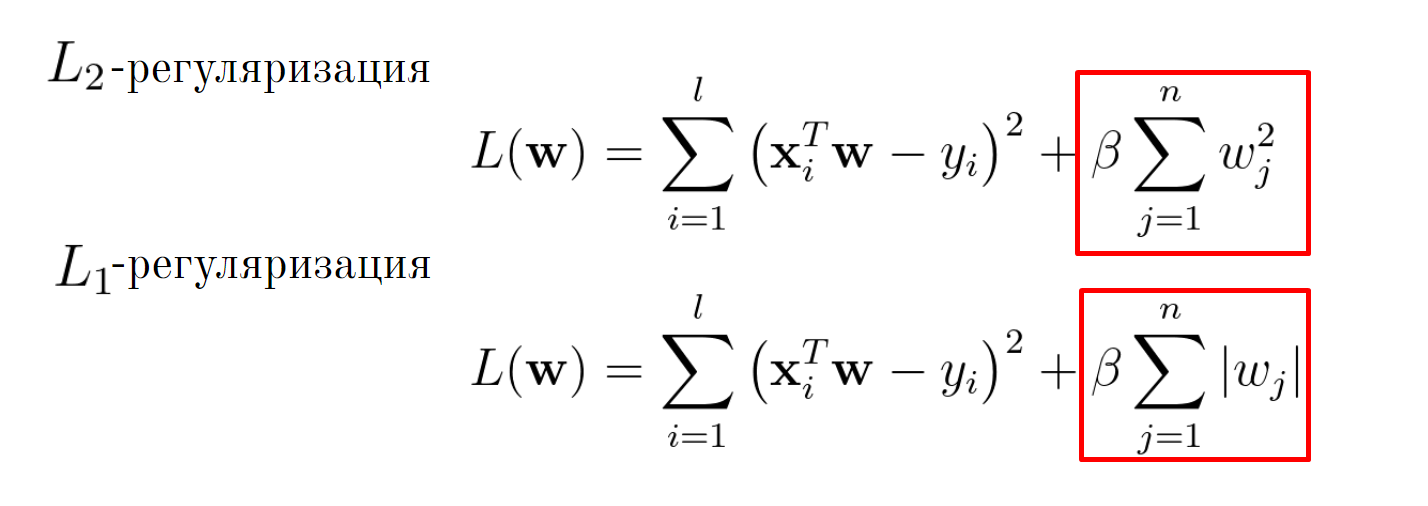
### 

### 14. Переобучение модели и регуляризация. Dropout.

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.

### 

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

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

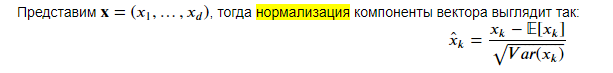
Сопутствующая проблема: "насыщение" функций активации

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:

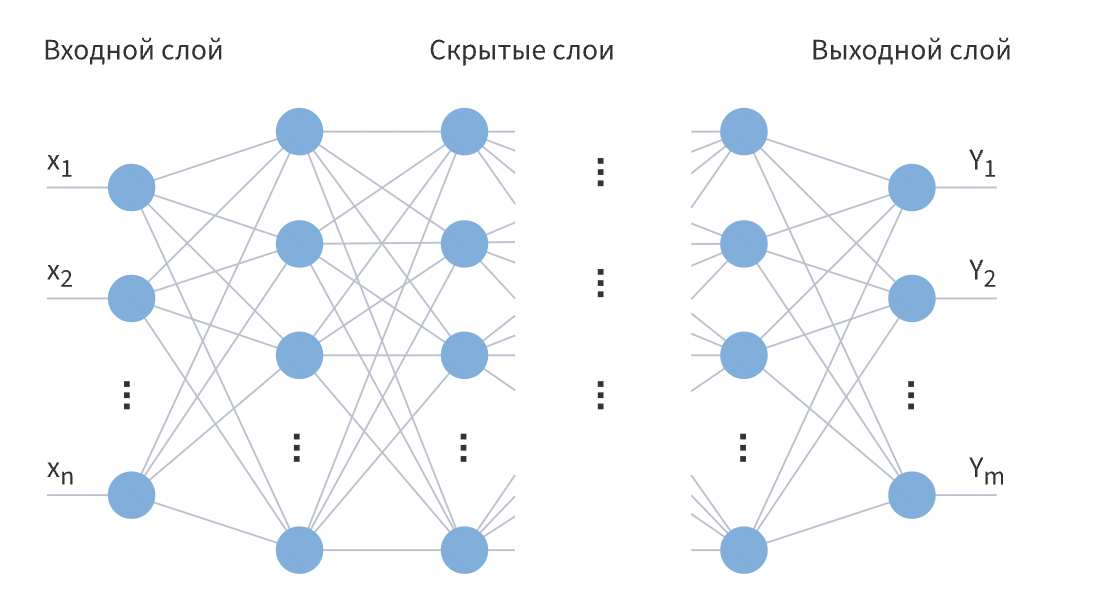
Если используется сигмоидная функция активации (например: 𝑓(𝑥)=𝜎(𝑥) , 𝑓(𝑥)=tanh(𝑥) ), то после нормализации ее аргумента нелинейность по сути пропадает.

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 .

### 

### 16. Многослойные сети. Граф потока вычислений.

Многослойными называются [нейронные сети](https://wiki.loginom.ru/articles/neural-network.html), в которых [нейроны](https://wiki.loginom.ru/articles/artificial-neuron.html) 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](https://wiki.loginom.ru/articles/supervised-learning.html). Наиболее популярным алгоритмом обучения для них является алгоритм [обратного распространения ошибки](https://wiki.loginom.ru/articles/back-propagation-algorithm.html) 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.

## 

## 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.

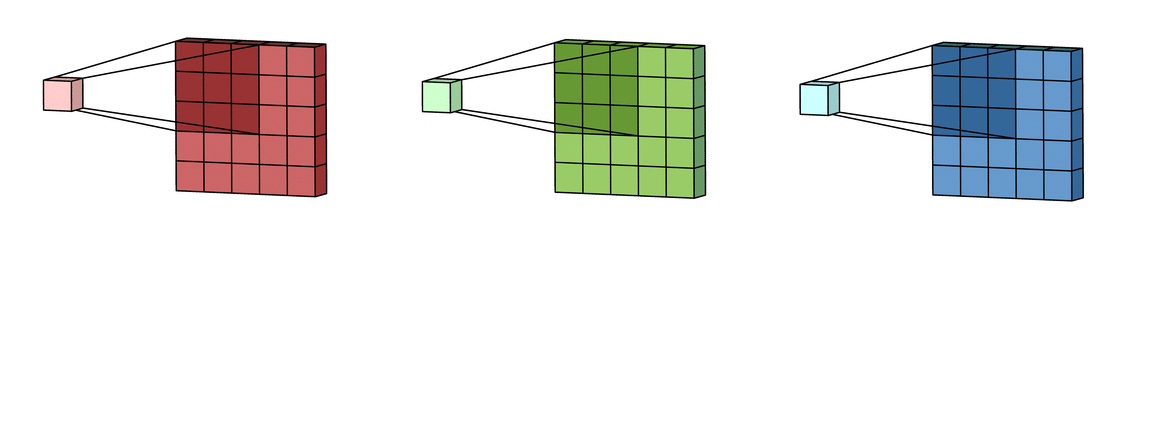
Свёрточные нейронные сети состоят из базовых блоков, благодаря чему их можно собирать как конструктор, добавляя слой за слоем и получая всё более мощные архитектуры. Основными блоками свёрточных нейронных сетей являются свёрточные слои, слои подвыборки (пулинга), слои активации и полносвязные слои.

