

Содержание

1 Основы глубинного обучения. Соколов. ИАД	4
1.1 Классическое компьютерное зрение	4
1.2 Современный подход к распознаванию объектов	4
1.3 Классическое NLP	4
1.4 Современное NLP	4
1.5 Полезные ссылки	4
1.6 Зачем нужны нейронные сети?	4
1.6.1 Предсказание стоимости квартиры	4
1.6.2 Нелинейные закономерности	5
1.6.3 Лирическое отступление. Нейрон	6
1.7 Граф вычислений (или нейронная сеть)	6
1.8 Полносвязный слой (fully connected, FC)	7
1.9 Как объединить слои в мощную модель?	7
1.9.1 Рассмотрим три полносвязных слоя	7
1.9.2 Нелинейность	8
1.9.3 Типичная полносвязная сеть	8
1.10 Теорема Цыбенко	8
2 Основы глубинного обучения. Обратное распространение ошибки, свёрточные сети. Соколов. ИАД	9
2.1 Повторение градиентного спуска	9
2.2 Обучение нейронных сетей	9
2.3 Как считать производные?	9
2.4 Backprop	10
2.5 Полносвязные нейронные сети для изображений	10
2.5.1 Решение на двух полносвязных слоях	10
2.5.2 Минусы	10
2.5.3 Итог	11
2.6 Свёртки	11
2.6.1 Эксперименты со зрительной корой	11
2.7 Свёртка	12
2.7.1 Как меняется размер картинки после применения фильтра определённого размера	12
2.7.2 Максимум свёртки инвариантен к сдвигам	13
2.8 Свёртки в компьютерном зрении	13
3 Автоматическое дифференцирование, полносвязные нейронные сети. Лекция Ильдуса ПМИ. Лекция 1	15
3.1 Описание решения проблемы произвольной модели машинного обучения	15
3.2 Главные формулы, описывающие DL	15
3.3 Chain rule	15
3.4 Stack more layers	15
3.5 Backward и forward propagation	15
3.6 Веса не только на первом слое	16
3.6.1 Пример вычисления градиента	16
3.7 Построение нейронки	17
3.8 Сигмоида	17
3.9 ReLU	18
3.9.1 Leaky-ReLU	19
3.9.2 Замечание про производную в нуле	19
3.10 MLP/FC(N)	20
3.11 Классификация	20
3.11.1 Базовое представление	20
3.11.2 Функция потерь	20
3.11.3 Softmax	21

3.11.4 Реализация Softmax в PyTorch	21
4 Учебник по машинному обучению Яндекс. Нейронные сети	22
4.1 Первое знакомство с полносвязными нейросетями	22
4.1.1 Основные определения	22
4.1.2 Forward backward propagation	25
4.1.3 Архитектуры для простейших задач	26
4.1.3.1 Бинарная классификация.	27
4.1.3.2 Многоклассовая классификация.	27
4.1.3.3 (Множественная) регрессия.	28
4.1.3.4 Всё вместе.	28
4.1.4 Популярные функции активации	28
4.1.4.1 ReLU.	28
4.1.4.2 Leaky ReLU.	29
4.1.4.3 PReLU.	29
4.1.4.4 ELU.	29
4.1.4.5 Sigmoid.	30
4.1.4.6 Tanh.	30
4.1.5 Зачем нужны функции активации?	31
4.1.5.1 Примечание.	31
4.1.6 Немного о мощи нейросетей	32
4.1.6.1 Теорема Цыбенко (1989).	32
4.2 Метод обратного распространения ошибки	32
4.2.1 Backpropagation в одномерном случае	32
4.2.2 Почему же нельзя просто пойти и начать везде вычислять производные?	33
4.2.3 Градиент сложной функции	34
4.2.4 Backpropagation в общем виде	34
4.2.5 Backpropagation для двуслойной нейронной сети. Автоматизация и autograd	35
4.3 Тонкости обучения	35
4.3.1 Инициализируем правильно	35
4.3.1.1 Наивный подход №0: инициализация нулем/константой.	36
4.3.1.2 Эвристический подход №1: инициализация случайными числами.	37
4.3.1.3 Подход №2: Xavier and Normalized Xavier initialization.	38
4.3.1.4 Подход №3: He (or Kaiming) initialization.	39
4.3.1.5 Выводы:	40
4.3.2 Методы оптимизации в нейронных сетях	40
4.3.3 Регуляризация нейронных сетей	41
4.3.4 Регуляризация через функцию потерь.	41
4.3.5 Регуляризация через ограничение структуры модели.	41
4.3.5.1 Dropout.	42
4.3.5.2 Batch normalization	43
4.3.6 Регуляризация через изменение данных	45
5 Лекция 1. Введение в PyTorch. Наша лекция.	46
5.1 Базовые базовые операции	46
5.2 Работа с torch и в идеокартами	49
5.2.1 Вычисления на m1 и m2	50
5.3 Pipeline обучения	50
6 Оптимизация нейронных сетей, dropout, batch-нормализация. Лекция Ильдуса ПМИ.	52
Лекция 2.	52
6.1 Теорема Цыбенко	52
6.2 Проблемы нейронных сетей	52
6.2.1 Обучение с помощью градиентного спуска	52
6.2.2 Переобучение	52
6.2.3 Ширина	52
6.3 Оптимизация нейронных сетей	52
6.3.1 SGDMomentum	52
6.3.2 Adam	53
6.3.3 Learning Rate Scheduling	53
6.4 Dropout слой	54

6.5	Batch normalization слой	54
7	Лекция 2. Построение и обучение нейронной сети. Наша лекция	55
7.1	Построение архитектуры сети	55
7.2	Как обучать модель	55
7.3	Визуализация обучения	57
7.4	Сохранение нашей модели	59
7.5	Загрузка модели	59
7.6	Обучение на GPU	59

1. Основы глубинного обучения. Соколов. ИАД

1.1. Классическое компьютерное зрение

Что мы делаем в классическом компьютерном зрении?

1. Считаем признаки (есть ли усы, какой формы уши, какой длины хвост, и так далее)
2. Обучаем на них градиентный бустинг.

Но посчитать признаки - отдельная история. Их нужно придумать, правильно находить. Животное может быть с разных сторон сфотографировано.

Так и решалось раньше. Работало - так себе. Градиентный бустинг если мы хорошо умеем обучать - признаки считать не очень.

1.2. Современный подход к распознаванию объектов

В какой-то момент стали искать другие подходы и оказалось интересной идеей делать end-to-end подход:

1. Возьмём нашу картинку - набор пикселей.
2. Подаём на вход какой-то хреновине/штуке/функции, которая на вход принимает картинку.
3. На выходе функция выдаёт число:
 - 1, если кот
 - 0, если собака

И теперь нам не нужно искать признаки, нужно просто искать эту функцию f , и подбирать параметры θ для неё. Получается, что у нас получается так, что функция уже выдаёт готовый результат.

И вот когда мы придумываем очень сложную функцию, это очень сильно помогло. Качество получается лучше, чем в классическом компьютерном зрении.

1.3. Классическое NLP

По простому NLP - обработка текста. И давайте рассмотрим, как классически решается генерация текста:

1. Подсчитываем статистику, как часто то или иное слово встречается после данного.
2. Генерируем следующее слово из этого распределения.

1.4. Современное NLP

Современное GPT уже работает на нейронных сетях. Как пример - GPT-3: она выводит очень связный текст.

1.5. Полезные ссылки

- <https://cs231n.github.io/convolutional-networks>
- <https://stepik.org/course/50352/promo>

1.6. Зачем нужны нейронные сети?

1.6.1. Предсказание стоимости квартиры

Мы помним, что мы умеем предсказывать стоимость квартиры с помощью **линейной модели** $\alpha(x) = w_0 + w_1 \cdot x_1 + \dots + w_n \cdot x_n$. Такая модели имеет минусы, имеет плюсы. Как пример - нет связности признаков, но быстро обучается.

Мы можем пытаться усилить связность признаков: добавить квадраты признаков, перемноженные признаки. Но проблема которая возникает - сколько признаков добавлять? Возможных комбинаций можно придумать очень много, настолько, что даже память может закончиться.

Также мы можем решить такую задачу **градиентным бустингом**. Мы берём кучу базовых моделей и постепенно добавляем в сумму:

$$\alpha_N(x) = \sum_{n=1}^N b_n(x)$$

И обучение N -й модели происходит как

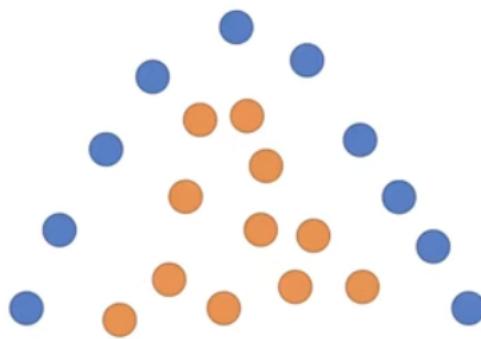
$$\frac{1}{l} \sum_{i=1}^l L(y_i, alpga_{N-1}(x_i) + b_N(x_i)) \rightarrow \min_{b_N(x)}$$

И такие модели, например, уже трудно обучить.

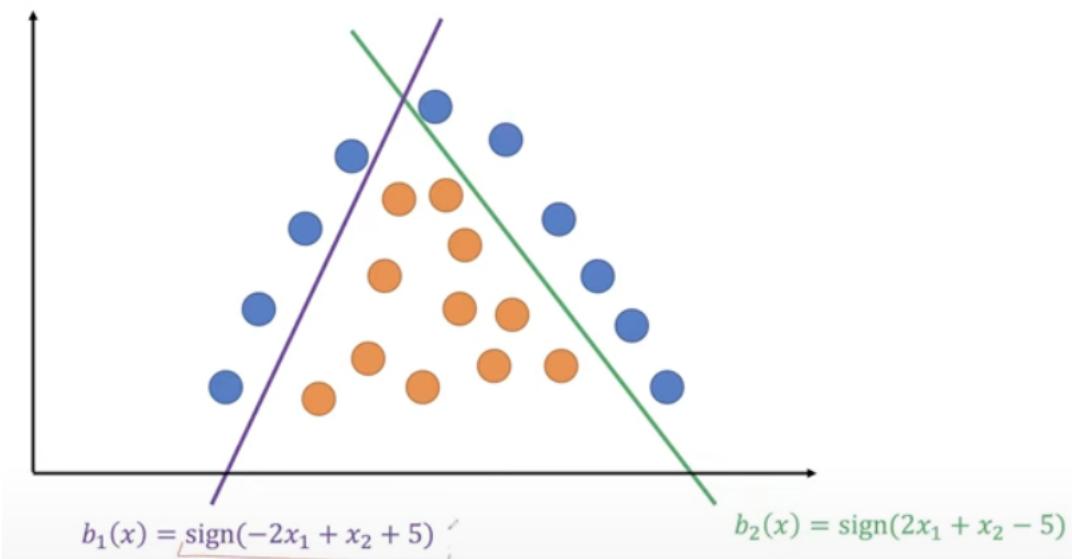
Хочется и чтобы модель была сложная, но и чтобы мы могли её легко стандартными подходами по типу градиентного спуска.

1.6.2. Нелинейные закономерности

Представьте, что у нас есть задача бинарной классификации:

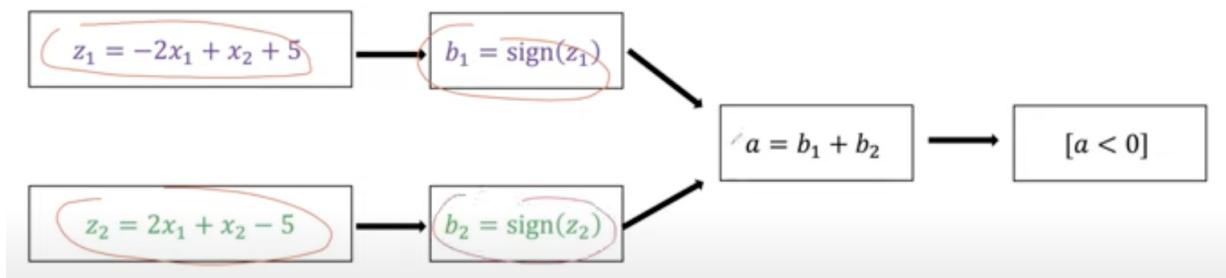


Прямой нельзя отделить оранжевые точки от синих. Линейная модель не применима. А что если мы возьмём две модели, и попробуем отделить синие точки от оранжевых?



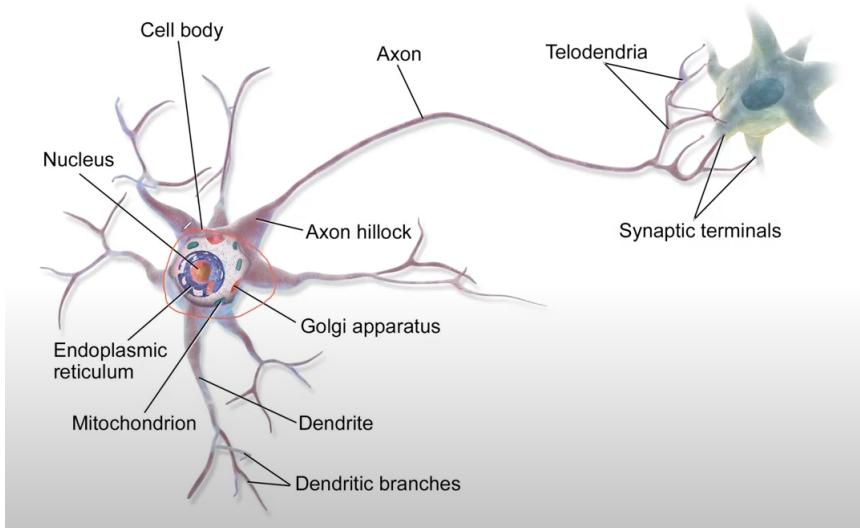
Получается, что оранжевые точки будут попадать в неотрицательную область (если сложить результат моделей), а синие в отрицательную.

Давайте это нарисуем в виде схемы:



Мы считаем формулы прямых. Берём их знаки. Складываем знаки. Смотрим меньше ли нуля эта сумма.

1.6.3. Лирическое отступление. Нейрон



Вот так примерно выглядит нейрон в школьных учебниках по биологии.

Нас ядро не сильно сейчас интересует, а интересует то, как нейрон функционирует.

К нейрону подходят дендриты. Это отростки других нейронов. Все эти сигналы стекаются в ядро. Но, перед тем, как электрический сигнал попадёт в ядро, есть синапсы, которые регулируют сигнал. Клетка может настраивать, как клетка воспринимает сигнал от определённого дендрита. Тогда клетка не реагирует на эти сигналы. И наоборот - клетка может усиливать сигнал.

Дальше все эти сигналы суммируются внутри ядра и по аксону (выходу клетки) этот просуммированный сигнал отправляется дальше.

Сама концепция ослабления/усиления и отправления сигналов похожа на линейную регрессию. Ослабление и усиление сигналов - веса.

Если много нейронов определить - получается нейронная сеть. То есть даже то, что две модели мы объединили, уже можно считать за нейронную сеть.

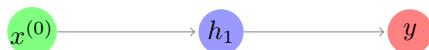
Потом оказалось, что эта модель нейронной сети неверна: оказалось, что ядро сразу не отправляет сигналы, а ядро их накапливает, и когда достигает пикового значения - отправляет по аксону. Эта модель называется моделью Ходжкина-Хаксли.

Получается, то что мы придумали - это не нейрон в голове. Странно на это смотреть как на модель человеческого мозга. Формально - неправильно называть это нейронной сетью и это называют графом вычислений. Про то что так говорить неправильно - мы запомним, но уже просто вошло в жаргонизм такое определение, поэтому оно будет встречаться.

1.7. Граф вычислений (или нейронная сеть)

Граф вычислений выглядит как:

- $x^{(0)}$ - признаки объекта
- $h_1(x)$ - преобразование ("слой")
- $x^{(1)}$ - результат



И вот по сути h_1 - это преобразование. Для модели это может быть череда преобразований. Будем рассматривать такие модели, которые последовательно применяют преобразования и получают ответ.

Это звучит очень абстрактно: что за преобразования, а как они работают, а как они картинку на вход могут принимать, а звук. Конкретика будет позже.

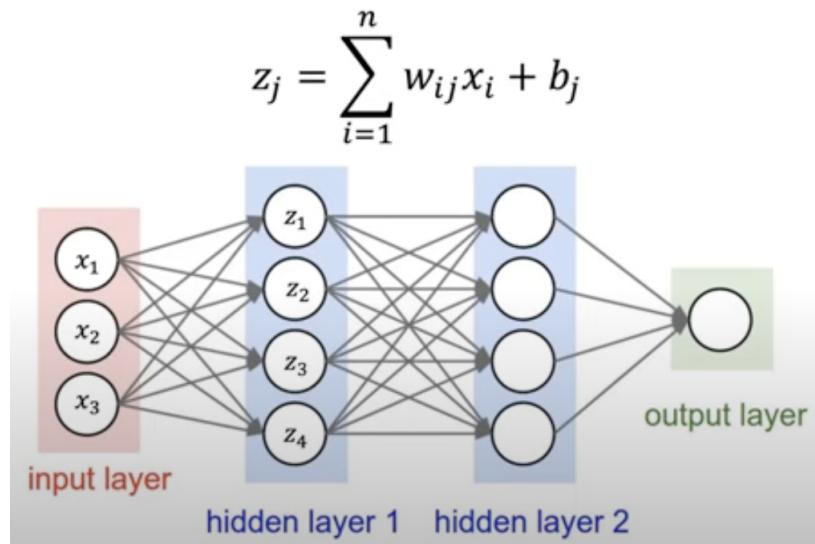
При этом графы вычислений могут быть сложнее: можем представить как бранчевание в git, где есть несколько отбранчёванных веток. Каждая из них развивается по своему. Только вместо коммитов у нас у каждого отбранчевания - преобразования.

1.8. Полносвязный слой (fully connected, FC)

- На входе n чисел, на выходе m чисел
- x_1, \dots, x_n - входы
- z_1, \dots, z_m - выходы
- Каждый выход - линейная модель над входами:

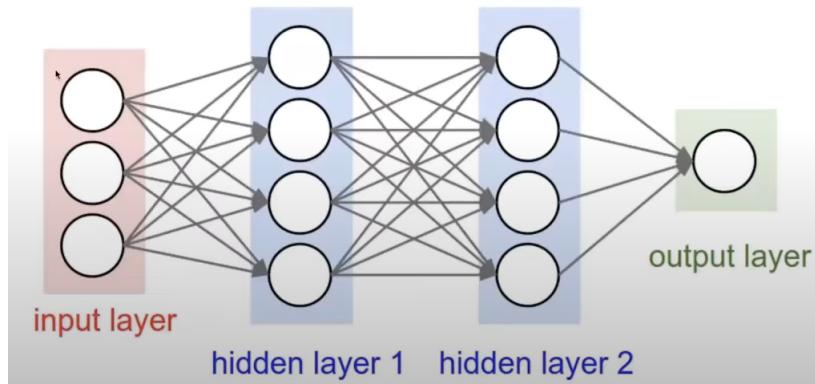
$$z_j = \sum_{i=1}^n w_{ji}x_i + b_j$$

- m линейных моделей, в каждой $n+1$ параметров
- Всего примерно mn параметров в полносвязном слое



1.9. Как объединить слои в мощную модель?

1.9.1. Рассмотрим три полносвязных слоя



Если мы сделаем три полносвязных слоя подряд - это не будем иметь смысла. Рассмотрим два полносвязных слоя

$$s_k = \sum_{j=1}^m v_{kj}z_j + c_k = \sum_{j=1}^m v_{kj} \sum_{i=1}^n w_{ji}x_i + \sum_{j=1}^m v_{kj}b_j + c_k = \sum_{j=1}^m \left(\sum_{i=1}^n v_{kj}w_{ji}x_i + v_{kj}b_j + \frac{1}{m}c_k \right)$$

И это ничем не лучше одного полносвязного слоя подряд. Это буквально получилась та же формула. Соответственно при больше, чем двух полносвязных слоях - они редуцируются до одного.

1.9.2. Нелинейность

Чтобы у нас появлялся какой-то профит - нужно добавлять нелинейную функцию после полносвязного слоя:

$$z_j = f \left(\sum_{i=1}^n w_{ji}x_i + b_j \right)$$

Примеры нелинейных функций f :

- Сигмоида - $f(x) = \frac{1}{1+exp(-x)}$
- ReLU - $f(x) = \max(0, x)$

1.9.3. Типичная полносвязная сеть

Типичная полносвязная сеть выглядит как $x^{(0)} \rightarrow FC_1 \rightarrow f \rightarrow FC_2 \rightarrow f \rightarrow \dots$

- На входе признаки
- В последнем слое выходов столько, сколько целевых переменных мы предсказываем

1.10. Теорема Цыбенко

Вольное изложение:

- Пусть $g(x)$ - непрерывная функция
- Тогда можно построить двуслойную нейронную сеть, приближающую $g(x)$ с любой заранее заданной точностью

Иначе говоря - **двуслойные нейронные сети ОЧЕНЬ мощные!**.

2. Основы глубинного обучения. Обратное распространение ошибки, свёрточные сети. Соколов. ИАД

Видео лекции - <https://www.youtube.com/watch?v=aSTwlPjJfso>

В прошлой лекции мы обсудили пример нейронной сети. Будут другие примеры, но они будут похожи. Но вопрос: как это обучать?

Мы с вами знаем как обучать разрешающие деревья. есть линейные модели - они обучались либо через анилитическую форму, либо через градиентный спуск. Вот про градиентный спуск давайте вспомним.