**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.

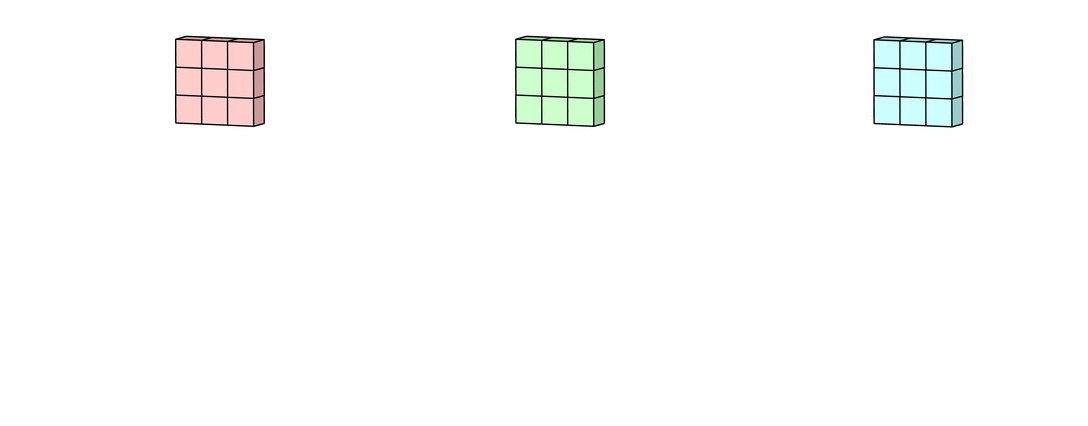
Для того, чтобы слой мог выделять признаки, в нём имеются так называемые фильтры (или ядра). Ядра — это всего лишь набор тензоров. Эти тензоры имеют один и тот же размер, а их количество определяет глубину выходного 3D массива. При этом глубина самих фильтров совпадает с количеством каналов входного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



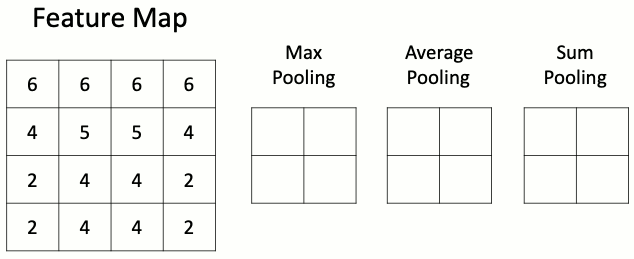
Затем числа полученных матриц суммируются в единую матрицу — результат применения фильтра.



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

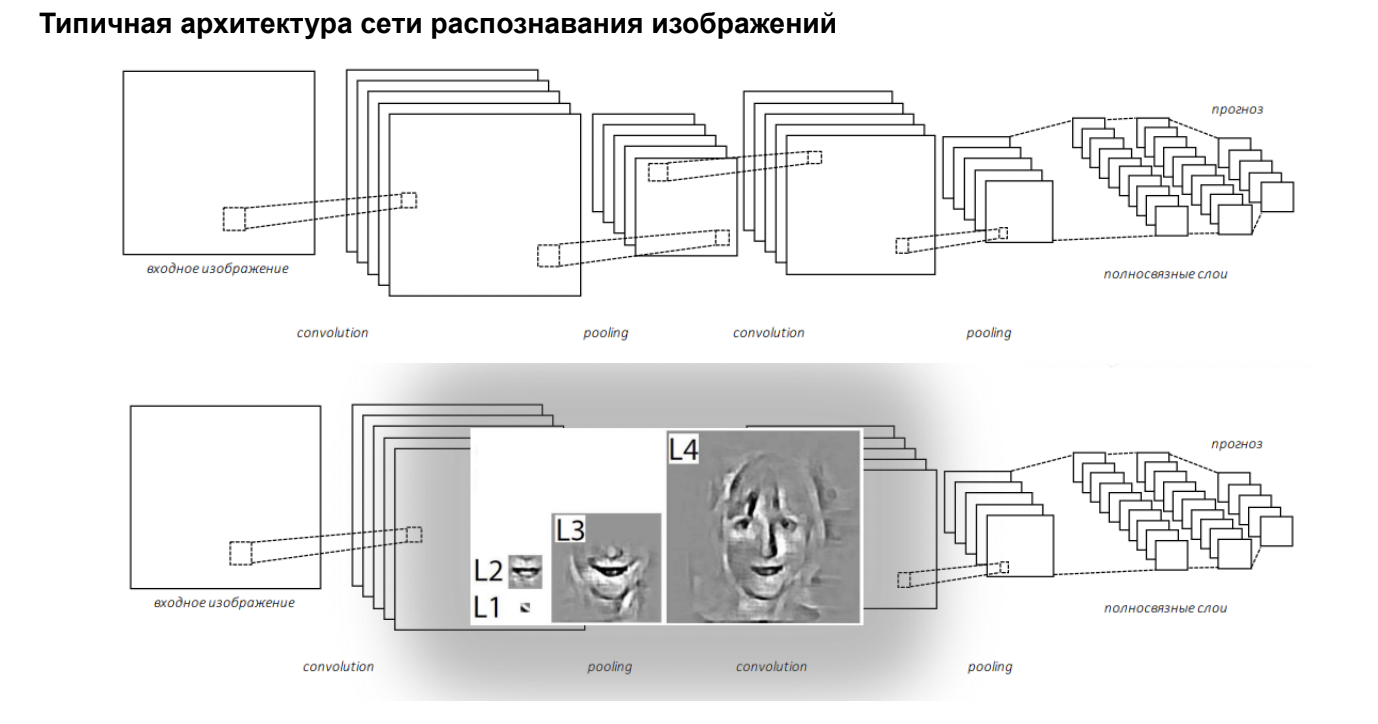
Данный слой содержит матрицу весовых коэффициентов и вектор смещений и ничем не отличается от такого же слоя в обыкновенной полносвязной сети. Единственным гиперпараметром слоя является количество выходных значений. При этом результатом применения слоя является вектор или тензор, у которого матрицы в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

****

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.

### 

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

**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.

### 

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

Эмбеддинг (embedding) ~ векторное представление. Решение задачи машинного обучения (с учителем) требует конструирования признаков для каждой конкретной задачи!

Для применения NN дискретные объекты (например слова) нужно представить в виде непрерывных, дифференцируемых объектов (чтобы потом можно было решать задачу градиентного спуска при обучении).

**Дистрибутивная семантика**

Дистрибутивная гипотеза - лингвистические единицы, встречающиеся в схожих контекстах, имеют близкие значения.

Контексты слова - это множество слов которые встречаются рядом с рассматриваемом словом в тексте. Для получения контекстов можно использовать:

* скользящие окна фиксированной ширины
* рассматривать в качестве границ окон, определяющих контекст, границы предложений, абзацев
* Каждому слову (или понятию) из словаря присваивается свой контекстный вектор
* Множество векторов формирует векторное пространство слов
* Семантическое расстояние между словами (или понятиями) естественного языка, обычно вычисляется как косинусное расстояние между векторами векторного пространства слов
* Для определения значения векторного представления слова (или понятия) используется множество его контекстов найденных в рассматриваемом текстовом корпусе

**Матрица совместной встречаемости**

Матрица совместной встречаемости (co-occurence matrix) - матрица в которой строки соответствуют определенным сущностям и столбцы соответствуют определенным сущностям (набор сущностей для строк и столбцов может отличаться, тогда матрица будет прямоугольной или набор сущностей может совпадать, тогда матрица будет квадратной) - элементы матрицы неотрицательные целые числа равные количеству раз когда сущность, которой соответствует строка, и сущность, которой соответствует столбец, совместно встречались в общем контексте.