2.1. Повторение градиентного спуска

Формула для шага градиентного спуска выглядит как $w^t = w^{t-1} - \eta \nabla Q(w^{t-1})$

Останавливаемся, если $\|w^t - w^{t-1}\| < \varepsilon$ или другой вариант: $\|\nabla Q(w^T)\| < \varepsilon$

2.2. Обучение нейронных сетей

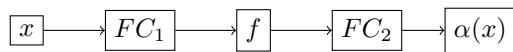
А в глубинном обучении мы останавливаемся обычно, когда ошибка на тестовой выборке перестаёт убывать. Какие-то более сложные критерии останова сложно придумать.

Формула для градиентного спуска есть. Шагаем в сторону антиградиента и обучаемся таким образом. Мы можем так делать, потому что нейронные сети дифференцируемы.

Мы специально будем так строить нейросети, чтобы все функции были дифференцируемыми. Благодаря этому они будут обучаться одинаково, с одними и теми же трюками, эвристиками.

Таким образом, мы можем забыть о том, как обучать и сконцентрироваться только на придумывании нейросетей.

Допустим у нас есть модель:

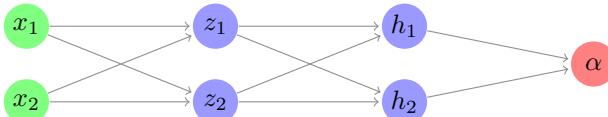


Получается $\alpha(x) = FC_2(f(FC_1(x)))$.

Что здесь являются параметрами (не гипер-параметрами)? Напиши веса на FC_1 и FC_2 - именно их мы и будем тренировать.

Хоть и не все нейронные сети дифференцируемы, но достаточное большинство.

2.3. Как считать производные?



z, h и α получаются тремя полно связанными слоями.

дорисовать блядской граф

Производная у нас берётся по входным данным. При этом заметим, что сейчас у нас в качестве переменных, по которым меняется прогноз - входы и веса. При этом в текущем положении производная любого входа по весу из 0 в -13 всегда будет равна нулю. Если менять нулевой вес, то прогноз нейросети изменится.

Получается можно считать производную любого узла по любому узлу до него.

Вот допустим есть $\alpha(x) = p_{11}h_1(x) + p_{21}h_2(x)$. Как найти $\frac{\partial \alpha}{\partial p_{11}}$? Да на самом деле, если просто от p_{11} , то будет просто $h_1(x)$

А вот найти $\frac{\partial \alpha}{\partial v_{11}}$ - сложнее. Давайте сначала скажем, что $\alpha(x) = p_{11}f(v_{11}z_1(x) + v_{21}z_2(x)) + p_{21}h_2(x)$. Тогда $\frac{\partial \alpha}{\partial v_{11}} = \frac{\partial \alpha}{\partial h_1} \frac{\partial h_1}{\partial v_{11}} = (p_{11}) \cdot (z_1(x))$.

Вот мы рассматриваем, как влияет w_{11} на α . Ответ на этот вопрос зависит от того, чему равно v_{11} . Да: w_{11} влияет на то, что передаёт v_{11} , а v_{11} влияет на выход. Точно также и с v_{12} . А вот изменение w_{11} не влияет на w_{22} .

Тогда, чтобы получить частную производную выхода по w_{11} , нужно найти все пути, которые идут из w_{11} в выход, и просуммировать все частные производные вдоль него:

$$\frac{\partial \alpha}{\partial w_{11}} = \frac{\partial \alpha}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial w_{11}} + \frac{\partial \alpha}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial w_{11}}$$

Получается, чтобы посчитать производные по какому-то параметру - нужно знать производные по параметрам более поздних слоёв. Чтобы это делать - нужно как бы идти считать производные с конца производной к началу. Называется это метод обратного распространения (**backpropagation**).

2.4. Backprop

- Во многие формулы входят одни и те же производные
- В backprop каждая частная производная вычисляется один раз - вычисление производных по слою N сводится к перемножению матрицы производных по слою $N + 1$ и некоторых векторов

2.5. Полносвязные нейронные сети для изображений

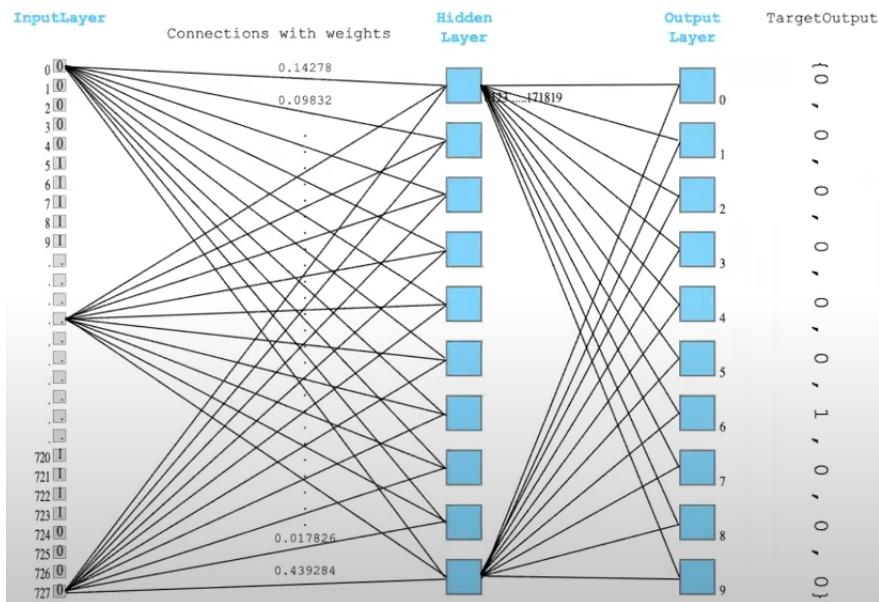
Давайте обсудим на примере конкретной задачи: есть датасет MNIST. Он содержит рукописные цифры:

- Изображения 28×28
- Изображения отцентрированы
- 60000 объектов в обучающей выборке

Пока мы знаем только одну архитектуру нейронной сети - полносвязная. Как мы можем использовать полносвязную сеть, чтобы решить эту задачу?

2.5.1. Решение на двух полносвязных слоях

Допустим у нас будет два FC слоя. Эти картинки $28 \times 28 = 784$ пикселя. Тогда мы картинку вытянем в вектор и будем использовать 784 входа.



В первом скрытом слое будет 10 нейронов - пусть это будет сумма входных пикселей с какими-то весами. Выходной слой - FC слой с 10-ю нейронами, каждый из которых будет отвечать за свой класс.

В MNIST'е есть также серые пиксели. Это можно отмасштабировать, чтобы они были на отрезке $[0; 1]$. Это по сути будет шкала от белого до чёрного.

Но что делает каждый нейрон? Например, нейрон на Hidden layer суммирует все входные пиксели с какими-то весами. При этом - веса могут быть нулевыми, и какой-то нейрон может брать пиксели только из какого-то квадрата.

2.5.2. Минусы

Но что если цифра смешена от центра? До этого наша сеть обучалась только тому, чтобы конкретные области смотреть. Если ничего не делать с нашей сетью - она просто будет выучивать положение цифр на картинке. Если единица будет не посередине, а сбоку - сеть не распознает это. Это, по сути, переобучение.

А ещё у нас получается очень много параметров:

- 784 входа
 - Полносвязный слой: 1000 нейронов
 - Выходной слой: 10 нейронов
 - Весом между входным и полносвязным слоями: $(784 + 1) * 1000 = 785000$
 - Весов между полносвязным и выходными слоями: $(1000 + 1) * 10 = 10010$
- Параметров больше, чем данных - не очень клёво.

2.5.3. Итог

- Очень много параметров
- Сети легко могут переобучиться
- Не учитывается специфика изображений (сдвиги, небольшие изменения формы и т.д.)
- Один из лучших способов борьбы с переобучением - снижение числа параметров

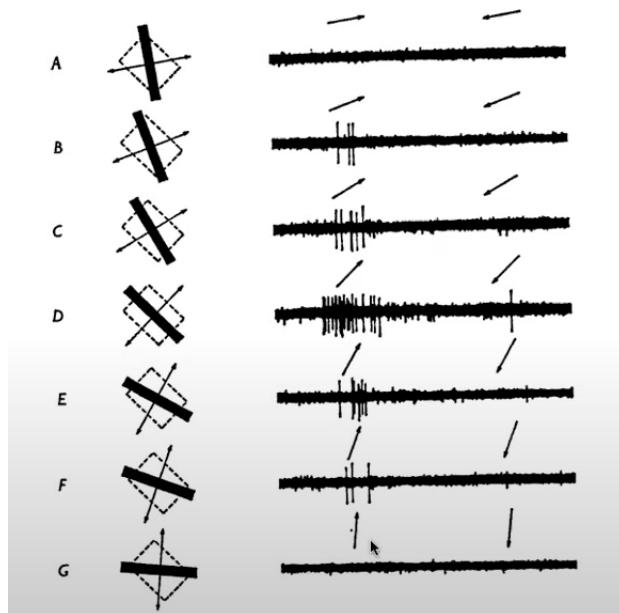
2.6. Свёртки

2.6.1. Эксперименты со зрительной корой

Этому эксперименту уже не один десяток лет. Некоторые исследователи заинтересовались как в голове устроено visual cortex - участок мозга, в который приходит зрительный нерв, и который обрабатывает что приходит со зрительного нерва.

Взяли кота и ему в visual cortex ввели некоторый электрод, который подошёл к конкретному нейрону в этой визуальной коре. Дальше этот электрод измерял активность этого нейрона. Дальше смотрели - как изменяется этот нейрон, в зависимости от того, что у кота перед глазами.

Коту показывают палку перед глазами и смотрят что происходит:



Разместили палку вертикально - ничего не происходит (картинка А). Дальше палку начинают вращать и активность нейрона увеличивается ближе к тому, как её положение становится диагонально (картины В-Д). А после того, как вращение продолжилось - активность при вращении его к горизонтальному положению уменьшалась (картины Е-Г). Ну и видимо этот нейрон, в который попали - реагирует на диагональные объекты. Воткнули в другой нейрон - он реагирует на другие объекты. Провели ещё много экспериментов и выяснилось, что есть много за что ответственных нейронов - за яркость, конкретные цвета итд.

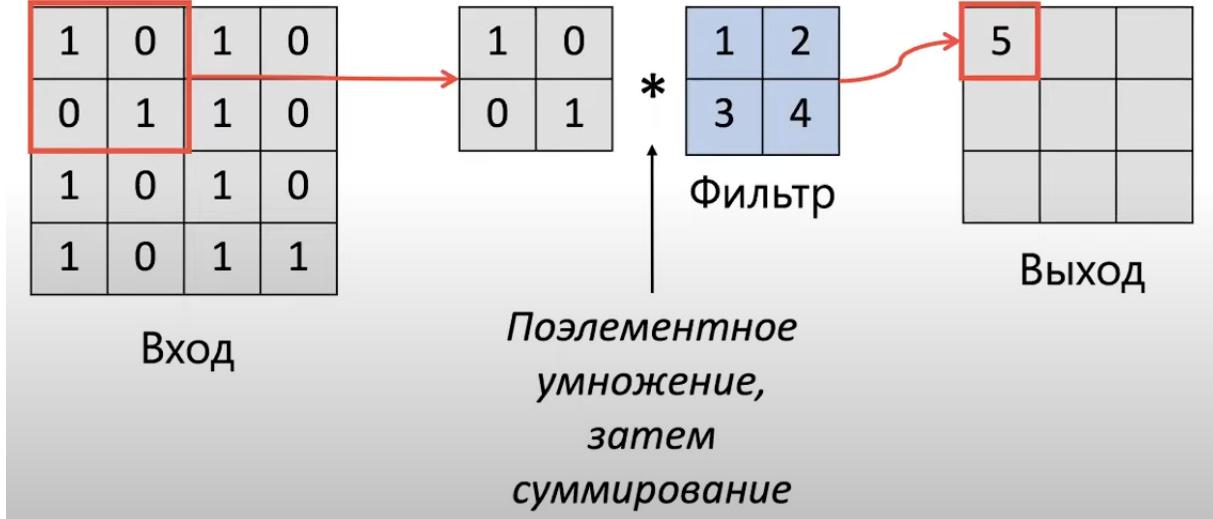
По итогу был сделан вывод в визуальной коре нейроны отвечают за распознавание нехитрых форм, но в совокупности эти нейроны позволяют решать нам безумно сложные визуальные задачи, которые перед нами стоят. Неизвестно вывод правильный или нет.

Свёртки о которых мы будем говорить - выполняют похожую вещь: они распознают какие-то важные паттерны на изображении.

2.7. Свёртка

Что такое операция свёртки (convolution)?

Допустим на входе есть какая-то картинка ЧБ. Также есть некоторый фильтр (ядро свёртки).



Мы каждый квадратик поэлементно умножим и получим какое-то число. Откуда взять сам фильтр? Вот его мы и будем обучать. И важно, что фильтр одинаковый для всей картинки целиком.

Можно ли выход пропустить через фильтр? Да, можно. Размер фильтра - гиперпараметр. Условий на начальный фильтр нет.

Какой смысл имеет свёртка? Представьте, что фильтр - диагональная матрица:

Свёртка

$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$	$*$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$=$	2
$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$	$*$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$=$	2
$\begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix}$	$*$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$=$	6
$\begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix}$	$*$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$=$	1
$\begin{bmatrix} 5 & 0 \\ 0 & 5 \end{bmatrix}$	$*$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$=$	10
$\begin{bmatrix} 0 & 2 \\ 3 & 0 \end{bmatrix}$	$*$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$=$	0

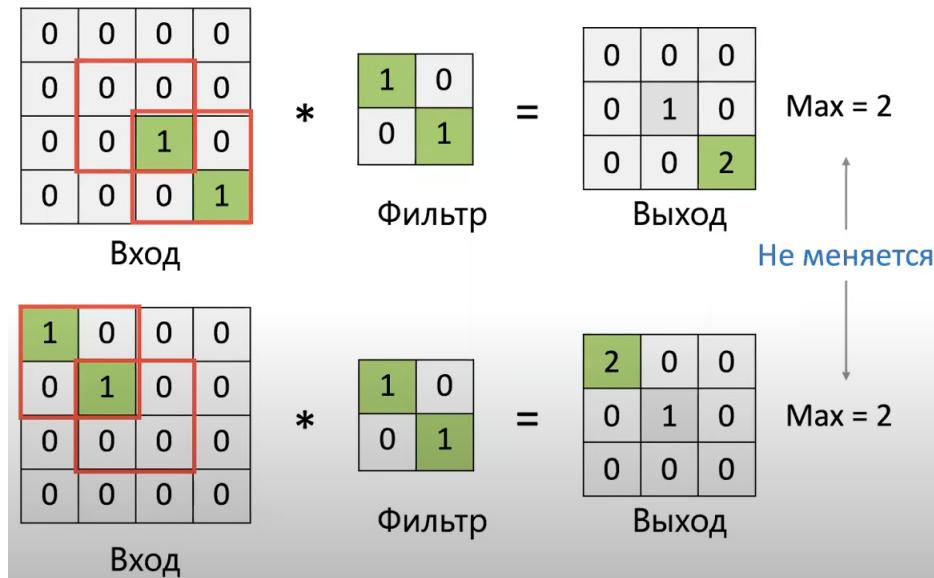
То что фильтр выдаёт в виде единичного значения называется откликом (response).

- Операция свёртки выявляет наличие на изображении паттерна, который задаётся фильтром
- Чем сильнее на участке изображения представлен паттерн, тем больше будет значение свёртки

2.7.1. Как меняется размер картинки после применения фильтра определённого размера

<https://arxiv.org/pdf/1603.07285.pdf>

2.7.2. Максимум свёртки инвариантен к сдвигам

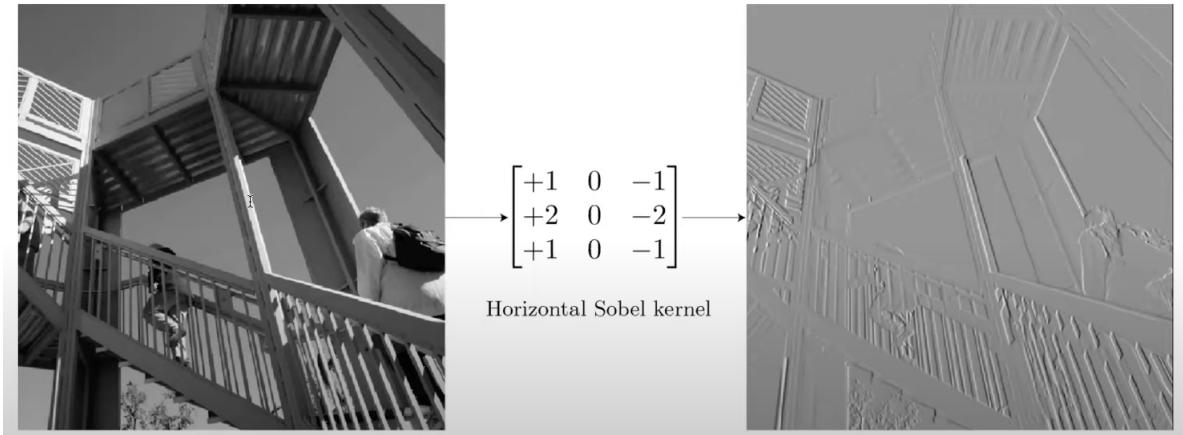


Это означает то, что у нас изображение может быть в разной части картинки и паттерн будет задектирован в независимо какой части изображения.

2.8. Свёртки в компьютерном зрении

Ещё до нейронных сетей придумали использовать фильтры. Есть фильтр Собеля (Horizontal Sobel kernel), который позволяет находить границы.

Вот пример его применения:



Почитать про это можно по ссылке <https://towardsdatascience.com/intuitively-understanding-convolutions-for-image-recognition-2d-filters-11fca8c8a33e>.
Есть также фильтры повышения резкости:

$$\begin{array}{c}
 \text{original} \\
 + a \\
 \text{detail} \\
 = \\
 \text{sharpened}
 \end{array}$$

$$\begin{array}{ccc}
 \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} & + & \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} - \frac{1}{9} \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 2 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} - \frac{1}{9} \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}
 \end{array}$$

<https://ai.stanford.edu/~syyeung/cvweb/tutorial1.html>

Поэтому, как мы видим, в компьютерном зрении давно придумали использовать свёртки, но сами ядра придумывали вручную, никакого обучения не было.

3. Автоматическое дифференцирование, полно связные нейронные сети. Лекция Ильдуса ПМИ. Лекция 1

<https://www.youtube.com/watch?v=g552oCpg-NE>

3.1. Описание решения проблемы произвольной модели машинного обучения

Напишем вот такую вещь: $\alpha(x) = w_1x_1 + \dots + w_nx_n + w_0 = f(x, w)$ - функция, у которой вход это признаковые описания объекта и веса модели. Так описывается большая часть современного машинного обучения.

Давайте теперь скажем, что $f(x, w)$ не обязательно линейная функция - она произвольная функция. На самом деле не совсем произвольная, какие нам подойдут - позже поговорим.

Допустим у нас есть модель и мы хотим её обучить. Возьмём выборку $\{X_i, y_i\}_{i=1}^l$. Если у нас есть такая функция f , что нам нужно сделать, чтобы модель обучить? Нам нужно посчитать функцию ошибки $\frac{1}{l} \sum_{i=1}^l L(f(x_i, w), y_i) \rightarrow \min_w$ - это и есть обучение. И чтобы подобрать эти веса и нужен градиент.

Нам нужно будет находить $\nabla_w L(f(x_i, w), y_i)$, и как следствие ещё находить $\nabla_w f(x_i, w)$. Это главное ограничение, которое у нас возникает. Мы хотим, чтобы наша функция f была не совсем произвольной, а **дифференцируемой**, иначе никакие наши методы для оптимизации работать не будут.

3.2. Главные формулы, описывающие DL

3.3. Chain rule

Мы будем пользоваться chain rule. У нас есть $f = f(g(x))$. Тогда если мы хотим посчитать $\frac{\partial f}{\partial x}$, то мы получаем $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$. На самом деле более продвинутая форма будет верна, где f зависит от нескольких функций: $f = f(g_1(x), g_2(x), \dots, g_n(x)) = \sum_{i=1}^n \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x}$

3.4. Stack more layers

Сначала скажем вот какую вещь: Давайте скажем, что у нас есть функция потерь от одного элемента $L(f(x, w), y)$, у нас есть целая выборка, но нам бы по одному элементу научиться дифференцировать. Чтобы нам делать градиентный спуск, нам нужна $\nabla_w L = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial x}$.

Давайте теперь засунем функции внутрь:

$f = f_4(f_3(f_2(f_1(x))))$. Давайте запишем chain rule для этой функции: $\frac{\partial f}{\partial x} = \frac{\partial f_4}{\partial f_3} \frac{\partial f_3}{\partial f_2} \frac{\partial f_2}{\partial f_1} \frac{\partial f_1}{\partial x}$.

3.5. Backward и forward propagation

Когда у нас есть вот такое представление - уже более-менее понятно что делать. У нас есть простые понятные функции, которые мы умеем вычислять и дифференцировать. Мы можем дифференцировать двумя способами: либо идти начиная от $\frac{\partial f_4}{\partial f_3}$ к концу, либо наоборот от $\frac{\partial f_1}{\partial x}$ к началу. Теперь хорошо бы понять как какая штука будет выглядеть, потому что они не одинаковы.