**Представление слов в виде векторов малой размерности**

Проблемы с представлением слов в виде вектора матрицы совместной встречаемости:

* большой размер вектора
* квадратичный рост хранимого объема информации при увеличении словария
* высокая разреженность матрицы совместной встречаемости
* неэффективное хранение информации
* низкая надежность использования меры сходства

Решение: построить представление слов малой размерности. Используется алгоритм Word2Vec и его производные. В этом алгоритме на вход однослойного перцептрона подается набор векторов представления слов из контекста, смоделированный, например, алгоритмом Bag-of-Words, на выходе такого перцептрона может быть получен такой же вектор для центрального слова в рассматриваемом окне. Скрытый слой такой нейросети имеет вектор небольшой размерности относительно всего лексикона — порядка нескольких сотен компонентов.

* обычно вектора малой размерности для слов имеют размерность от 25 до 1000 компонент (словари могут содержать многие десятки тысяч слов)
* вектора плотные и хранят "самую ценную" информацию т.е. мера сходства для плотных векторов в среднем должна давать близкий результат к мере сходства для векторов совместной встречаемости
* векторное представление слов назвают: "word vectors", "word embeddings", "word representations"

Как уменьшить размерность векторов матрицы совместной встречаемости?

Существует несколько методов:

* методы снижения размерности для матриц совместной встречаемости
* построения векторов малой размерности с помощью нейронных сетей

Изображение выглядит как стол

Автоматически созданное описание

Local Representation (в частности one-hot) – каждое измерение соответствует отдельному объекту.

Изображение выглядит как стол

Автоматически созданное описание

Распределенное представление (Distributed representation) – за измерениями не закреплен смысл, но расстояние между векторами несет смысловую нагрузку.

### 22. Word2vec: модель CBOW.

Существует несколько видов преобразования слов в числа. Одним из таких NLP-методов является Word Embeddings. Рассмотрим наиболее популярную разновидность Word Embeddings – нейросеть Word2Vec. Идея word2vec состоит в том, чтобы напрямую обучать вектора слов малой размерности

Нейронная сеть Word2Vec имеет две реализации: Skip-gram и CBOW (Сontinuous bag-of-words).

CBoW (англ. Continuous Bag of Words, «непрерывный мешок со словами», bag — мультимножество) предсказывает текущее слово, исходя из окружающего его контекста.

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

* окно для предсказания может быть любым (не обязательно предсказывать следующее слово по предыдущим), обычно предсказывают центральное слово в окне по левому и правому контексту
* по сути модель CBOW это неглубокая нейронная сеть с одним скрытым слоем
* каждый вход сети - это вектор в one-hot представлении размерности |𝑉| (размер словаря)
* в CBOW 2𝑚 входов: ( 𝑚 - ширина окна) и 1 выход 𝑦(𝑐)

Изображение выглядит как текст

Автоматически созданное описание

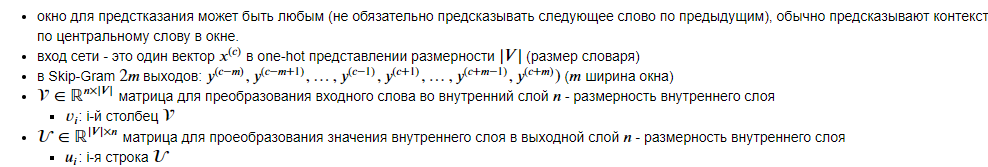
### 

### 23. Word2vec: модель Skip-Gram.

Существует несколько видов преобразования слов в числа. Одним из таких NLP-методов является Word Embeddings. Рассмотрим наиболее популярную разновидность Word Embeddings – нейросеть Word2Vec. Идея word2vec состоит в том, чтобы напрямую обучать вектора слов малой размерности

Нейронная сеть Word2Vec имеет две реализации: Skip-gram и CBOW (Сontinuous bag-of-words).

Модель Skip-Gram учится как можно лучше по заданному слову восстановить его контекст. Дано конкретное слово в середине предложения (входное слово). Алгоритм наугад выбирает одно слово из близких к входному и для каждого слова в словаре предсказывает вероятность того, что наугад выбрали именно его.



Изображение выглядит как текст

Автоматически созданное описание

### 

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

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

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

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

### 

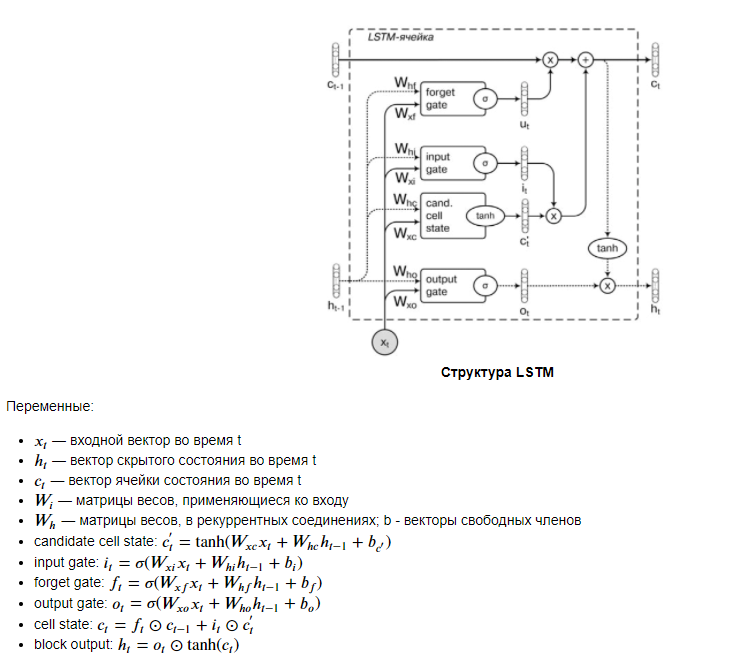
### 25. Модуль LSTM.

LSTM (Long Short-Term Memory)

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

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

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



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

### 

### 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. Архитектура Transformer.

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

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

Изображение выглядит как текст

Автоматически созданное описание

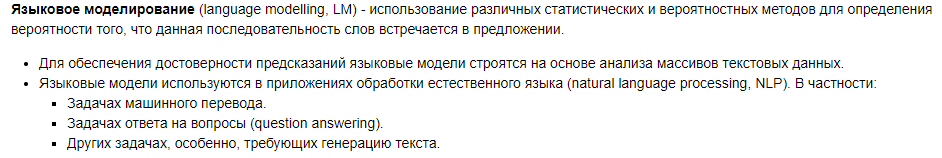
Изображение выглядит как текст

Автоматически созданное описание

### 28. Модель BERT.

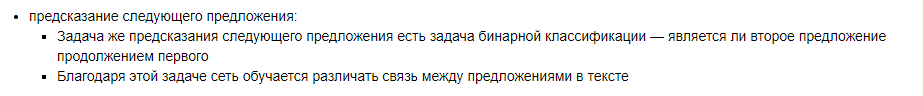
Bidirectional Encoder Representations from Transformers (BERT) - языковая модель, основанная на архитектуре трансформер, предназначенная для предобучения языковых представлений с целью их последующего применения в широком спектре задач обработки естественного языка. BERT был создан и опубликован в 2018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

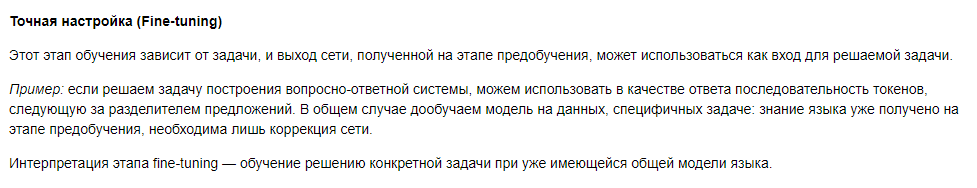
Автоматически созданное описание 

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание



## 

## PyTorch и задачи глубокого обучения

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

torch.Tensor- это многомерная матрица, содержащая элементы одного типа данных. Torch определяет 10 типов тензоров с вариантами CPU и GPU.

Тензор (в линейной алгебре) — объект линейной алгебры, линейно преобразующий элементы одного линейного пространства в элементы другого. Частными случаями тензоров являются скаляры, векторы, билинейные формы и т. п.

Часто тензор представляют как многомерную таблицу 𝑑×𝑑×⋯×𝑑 , заполненную числами - компонентами тензора (где 𝑑 — размерность векторного пространства, над которым задан тензор, а число размерностей совпадает рангом (валентностью) тензора. В случае ранга 2 запись тензора на письме выглядит как матрица.

Изображение выглядит как текст

Автоматически созданное описание



Изображение выглядит как текст

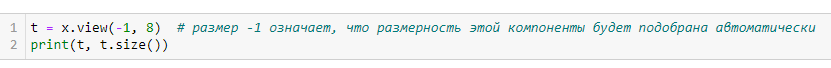
Автоматически созданное описание



Изменяющие размер операции

Изображение выглядит как текст

Автоматически созданное описание



Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

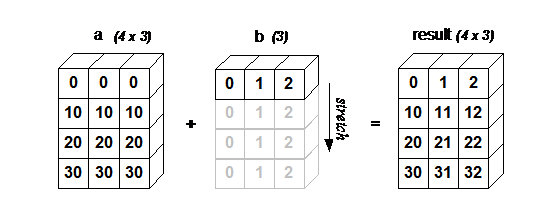
Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Broadcasting аналогичен NumPy



Операции агрегации



Изображение выглядит как текст

Автоматически созданное описание

### 

### 30. Принципиальная логика обучения нейронной сети.

Обучение нейронной сети основано на использовании алгоритма обратного распространения ошибки (backpropagation). Этот алгоритм используется для оптимизации весов нейронной сети так, чтобы ее выходные данные максимально соответствовали ожидаемым.

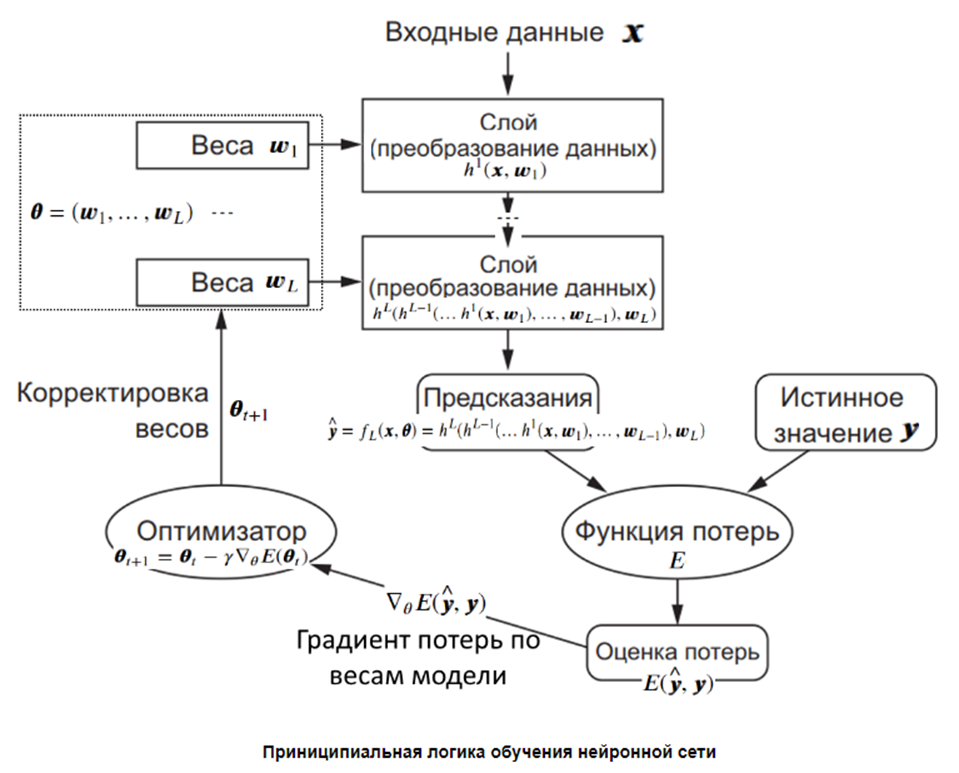
Обучение начинается с инициализации случайными весами нейронной сети. Затем происходит обучение на обучающем наборе данных. На каждом шаге алгоритма входные данные проходят через сеть, и выходные данные сравниваются с ожидаемыми результатами. На основе этой информации вычисляется ошибка сети, которая используется для актуализации весов. Этот процесс повторяется много раз для каждого набора данных, до тех пор, пока ошибка не станет достаточно малой.

После окончания обучения нейронной сети, ее можно использовать для предсказания результатов на новых данных. Это достигается с помощью форвардного распространения входных данных через сеть и получения выходных данных.

Важным аспектом обучения нейронных сетей является предотвращение переобучения. Это происходит, когда сеть излишне адаптируется к обучающему набору данных и не способна обобщить свои знания на новые данные. Различные методы, такие как регуляризация и использование валидационного набора данных, могут помочь в предотвращении переобучения.

Основной проблемой является не применение модели к входным данным и оценка ошибки на основе полученных и истинных значений, а обучение модели, то есть определение наилучших параметров W модели. В случае нейронной сети обучение сводится к поиску весов слоев сети W=(w1, …, wn) , которые в совокупности являются параметрами модели W.

Цель обучения – найти оптимальное значение параметров W\*, минимизирующее ошибку на обучающей выборке, то есть W\* = argmin 𝐸(W) = argmin ∑ ( E(f(x, W), y) ) для каждого значений x, y обучающей выборки D. То есть задача обучения сводится к задаче оптимизации, но на самом деле все сложнее: хороший результат на выборке D может плохо обобщаться (модель может давать низкое качество на другой выборке из той же генеральной совокупности) – так возникает проблема переобучения.



### 

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

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.

### 

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

Принципиальная логика организации работы с данными в PyTorch:

* Создается объект Dataset
  + Dataset обеспечивает доступ к данным (с помощью интерфейса)
  + в параметр transform конструктора Dataset передается вызываемый объект, обеспечивающий трансформацию исходных данных
* Dataset передается в DataLoader
  + DataLoader обеспечивает загрузку данных батчами, распараллеливает загрузку данных и т.п.

Наиболее важным аргументом конструктора DataLoader является dataset, который указывает объект dataset для загрузки данных. PyTorch поддерживает два различных типа наборов данных: map-style (\_\_getitem\_\_\_(), \_\_len\_\_()) и iterable (\_\_iter\_\_()).

Изображение выглядит как текст

Автоматически созданное описание

Transforms. Часто необходим некоторый препроцессинг данных получаемых из данных, оформленных в виде Dataset.

Изображение выглядит как текст

Автоматически созданное описание

DataLoader подгружает данные, предоставляемые классом Dataset, во время тренировки и группирует их в батчи. Он дает возможность указать Sampler, который выбирает, какие примеры из датасета использовать для тренировки. Этот параметр можно использовать для разделения данных на training и validation.

Изображение выглядит как текст

Автоматически созданное описание

### 

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

torch.nn.Module

Базовый класс для всех модулей нейронной сети. Ваши модели также должны быть подклассом этого класса. Модули также могут содержать другие модули, что позволяет размещать их в древовидной структуре. Вы можете назначить подмодули как обычные атрибуты:

Изображение выглядит как текст

Автоматически созданное описание

nn.Module - это базовый класс всех модулей нейронной сети, который реализует определение каждого уровня сети, а также механизм прямого вычисления и обратного распространения в классе. На практике, если вы хотите реализовать определенную нейронную сеть, вам необходимо унаследовать nn.Module.