Теперь давайте подставим $\frac{\partial f}{\partial w}$. Но давайте вот ещё что скажем: чтобы понять, насколько хорошо то, что мы делаем - посчитаем размеры производных. У нас всегда размерно производной это размерность того, что стоит в числите на размерность того, что стоит в знаменателе (делаем оговорку, что мы предсказываем одно число):

- Размерность $\frac{\partial L}{\partial f} - 1 \times 1$
- Размерность $\frac{\partial f}{\partial w} - 1 \times d$, где d - размерность весов.
- Размерность $\frac{\partial f_4}{\partial f_3} - 1 \times d_3$, 1 сначала, потому что f_4 - функция выхода, а d_3 - просто размерность f_3 .
- Размерность $\frac{\partial f_3}{\partial f_2} - d_3 \times d_2$
- Размерность $\frac{\partial f_2}{\partial f_1} - d_2 \times d_1$
- Размерность $\frac{\partial f_1}{\partial w} - d_1 \times d$

Мы тут немного делаем оговорку, что веса могут не только на самом нижнем уровне играть роль, но для понимания конструкции нам это не важно.

Мы будем на каждой итерации цикла считать $\frac{\partial f}{\partial f_i}$ с размерностью $1 \times d$ или, если идём с другого конца - $\frac{\partial f_i}{\partial w}$ с размерностью $d_i \times d$.

Когда мы идём от начала к концу - это называется backward propagation, а когда идём от конца к началу - forward propagation.

Мы хотим использовать backward propagation, если хотим продифференцировать скаляр.

Если мы хотим из скаляра получить вектор - то нам нужно воспользоваться forward propagation.

3.6. Веса не только на первом слое

Рассмотрим функцию $f_2(f_1(x, w_1), w_2)$.

У нас есть где-то посчитанная $\frac{\partial L}{\partial f}$. Мы хотим посчитать $\frac{\partial L}{\partial w_2}$ и делаем это как $\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial w_2}$.

Нужно найти $\frac{\partial f}{\partial f_1} = \frac{\partial f}{\partial f_2} \frac{\partial f_2}{\partial f_1}$.

Чтобы нам найти производную по весам, мы следующим шагом сделаем так: $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f_1} \frac{\partial f_1}{\partial w_1}$

3.6.1. Пример вычисления градиента

Давайте возьмём функцию $y = x_1 w_1 + 3(x_2^2 - w_2)$.

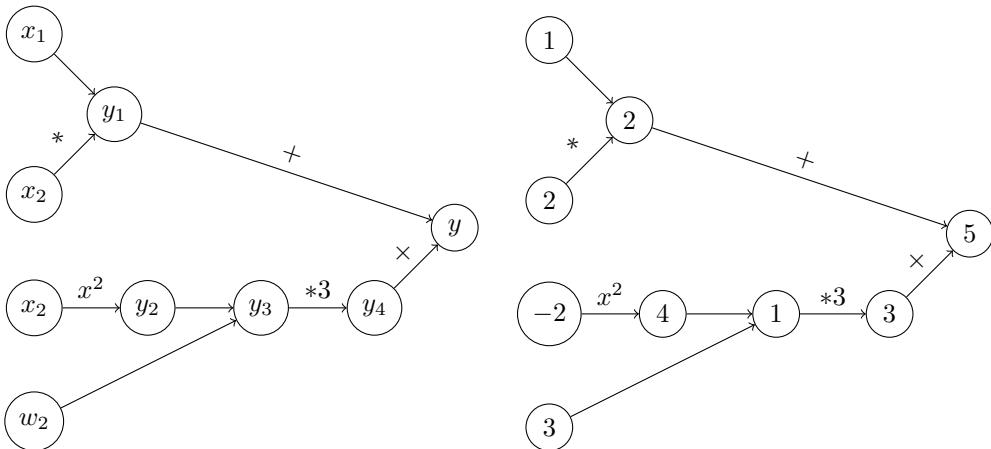
Чтобы численно посчитать производные, давайте обозначим чему равны наши параметры и переменные:

- $x_1 = 1$
- $w_1 = 2$
- $x_2 = -2$
- $w_2 = 3$

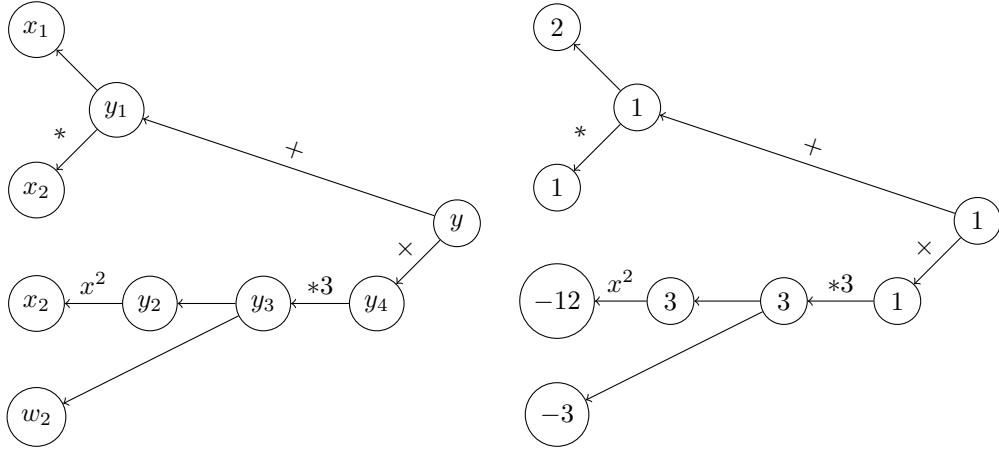
Также обозначим:

- $y_1 = x_1 w_1$
- $y_2 = x_2^2$
- $y_3 = y_2 - w_2$
- $y_4 = y_1 + 3y_2$

Граф вычислений выглядит так:



Теперь посчитаем в обратную сторону:



В таком подсчёте разницы между x и w нет. Мы её чувствуем, только когда строим реальную модель.

3.7. Построение нейронки

У нас исходно была просто линейная комбинация x : $w_1x_1 + \dots + w_nx_n = wx$

Давайте вместо одного выхода сделаем много: $w \in \mathbb{R}^d \rightarrow W \in \mathbb{R}^{d_1 \times d}$

А теперь скажем, что $y_1 = f_1(x, w_1) = (W_1x) \in \mathbb{R}^{d_1}$

Дальше возьмём ещё вектор $y_2 = f_2(y_1, w_2) = W_2y_1 \in \mathbb{R}^{d_2}$

...

$$y_n = f_n(y_{n-1}, w_n) = W_n y_{n-1}$$

Вот мы и построили нейронку.

Мы сделали очень много лишней работы, но у нас всё ещё осталась линейная функция: $y_n = \underbrace{W_n W_{n-1} \dots W_1}_W x$.

Что тогда делать, чтобы такого не было? Самое простое решение - давайте после каждого умножения на матрицу добавим некоторую поэлементную функцию, которая будет преобразовывать каждое число в векторе нелинейным преобразованием. Обозначим эту функцию за σ : $y_1 = f_1(x, w_1) = \sigma(W_1x) \in \mathbb{R}^{d_1}$

$$y_2 = f_2(y_1, w_2) = \sigma(W_2y_1) \in \mathbb{R}^{d_2}$$

...

$$y_n = f_n(y_{n-1}, w_n) = \sigma(W_n y_{n-1})$$

Но очень часто мы будем делать афинные преобразования: $y_1 = f_1(x, w_1) = \sigma(W_1x + b_1) \in \mathbb{R}^{d_1}$

$$y_2 = f_2(y_1, w_2) = \sigma(W_2y_1 + b_2) \in \mathbb{R}^{d_2}$$

...

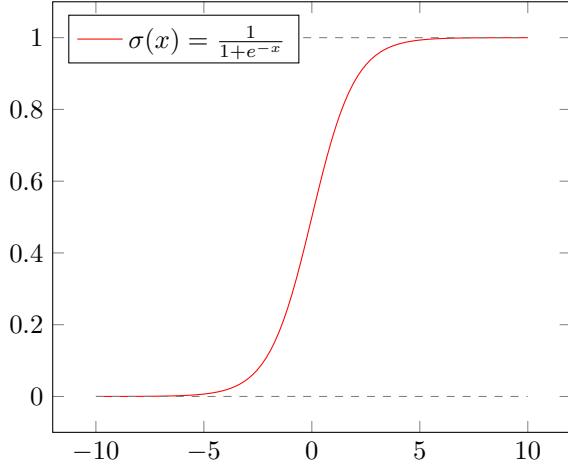
$$y_n = f_n(y_{n-1}, w_n) = \sigma(W_n y_{n-1} + b_n)$$

Нам нужна эта функция активации, потому что если мы её не добавим - у нас все матрицы схлопываются в одну W матрицу (получается простая линейная модель, хоть мы и ввели много весов, которые параметризуют её).

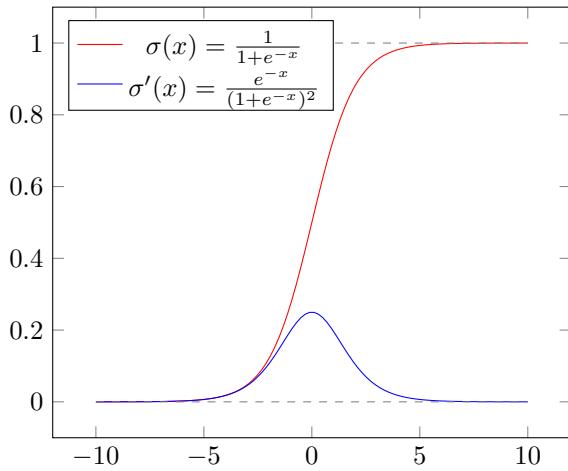
То есть по итогу наша функция выглядит как $y_n = \sigma(w_n \sigma(\dots \sigma(w_1x + b_1)) + b_{n-1}) + b_n$

3.8. Сигмоида

Сигмоида выглядит как $\sigma(x) = \frac{1}{1+e^{-x}}$. Эта функция обладает свойством, что она всю числовую прямую $(-\infty, +\infty)$. Переводит в отрезок $[0; 1]$.



Эта функция нелинейная и она нам подойдёт. Изначально нейронки и учили из этой функции. Производная этой функции выглядит как $\frac{\partial \sigma}{\partial x} = \frac{1}{1+e^{-x}} \cdot (1 - \frac{1}{1+e^{-x}}) = \frac{e^{-x}}{(1+e^{-x})^2}$



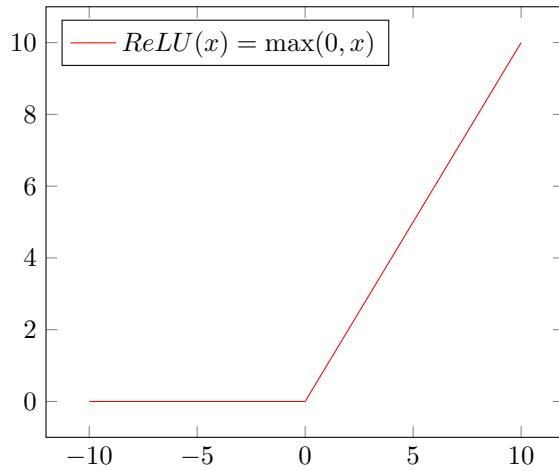
Какая есть проблема? Если слишком большое значение активации, то производная становится очень маленькой, почти нулём. Это проблема, когда мы хотим считать производную, потому что производная идёт с переда назад, и по сути производная передаётся самым глубоким слоям.

Если мы напишем производную на очередном шаге $y = \sigma(w, x)$, мы посчитаем производную $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \frac{\partial \sigma}{\partial (w_i, x)} \cdot w_i$. И каждый раз когда мы передаём эту вещь для подсчёта back-propagation, мы каждый раз передаём $\frac{\partial \sigma}{\partial (w_i, x)}$. Если эта штука маленькая, то и дальше производная будет маленькая. Чем больше слоёв настакано, тем меньше будет становиться производная. До самых нижних слоёв дойдём очень маленькая производная и ничего учиться не будет.

Давайте тогда не будем использовать эту активацию и возьмём то, что будет иметь производную получше.

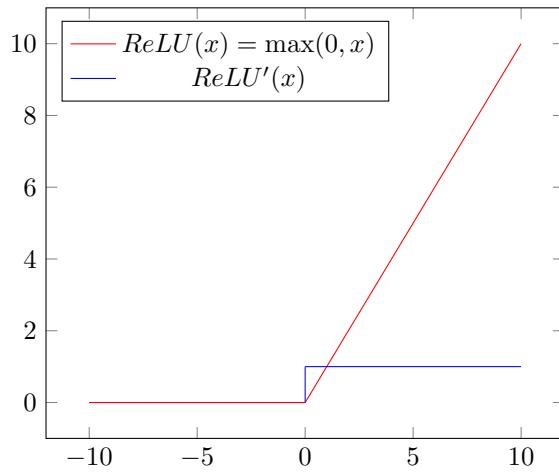
3.9. ReLU

Пример такой функции - *ReLU* (Rectified Linear Unit) = $\max(0, x)$. Вычисляется быстрее, чем экспоненты.



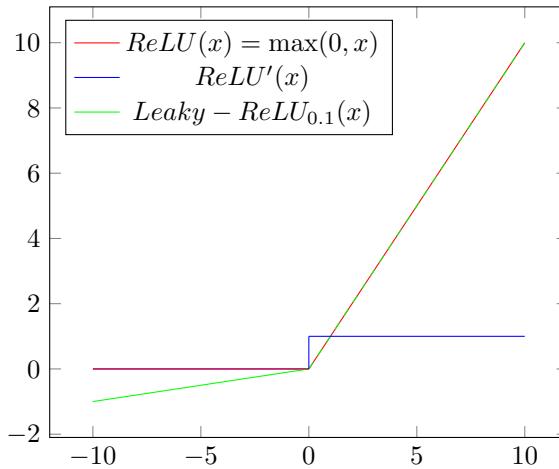
При этом эта функция нелинейная! Все те вещи, которые мы говорили про вынесении матриц в одну мы тут не можем сказать, потому что функция нелинейная.

А производная для неё будет выглядеть вот так:



3.9.1. Leaky-ReLU

Есть ёщё Leaky-ReLU: есть какой-то гиперпараметр u , и $\text{Leaky-ReLU}_u(x) = \max(ux, x)$.



Чисто эвристически - должно лучше работать.

3.9.2. Замечание про производную в нуле

Нужно также сказать, что в теории в нуле - производной нет. Но так как мы генерируем случайные числа - в ноль мы никогда не попадём.

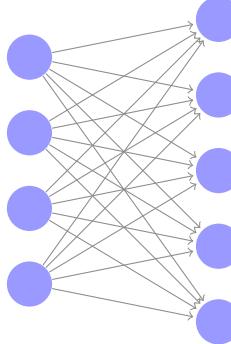
То же самое, на самом деле касается и МАЕ - ведь она в нуле не дифференцируема.

3.10. MLP/FC(N)

Очень часто можно встретить аббревиатуры MLP и FC:

- MLP - Multi-Layer Perceptron
- FC Fully-Connected network

Это одно и то же. Это обозначает, что между парой слоёв у всех нейронов со всеми есть связь:



Пример полносвязанного слоя

3.11. Классификация

3.11.1. Базовое представление

В курсе Машинного Обучения мы разделяли как вид задач бинарную и многоклассовую классификацию. Здесь же мы всегда будем думать о многоклассовой классификации.

Допустим у нас есть K классов $\{1, \dots, K\}$, и нам нужно придумать какую-то нейронку, которая будет выдавать вероятности классов.

Выход последнего слоя будет состоять из K чисел и i -я вершина будет хранить p_i - вероятность, что объект принадлежит классу i , иными словами мы получаем:

$$y_{out} = \begin{pmatrix} \mathbb{P}(y_i = 1) \\ \mathbb{P}(y_i = 2) \\ \dots \\ \mathbb{P}(y_i = K) \end{pmatrix}$$

При этом у нас получаются следующие ограничения на этот вектор:

- $\forall k \in 1, \dots, K : \mathbb{P}(y_i = k) \geq 0$

- $\sum_{k=1}^K \mathbb{P}(y_i = k) = 1$

Вообще говоря - всё обучение это по сути - метод максимального правдоподобия.

3.11.2. Функция потерь

Функция потерь будет выглядеть следующим образом: давайте возьмём вероятность правильного класса и возьмём логарифм от этой вероятности с отрицательным знаком и будем минимизировать:

$$-\sum_{k=1}^K [y_i = k] \log \mathbb{P}(y_i = k)$$

Negative log likelihood.

По общим соглашениям мы все функции потерь хотим минимизировать, при этом вероятность мы хотим максимизировать.

Как нам теперь взять какой-то выход нейронки, чтобы он отображал вероятности?

3.11.3. Softmax

Закончить нашу нейронку нужно линейным слоем, который выдаёт K чисел. Возьмём такую функцию:

$$\text{Softmax}(x) = \frac{e^{x_k}}{\sum_{j=1}^K e^{x_j}}$$

Таким образом каждая координата больше нуля, и в сумме они будут давать единицу.

Давайте предположим, что f - выход нейронной сети, тогда нам нужно минимизировать функцию $\underset{w}{\text{NLLLoss}}(\text{Softmax}(f), y) \rightarrow \min$.

На самом деле делать так никогда не надо!. Численно это очень нестабильная ситуация: если какой-то x будет большим, мы возьмём от него экспоненту, которая нам сразу всё сломает.

Сделать то же самое, но по другому, можно следующим способом:

В нашей функции потерь используется \log вероятности. Давайте сразу посчитаем \log в *Softmax*:

$$\log(\text{Softmax}(x)_k) = x_k - \underbrace{\log\left(\sum_{j=1}^K e^{x_j}\right)}_{\log \text{ sum exp}}$$

3.11.4. Реализация Softmax в PyTorch

Есть класс **CrossEntropyLoss**, который реализует эту функцию потерь. В него мы подаём logits и labels.

- logits - это наши x , которые мы подаём в Softmax
- labels - истинные классы

Другой вариант в PyTorch: самим посчитать LogSoftmax, передаём логарифмы вероятностей в NLLLoss, и там получаем $\log_{p}robs$.

4. Учебник по машинному обучению Яндекс. Нейронные сети

<https://academy.yandex.ru/handbook/ml/article/nejronnye-seti>

В этой главе вы познакомитесь с **нейронными сетями** – семейством моделей, которое начиная с 2012-го постепенно добивается превосходства во всём новых и новых приложениях, во многих став де-факто стандартом. Они были придуманы ещё в 70-х, но техническая возможность и понимание того, как обучать нейросети большого размера, появились лишь примерно к 2011 году, и это дало мощный толчок к их развитию. Совокупность нейросетевых подходов и сама наука о нейросетях носит название глубинного обучения или deep learning. Во многом глубинное обучение основано на двух идеях. Во-первых, это стремление к переходу от построения сложных пайплайнов, каждая компонента которых тренируется сама по себе решать кусочек задачи, к end-to-end обучению всей системы, как одного целого. Так, мы можем обучать не каждый слой отдельно, а все вместе, и не учить какие-нибудь линейные модели поверх случайных лесов, а работать с одной цельной моделью. Во-вторых, это обучение представлений объектов – информативных признаковых описаний, учитывающих структуру данных, построенных лишь на основе самих данных, зачастую неразмечённых. Это позволяет автоматизировать процесс отбора признаков: теперь вместо того, чтобы руками экспертов искать «более информативное или более простое» признаковое описание наших объектов, мы можем получить их автоматически, притом используя не только данные, доступные нам для задачи, но и, к примеру, все тексты мира.

Обучению представлений будет посвящён отдельный раздел, а в этой главе мы постараемся убедить вас, что нейросети – это гибкий инструмент для решения самых разных задач и для работы с самыми разными типами данных.

Надо признать, впрочем, что современные модели весьма сложны и мало напоминают своих элегантных предшественников из 2012 года. Развитие нейросетей во многом мотивируется нуждами и возможностями индустрии, ростом производительности компьютеров и объёмов доступных данных; при этом теория отчаянно не успевает за практикой. В глубинном обучении весьма распространён чисто инженерный подход к построению алгоритмов, изобилие деталей, основанных на интуиции и обосновывающихся фразой «просто потому, что так работает, а иначе – нет», поэтому ряд учёных продолжает относиться к нейросетям скептически. Однако результаты, достигнутые с их помощью за последние 10 лет, столь впечатляющие, что их нельзя игнорировать. Особенно существенный прогресс был достигнут в области анализа данных, обладающих некоторой внутренней структурой: текстов, изображений, видео, облаков точек, графов и т.д. Тем не менее, есть и подходы к теоретическому осмысливанию того, почему нейросети работают так хорошо, и мы познакомим вас с ними в отдельной главе, посвящённой теории машинного обучения.

Но довольно предисловий! Давайте набросаем план нашего вторжения в мир глубинного обучения:

1. Первое знакомство с полносвязными нейросетями. Вы впервые встретитесь с самой простой нейросетевой архитектурой, узнаете, как строятся и как применяются нейронные сети.
2. Обратное распространение ошибки. Вы узнаете, как легко и быстро дифференцировать по параметрам сложного вычислительного графа, после чего будете (теоретически) понимать, как обучить несложную нейросеть.
3. Тонкости обучения. Нейросети – могущественный, но капризный инструмент, и чем сложнее глубинная модель, тем труднее обучить её. В этом разделе мы начнём знакомить вас с разными приёмами, которые позволяют повысить вероятность успеха, а также с регуляризационными техниками для нейросетей.

В первых трёх главах вы познали основы глубинного обучения, но в основном имели дело с ситуацией, когда и данные, и выходы представляют из себя непривязательные векторы. А что делать, если мы должны работать с чем-то более сложно устроенным? Оказывается, за счёт своей гибкости и возможности сочетать в себе самые разные компоненты нейросети отлично справляются с данными, имеющими однородную структуру (изображениями, текстами, облаками точек). Для работы с каждым из этих типов данных требуются специфические инженерные решения, и в дальнейшем тут появятся главы, посвящённые архитектуре для работы с различными структуризованными данными.

4.1. Первое знакомство с полносвязными нейросетями

4.1.1. Основные определения

Искусственная нейронная сеть (далее — **нейронная сеть**) — это сложная дифференцируемая функция, задающая отображение из исходного признакового пространства в пространство ответов, все параметры которой могут настраиваться одновременно и взаимосвязанно (то есть сеть может обучаться end-to-

end). В частном (и наиболее частом) случае представляет собой последовательность (дифференцируемых) параметрических преобразований.

Внимательный читатель может заметить, что под указанное выше определение нейронной сети подходят и логистическая, и линейная регрессия. Это верное замечание: и линейная, и логистическая регрессии могут рассматриваться как нейронные сети, задающие отображения в пространство ответов и логитов соответственно.

Сложную функцию удобно представлять в виде суперпозиции простых, и нейронные сети обычно представляют перед программистом в виде конструктора, состоящего из более-менее простых блоков (**слоёв**, *layers*). Вот две простейшие их разновидности:

- **Линейный слой (linear layer, dense layer)** — линейное преобразование над входящими данными (его обучаемые параметры — это матрица W и вектор b): $x \mapsto xW + b$, где

- $W \in \mathbb{R}^{d \times k}$
- $x \in \mathbb{R}^d$
- $b \in \mathbb{R}^k$

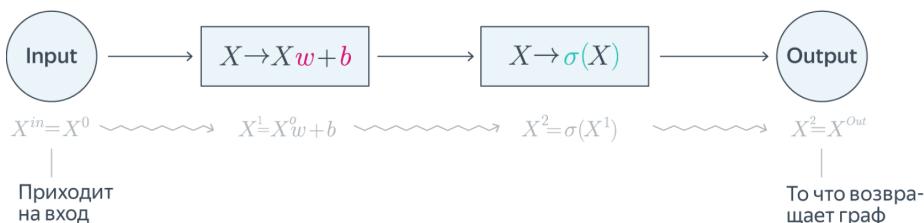
Такой слой преобразовывает d -мерные векторы в k -мерные.

- **Функция активации (activation function)** — нелинейное преобразование, поэлементно применяющееся к пришедшему на вход данным. Благодаря функциям активации нейронные сети способны порождать более информативные признаковые описания, преобразуя данные нелинейным образом. Может использоваться, например, ReLU (rectified linear unit) $\text{ReLU}(x) = \max(0, x)$ или уже знакомая из логистической регрессии сигмоида $\sigma(x) = \frac{1}{1+e^{-x}}$. К более глубокому рассмотрению разновидностей и свойств различных функций активации вернёмся позднее.

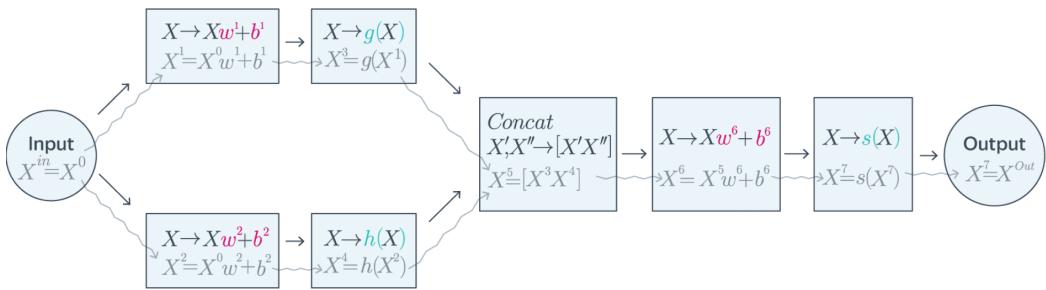
Даже самые сложные нейронные сети обычно собираются из относительно простых блоков, подобных этим. Таким образом, их можно представить в виде **вычислительного графа (computational graph)**, где вершинам промежуточным соответствуют преобразования. На иллюстрации ниже приведён вычислительный граф для логистической регрессии.



Применение



Не правда ли, похоже на слоёный пирог из преобразований? Отсюда и слои.
Графы могут быть и более сложными, в том числе нелинейными:

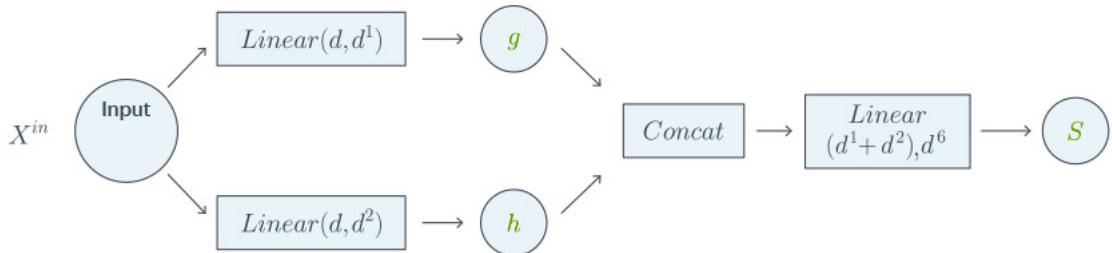


Давайте разберёмся, что тут происходит.

1. Input — это **вход** нейросети, который получает исходные данные. Обычно требуется, чтобы они имели вид матрицы («объекты-признаки») или тензора (многомерной матрицы). Вообще говоря, входов может быть несколько: например, мы можем подавать в нейросеть картинку и какие-нибудь ещё сведения о ней — преобразовывать их мы будем по-разному, поэтому логично предусмотреть два входа в графе.
2. Дальше к исходным данным X^0 применяются два линейных слоя, которые превращают их в **промежуточные (внутренние, скрытые) представления** X^1 и X^2 . В литературе они также называются **активациями** (не путайте с функциями активации).
3. Каждое из представлений X^1 и X^2 подвергается нелинейному преобразованию, превращаясь в новые промежуточные представления X^3 и X^4 соответственно. Переход от X^0 к двум новым матрицам (или тензорам) X^3 и X^4 можно рассматривать как построение двух новых (возможно, более информативных) признаковых описаний исходных данных.
4. Затем представления X^3 и X^4 конкатенируются (то есть признаковые описания всех объектов объединяются).
5. Дальше следует ещё один линейный слой и ещё одна активация, и полученный результат попадает на **выход** сети, то есть отдаётся обратно пользователю.

Нейросеть, в которой есть только линейные слои и различные функции активации, называют **полносвязанной (fully connected)** нейронной сетью или **многослойным перцептроном (multilayer perceptron, MLP)**.

Посмотрим, что происходит с размерностями, если на вход подаётся матрица $N \times d$:

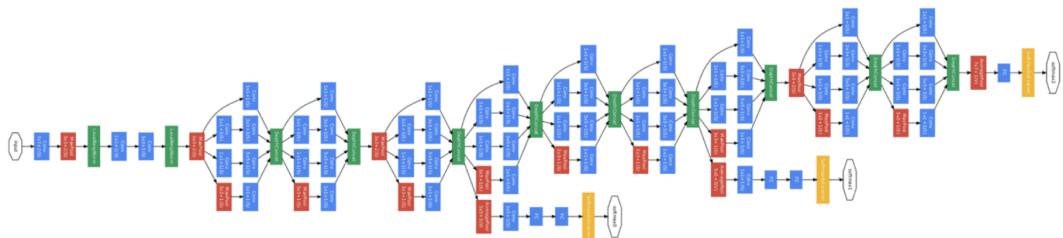


Примечание о терминологии

В литературе, увы, нет единства терминологии. Так, например, никто не мешает нам объявить «единичным и неделимым слоем» композицию линейного слоя и активации (в ознаменование того, что мы почти

никогда не используем просто линейный слой без нелинейности). Например, в фреймворке `keras` активацию можно указать в линейном слое в качестве параметра.

А вот и настоящий пример из реальной жизни. GoogLeNet (она же Inception-v1), показавшая SotA-результат на ILSVRC 2014 (ImageNet challenge), выглядит так:

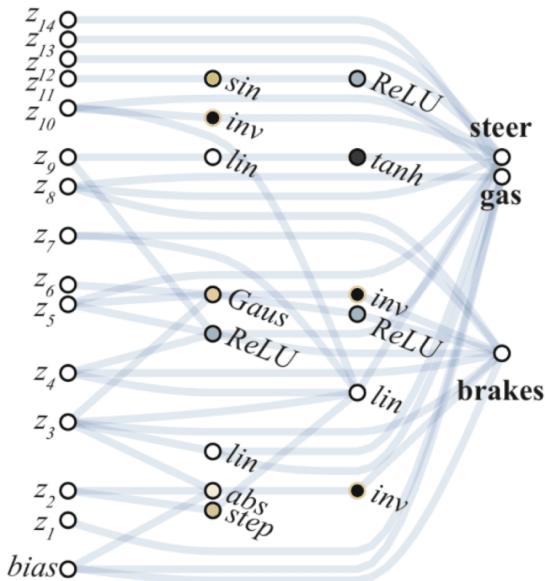


Здесь каждый кирпичик — это некоторое относительно простое преобразование, а белым помечены входы и выходы вычислительного графа.

Современные же сети часто выглядят и ещё сложней, но всё равно они собираются из достаточно простых кирпичиков-слоёв.

Примечание

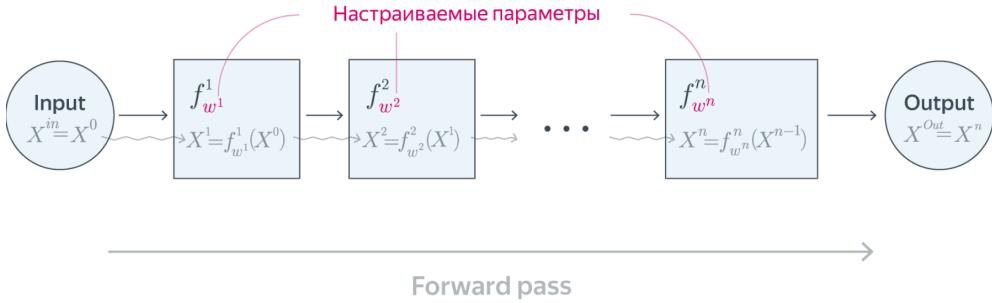
Стоит отметить, впрочем, что в общем случае нейронная сеть представляет собой просто некоторую сложную функцию (или, что эквивалентно, граф вычислений), и в некоторых (очень нетривиальных) случаях её нет смысла разбивать на слои. В качестве иллюстрации ниже приведены структуры агностических нейронных сетей WANN, представленных в работе [Weight Agnostic Neural Networks, NeurIPS 2019](<https://weightagnostic.github.io>).



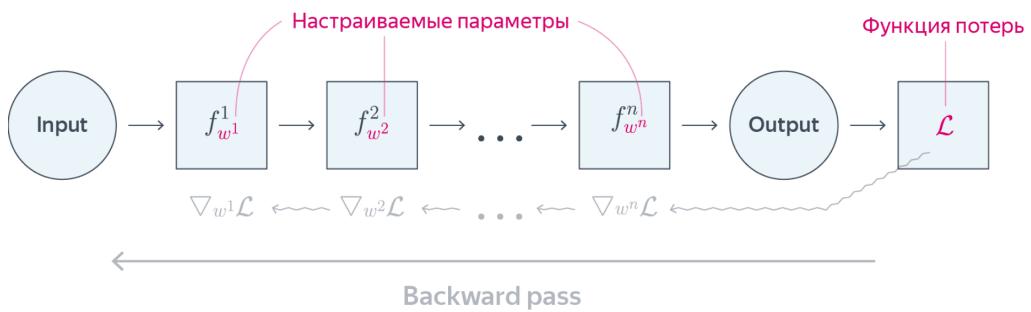
4.1.2. Forward backward propagation

Информация может течь по графу в двух направлениях.

Применение нейронной сети к данным (вычисление выхода по заданному входу) часто называют **прямым проходом**, или же **forward propagation (forward pass)**. На этом этапе происходит преобразование исходного представления данных в целевое и последовательно строятся промежуточные (внутренние) представления данных — результаты применения слоёв к предыдущим представлениям. Именно поэтому проход называют прямым.



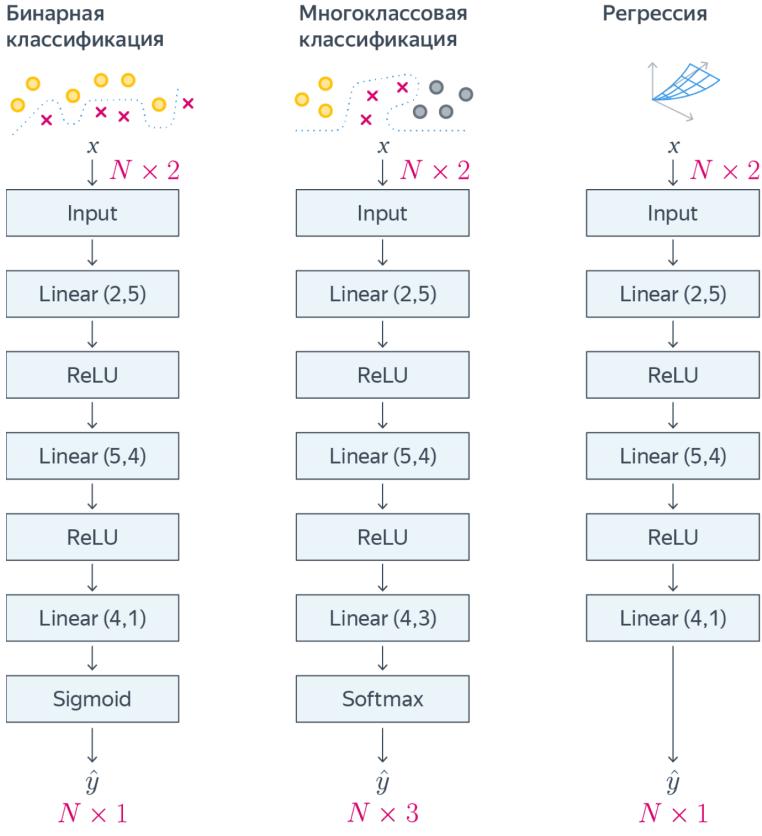
При **обратном проходе**, или же **backward propagation (backward pass)**, информация (обычно об ошибке предсказания целевого представления) движется от финального представления (а чаще даже от функции потерь) к исходному через все преобразования. Механизм обратного распространения ошибки, играющий важнейшую роль в обучении нейронных сетей, как раз предполагает обратное движение по вычислительному графу сети (с ним вы познакомитесь в следующей главе).



4.1.3. Архитектуры для простейших задач

Как мы уже упоминали выше, нейросети — это универсальный конструктор, который из простых блоков позволяет собрать орудия для решения самых разных задач. Давайте посмотрим на конкретные примеры. Безусловно, мир намного разнообразнее того, что мы покажем вам в этой главе, но с чего-то ведь надо начинать, не так ли?

В тех несложных ситуациях, которые мы сейчас рассмотрим, архитектура будет отличаться лишь самыми последними этапами вычисления (у сетей будут разные «головы»). Для иллюстрации приведём примеры нескольких игрушечных архитектур для решения игрушечных задач классификации и регрессии на двумерных данных:



4.1.3.1. Бинарная классификация. Для решения задачи бинарной классификации подойдёт любая архитектура, на выходе у которой одно число от 0 до 1, интерпретируемое как «вероятность класса 1». Обычно этого добиваются, взяв

$$\hat{y} = \sigma(f(X^m))$$

где f - некоторая функция, превращающая представление X^m в число (если X^m — матрица, то подойдёт $f(X^m) = X^m\omega + b$, где ω - вектор столбец), а σ наша любимая сигмоида. При этом X^m может получаться как угодно, лишь бы хватало оперативной памяти и не было переобучения.

В качестве функции потерь удобно брать уже знакомый нам log loss.

4.1.3.2. Многоклассовая классификация. Работая с другими моделями, мы порой вынуждены были выдумывать сложные стратегии многоклассовой классификации; нейросети позволяют это делать легко и элегантно. Достаточно построить сеть, которая будет выдавать K неотрицательных чисел, суммирующихся в 1 (где K — число классов); тогда им можно придать смысл вероятностей классов и предсказывать тот класс, «вероятность» которого максимальна (в главе про вероятностные модели мы обсудим, почему это вовсе не обязаны быть настоящие вероятности). Превратить произвольный набор из K чисел в набор из неотрицательных чисел, суммирующихся в 1, позволяет, к примеру, функция

$$softmax(x_1, \dots, x_K) = \left(\frac{e^{x_1}}{\sum_{i=1}^K e^{x_i}}, \dots, \frac{e^{x_K}}{\sum_{i=1}^K e^{x_i}} \right)$$

Наиболее популярные архитектуры для многоклассовой классификации имеют вид

$$\hat{y} = softmax(f(X^m))$$

где f - функция, превращающая X^m в матрицу $B \times K$ (где B - размер батча), а X^m может быть получен любым приятным образом.

Но какой будет функция потерь для такой сети? Мы должны научиться сравнивать «распределение вероятностей классов» с истинным (в котором на месте истинного класса стоит 1, а в остальных местах 0).

Сделать это позволяет **кросс-энтропия** (она же **negative log-likelihood**), которая является некоторым аналогом расстояния между распределениями:

$$L(\hat{y}, y) = -\frac{1}{B} \sum_{i=1}^B \sum_{k=1}^K y_{ik} \log \hat{y}_{ik}$$

где снова B - размер батча, а K - число классов. Легко видеть, что при $k = 2$ получается та же самая функция потерь, которую мы использовали для обучения бинарной классификации.

4.1.3.3. (Множественная) регрессия. С помощью нейросетей легко создать модель, которая предсказывает не одно число, а сразу несколько (например, координаты ключевых точек лица — кончика носа, уголков рта и так далее — по фотографии). Достаточно сделать, чтобы последнее представление было матрицей $B \times M$, где B - размер батча, а M - количество предсказываемых чисел. Особенностью большинства моделей регрессии является то, что после последнего слоя (часто линейного) не ставят функций активации. Вы тоже этого не делайте, если только чётко не понимаете, зачем вам это. В качестве функции потерь можно брать, например, MSE по всей матрице $B \times M$

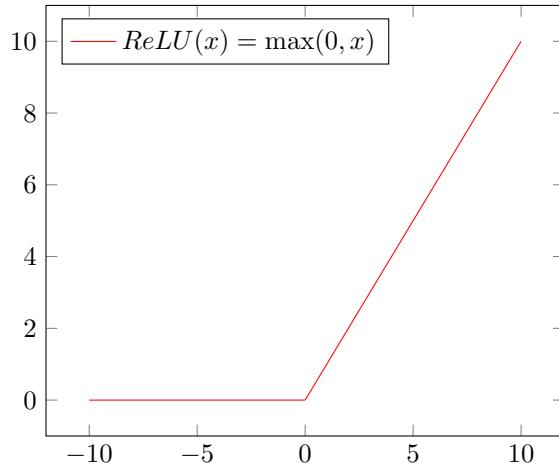
4.1.3.4. Всё вместе. Если вы используете нейросети, то ваши таргеты могут иметь и различную природу. Например, можно соорудить одну-единственную сеть, которая по фотографии нескольких котиков определяет их количество (регрессия) и породу каждого из них (многоклассовая классификация). Лосс для такой модели может быть равен (взвешенной) сумме лоссов для каждой из задач (правда, не факт, что это хорошая идея). Так что, по крайней мере в теории, сетям подвластны любые задачи. На практике, конечно, всё гораздо хитрее: для обучения слишком сложной сети у вас может не хватить данных или вычислительных мощностей.

4.1.4. Популярные функции активации

4.1.4.1. ReLU. Rectified linear unit

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

$$ReLU(x) : \mathbb{R} \mapsto [0; +\infty]$$



ReLU представляет собой простую кусочно-линейную функцию. Одна из наиболее популярных функций активации. В нуле производная доопределется нулевым значением (*На первой лекции Ильдуса говорили, что там нет производной*).

Минусы:

- область значений является смешённой относительно нуля
- для отрицательных значений производная равна нулю, что может привести к затуханию градиента

Плюсы:

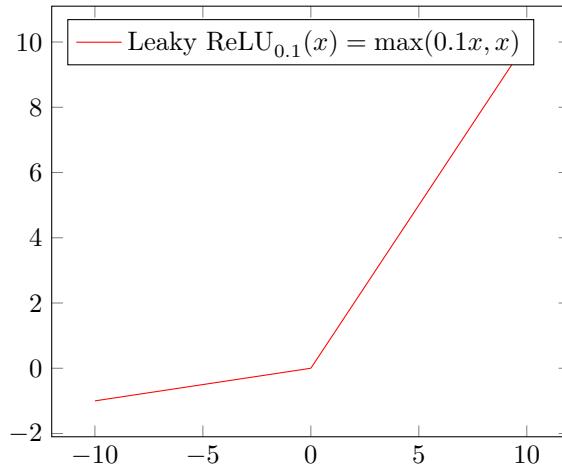
- простота вычисления активации и производной

ReLU и её производная очень просты для вычисления: достаточно лишь сравнить значение с нулём. Благодаря этому использование ReLU позволяет достигать прироста в скорости до четырёх-шести раз относительно сигмоиды.

4.1.4.2. Leaky ReLU.

$$\text{Leaky ReLU}(x) = \max(\alpha x, x), \quad \alpha = \text{const}, \quad 0 < \alpha \ll 1$$

$$\text{Leaky ReLU} : \mathbb{R} \mapsto (-\infty, +\infty)$$



Гиперпараметр α обеспечивает небольшой уклон слева от нуля, что позволяет получить более симметричную относительно нуля область значений. Также меньше провоцирует затухание градиента благодаря наличию ненулевого градиента и слева, и справа от нуля.