Изображение выглядит как текст

Автоматически созданное описание

Методы

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

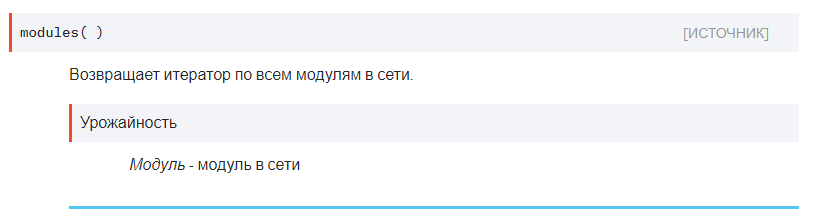
Автоматически созданное описание 

 Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание



Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

### 

### 34. Линейные слои (Linear Layers).

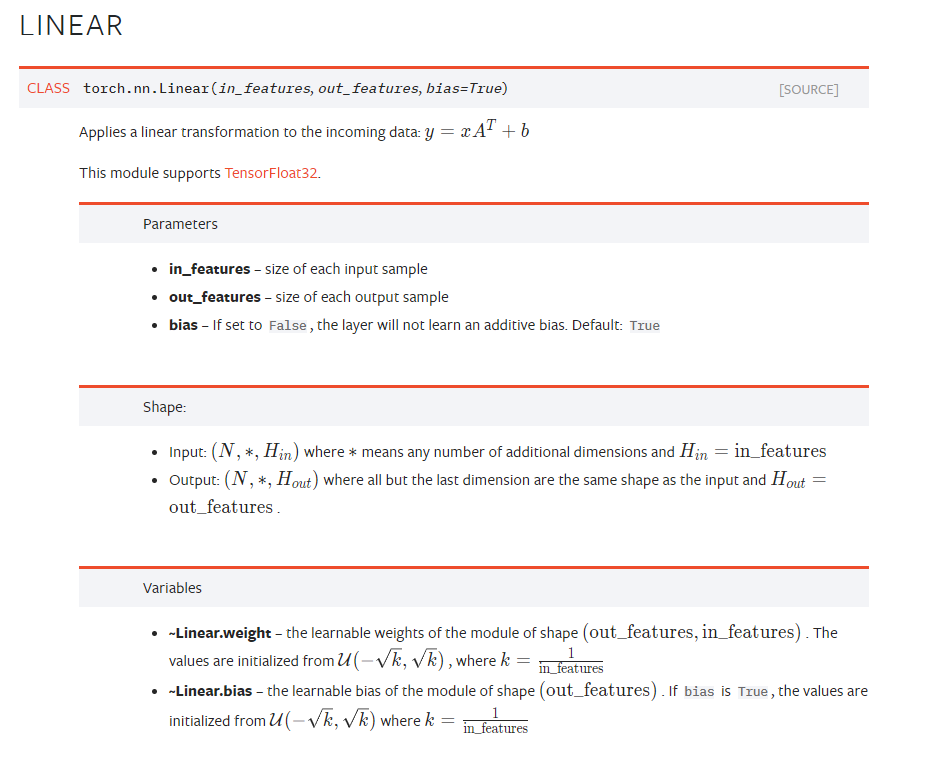
Линейный слой (Linear Layer) является одним из основных компонентов нейронных сетей. Он осуществляет линейное преобразование входных данных и используется для выделения признаков из исходных данных.

Линейный слой может быть представлен как матричное умножение входных данных на матрицу весов и сложение с вектором сдвига(bias). Это может быть записано как y = Wx + b , где x - входные данные, W - матрица весов, b - вектор сдвига, а y - выходные данные.

Линейный слой используется в многослойных нейронных сетях для извлечения признаков из исходных данных и их передачи на следующий слой. В зависимости от конкретной задачи, может использоваться один или несколько линейных слоев подряд.

Обычно в нейронных сетях, линейные слои используются в качестве скрытых слоев между входным и выходным слоем. Это позволяет модели извлекать сложные признаки из исходных данных и использовать их для предсказания. В модели нейронных сетей линейный слой используется в сочетании с нелинейными слоями, такими как слой с ReLU или sigmoid, что дает модели большую гибкость и возможность решать сложные задачи.

**class torch.nn.Linear(in\_features, out\_features, bias=True)**

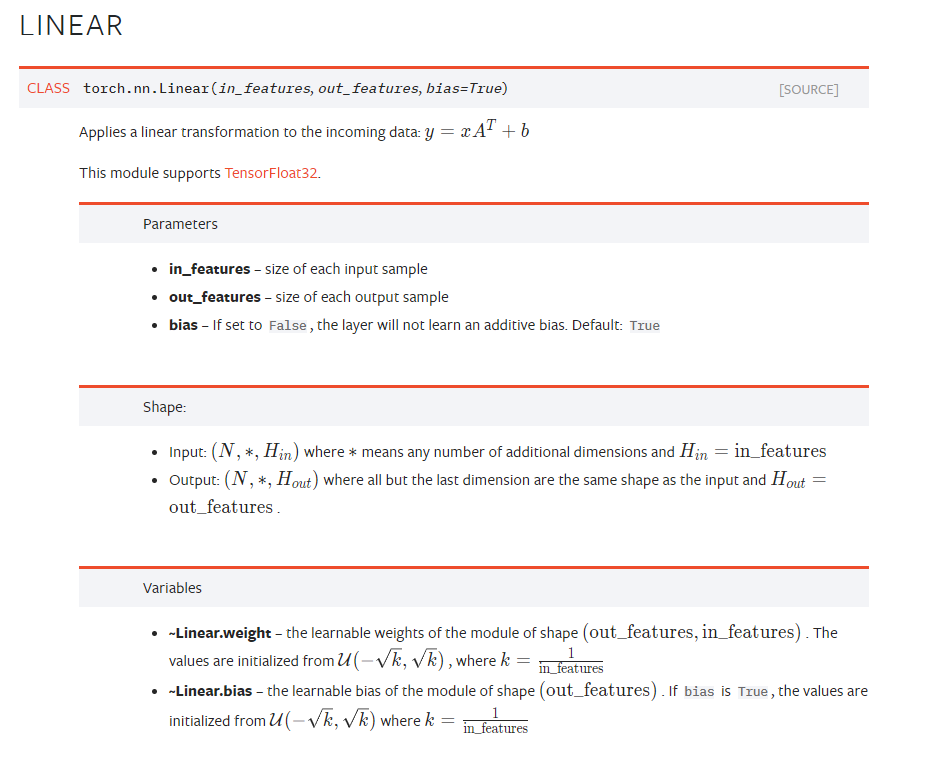
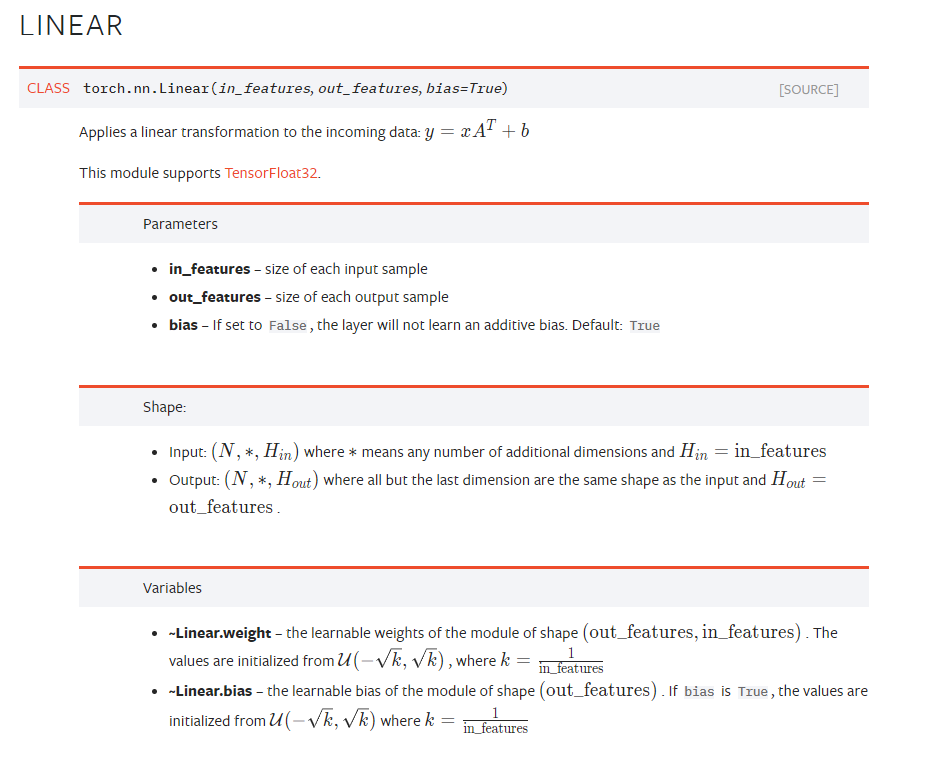
Применяет линейное преобразование к входящим данным:

PyTorch-nn.Linear Linear(n,m)-это модуль,который создает однослойную сеть прямого хода с n входами и m выходами.Математически этот модуль предназначен для вычисления линейного уравнения Ax=b,где x-вход,b-выход,A-вес.

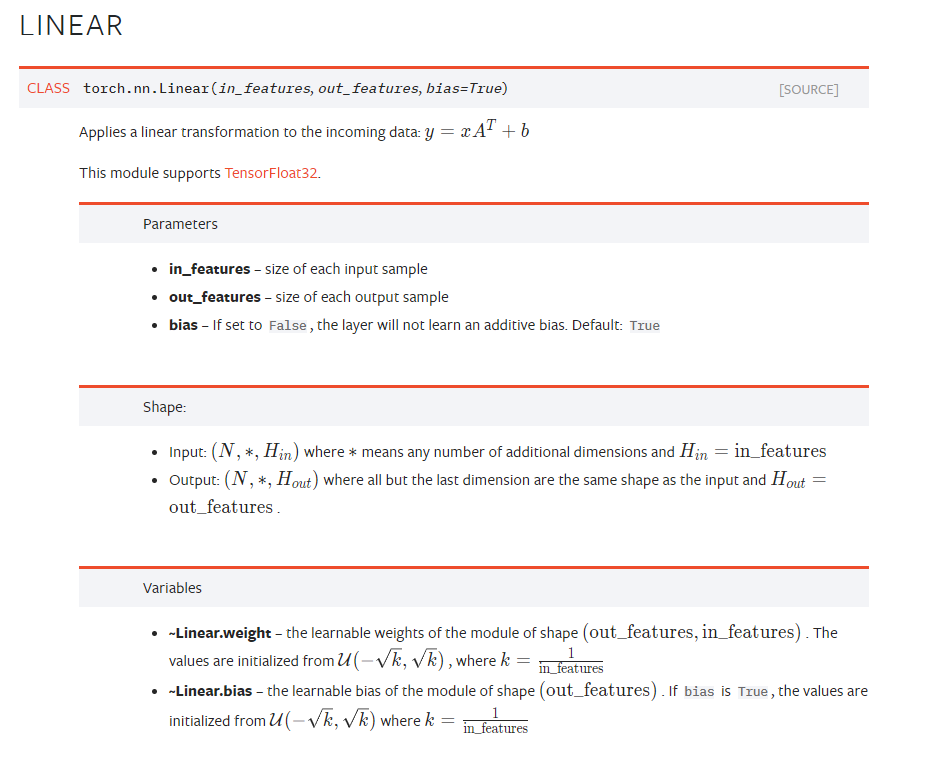
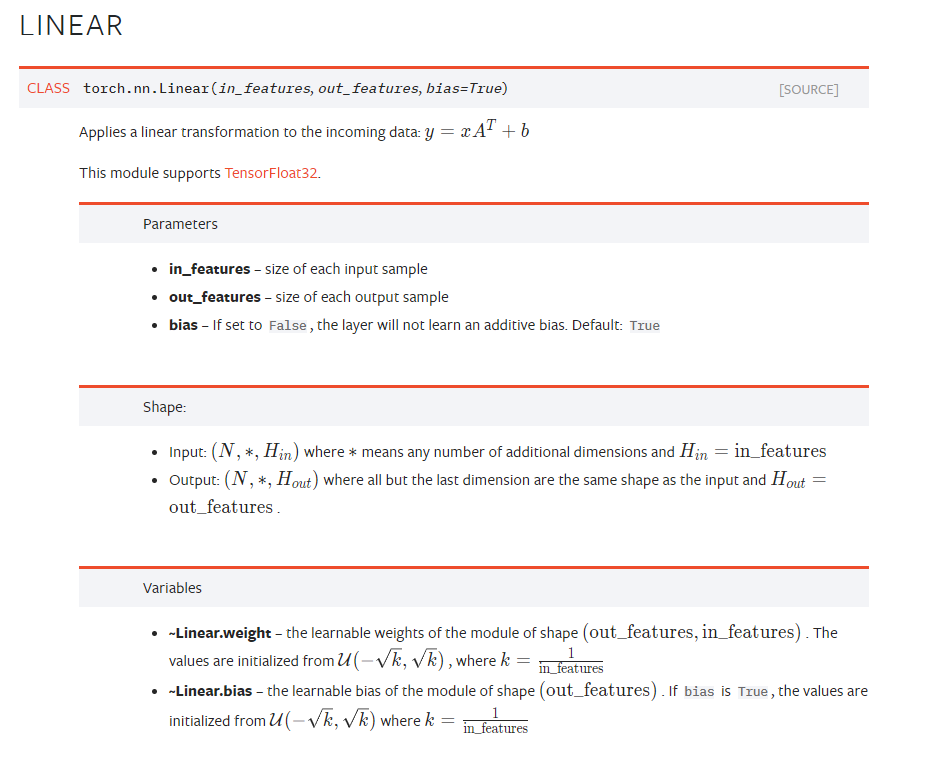
**Parameters**

* in\_features — размер каждого входного образца
* out\_features — размер каждого выходного образца
* смещение — если установлено значение False , слой не будет изучать аддитивное смещение. По умолчанию: True

**Shape:**

* Input: where\* означает любое количество дополнительных измерений H\_in = in\_features
* Output: где все измерения,кроме последнего,имеют ту же форму,что и входные данные, и H\_out = out\_features .

**Variables**

* ~Linear.weight – обучаемые веса модуля формы out\_features, in\_features . Значения инициализируются из 
* ~Linear.bias — обучаемое смещение модуля формы(\text{out\\_features}) . Если bias является True , значения инициализируются 



### 

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

Слои нелинейной активации (Non Linear Activations) - это специальные слои в нейронных сетях, которые используются для добавления нелинейности к модели. Они применяются к выходу каждого слоя для того, чтобы добавить нелинейность и повысить возможности модели для обработки сложных задач.