4.1.4.3. PReLU. Parametric ReLU

$$PReLU(x) = \max(\alpha x, x), \quad 0 < \alpha \ll 1$$

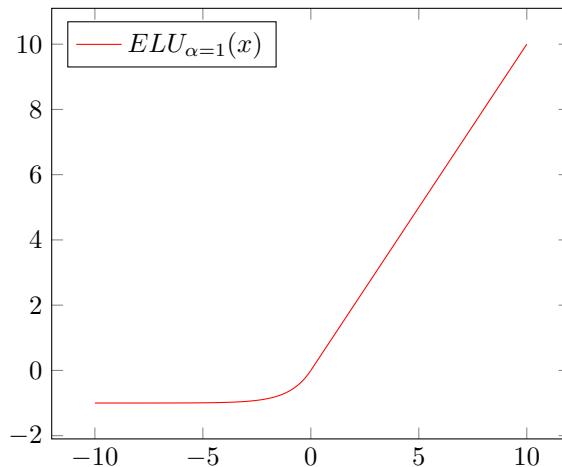
$$PReLU(x) : \mathbb{R} \mapsto (-\infty, +\infty)$$

Аналогична Leaky ReLU, но параметр α настраивается градиентными методами.

4.1.4.4. ELU. Гладкая аппроксимация ReLU. Обладает более высокой вычислительной сложностью, достаточно редко используется на практике.

$$ELU(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}, \quad \alpha = \text{const}, \quad 0 < \alpha$$

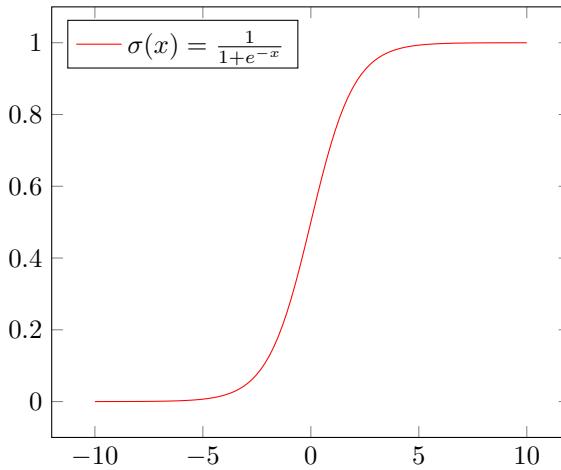
$$ELU : \mathbb{R} \mapsto (-\alpha; +\infty)$$



4.1.4.5. Sigmoid. Сигмоида

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

$$\sigma : \mathbb{R} \mapsto (0, 1)$$



Исторически одна из первых функций активации. Рассматривалась в том числе и как гладкая аппроксимация порогового правила, эмулирующая активацию естественного нейрона.

К минусам сигмоиды можно отнести:

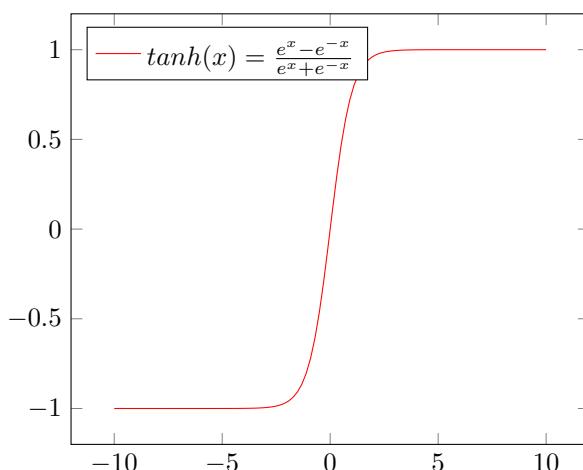
- область значений смещена относительно нуля
- сигмоида (как и её производная) требует вычисления экспоненты, что является достаточно сложной вычислительной операцией. Её приближённое значение вычисляется на основе ряда Тейлора или с помощью полиномов, Stack Overflow question 1, question 2
- на «хвостах» обладает практически нулевой производной, что может привести к затуханию градиента
- максимальное значение производной составляет 0.25, что также приводит к затуханию градиента

На практике редко используется внутри сетей, чаще всего в случаях, когда внутри модели решается задача бинарной классификации (например, вероятность забывания информации в LSTM).

4.1.4.6. Tanh. Гиперболический тангенс

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

$$\tanh : \mathbb{R} \mapsto (-1, 1)$$



Плюсы:

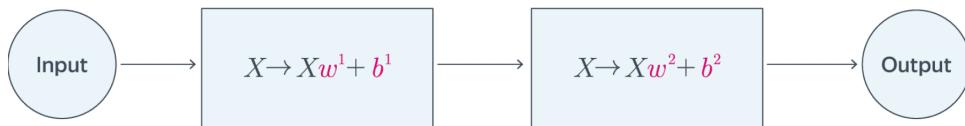
- как и сигмоида, имеет ограниченную область значений
- в отличие от сигмоиды, область значений симметрична

Минусы:

- требует вычисления экспоненты, что является достаточно сложной вычислительной операцией
- на «хвостах» обладает практически нулевой производной, что может привести к затуханию градиента

4.1.5. Зачем нужны функции активации?

Казалось бы, можно последовательно выстраивать лишь линейные слои, но так не делают: после каждого линейного слоя обязательно вставляют функцию активации. Но зачем? Попробуем разобраться. Рассмотрим нейронную сеть из двух линейных слоёв. Что произойдёт, если между ними будет отсутствовать нелинейная функция активации?



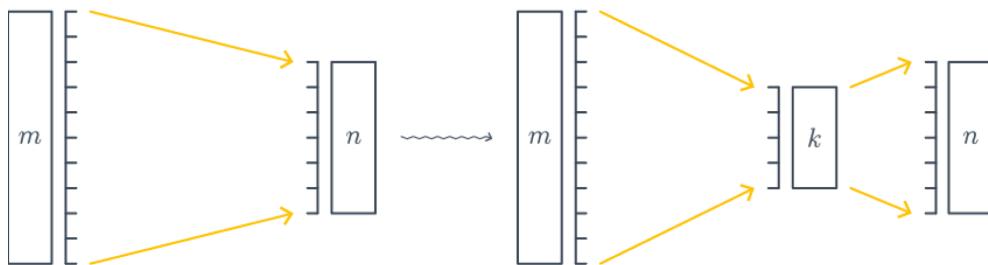
$$\hat{y} = X_{out} = X_1 W_2 + b_2 = (X_0 W_1 + b_1) W_2 + b_2 = X_0 W_1 W_2 + b_1 W_2 + b_2 = X_0 \tilde{W} + \tilde{b}$$

Линейная комбинация линейных отображений есть линейное отображение, то есть два последовательных линейных слоя эквивалентны одному линейному слою.

Добавление функций активации после линейного слоя позволяет получить нелинейное преобразование, и подобной проблемы уже не возникает. Вдобавок правильный выбор функции активации позволяет получить преобразование, обладающее подходящими свойствами.

В качестве функции активации может использоваться, например, уже знакомая вам из логистической регрессии сигмоида или ReLU. К более глубокому рассмотрению разновидностей и свойств различных функций активации вернёмся позднее.

4.1.5.1. Примечание. На самом деле бывают ситуации, когда два линейных слоя подряд — это полезно. Например, если вы понимаете, что у вас очень много параметров, а информации в данных не так много, вы можете заменить линейный слой, превращающий m -мерные векторы в n -мерные, на два, вставив посередине k -мерное представление, где $k \ll m, n$



С точки зрения линейной алгебры это примерно то же самое, что потребовать, чтобы матрица исходного линейного слоя имела ранг не выше k . И с точки зрения сужения «информационного канала» это иногда может сработать. Но в любом случае вы должны понимать, что два линейных слоя подряд стоит ставить, только если вы хорошо понимаете, чего хотите добиться.

4.1.6. Немного о мощи нейросетей

Рассмотрим для начала задачу регрессии. Ясно, что линейная модель (то есть однослойная нейросеть) может приблизить только линейную функцию, но уже двуслойная нейросеть может приблизить почти что угодно. Есть ряд теорем на эту тему, мы упомянем одну из них (обратите внимание на год: как мы уже упоминали, нейросети начали серьёзно изучать задолго до того, как они начали превращаться в state of the art).

4.1.6.1. Теорема Цыбенко (1989). Для любой непрерывной функции $f(x) : \mathbb{R}^m \mapsto \mathbb{R}$ и для любого $\varepsilon > 0$ найдётся число N , а также числа $\omega_1, \dots, \omega_N, b_1, \dots, b_N, \alpha_1, \dots, \alpha_n$, для которых

$$\left| f(x) - \sum_{i=1}^N \alpha_i \sigma(\langle x, w_i \rangle + b_i) \right| < \varepsilon$$

для любых x из единичного куба $[0, 1]^m$ в \mathbb{R}^m .

В сумме из теоремы Цыбенко легко опознать двуслойную нейросеть с сигмоидной функцией активации. В самом деле, сперва мы превращаем x в $\langle x, w_i \rangle + b_i$ - это можно представить в виде одной матричной операции (линейный слой!):

$$x \mapsto x^{(1)} = x \cdot (\omega_1 \dots \omega_N) + (b_1 \dots b_N)$$

где ω_i - вектор-столбцы, а каждое из b_i прибавляется к i -му столбцу, после чего поэлементно берём от $x^{(1)}$ сигмоиду (активацию) $x^{(2)} = \sigma(x^{(1)})$, после чего вычисляем

$$\sum_{i=1}^N \alpha_i x_i^{(2)} = (\alpha_1, \dots, \alpha_N) \cdot x$$

и это второй линейный слой (без свободного члена).

Правда, теорема не очень помогает находить такие функции, но это уже другое дело. В любом случае — если дать нейросети достаточно данных, она действительно может выучить почти что угодно.

4.2. Метод обратного распространения ошибки

Нейронные сети обучаются с помощью тех или иных модификаций градиентного спуска, а чтобы применять его, нужно уметь эффективно вычислять градиенты функции потерь по всем обучающим параметрам. Казалось бы, для какого-нибудь запутанного вычислительного графа это может быть очень сложной задачей, но на помощь спешит метод обратного распространения ошибки.

Открытие метода обратного распространения ошибки стало одним из наиболее значимых событий в области искусственного интеллекта. В актуальном виде он был предложен в 1986 году Дэвидом Э. Румельхартом, Джоном Э. Хинтоном и Рональдом Дж. Вильямсом и независимо и одновременно красноярскими математиками С. И. Барцевым и В. А. Охониным. С тех пор для нахождения градиентов параметров нейронной сети используется метод вычисления производной сложной функции, и оценка градиентов параметров сети стала хоть сложной инженерной задачей, но уже не искусством. Несмотря на простоту используемого математического аппарата, появление этого метода привело к значительному скачку в развитии искусственных нейронных сетей.

Суть метода можно записать одной формулой, тривиально следующей из формулы производной сложной функции: если $f(x) = g_m(g_{m-1}(\dots(g_1(x))\dots))$, то $\frac{\partial f}{\partial x} = \frac{\partial g_m}{\partial g_{m-1}} \frac{\partial g_{m-1}}{\partial g_{m-2}} \dots \frac{\partial g_2}{\partial g_1} \frac{\partial g_1}{\partial x}$. Уже сейчас мы видим, что градиенты можно вычислять последовательно, в ходе одного обратного прохода, начиная с $\frac{\partial g_m}{\partial g_{m-1}}$ и умножая каждый раз на частные производные предыдущего слоя.

4.2.1. Backpropagation в одномерном случае

В одномерном случае всё выглядит особенно просто. Пусть ω_0 - переменная, по которой мы хотим про-дифференцировать, причём сложная функция имеет вид

$$f(\omega_0) = g_m(g_{m-1}(\dots g_1(\omega_0)\dots))$$

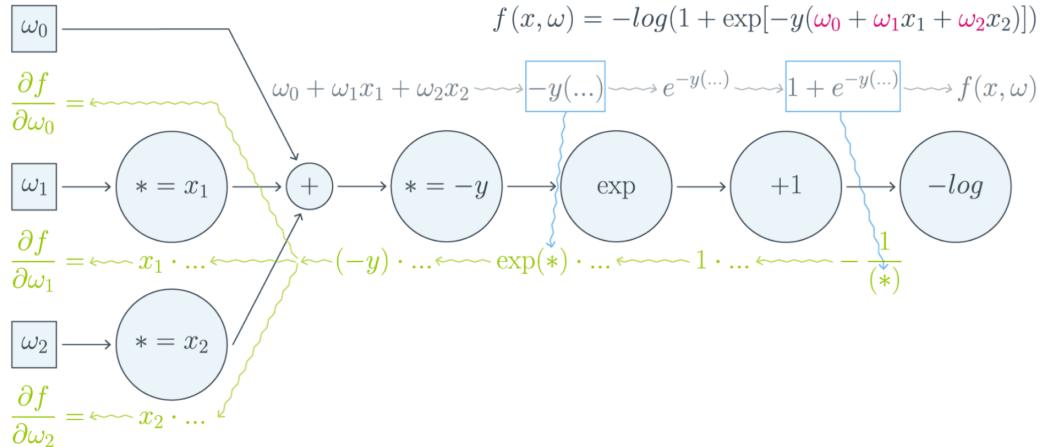
где g_i - скалярные. Тогда

$$f'(\omega_0) = g'_m(g_{m-1}(\dots g_1(\omega_0)\dots)) \cdot g'_{m-1}(g_{m-2}(\dots g_1(\omega_0)\dots)) \cdot \dots \cdot g'_1(\omega_0)$$

Суть этой формулы такова. Если мы уже совершили forward pass, то есть уже знаем $g_1(\omega_0), g_2(g_1(\omega_0)), \dots, g_{m-1}(\dots g_1(\omega_0)\dots)$, то мы действуем следующим образом:

1. Берём производную g_m в точке $g_{m-1}(\dots g_1(\omega_0) \dots)$
2. Умножаем на производную g_{m-1} в точке $g_{m-2}(\dots g_1(\omega_0) \dots)$
3. И так далее, пока не дойдём до производной g_1 в точке ω_0

Проиллюстрируем это на картинке, расписав по шагам дифференцирование по весам ω_i функции потерь логистической регрессии на одном объекте (то есть для батча размера 1):



Собирая все множители вместе, получаем:

- $\frac{\partial f}{\partial \omega_0} = (-y) \cdot e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)} \cdot \frac{-1}{1 + e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)}}$
- $\frac{\partial f}{\partial \omega_1} = x_1(-y) \cdot e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)} \cdot \frac{-1}{1 + e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)}}$
- $\frac{\partial f}{\partial \omega_2} = x_2(-y) \cdot e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)} \cdot \frac{-1}{1 + e^{-y(\omega_0 + \omega_1 x_1 + \omega_2 x_2)}}$

Таким образом, мы видим, что сперва совершается forward pass для вычисления всех промежуточных значений (и да, **все промежуточные представления нужно будет хранить в памяти**), а потом запускается backward pass, на котором в один проход вычисляются все градиенты.

4.2.2. Почему же нельзя просто пойти и начать везде вычислять производные?

В главе, посвящённой матричным дифференцированиям, мы поднимаем вопрос о том, что вычислять частные производные по отдельности — это зло, лучше пользоваться матричными вычислениями. Но есть и ещё одна причина: даже и с матричной производной в принципе не всегда хочется иметь дело. Рассмотрим простой пример. Допустим, что X^r и X^{r+1} — два последовательных промежуточных представления $N \times M$ и $N \times K$, связанных функцией $X^{r+1} = f^{r+1}(X^r)$. Предположим, что мы как-то посчитали производную $\frac{\partial L}{\partial X_{ij}^{r+1}}$ функции потерь L , тогда

$$\frac{\partial L}{\partial X_{st}^r} = \sum_{i=1}^N \sum_{j=1}^K \frac{\partial f_{ij}^{r+1}}{\partial X_{st}^r} \frac{\partial L}{\partial X_{ij}^{r+1}}$$

И мы видим, что оба градиента $\frac{\partial L}{\partial X_{ij}^{r+1}}$ и $\frac{\partial f_{ij}^{r+1}}{\partial X_{st}^r}$ являются просто матрицами, в ходе вычислений возникает «четырёхмерный кубик» $\frac{\partial f_{ij}^{r+1}}{\partial X_{st}^r}$, даже хранить который весьма болезненно: уж больно много памяти он требует ($N^2 MK$ по сравнению с безобидными $NM + NK$), требуемыми для хранения градиентов). Поэтому хочется промежуточные производные $\frac{\partial f_{ij}^{r+1}}{\partial X_{st}^r}$ рассматривать не как вычисляемые объекты $\frac{\partial f_{ij}^{r+1}}{\partial X_{st}^r}$, а как преобразования, которые превращают $\frac{\partial L}{\partial X_{ij}^{r+1}}$ в $\frac{\partial L}{\partial X_{st}^r}$. Целью следующих глав будет именно это: понять, как преобразуется градиент в ходе error backpropagation при переходе через тот или иной слой.

4.2.3. Градиент сложной функции

Напомним, что формула производной сложной функции выглядит следующим образом:

$$[D_{x_0}(u \circ v)](h) = [D_{v(x_0)}u]([D_x v](h))$$

Теперь разберёмся с градиентами. Пусть $f(x) = g(h(x))$ — скалярная функция. Тогда

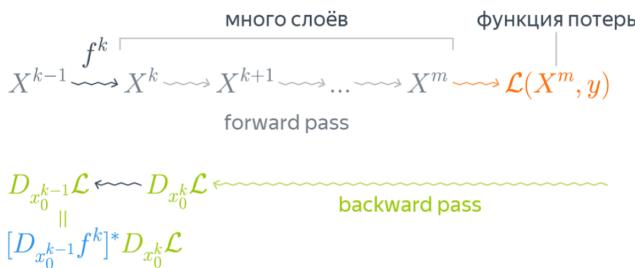
$$[D_{x_0}f](x - x_0) = \langle \nabla_{x_0} f, x - x_0 \rangle$$

С другой стороны,

$$[D_{h(x_0)}g]([D_{x_0}h](x - x_0)) = \langle \nabla_{h(x_0)}g, [D_{x_0}h](x - x_0) \rangle = \langle [D_{x_0}h] \cdot \nabla_{h(x_0)}g, x - x_0 \rangle$$

То есть $\nabla_{x_0}f = [D_{x_0}h] \cdot \nabla_{h(x_0)}g$ — применение сопряжённого к $D_{x_0}h$ линейного отображения к вектору $\nabla_{h(x_0)}g$.

Эта формула — сердце механизма обратного распространения ошибки. Она говорит следующее: если мы каким-то образом получили градиент функции потерь по переменным из некоторого промежуточного представления X^k нейронной сети и при этом знаем, как преобразуется градиент при проходе через слой f^k между X^{k-1} и X^k (то есть как выглядит сопряжённое к дифференциалу слоя между ними отображение), то мы сразу же находим градиент и по переменным из X^{k-1} :



Таким образом слой за слоем мы посчитаем градиенты по всем X^i вплоть до самых первых слоёв.

Далее мы разберёмся, как именно преобразуются градиенты при переходе через некоторые распространённые слои.

ЗАТЕХАТЬ РАЗБОР ПРИМЕРА

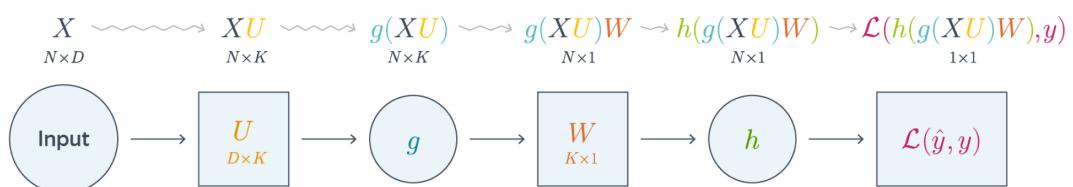
4.2.4. Backpropagation в общем виде

Подытожим предыдущее обсуждение, описав алгоритм **error backpropagation** (**алгоритм обратного распространения ошибки**). Допустим, у нас есть текущие значения весов W_0^i и мы хотим совершить шаг SGD по мини-батчу X . Мы должны сделать следующее:

1. Совершить forward pass, вычислив и запомнив все промежуточные представления $X = X_0, X_1, \dots, X_m = \hat{y}$
2. Вычислить все градиенты с помощью backward pass.
3. С помощью полученных градиентов совершить шаг SGD.

Проиллюстрируем алгоритм на примере двуслойной нейронной сети со скалярным output'ом. Для простоты опустим свободные члены в линейных слоях.

Рассмотрим простую нейросеть



Обучаемые параметры – матрицы U и W . Как найти градиенты по ним в точке U_0 и W_0 ?

$$\nabla_{\omega_0} L = \nabla_{\omega_0} (L \circ h \circ [W \mapsto g(XU_0)W]) = g(XU_0)^T \nabla_{g(XU_0)W_0} (L \circ h) = \underbrace{g(XU_0)^T}_{k \times N} \cdot \left[\underbrace{h'(g(XU_0)W_0)}_{N \times 1} \circ \underbrace{\nabla_{h(g(XU_0)W_0)} L}_{N \times 1} \right]$$

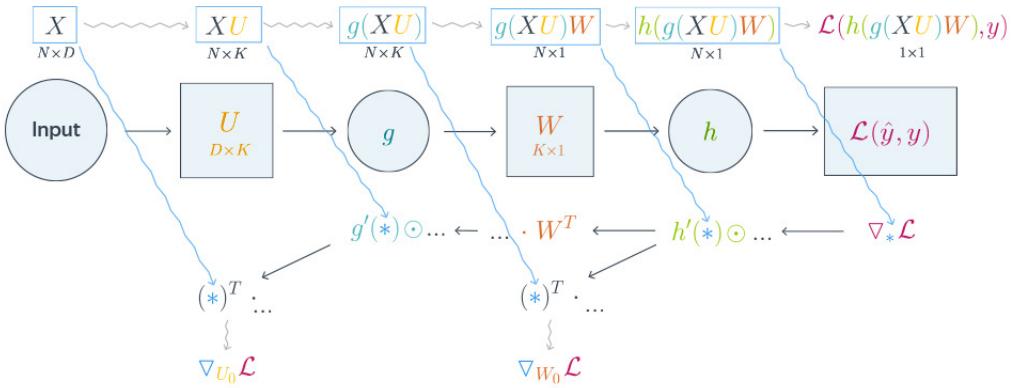
Итого матрица $k \times 1$, как и W_0

$$\nabla_{U_0} L = \nabla_{U_0} (L \circ h \circ [Y \mapsto YW_0] \circ g \circ [U \mapsto XU]) = X^T \cdot \nabla_{XU_0} (L \circ h \circ [YYW_0] \circ g) = \dots =$$

$$= X^T \cdot \left(\underbrace{g'XU_0}_{N \times K} \circ \left[\underbrace{h'(g(XU_0)W_0)}_{N \times 1} \circ \underbrace{\nabla_{h(g(XU_0)W_0)} L}_{N \times 1} \right] \cdot \underbrace{W^T}_{1 \times K} \right)$$

Итого $D \times K$, как и U_0

Схематически это можно представить следующим образом:



4.2.5. Backpropagation для двуслойной нейронной сети. Автоматизация и autograd

Итак, чтобы нейросеть обучалась, достаточно для любого слоя $f^k : X^{k-1} \mapsto X^k$ с параметрами W^k уметь:

- превращать $\nabla_{X_0^k} L$ в $\nabla_{X_0^{k-1}} L$ (градиент по выходу в градиент по входу)
- считать градиент по его параметрам $\nabla_{W_0^k} L$

При этом слою совершенно не надо знать, что происходит вокруг. То есть слой действительно может быть запрограммирован как отдельная сущность, умеющая внутри себя делать forward pass и backward pass, после чего слои механически, как кубики в конструкторе, собираются в большую сеть, которая сможет работать как одно целое.

Более того, во многих случаях авторы библиотек для глубинного обучения уже о вас позаботились и создали средства для **автоматического дифференцирования выражений (autograd)**. Поэтому, программируя нейросеть, вы почти всегда можете думать только о forward-проходе, прямом преобразовании данных, предоставив библиотеке дифференцировать всё самостоятельно. Это делает код нейросетей весьма понятным и выразительным (да, в реальности он тоже бывает большим и страшным, но сравните на досуге код какой-нибудь разухабистой нейросети и код градиентного бустинга на решающих деревьях и почувствуйте разницу).

4.3. Тонкости обучения

Если открыть случайную научную статью по глубинному обучению и попробовать воспроизвести её результаты, можно запросто потерпеть крах, и даже код на github, если он есть, может не помочь. А дело в том, что обучение сложной модели — это сложная инженерная задача, в которой успеху сопутствует огромное число разных хаков, и изменение какого-нибудь безобидного параметра может очень сильно повлиять на результат. В этой главе мы познакомим вас с некоторыми из таких приёмов.

4.3.1. Инициализируем правильно

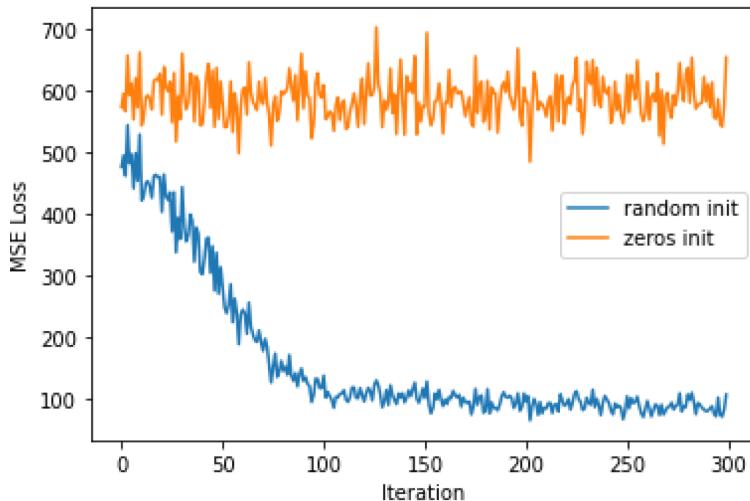
Как вы уже успели заметить, нейронные сети – достаточно сложные модели, чувствительные к изменениям архитектуры, гиперпараметров, распределения данных и пр. Поэтому значительную роль играет начальная инициализация весов вашей сети. Стоит отметить, что здесь речь идет именно о начальной

инициализации параметров сети, вопрос дообучения (и использования предобученных сетей в качестве backbone) в данной главе рассматриваться не будет.

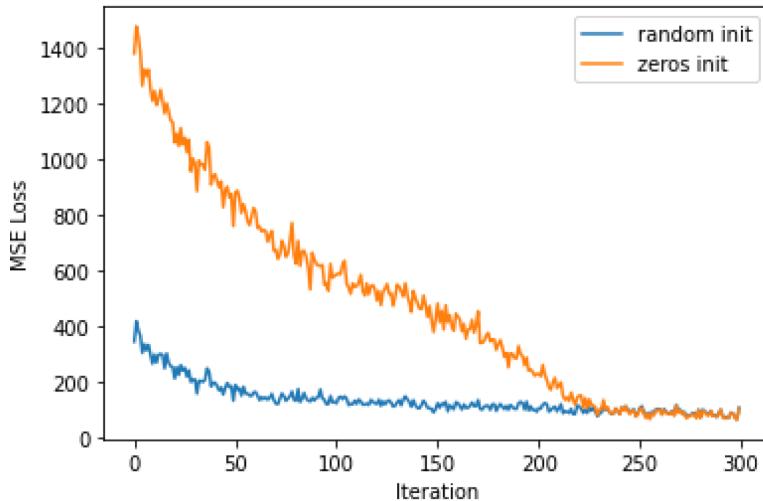
Нейронные сети включают в себя различные преобразования, и инициализация по-хорошему также должна зависеть от типа используемого преобразования. На практике вопрос часто остается без внимания, так как в большинстве современных фреймворков уже реализованы методы инициализации, зависящие от используемой функции активации и гиперпараметров слоя, и пользователь может не задумываться об этом. Но всё же важно понимать, какие соображения привели к появлению тех или иных стратегий инициализации.

Давайте разберём несколько методов инициализации и обсудим их свойства.

4.3.1.1. Наивный подход №0: инициализация нулем/константой. Казалось бы, инициализация параметров слоя нулями – это достаточно просто и лаконично. Но инициализация нулём (как и любой другой константой) ведёт к катастрофе! Вот пример того, что может получиться:



Стоит, впрочем, отметить, что из-за численных ошибок значения параметров могут всё-таки сдвинуться с мёртвой точки, и тогда нейросеть что-нибудь выучит:



Математическая иллюстрация того, почему плохо инициализировать нулями:

Нам понадобится сеть с хотя бы двумя слоями (иначе ничего не получится).

Рассмотрим несколько последовательных скрытых представлений:

1. X_1
2. $X_2 = X_1W$, где W - нулевая матрица
3. $X_3 = h(X_2)$, где h - поэлементная нелинейность
4. $X_4 = X_3U$, где U - нулевая матрица

Проследим, что происходит во время forward pass.

$$X_2 = X_1 \times W$$

Получилась снова матрица с одинаковыми столбцами, и это сохраняется дальше:

$$X_3 = h(0)$$

$$X_4 = X_3 U = 0$$

Теперь рассмотрим обратный проход. Допустим, мы вычислили градиент $\nabla_{X_4} L$. Тогда:

$$\nabla_U L = X_3^T \nabla_{X_4} L = 0$$

То есть матрица U никак не обновится. Далее:

$$\nabla_{X_3} L = \nabla_{X_4} L U^T = 0$$

$$\nabla_{X_2} L = \nabla_{X_3} L \circ h'(X_2)$$

$$\nabla_W L = X_1^T \nabla_{X_2} L = 0$$

То есть веса W тоже не обновятся. Таким образом, обучение происходить не будет.

А что же будет с однослоиной нейросетью? Вспомним, что веса логистической регрессии (нейросети с одним-единственным, последним слоем) можно инициализировать чем угодно, в том числе нулями: функция потерь выпукла, поэтому градиентный спуск сходится из любой точки (по модулю численных эффектов).

Здесь также стоит привести цитату из замечательной Deep Learning book (страница 301):

“Perhaps the only property known with complete certainty is that the initial parameters need to “break symmetry” between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters. If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way.”

4.3.1.2. Эвристический подход №1: инициализация случайными числами. Если константная инициализация не подходит, можно инициализировать случайными числами. Допустим, веса пришли из распределения с нулевым средним и дисперсией σ^2 , например из нормального распределения $\mathcal{N}(0, \sigma^2)$

Пусть теперь на вход линейному слою с весами ω размерности n_{in} пришёл вектор x аналогичной размерности.

Замечание. Можем считать, что мы рассматриваем лишь одну компоненту следующего промежуточного представления z .

Все компоненты x распределены одинаковым образом и обладают нулевым средним. Тогда дисперсия их произведения y имеет вид:

$$Var(y) = Var(\omega^t x) = \sum_{i=1}^n [\mathbb{E}[x_i]^2 Var(\omega_i) + \mathbb{E}[x_i]^2 Var(x_i) + Var(\omega_i) Var(x)]$$

Первое и второе слагаемые равны нулю так как математическое ожидание и весов, и значений x равны нулю. Стоит заметить, что это будет верно и для промежуточных слоев в случае использования симметричной относительно нуля функции активации, например, \tanh . Поскольку все веса пришли из одного распределения, можно выразить дисперсию результата следующим образом:

$$Var(W^t x) = n_{in} Var(\omega) Var(x)$$

где $Var(X)$ - это дисперсия любой компоненты x (как было оговорено ранее, они распределены одинаково), а $Var(\omega) = \sigma^2$ – дисперсия компоненты w .

Следовательно, дисперсия результата линейно зависит от дисперсии входных данных с коэффициентом $n_{in} Var(\omega)$.

Увеличение дисперсии промежуточных представлений с каждым новым преобразованием (слоем) может вызвать численные ошибки или насыщение функций активации (таких как \tanh и sigmoid), что не лучшим образом скажется на обучении сети.

Снижение дисперсии может привести к почти нулевым промежуточным представлениям (плюс «линейному» поведению \tanh и sigmoid в непосредственной близости от нуля), что тоже негативно повлияет на результаты обучения.

Поэтому для начальной инициализации весов имеет смысл использовать распределение, дисперсия которого позволила бы сохранить дисперсию результата. Например, $\forall i, \omega_i \sim \mathcal{N}(0, \frac{1}{n_{in}})$, или же в общем случае

$$\forall i, Var(\omega_i) = \frac{1}{n_{in}}$$

Данный подход часто упоминается как **calibrated random numbers initialization**.

4.3.1.3. Подход №2: Xavier and Normalized Xavier initialization. Если обратиться к предыдущему подходу, можно обнаружить, что все выкладки верны как для «прямого» прохода (forward pass), так и для обратного (backward pass). Дисперсия градиента при этом меняется в $n_{out}Var(\omega)$ раз, где n_{out} - размерность следующего за x промежуточного представления.

И если мы хотим, чтобы сохранились дисперсии и промежуточных представлений, и градиентов, у нас возникают сразу два ограничения:

$$\begin{cases} \forall i, Var(\omega_i) = \frac{1}{n_{in}} \\ \forall i, Var(\omega_i) = \frac{1}{n_{out}} \end{cases}$$

Легко заметить, что оба этих ограничения могут быть выполнены только в случае, когда размерность пространства не меняется при отображении, что случается далеко не всегда.

В работе Understanding the difficulty of training deep feedforward neural networks за авторством Xavier Glorot и Yoshua Bengio в качестве компромисса предлагается использовать параметры из распределения с дисперсией

$$\forall i, Var(\omega_i) = \frac{2}{n_{in} + n_{out}}$$

Подробный вывод данного результата можно найти в оригинальной статье в формулах 2-12. Обращаем ваше внимание, что данная инициализация хорошо подходит именно для \tanh , т.к. в выводе явно учитывается симметричность функции активации относительно нуля.

В случае использования равномерного распределения U для инициализации весов с учетом описанных выше ограничений мы получим **normalized Xavier initialization**:

$$\forall i, Var(\omega_i) \sim U \left[-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}} \right]$$

Замечание. Здесь используется тот факт, что дисперсия непрерывного равномерного распределения $Var[U[a, b]] = \frac{1}{12}(b - a)^2$.

Сравнение подобной инициализации для поведения промежуточных представлений (сверху) и градиентов (снизу) проиллюстрированы ниже (иллюстрации из оригинальной статьи):

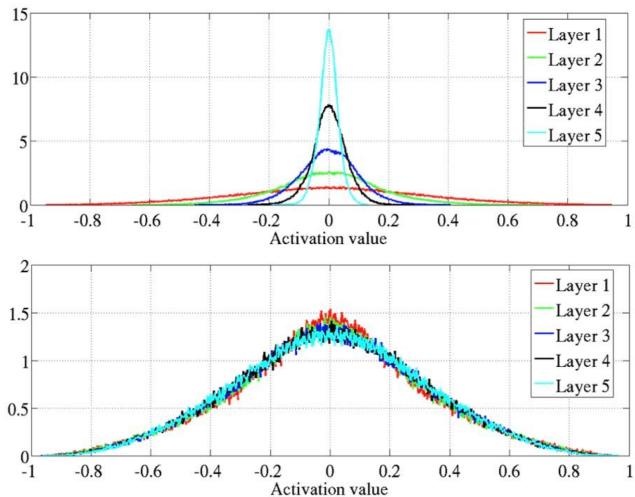


Figure 6: Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.

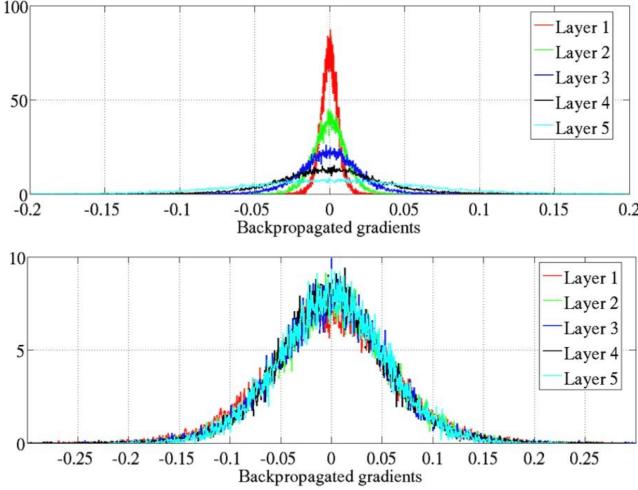


Figure 7: Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.

4.3.1.4. Подход №3: He (or Kaiming) initialization. Вы могли обратить внимание, что Xavier initialization во многом опиралась на поведение функции активации \tanh . Данный тип инициализации и впрямь лучше подходит для нее, но само использование гиперболического тангенса приводит к некоторым сложностям (например, к затуханию градиентов). В 2015 году в работе [Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](#) за авторством Kaiming He, Xiangyu Zhang, Shaoqing Ren и Jian Sun были рассмотрены особые свойства функции активации ReLU, в частности, существенно смещенная относительно нуля область значений.

Пусть представление на входе было получено после применения данной функции активации к предыдущему представлению z_{prev}

$$x = \text{ReLU}(z_{prev})$$

где z_{prev} , в свою очередь, – это выход предыдущего линейного слоя с нулевым средним для каждой компоненты весов, то есть, в частности, $\mathbb{E}(z_{prev}) = 0$

В таком случае дисперсия выхода следующего линейного слоя примет вид:

$$\text{Var}(\omega^T x) = \sum_{i=1}^n [\mathbb{E}[x_i]^2 \text{Var}(\omega_i) + \mathbb{E}[\omega_i]^2 \text{Var}(x_i) + \text{Var}(\omega_i) \text{Var}(x_i)] = \sum_{i=1}^n (\mathbb{E}[x_i]^2 + \text{Var}(x_i)) \text{Var}(\omega_i)$$

В данном случае первый член не может быть проигнорирован, так как ReLU имеет асимметричную область значений, а значит, распределения x_i будут смещёнными.

С учётом того, что $\text{Var}(x) = \mathbb{E}[x_i^2] - \mathbb{E}[x_i]^2$, выражение выше примет итоговый вид:

$$\text{Var}(\omega^T x) = \text{Var}(\omega^T x) = n_{in} \text{Var}(\omega_i) \mathbb{E}(x_i^2)$$

С учётом поведения ReLU и того, что $\mathbb{E}(z_{prev}) = 1$, можно сказать, что

$$\mathbb{E}(x_i^2) = \frac{1}{2} \text{Var}(z_{prev})$$

то есть

$$\text{Var}(\omega^T x) = \frac{1}{2} n_{in} \text{Var}(\omega_i) \text{Var}(z_{prev})$$

Получается, что использование ReLU приводит к необходимости инициализировать веса из распределения, чья дисперсия удовлетворяет следующему ограничению:

$$\forall i : \frac{1}{2} n_{in} \text{Var}(\omega_i) = 1$$

Например, подходит нормальное распределение $\mathcal{N}(0, \frac{2}{n_{in}})$.

Данный способ инициализации (и его сравнение с Xavier initialization) проиллюстрирован ниже (иллюстрации из [оригинальной статьи](#)):

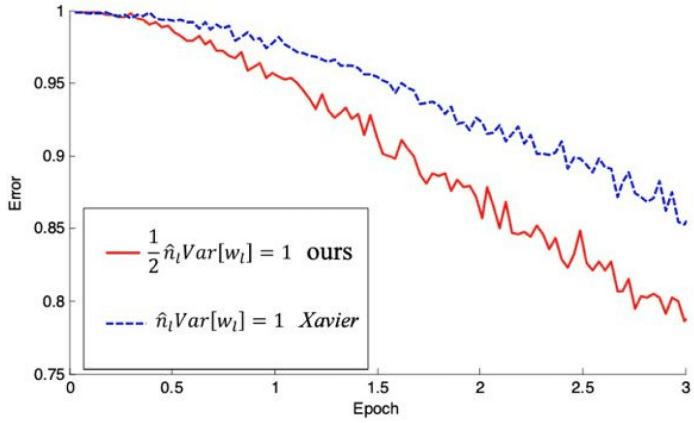


Figure 2. The convergence of a **22-layer** large model (B in Table 3). The x-axis is the number of training epochs. The y-axis is the top-1 error of 3,000 random val samples, evaluated on the center crop. We use ReLU as the activation for both cases. Both our initialization (red) and “Xavier” (blue) [7] lead to convergence, but ours starts reducing error earlier.

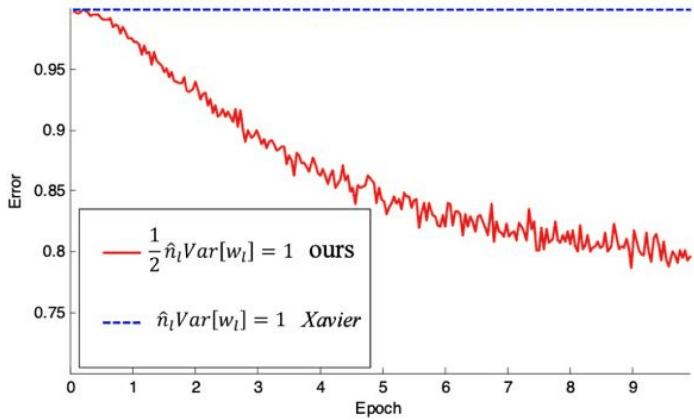


Figure 3. The convergence of a **30-layer** small model (see the main text). We use ReLU as the activation for both cases. Our initialization (red) is able to make it converge. But “Xavier” (blue) [7] completely stalls - we also verify that its gradients are all diminishing. It does not converge even given more epochs.

4.3.1.5. Выводы: Рассмотренные способы инициализации используют достаточно много предположений, но все-таки они работают и позволяют нейронным сетям в некоторых случаях значительно быстрее сходиться. Понимание принципов работы даже таких небольших механизмов – ключ к глубокому освоению области глубокого обучения :)

4.3.2. Методы оптимизации в нейронных сетях

Так как мы договорились, что нейросети представляют собой параметризованные дифференцируемые функции и для каждого параметра мы можем посчитать градиент, то, так же как и линейные модели, их можно настраивать с помощью градиентных методов. В главе про линейные модели мы под этим подразумевали обычно стохастический градиентный спуск на батчах, и это совершенно подходящий способ и для нейросетей тоже. Но существует множество модификаций и эвристик, позволяющих ускорить его схо-

димость. О них вы сможете прочитать в отдельной главе про методы оптимизации в машинном обучении, которая выйдет этой весной.

4.3.3. Регуляризация нейронных сетей

Смысл термина **регуляризация** (англ. regularization) гораздо шире привычного вам прибавления L_1 или L_2 нормы вектора весов к функции потерь. Фактически он объединяет большое количество техник для борьбы с переобучением и для получения более подходящего решения с точки зрения эксперта. Каждая из них позволяет навязать модели определённые свойства, пусть даже и ценой некоторого снижения качества предсказания на обучающей выборке. Например, уже знакомая читателю L_1 или L_2 регуляризация в задаче линейной регрессии (регуляризация Тихонова) позволяет исключить наименее значимые признаки (для линейной модели) или же получить устойчивое (хоть и смещённое) решение в случае мультиколлинеарных признаков.

В нейронных сетях техники регуляризации можно разделить на три обширные группы:

- связанные с изменением функции потерь
- связанные с изменением структуры сети
- связанные с изменением данных

4.3.4. Регуляризация через функцию потерь.

Изменение функции потерь — классический способ получить решение, удовлетворяющее определённым условиям. В глубоком обучении часто используется техника Weight Decay, очень близкая к регуляризации Тихонова. Она представляет собой аналогичный штраф за высокие значения весов нейронной сети с коэффициентом регуляризации λ :

$$L_{\text{with regularization}} = L_{\text{original}} + \lambda \|W\|_2$$

Данная техника регуляризации была совмещена с методом градиентной оптимизации Adam, в результате чего был получен метод AdamW (описанный в главе про методы оптимизации).