Существует множество различных функций активации, но некоторые из самых популярных:

* ReLU (Rectified Linear Unit): эта функция активации возвращает значение входного значения, если оно больше 0, и 0 в противном случае. ReLU широко используется в сверточных нейронных сетях, так как она позволяет ускорить процесс обучения и улучшить качество модели.
* Sigmoid: эта функция активации имеет S-образную форму и используется для преобразования линейных значений в вероятности в диапазоне от 0 до 1. Она часто используется в задачах классификации.
* Tanh (Hyperbolic tangent): эта функция активации имеет гиперболический тангенс в качестве формы и используется для преобразования линейных значений в диапазон от -1 до 1. Она часто используется в рекуррентных нейронных сетях.
* Softmax: эта функция активации используется для преобразования линейных значений в вероятностный вектор в диапазоне от 0 до 1. Она часто используется в задачах классификации для определения вероятности каждого класса.

Важно отметить, что выбор функции активации может иметь значительное влияние на качество и скорость обучения модели. В зависимости от типа задачи и данных, одна или несколько функций активации могут быть более подходящими для использования. Часто используются сочетания различных функций активации для решения сложных задач.

**class torch.nn.ReLU(inplace=False)**

Применяет выпрямленную линейную единичную функцию поэлементно:

Изображение выглядит как текст

Автоматически созданное описание

**Parameters**

inplace — при желании можно выполнить операцию на месте. По умолчанию: False

**Shape:**

* Input:(N, \*) где \* означает любое количество дополнительных измерений.
* Output:(N, \*) , та же форма, что и вход.

Examples:

>>> m = nn.ReLU()

>>> input = torch.randn(2)

>>> output = m(input)

An implementation of CReLU - https://arxiv.org/abs/1603.05201

>>> m = nn.ReLU()

>>> input = torch.randn(2).unsqueeze(0)

>>> output = torch.cat((m(input),m(-input)))

**class torch.nn.Sigmoid**

Применяет функцию "по элементам" :

Изображение выглядит как текст

Автоматически созданное описание

**Shape:**

* Input:(N, \*) где \* означает любое количество дополнительных измерений
* Output:(N, \*) , та же форма, что и вход

Examples:

>>> m = nn.Sigmoid()

>>> input = torch.randn(2)

>>> output = m(input)

**class torch.nn.Tanh**

Применяет функцию "по элементам":

Изображение выглядит как текст

Автоматически созданное описание

**Shape:**

* Input:(N, \*) где \* означает любое количество дополнительных измерений
* Output:(N, \*) , та же форма, что и вход

Examples:

>>> m = nn.Tanh()

>>> input = torch.randn(2)

>>> output = m(input)

### 

### 36. Слои нормализации (Normalization Layers).

Слои нормализации (Normalization Layers) - это специальные слои в нейронных сетях, которые используются для нормализации данных. Они используются для того, чтобы сделать входные данные более устойчивыми к изменениям и повысить скорость обучения и точность модели.

Существует несколько различных методов нормализации, но три наиболее популярных из них:

* Batch Normalization (BN): Этот слой используется для нормализации данных на каждом слое нейронной сети. BN считает среднее и стандартное отклонение по батчу (набору) данных и использует их для нормализации каж дого входного значения.
* Layer Normalization (LN): Этот слой используется для нормализации данных в каждом слое нейронной сети. LN считает среднее и стандартное отклонение по каждому входному значению. в отличие от Batch Normalization, которая использует статистику по всему батчу, Layer Normalization использует статистику по каждому нейрону.
* Instance Normalization (IN): Этот слой нормализует каждое входное изображение отдельно. Он считает среднее и стандартное отклонение для каждого изображения и использует их для нормализации. Этот метод полезен для нормализации данных, которые не являются взаимозависимыми, например, для разных изображений.

Использование слоев нормализации может значительно улучшить работу модели и ускорить процесс обучения. Они помогают избежать проблем с исчезающим и взрывающимся градиентом, которые могут возникнуть из-за большой разницы между значениями входных данных. Они также позволяют модели более легко обрабатывать различные размеры и масштабы входных данных.

Слои нормализации обычно используются в сочетании с другими слоями, такими как сверточные слои или полносвязные слои, и располагаются между ними для достижения лучшей точности и скорости обучения.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

### 

### 37. Слои регуляризации (Dropout Layers).

Dropout является одной из самых популярных методов регуляризации нейронных сетей. Он используется для предотвращения переобучения, так как заставляет слои нейронной сети справляться с различными сэмплами и помогает уменьшить связность между ними.

Принцип работы Dropout: в каждый момент обучения случайным образом выбираются некоторые нейроны и они "выключаются" (то есть, их выходные значения устанавливаются равными нулю) во время одной итерации обучения. Это заставляет сеть находить другие пути для прогнозирования в зависимости от данных.

Частота выключения нейронов задается с помощью гиперпараметра "Dropout rate", который обычно задается в диапазоне от 0 до 1. Если dropout rate = 0.1, значит 10% нейронов будут выключены в каждой итерации обучения.

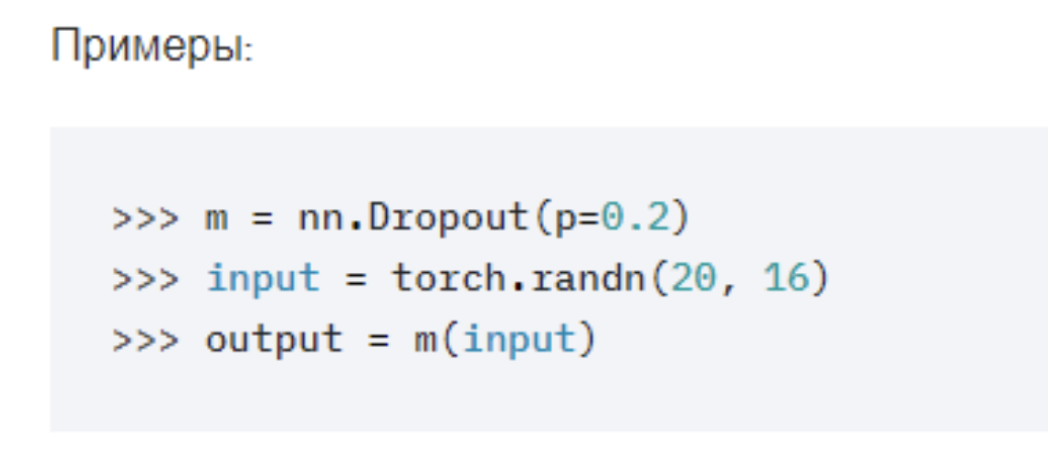
Dropout обычно используется в слоях сверточных или полносвязных нейронных сетей. Он может быть добавлен как в процессе создания модели, так и во время тренировки, используя специальные слои Dropout.

Но стоит отметить что dropout применяется во время тренировки, а не на тесте, или в боевых условиях, это сделано для того чтобы сгладить выходной результат модели, и повысить ее точность.

torch.nn.Dropout(p=0,5;inplace = False)

Параметры:

* p - вероятность обнуляния элемента. По умолчанию 0.5
* inplace - если установлено True, то хэта операция выполняется на месте. По умолчанию False



Также существует torch.nn.AlphaDropout с теми же аргументами

AlphaDropout это модификация слоя Dropout, которая используется в нейронных сетях с активацией "alpha-ReLU" (альфа-ReLU). AlphaDropout является частным случаем слоя Dropout, где маска отбора значений нейронов создается с помощью случайного распределения "alpha-dropout" (альфа-dropout).

Alpha-ReLU - это активационная функция, которая имеет дополнительный параметр "alpha", который задает нелинейность. Если alpha = 0, то это обычный ReLU, если alpha > 0, то это нелинейность выглядит как полупрямая, со слабой затухающей кривой.

AlphaDropout отличается от обычного Dropout тем, что вместо бинарной маски (0 или 1), он использует случайное распределение "alpha-dropout", которое определяется значением alpha. Это помогает сохранять нелинейность активационной функции и улучшает работу сети.