Также достаточно часто в качестве регуляризационного члена встречается энтропия распределения, предсказанного нейронной сетью. Представьте, что вы рекомендуете пользователю товары по истории его взаимодействия с сервисом, семплируя товары для показа в соответствии с распределением предсказанной релевантности. Вам может быть важно, чтобы рекомендации не были фиксированными (менялись при обновлении страницы), ведь это повысит вероятность того, что пользователь найдёт что-то интересное, а вы узнаете о нём что-нибудь новое. В такой ситуации при обучении модели вы можете потребовать, чтобы распределение предсказаний не сходилось к вырожденному, и в качестве дополнительной штрафной функции может выступать энтропия этого распределения. Энтропия дифференцируема, как и сами предсказанные величины, и может быть использована в качестве регуляризационного члена. Для задачи классификации он будет выглядеть следующим образом:

$$\hat{p} = f(x, \theta)$$

$$L_{\text{with regularization}} = L_{\text{original}} - \lambda \sum_k \hat{p}_k \log \hat{p}_k$$

где λ - коэффициент регуляризации, \hat{p} - предсказанные вероятности. Тем самым эксперт привносит своё знание непосредственно в процесс обучения модели в подходящей математической форме: «предсказания должны быть разнообразными» \rightarrow «распределение не должно быть вырожденным» \rightarrow «энтропия не должна быть слишком низкой».

4.3.5. Регуляризация через ограничение структуры модели.

Внесение подходящих преобразований в структуру сети также может быть хорошим способом добиться желаемых результатов. Огромное влияние на развитие нейронных сетей оказали техники dropout (2014) и batch normalization (2015), позволившие сделать нейронные сети более устойчивыми к переобучению и многократно ускорить их сходимость соответственно.

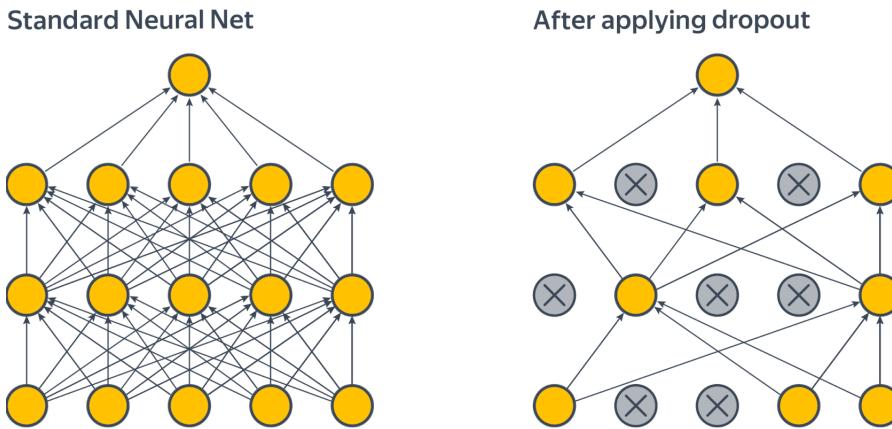
4.3.5.1. Dropout. Обратимся к простым полносвязным (FC/Dense) сетям из нескольких слоёв. Каждый из слоёв порождает новое признаковое описание x^k объекта x_{in} , который пришёл на вход:

$$x_k = f_k(x_{k-1})$$

Но как можно гарантировать, что модель будет эффективно использовать все доступные параметры, а не переобучится под использование лишь небольшого их подмножества, поделив для себя внутреннее представление на сигнал и шум?

$$x_{k,\text{overfitted}} = [\text{signal}, \text{noise}]$$

Для этого можно было бы случайным образом «выключать» доступ к некоторым координатам внутренних представлений на этапе обучения. Тогда при выключении «полезных» координат произойдёт резкое изменение предсказаний модели, что приведёт к увеличению ошибки, а полученные градиенты этой ошибки укажут, как её исправить с использованием (и изменением) других координат. Сравнение тока информации по исходной модели и по модели с «выключенными» координатами внутренних представлений можно проиллюстрировать с помощью классической картинки:



Обращаем ваше внимание, что «выключать» можно как оригинальные признаки, так и признаки, возникающие на любом другом уровне представления объектов. С точки зрения $(k+1)$ -го слоя нейронной сети данные приходят откуда-то извне: при $k = 0$ - из реального мира, а при $k > 1$ - с предыдущих слоёв.

Технически это осуществляется следующим образом: некоторые координаты внутреннего представления домножаются на ноль. То есть добавляется ещё одно преобразование, которое представляет собой домножение выхода предыдущего слоя на маску из нулей и единиц.

$$x_{k+1} = \frac{1}{1-p} x_k \circ \text{mask}$$

$$\text{mask}_i \sim \text{Bernoulli}(1-p)$$

где $(1-p)$ (вероятность обнуления координаты) является гиперпараметром слоя (отметим, что во многих фреймворках для глубинного обучения в качестве параметра слоя указывается именно вероятность обнуления, а не выживания). Данная маска участвует и при подсчёте градиентов:

$$\nabla_{x_k} L = \frac{1}{1-p} \nabla_{x_{k+1}} L \circ \text{mask}$$

Как правило, маска генерируется независимо на каждом шаге градиентного спуска. Важно отметить, что на этапе предсказания dropout ничего не меняет, то есть $x_{k+1} = x_k$

Множитель $\frac{1}{1-p}$ нужен для того, чтобы распределение x^{k+1} на этапе предсказания совпадало с распределением на этапе обучения. В самом деле, если даже математическое ожидание x_k было равно нулю, выборочная дисперсия $x_k \circ \text{mask}$ ниже, чем у x_k : ведь часть значений обнулилась.

На этапе предсказания dropout «выключается»: внутренние представления используются как есть, без умножения на маску. А чтобы слой знал, обучается он сейчас или предсказывает, в нейросетевых библиотеках в классе слоя обычно реализовано переключение между этими режимами (например, булев флаг `training` в pytorch-модулях).

Примечание.

Полезно провести аналогию с другим алгоритмом, использующим техники ансамблирования и метод случайных подпространств: речь о случайному лесе (Random Forest). При обучении сети на каждом шаге обучается лишь некоторая подсеть (некоторый подграф вычислений из исходного графа). При переходе в режим inference (то есть применения к реальным данным с целью получения результата, а не обучения) активируются сразу все подсети, и их результаты усредняются. Таким образом, сеть с dropout можно рассматривать как ансамбль из экспоненциально большого числа сетей меньшего размера (подробнее можно прочитать [здесь](<https://arxiv.org/abs/1706.06859>)). Это приводит к получению более устойчивой оценки значений целевой переменной.

В этом свойстве кроется и главное коварство dropout (как и большинства других техник регуляризации): благодаря получению более устойчивой оценки целевой переменной путём усреднения множества подсетей, эффективная обобщающая способность итоговой сети снижается! В самом деле, пусть при обучении каждый раз модели была доступна лишь половина параметров. В таком случае итоговая модель представляет собой усреднение множества более слабых моделей, в которых вдвое меньше параметров. Её предсказания будут более устойчивы к шуму, но при этом она неспособна выучить столь сложные зависимости, как сеть аналогичной структуры, но без dropout. То есть за более устойчивые предсказания (и получение менее переобученной модели) приходится расплачиваться и меньшей обобщающей способностью.

Стоит отметить, что dropout может применяться и к входным данным (то есть слой dropout может стоять первым в сети), и это может приводить к получению более качественных результатов. Например, если в данных множество мультикоррелирующих признаков или присутствует шум, наличие dropout позволит избежать обусловливания модели на лишь их подмножество и позволит учитывать их все. Так, подобный подход может быть использован, если данные представляют собой сильно разреженные векторы высокой размерности (скажем, сведения об интересе пользователя к тем или иным товарам).

4.3.5.2. Batch normalization Появление техники batch normalization привело к значительному ускорению обучения нейронных сетей. В данной главе мы рассмотрим лишь основные принципы работы batch normalization. Дискуссия о свойствах и причинах эффективности batch normalization всё ещё ведётся, рекомендуем обратить внимание на [статью с NeurIPS 2018](#). Нам, впрочем, кажется, что, несмотря на активную критику в его адрес, полезно знать и предложенное авторами подхода объяснение необходимости batch normalization.

Предложенная авторами мотивация. Обратимся к механизму обратного распространения ошибки. Пусть мы находимся на этапе обновления параметров W^k некоторого k -го слоя:

$$x_k = f(x_{k-1}, W_k)$$

где f - некоторая функция, которая вычисляется на данном слое. В общем случае W_k и x_{k-1} не обязательно взаимодействуют линейным образом. Функция f может быть и нелинейной. В ходе error backpropagation мы вычисляем градиент:

$$\nabla_{w_k} L = g(x_{k-1}, x_k, x_{k+1}, \dots, W_k)$$

где g - некоторая функция, в которой будут участвовать представления со слоёв, начиная с $(k-1)$ -го (вычисленные в ходе forward pass и запомненные). Новое значение параметров примет вид:

$$W_{k,\text{new}} = W_k - \alpha \nabla_{W_k} L = W_k - \alpha g(x_{k-1}, x_k, \dots)$$

После обновления параметров W_k мы перейдём к обновлению параметров предыдущего слоя W_{k-1} и обновим их аналогичным образом.

Но это приведёт к изменению представления, которое пришло на вход k -му слою, которое мы не учитываем:

$$x_{k,\text{new}} = f(x_{k-1}, W_{k,\text{new}}) = f(x_{k-1}, W_k + \phi)$$

где ϕ - разница между предыдущими и новыми параметрами W_k .

То есть параметры $(k-1)$ -го слоя будут обновлены исходя из предположения, что данные приходят из некоторого распределения на x^k , которое параметризовалось W_{k-1} , но теперь параметры изменились и данные могут **обладать иными свойствами**. Например, может существенно измениться среднее или дисперсия (что может привести, например, к попаданию на «хвосты» функции активации и затуханию градиента).

До появления batch normalization с этой проблемой боролись достаточно просто: использовали небольшие значения шага обучения (learning rate) α . Благодаря этому изменения были не слишком большими и можно было предположить, что и распределение внутренних представлений поменялось незначительно.

И так, вернёмся к самому применению batch normalized.

Использование batch normalization гарантирует, что каждая компонента представления на выходе будет иметь контролируемое среднее и дисперсию. Достигается это следующим образом:

- Сперва идёт собственно слой batch normalization, на котором текущий батч приводится к нулевому среднему и единичной дисперсии:

$$X_{k+1} = \frac{X_k - \mu}{\sqrt{\sigma^2}} + \varepsilon$$

где μ и σ^2 - среднее и дисперсия признаков по обрабатываемому батчу, а ε - гиперпараметр слоя, небольшое положительное число, добавляемое для улучшения численной устойчивости. Отметим, что μ и σ , будучи функциями от X_k , тоже участвуют в вычислении градиентов.

В ходе **предсказания** (или, как ещё говорят, **инфереенса**, от английского **inference**) используются фиксированные значения μ_* и σ_* , которые были получены в ходе обучения как скользящее среднее всех μ и σ^2 . Более точно: на каждой итерации forwards pass мы вычисляем

$$\mu_* = \mu_* \lambda + \mu(1 - \lambda)$$

$$\sigma_*^2 = \sigma_*^2 \lambda + \sigma^2(1 - \lambda)$$

где λ также является гиперпараметром слоя.

- Далее идёт слой channelwise scaling, который позволяет выучить оптимальное шкалирование для всех признаков X_{k+2} :

$$X_{k+2} = \beta X_{k+1} + \gamma$$

где β и γ - обучаемые параметры, позволяющие настраивать в ходе обучения оптимальные значения матожидания и дисперсии выходного слоя X_{k+2}

Ниже приведён алгоритм из оригинальной статьи 2015 года за авторством Сергея Иоффе и Кристиана Сегеди:

Input: Network N with trainable parameters Θ ;
subset of activations $\{x^{(k)}\}_{k=1}^K$

Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$

- $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network
- for** $k = 1 \dots K$ **do**
- Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)}, \beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)
- Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
- end for**
- Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^K$
- $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen
 // parameters
- for** $k = 1 \dots K$ **do**
- // For clarity, $x \equiv x^{(k)}$, $\gamma \equiv \gamma^{(k)}$, $\mu_B \equiv \mu_B^{(k)}$, etc.
- Process multiple training mini-batches \mathcal{B} , each of size m , and average over them:
- $E[x] \leftarrow E_{\mathcal{B}}[\mu_B]$
- $\text{Var}[x] \leftarrow \frac{m}{m-1} E_{\mathcal{B}}[\sigma_B^2]$
- end for**
- In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma, \beta}(x)$ with
 $y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma E[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$
- end for**

Причина популярности batch normalization заключается в значительном ускорении обучения нейронных сетей и в улучшении их сходимости в целом. Рассмотрим график из оригинальной статьи:

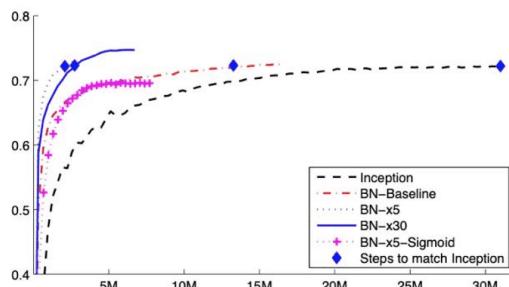


Figure 2: Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.

Как видно на иллюстрации выше, использование batch normalization позволило ускорить обучение в несколько раз и даже добиться лучших результатов, чем SotA-подход 2014 года Inception (структура которого была приведена на одной из иллюстраций в начале этой главы).

Значительное ускорение достигается в том числе благодаря использованию более высокого learning rate: благодаря нормировке связь между слоями не нарушается столь сильно.

Стоит заметить, что причины столь эффективной работы batch normalization до сих пор являются поводом для дискуссий и строгого теоретического объяснения эффекта от batch normalization ещё нет. Несмотря на это, он перевернул область глубинного обучения и вошёл в стандартный инструментарий при обучении нейронных сетей.

Примечание: стоит заметить, что в настоящее время существуют и другие способы нормировать промежуточные представления: instance normalization, layer normalization и др.

В завершение рекомендуем ознакомиться с данной статьёй о работе метода обратного распространения ошибки в слое batch normalization.

4.3.6. Регуляризация через изменение данных

Внесение изменений в данные (аугментация данных) также является популярной техникой регуляризации. Рассмотрим её на примере. Пусть перед нами фотография самолёта. Добавим мелкодисперсный шум к изображению. Мы всё ещё сможем увидеть на фотографии самолёт, но, с точки зрения модели машинного обучения (в данном случае — нейронной сети), полученное изображение является новым объектом! Повернём изображение самолёта на 10 градусов по часовой стрелке. В нашем распоряжении ещё одно изображение с известной целевой меткой (например, меткой класса «самолёт»), в котором присутствует поворот. Таким образом, внесение новых данных позволяет дать модели понять, какие преобразования над данными являются допустимыми, и она уже будет более устойчивой к наличию небольшого шума в данных или к поворотам (к которым чувствительна операция свёртки). Вдобавок аугментации позволяют значительно увеличить объём обучающей выборки. Особую популярность аугментации приобрели в области компьютерного зрения. В качестве примера приведём отличную библиотеку, позволяющую производить аугментацию изображений.

Стоит обратить внимание, что используемые аугментации должны быть адекватны решаемой задаче. Инвертирование цветов на фотографии, внесение значительного количества шумов или переворот изображения могут привести к негативным результатам (по сути, просто сделают выборку более запущённой или даже заставят сеть учиться на данных, которые она никогда не встретит в реальности), так как обобщающая способность сети ограничена. Можно сказать, что аугментированные данные должны принадлежать к той же генеральной совокупности, что и оригиналный датасет.

Итак, эксперт может привнести своё понимание задачи и на уровне аугментации данных: если данное преобразование является допустимым (то есть преобразованный объект мог бы попасть в обучающую выборку и самостоятельно — как фотография с другого устройства или запись речи другого человека с опечаткой), то модель должна быть устойчива к данным с подобными преобразованиями.

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid	15M	69.8%

Figure 3: For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.

5. Лекция 1. Введение в PyTorch. Наша лекция.

5.1. Базовые баз основные операции

Ну для начала, импорт:

```
import torch
```

В torch'е точно также, как и в numpy создавать многомерные массивы, только они называются по другому:

Например, код для двумерного массива:

```
a = torch.tensor([
    [1., 2., 3.],
    [4., 5., 6.]
])
a
# tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

Можно точно также посмотреть тип данных:

```
type(a)
# torch.Tensor

a.dtype
# torch.float32
```

Точно также можно создавать случайные массивы. Например, если хотим сгенерировать массив из случайных нормальных чисел:

```
a = torch.randn(3, 4)
a
# tensor([[-0.3398, -0.0950,  1.6260, -1.8649],
        [ 0.6572,  0.7548,  0.4294, -0.6261],
        [ 1.1535, -0.6050, -0.9054, -0.1503]])
```

Точно также, как и в numpy, можно посмотреть размер тензора:

```
a.shape
# torch.Size([3, 4])
```

И, например, можно создать массив 3×4 , но состоять не из нормальных, а из равномерных от 0 до 1:

```
b = torch.rand(3, 4)
b
# tensor([[0.5950,  0.6613,  0.2369,  0.4959],
        [0.3892,  0.2089,  0.8794,  0.0487],
        [0.3155,  0.2532,  0.2157,  0.4078]])
```

Также можно два тензора сложить - это будет покоординатное сложение:

```
a + b
# tensor([[ 0.2552,  0.5663,  1.8629, -1.3690],
        [ 1.0464,  0.9637,  0.3088, -0.5775],
        [ 1.4690, -0.3518, -0.6897,  0.2575]])
```

Можно перемножить поэлементно:

```
a * b
# tensor([[-0.2022, -0.0628,  0.3852, -0.9248],
        [ 0.2558,  0.1577,  0.3776, -0.0305],
        [ 0.3639, -0.1532, -0.1953, -0.0613]])
```

А можно взять синус от одного тензора, и поделить на экспоненту другого:

```
torch.sin(a) / torch.exp(b)
# tensor([[-0.1838, -0.0490,  0.7879, -0.5829],
        [ 0.4139,  0.5560,  0.1728, -0.5582],
        [ 0.6668, -0.4416, -0.6341, -0.0996]])
```

Давайте ещё сделаем умножение матриц как в numpy (и линале)

```
a = torch.randn(5, 2)
b = torch.randn(2, 4)
a @ b
# tensor([[ 0.9109, -0.6142,  0.1560,  0.0635],
#         [-1.2903,  1.0413, -0.7832, -0.1258],
#         [ 0.6881,  0.5642, -3.2551, -0.1674],
#         [-1.7927,  1.4251, -1.0169, -0.1703],
#         [-0.3289, -0.9621,  3.8271,  0.2550]])
(a @ b).shape
# torch.Size([5, 4])
```

Также можно считать градиент:

```
a = torch.randn(5, requires_grad=True)
a
# tensor([1.4558, 1.3855, 0.0984, 0.2864, 1.0203])
```

Что это значит? Если вспомним лекцию, то в нейронных сетях у нас есть какие-то веса, которые нам задают слои. Кроме есть ешё вектор вектор сдвига, который мы добавляем ешё после матричного умножения. Сами по себе параметры - это тоже какие-то многомерные тензоры, и фактические наше обучение нейронной сети сводится к тому, что мы будем делать градиентный спуск.

То есть мы на каждой итерации нашего обучения будем некоторым образом считать по всем параметром, которые есть в нашей модели и будем делать шаг градиентного спуска. Соответственно, когда мы прописываем `requires_grad=True`, мы говорим, что потенциально этот тензор мы хотим использовать для подсчёта градиента ошибки.

Давайте разберём простейший пример:

Сначала запустим без `requires_grad`

```
a = torch.randn(5)
a
# tensor([ 0.5427, -1.0936,  0.4846, -0.6583,  1.6258])

l = (a ** 2).sum()
l
# tensor(0.3.8020)
```

Это просто скалярный тензор. Если мы посмотрим на его форму, то это будет тривиально просто одно число, а не массив:

```
l.shape
# torch.Size([])
```

Теперь добавим параметр для градиента:

```
a = torch.randn(5, requires_grad=True)
a
# tensor([-1.2647, 1.0210, 0.0850, -0.1612, -0.7875], requires_grad=True)
```

Как мы видим, даже в выводе появился `requires_grad=True`. Это ключевое слово говорит, что мы потенциально по этой матрице будем считать градиенты.

Давайте теперь ешё раз прогоним функцию, которая считает квадраты:

```
l = (a ** 2).sum()
l
# tensor(3.2954, grad_fn=<SumBackward0>)
```

Помимо того, что появилось другое число, потому что сгенерировались другие числа, появилось `grad_fn=<SumBackward0>`. Суть в том, чтобы делать алгоритм обратного распространения ошибки для дифференцирования нейронных сетей, нам нужно строить вычислительный граф. Для каждого элемента, который используется для вычисления итогового результата (в нашем случае 3.2954) нам нужно каждое промежуточное вычисление запомнить. Нам нужно делать это для того, чтобы потом делать проход назад по графу вычислений и в каждом узле для этого графа мы считали производную и передавали её дальше.

У нас как раз написано, что последняя операция, которая нас привела к результату - это функция суммы. И здесь написано, что здесь это последняя операция и мы продифференцируем её.

Теперь, чтобы нам посчитать функцию l , которая сумма квадратов - нам нужно вызвать `backward`. Состоит она в том, что у нас для точки совершается проход назад по графу вычислений и во всех листьях этого графа, где мы попросили, что для тензоров должен вычисляться градиент - он будет посчитан.

```
l.backward()
```

Ничего не произошло. Если мы посчитаем функцию l аналитически по a , то производная будет $2a$. И у нашей переменной a появился новый атрибут `grad`.