AlphaDropout обычно используется в нейронных сетях с сильно затухающими активационными функциями, такими как ReLU и LeakyReLU, так как они имеют тенденцию к затуханию градиента. AlphaDropout может улучшить работу сети в этих случаях, поскольку он сохраняет нелинейность активационной функции и уменьшает затухание градиента.

Но нужно иметь ввиду, что использование AlphaDropout также зависит от конкретной задачи и архитектуры сети, и может не давать ощутимого прироста качества в некоторых случаях.

### 

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

Слои свертки и слои объединения являются двумя наиболее важными компонентами сверточной нейронной сети (CNN). Слои свертки отвечают за извлечение объектов из входного изображения, в то время как слои объединения используются для уменьшения размера карт объектов, создаваемых слоями свертки.

Слои свертки - это строительные блоки CNN. Они используются для обнаружения шаблонов во входном изображении путем применения к изображению набора фильтров. Каждый фильтр представляет собой небольшую матрицу весов, которая применяется к изображению. Результатом слоя свертки является карта объектов, которая представляет собой представление входного изображения с выделенными шаблонами, обнаруженными фильтрами.

Объединяющие слои используются для уменьшения размера карт объектов, создаваемых слоями свертки. Это делается путем применения операции объединения к карте объектов. Наиболее распространенной операцией объединения является максимальное объединение, которое берет максимальное значение из области карты объектов и использует его в качестве выходных данных слоя объединения. Объединяющие слои используются для уменьшения размера карт объектов, что помогает снизить вычислительную сложность CNN.

слои свертки используются для извлечения объектов из входного изображения, в то время как слои объединения используются для уменьшения размера карт объектов, создаваемых слоями свертки. Оба эти компонента необходимы для построения успешного CNN.

**class torch.nn.Conv1d(in\_channels, out\_channels, kernel\_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding\_mode='zeros')**

Применяет одномерную свертку над входным сигналом,состоящим из нескольких входных плоскостей.

**class torch.nn.Conv2d-** Применяет двумерную свертку к входному сигналу,состоящему из нескольких входных плоскостей.

**class torch.nn.Conv3d-** Применяет 3D-свертку к входному сигналу,состоящему из нескольких входных плоскостей.

Изображение выглядит как текст

Автоматически созданное описание

**class torch.nn.MaxPool1d(kernel\_size, stride=None, padding=0, dilation=1, return\_indices=False, ceil\_mode=False)**

Применяет 1D max pooling над входным сигналом,состоящим из нескольких входных плоскостей.

**class torch.nn.MaxPool2d -** Применяет 2D max pooling к входному сигналу,состоящему из нескольких входных плоскостей.

**class torch.nn.MaxPool3d -** Применяет 3D max pooling к входному сигналу,состоящему из нескольких входных плоскостей.

**Parameters**

* **kernel\_size** — максимальный размер окна.
* **stride** – шаг окна. Значение по умолчанию — kernel\_size
* **padding** — неявное заполнение нулями для добавления с обеих сторон
* **расширение** — параметр, который управляет шагом элементов в окне.
* **return\_indices** — если True , вернет максимальные индексы вместе с выходными данными.
* **ceil\_mode** — при значении True будет использовать ceil вместо floor для вычисления выходной формы.

**Examples:**

>>> # пул размером = 3, шаг = 2

>>> m = nn.MaxPool1d(3, stride=2)

>>> input = torch.randn(20, 16, 50)

>>> output = m(input)

**class torch.nn.AvgPool1d(kernel\_size, stride=None, padding=0, ceil\_mode=False, count\_include\_pad=True)**

Применяет 1D усреднение по входному сигналу,состоящему из нескольких входных плоскостей.

**class torch.nn.AvgPool2d -** Применяет двумерное усреднение по входному сигналу,состоящему из нескольких входных плоскостей.

**class torch.nn.AvgPool3d -** Применяет трехмерное усреднение по входному сигналу,состоящему из нескольких входных плоскостей.

**Parameters**

* **kernel\_size** — размер окна
* **stride** – шаг окна. Значение по умолчанию — kernel\_size
* **padding** — неявное заполнение нулями для добавления с обеих сторон
* **ceil\_mode** — при значении True будет использовать ceil вместо floor для вычисления выходной формы.
* **count\_include\_pad** — при значении True будет включать заполнение нулями в расчет усреднения.

**Examples:**

>>> # pool with window of size=3, stride=2

>>> m = nn.AvgPool1d(3, stride=2)

>>> m(torch.tensor([[[1.,2,3,4,5,6,7]]]))

tensor([[[ 2., 4., 6.]]])

### 

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

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

### 

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

nn.Embedding это слой в PyTorch, который используется для обучения и применения эмбеддингов (embeddings) в нейронных сетях.

Эмбеддинги используются для представления слов в виде векторов чисел. Это позволяет использовать слова в качестве входных данных для нейронных сетей, так как многие алгоритмы машинного обучения работают только с числовыми данными.

nn.Embedding слой инициализируется с двумя аргументами: числом элементов в входном словаре (num\_embeddings) и размерностью эмбеддинга (embedding\_dim). Во время обучения модели параметры эмбеддинга обучаются на основе данных обучения, чтобы они могли лучше отражать смысл слов.

В задачах NLP эмбеддинги часто используются в качестве входных данных для сверточных или рекуррентных сетей, которые используются для задач NLP, таких как классификация текста, машинный перевод, или генерация текста. После обучения модели, эмбеддинги могут быть использованы для решения других задач, где требуется использовать слова в качестве входных данных.

В этом примере, используется слой nn.Embedding в качестве первого слоя в нейронной сети для классификации текста:

import torch.nn as nn

class TextClassifier(nn.Module):

def \_\_init\_\_(self, vocab\_size, embedding\_dim, num\_classes):

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

self.embedding = nn.Embedding(vocab\_size, embedding\_dim)

self.fc = nn.Linear(embedding\_dim, num\_classes)

def forward(self, x):

x = self.embedding(x)

x = self.fc(x)

return x

Здесь, vocab\_size это число элементов в входном словаре, embedding\_dim - размерность эмбеддинга, а num\_classes - число классов для классификации. В примере используется слой nn.Embedding в качестве первого слоя модели, а затем идет его результат передается в полносвязный слой nn.Linear, который используется для классификации.

В этом примере, входные данные для модели представляют собой целочисленный индекс слова из словаря. При прохождении через слой nn.Embedding, индекс слова используется для извлечения соответствующего эмбеддинга из обученной матрицы эмбеддингов. Результирующий вектор эмбеддинга затем используется как входные данные для дальнейшей обработки и классификации.

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

Простая таблица поиска, в которой хранятся вложения фиксированного словаря и размера.

Этот модуль часто используется для хранения вложенных слов и их извлечения с помощью индексов.

Параметры:

* num\_embeddings(int) - размер словаря вложений
* embedding\_dim(int) - размер каждого вектора встраивания

Переменные:

* Embedding.weight (Tensor) - обучаемые веса модуля формы (num\_embeddings, embedding\_dim), инициализированного из N (0, 1)

### 

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

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.



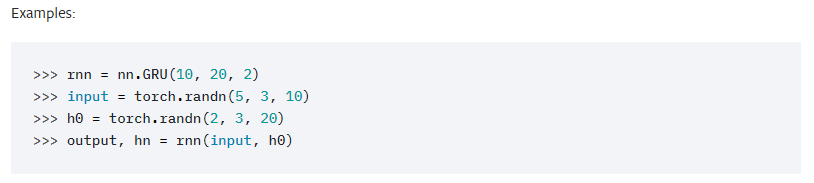
**class torch.nn.GRU(\*args, \*\*kwargs)**

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.