```
a.grad  
# tensor([-2.5295, 2.0419, 0.1699, -0.3229, -1.5750])
```

И действительно в нём записан некоторый тензор. И если мы на него посмотрим, то это действительно наш исходный тензор, каждая координата которого умножена на 2. Всё соотносится с нашей логикой.

Если мы заново запустим нашу функцию

```
l = torch.randn(5, requires_grad=True)  
a.grad
```

То мы увидим, что ничего не произошло. И действительно, пока мы не продифференцируем график, у нас никакого градиента нет.

Если мы попробуем запустить `backward` 2 раза, то вот что из этого выйдет:

```
a = torch.randn(5, requires_grad=True)  
a  
# tensor([-0.2471, 1.4925, -1.1152, -0.0602, 0.1761], requires_grad=True)  
  
l.backward()  
l.backward()  
  
a.grad  
# tensor([-0.9883, 5.9702, -4.4607, -0.2406, 0.7046])
```

Что произошло? В `a.grad` записано 4 a , это происходит потому что если у вас есть какой-то градиент, который уже записан в вашем тензоре, и вы хотите продифференцировать его ещё раз или от другой функции - это работает так, что новый градиент не зануляется, а складывается со старым.

На самом деле это сделано не просто так - есть техника аккумулирование градиентов.

Представьте ситуацию такую: мы вычисляем функцию l и мы используем какие-то тензоры, у которых написано `requires_grad=True`, происходит построение графа вычислений. Этот график включает в себя путь для того, чтобы дойти до l . Когда мы зовём `l.backward()` мы говорим, что мы хотим по этому нашему финальному элементу посчитать $\frac{\partial l}{\partial a}$. На самом деле будет считаться производная для всех элементов, для которых мы пометили `requires_grad=True`. То есть во всех тензорах будет храниться $\frac{\partial l}{\partial \text{тензор}}$.

Представляем ещё ситуацию: у вас есть очень большая модель, которая с трудом помещается на одну видеокарту. Из-за этого мы не можем делать стохастический градиентный спуск с большим `batch`: у нас помещается только один объект чтобы сделать шаг градиентного спуска, но вот если увеличить количество объектов, то пропорционально увеличатся затраты по памяти. Зачастую хочется, чтобы у нас градиентный спуск был не с `batch_size=1`, а побольше. И вместо того, чтобы страдать, что у нас ничего не влезает, что мы можем сделать. Предположим у нас `batch_size=8`, мы можем взять один объект, посчитаем на нём нашу функцию потерь, продифференцируем её, у нас все нужные градиенты сложатся в поле `grad`. Дальше сделаем то же самое со вторым объектом. Эти действия не занулят первый градиент, а сложатся с ними. Мы посчитали градиент суммы функции суммы функции потерь и на втором. То же самое мы сделаем и с остальными. По итогу мы получим суммарный градиент по всем 8-ми функциям потерь. Не смотря на то, что мы одновременно можем обрабатывать один объект - у нас по итогу градиент получится от всех 8-ми объектов, как будто бы считали ошибку по всем 8-ми объектам.

Сейчас мы могли посчитать градиент вручную, давайте возьмём какую-нибудь более сложную функцию.

Сделаем несколько тензоров:

```
a = torch.randn(2, 3, requires_grad=True)  
b = torch.randn(2, 3, requires_grad=False)  
c = torch.randn(2, 3, requires_grad=True)
```

И возьмём функцию

```
loss = (torch.cos(a / 3) + torch.sqrt(torch.abs(b * c))).sum(dim=0)  
loss  
# tensor(39.2590, grad_fn=<Prod@Backward1>)
```

И заметим отличие от numpy - axis в numpy это dim в pytorch.

Мы точно также можем продифференцировать:

```
loss.backward()
```

И теперь у a и c есть градиент:

```
a.grad  
# tensor([[ -0.4042,   0.4052,   0.2151],  
#           [ 1.6134,  0.9258, -2.6517]])  
  
c.grad  
# tensor([[ -3.0521, -102.4510,  
#           -6.2473],  
#           [ 6.4475,    3.5821,  
#           -9.8835]])
```

И при этом в b лежит None:

```
type(b.grad)  
# NoneType
```

5.2. Работа с torch и видеокартами

Все дальнейшие действия произведены в colab.

Есть команда nvidia-smi, которая показывает текущую информацию про видеокарту: какая версия CUDA стоит, какой драйвер, какая видеокарта.

```
!nvidia-smi
```

Дальше импортируем torch:

```
import torch
```

И дальше, что вам понадобится, если вы хотите пользоваться видеокартой - завести device:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

Что здесь происходит? Мы говорим, что если у нас есть CUDA, в таком случае мы работаем на cuda0 - это означает индекс видеокарты, которые есть на нашем сервере. Если бы у нас было несколько видеокарт и мы хотели бы работать со второй, то мы прописали бы cuda1. Чтобы эта штука не падала - мы прописали, что если видеокарты нет, то мы используем в качестве прогона - сри.

Дальше наш device имеет следующие характеристики при отображении:

```
device  
# device(type='cuda', index=0)
```

Давайте сгенерируем синтетический пример - просто умножим две матрицы:

```
A = torch.randn(10000, 10000)  
B = torch.randn(10000, 10000)
```

```
A.device  
# device(type='cpu')
```

Видим, что матрицы создаются на CPU, давайте посмотрим сколько занимает умножение этих матриц:

```
%time  
A @ B  
# CPU times: user 27 s, sys: 207 ms, total: 27.2 s  
Wall time: 27.8 s
```

Теперь давайте сделаем всё то же самое, но на GPU, сначала нам нужно определить матрицы:

```
A = A.to(device)  
B = B.to(device)
```

Можно, конечно переносить и так, но так делать не рекомендуется:

```
B = B.cuda()
```

Если у вас доступно несколько видеокарт, то не очень понятно на какую видеокарту перенесётся В. Ключевое слово to работает не только для перемещения на видеокарту, но и для других вещей:

```
c = torch.arange(6)
c.to(torch.float32)
# tensor([0., 1., 2., 3., 4., 5.])
```

Это тот же самый массив, просто исходно он был в int64, а мы ему дали другой тип.

Давайте теперь перемножим А и В на видеокарте:

```
%%time
A @ B
CPU times: 491 ms, sys: 279 ms, total: 720 ms
Wall time: 3.02 s
```

Видно, что мы очень сильно ускорились. В контексте обучения нейронных сетей разница будет в сотни и даже тысячи раз, поэтому, если у вас сеть не очень большая, то обучать нужно на видеокарте.

Но мы сделали не очень корректно - мы сначала тензор сделали на CPU, а потом перенесли на GPU, а могли бы сразу аллоцировать на GPU:

```
A = torch.randn(10000, 10000, device=device)
```

В таком случае тензор сразу аллоцируется на GPU.

Если хотим перенести на CPU, то можно сделать так:

```
A = A.cpu()
```

И теперь если захотим эти тензоры перемножить, то нас ждёт ошибка, так как А лежит на CPU, а В на GPU. Если в процессе выполнения функции оказывается так, что один тензор на одном устройстве, а другой на другом (даже оба на видеокартах, но на разных), то тогда вас ждёт ошибка. Иногда они выглядят не очевидно и нужно это смотреть.

5.2.1. Вычисления на m1 и m2

Для того, чтобы производить вычисления на m1 или m2, нам нужно взять другой device:

```
device = torch.device('mps')
device
# device(type='mps')
```

Поскольку память ограничена - нам хотелось бы хранить на GPU действительно только то, что нужно. Это веса самой модели. Мы их будем использовать постоянно.

5.3. Pipeline обучения

Для примера возьмём датасет MNIST.

```
from torchvision.datasets import MNIST
```

```
train_set = MNIST('~/.datasets/mnist', train=True, download=True)
test_set = MNIST('~/.datasets/mnist', train=False, download=True)
```

Напомним, что в MNIST 60000 рукописных картинок.

В питоне, в torch, MNIST предстаёт как класс. У этого класса есть родительский класс Dataset. Это просто абстрактный класс, который умеет следующее:

- С помощью __getitem__() можно обращаться с помощью квадратных скобок к какому-то очередному элементу.
- __len__() - ну соответственно тоже стандартная длина.

Например, если мы для нашего датасета MNIST захотим применить оператор квадратных скобок: вот что получится

```
out = train_set[12674]
out
# (<PIL.Image.Image image mode=L size=28x28>, 9)
```

Получается, один объект - это кортеж из объекта и целевой переменной.

А вот как хранится .data:

```
train_set.data.shape  
# torch.Size([60000, 28, 28])
```

На одном из ближайших занятий мы будем обучать что-то на произвольном датасете, которого нет в torchvision, и оказывается в этом случае очень удобно написать произвольный класс, отнаследованный от torch'ового.

Чтобы обучаться - нам удобно, чтобы возвращались тензоры вместо картинок, поэтому давайте сделаем это преобразование. Для этого в самом torch есть всякие transform:

```
import torchvision.transforms as T
```

```
train_set = MNIST('~/datasets/mnist', transform=T.ToTensor, train=True, download=True)  
test_set = MNIST('~/datasets/mnist', transform=TT.ToTensor, train=False, download=True)
```

И теперь вот чем является наш объект:

```
out = train_set[12675]  
out  
# (tensor ([[[...]]]))  
  
out[0].shape  
# torch.Size([1, 28, 28])
```

Где в размерности 1 - единственный цветовой канал, а 28×28 - сама картинка.

Мы также можем задать цепочку transform'ов. Для этого нам понадобится T.Compose.

Если мы захотим обратно тензор вернуть в картинку, то мы воспользуемся таким transform'ом:

```
to_image = t.ToPILImage()
```

Также у torch'a есть DataLoader. Это нужно для того, чтобы делать батчи. Он берёт в себя датасет и умеет нарезать его на батчи.

```
from torch.utils.data import DataLoader
```

```
train_loader = DataLoader(train_set, batch_size=64, shuffle=True, num_workers=4, pin_memory=True)  
test_loader = DataLoader(test_set, batch_size=64, shuffle=False, num_workers=4, pin_memory=True)
```

И вот если мы будем циклом for перебирать наши батчи:

```
for batch in train_loader
```

То отдельный batch это два объекта в листе: картинки и целевые переменные:

```
batch[0].shape  
# torch.Size([64, 1, 32, 32])  
  
batch[1].shape  
# torch.Size([64])
```

И обычно мы будем перебирать как:

```
for images, labels in train_loader():  
    # step of grad descent
```

ДА НЕУЖЕЛИ У МЕНЯ НЕ СОХРАНИЛИСЬ ИЗМЕНЕНИЯ

6. Оптимизация нейронных сетей, dropout, batch-нормализация. Лекция Ильдуса ПМИ. Лекция 2.

https://www.youtube.com/watch?v=09JV_Kgd31E

6.1. Теорема Цыbenко

Теоретический вопрос: насколько теоретически хорошо работают нейронки? Насколько теоретически мы хорошо подвигаем решение нейронкой к реальной функции решения?

Это изучает Universal Approximation Theorem (UAT), мы рассмотрим конкретно **теорему Цыбенко**:

Пусть у нас есть функция активации σ :

- Непрерывная
- Сигмоидальная: $\lim_{x \rightarrow -\infty} \sigma(x) = 0$ и $\lim_{x \rightarrow +\infty} \sigma(x) = 1$

Пусть у нас есть некоторый уровень ошибки $\forall \varepsilon > 0$, а $f(x)$ - непрерывная функция на $[0, 1]^d$.

Теорема утверждает, что любую такую функцию мы можем равномерно приблизить двуслойной нейронкой на множестве $[0, 1]^d$ с ошибкой не более, чем ε .

Нейронка выглядит как $g(x) = \overbrace{C}^{\mathbb{R}} + \sum_{i=1}^n \overbrace{v_i}^{\mathbb{R}} \sigma(\overbrace{\omega_i^T}^{\mathbb{R}^d} x + \overbrace{b_i}^{\mathbb{R}})$.

Давайте тогда подытожим:

$\forall \varepsilon > 0, \forall f(x), x \in [0, 1]^d$ для функции $g(x) = C + \sum_{i=1}^n v_i \sigma(\omega_i^T x + b_i)$ $\exists n, \omega_i, v_i, b_i, C$ такие, что $\forall x \in [0, 1]^d, |f(x) - g(x)| \leq \varepsilon$

6.2. Проблемы нейронных сетей

6.2.1. Обучение с помощью градиентного спуска

Сампо по себе обучение градиентным спуском больших проблем не несёт. Нужно давать такие функции, которые дифференцируемы и всё.

6.2.2. Переобучение

Из-за того, что у нас много параметров у модели: если у нас l слоёв и ширина нейронки d , то получается примерно $l \times d^2$ параметров. Это очень много.

Мы можем взять очень широкую очень длинную нейронку, у которой на обучающих данных будет loss 0, но на тестовых мы увидим, что она переобучилась.

6.2.3. Ширина

Нам гарантируется, что есть какая-то величина, согласно теореме Цыбенко, но на практике эта величина может быть очень большой. На практике глубокие сетки работают лучше, чем двуслойная с хоть сколько большой шириной.

6.3. Оптимизация нейронных сетей

6.3.1. SGDMomentum

Давайте скажем, что мы оптимизируем функционал $Q(\omega) = \frac{1}{B} \sum_{i=1}^B q_{t_i}(\omega)$, где q_{t_i} - loss на объекте t_i , а B - размер батча.

Давайте тогда запишем, как будет выглядеть SGD Momentum:

Для начала введём обозначение $g_t = \nabla_\omega Q(\omega_{t-1})$, а $m_t = \mu m_{t-1} + g_t$. Всё что нам осталось - обновить веса. Давайте соберём всё вместе:

$$\begin{cases} g_t &= \nabla_\omega Q(\omega_{t-1}) \\ m_0 &= g_1 \\ m_t &= \mu \cdot m_{t-1} + g_t \\ \omega_t &= \omega_{t-1} - \eta_t \cdot m_t \end{cases}$$

где μ - параметр нашего моментума. Если $\mu = 0$ - то это обычный градиентный спуск.

Тогда теперь мы можем сделать переход от $Q(\omega) \rightarrow Q_\lambda(\omega) = \frac{1}{B} \sum_{i=1}^B q_{t_i}(\omega) + \lambda \|\omega\|_2^2$.

Тогда $\nabla_\omega Q_\lambda(\omega) = \nabla_\omega Q(\omega) + \lambda \omega$.

Давайте теперь запишем нашу систему снова:

$$\begin{cases} g_t &= \nabla_\omega Q(\omega) + \lambda \omega \\ m_0 &= g_1 \\ m_t &= \mu \cdot m_{t-1} + g_t \\ \omega_t &= \omega_{t-1} - \eta_t \cdot m_t \end{cases}$$

На самом деле мы теперь можем расписать $\omega_t = \omega_{t-1} - \eta_t m_t = \omega_{t-1} - \eta_t (\mu m_{t-1} + \nabla_\omega Q(\omega_{t-1}) + \lambda \omega_{t-1}) = (1 - \eta_t \lambda) \omega_{t-1} - \eta_t (\mu m_{t-1} + \nabla_\omega Q(\omega_{t-1}))$ и эта штука называется weight decay (или же затухание веса):

$$\begin{cases} g_t &= \nabla_\omega Q(\omega) + \lambda \omega \\ m_0 &= g_1 \\ m_t &= \mu \cdot m_{t-1} + g_t \\ \omega_t &= (1 - \eta_t \lambda) \omega_{t-1} - \eta_t (\mu m_{t-1} + \nabla_\omega Q(\omega_{t-1})) \end{cases}$$

То есть это очень похоже на $L2$ регуляризацию.

6.3.2. Adam

Формула обучения выглядит следующим образом:

$$\begin{cases} g_t &= \nabla_\omega Q(\omega_{t-1}) + \lambda \omega_{t-1} \\ m_0 &= 0 \\ v_0 &= 0 \\ m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \cdot g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) \cdot g_t^2 \\ \hat{m}_t &= \frac{1}{1 - \beta_1^t} \cdot m_t \\ \hat{v}_t &= \frac{1}{1 - \beta_2^t} \cdot v_t \\ \omega_t &= \omega_{t-1} - \eta_t \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{cases}$$

И все операции у нас не матричные, а поэлементные.

Зачем нормировать m и v ?

$\mathbb{E} \hat{m}_t = \mathbb{E} g_t$, $\mathbb{E} \hat{v}_t = \mathbb{E} g_t^2 = Var(g_t) + (\mathbb{E} g_t)^2$.

И $Var(X) = \mathbb{E} X^2 - (\mathbb{E} X)^2$

Обычно гиперпараметры β берут $\beta_1 = 0.9$ и $\beta_2 = 0.999$ и вообще не трогаем.

Суть этого такова, что чем ближе мы приближаемся к оптимуму - тем меньше Adam делает шаги. За это отвечает слагаемое $\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$. Давайте распишем формулу перехода к новым весам:

$$\omega_t = \omega_{t-1} - \frac{\eta_t}{1 - \beta_1^t} \cdot \frac{1}{\sqrt{\hat{v}_t} + \epsilon} (\beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla_\omega Q(\omega) + (1 - \beta_1) \lambda \omega_{t-1}) = \omega_{t-1} \left(1 - \frac{\eta_t \cdot (1 - \beta_1)}{1 - \beta_1^t} \cdot \frac{1}{\sqrt{\hat{v}_t} + \epsilon} \right) - \dots$$

Получается чем мем больше v_t - тем больше регуляризуем. Это влияет на затухание. Это расходится с логикой того, что мы хотели делать в $L2$ регуляризации. У нас веса будут штрафоваться не одинаково. Это не то, что мы изначально хотели в нашей реализации.

Мы как бы и очень сильно оптимизируемся, но оптимизируемся слишком хорошо и это ведёт к переобучению.

6.3.3. Learning Rate Scheduling

Почти всегда, если вы используете константный LR, то вы теряете в качестве. Нужно использовать расписание LR для того, чтобы улучшать обучаемость модели.

- Step LR. Вы раз в сколько-то эпох изменяете LR. 1 эпоха это когда у вас каждый объект побывал в каком-то батче. Вы можете менять LR каждую эпоху. Если размер батча равен размеру датасета, то эпоха - это один шаг.
- Exponential LR - уменьшаем по экспоненте.
- Cosine LR - в самом начале оптимизации мы не уменьшаем LR, и держим большим, а ближе к концу обучения (ну типа) - уменьшаем.

- Reduce LR on Plato - мы проверяем как уменьшается loss на отложенной валидационной выборке, и когда loss на ней не убывает - уменьшаем LR.
- Cyclic LR - мы иногда понижаем LR, а иногда повышаем.

6.4. Dropout слой

Это эвристический способ регуляризовать нашу нейронку. Допустим мы сейчас обучаем на $x \in \mathbb{R}^{B \times N}$. Давайте сделаем бинарную маску $m \in \mathbb{R}^{B \times N}$ и её мы генерируем случайно: у нас есть гиперпараметр p и $m_{ij} \sim Bernoulli(1 - p)$. Когда мы будем применять наш слой и поэлементно умножим на эту маску: $f(x) = X \odot m$.

Фактически это будет означать, что у нас выбираются примерно p (в отношении) случайных клеток в x и зануляются. Эффект регуляризации будет в том, что мы какую-то информацию занулим и нейронка не сможет под неё подстроиться.

На самом деле делается ещё нормализация, чтобы математическое ожидание выхода было такое же, как математическое ожидание входа: $\mathbb{E}f(x) = \mathbb{E}[x \odot m] = x \odot \mathbb{E}[m] = (1 - p)x$. Тогда мы умножим на $\frac{1}{1-p}$ и получим $f(x) = \frac{x \odot m}{1-p}$, чтобы сохранить математическое ожидание.

Однако на инференсе мы ничего не будем занулять по причине того, что мы уже не обучаемся, и нам нужна вся возможная информация про объект, чтобы сделать предсказание как можно точнее. То есть $f(x) = x$.

6.5. Batch normalization слой

Если мы применяем линейные слои - то их выходы могут быть любыми. Если у нас слишком большие веса в линейном слое, то выходы могут становиться очень большими. Если мы неудачно инициализировали и происходит такая ситуация, или наоборот, линейный слой всё сжимает в точку.

Давайте тот слой, который мы сейчас определим - будет выпускло нормированным.

Допустим есть $x \in \mathbb{R}^{B \times N}$. Давайте сделаем переход от x к нормированному \hat{x} : $x_i \in \mathbb{R}^N, x_i \mapsto \hat{x}_i$.

Тогда возьмём $\mu = \frac{1}{B} \sum_{i=1}^B x_i$ и $\sigma^2 = \frac{1}{B} \sum_{i=1}^B (x_i - \mu)^2$, $\mu, \sigma^2 \in \mathbb{R}^N$.

Тогда будем считать $\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$, $y_i = \hat{x}_i \odot \omega + b$, $\omega, b \in \mathbb{R}^N$.

Теперь давайте попробуем продифференцировать, для этого нам надо посчитать производную по входу, чтобы передать на следующие слои. И на самом деле, когда мы дойдём до \hat{x}_i , то у нас x_i, μ и σ зависят от x_i . Дифференцировать это сложно.

Каждый раз, когда мы будем делать предсказание, на этом слое эти статистики (μ и σ) мы будем сглаживать. Для этого у нас есть ещё два параметра, который мы используем в проходе через режим обучения.

- $\text{running_mean} = (1 - m) \cdot \text{running_mean} + m \cdot \mu$
- $\text{running_var} = (1 - m) \cdot \text{running_var} + m \cdot \sigma^2 \cdot \frac{B}{B-1}$

где running_mean и running_var изначально как-то инициализированы и потом обновляются.

Как модель работает на предсказании?

$\hat{x}_i = \frac{x_i - \text{running_mean}}{\sqrt{\text{running_var} + \epsilon}}$, где параметры running_mean и running_var мы берём от обучения.
 $y_i = \hat{x}_i \odot \omega + b$

7. Лекция 2. Построение и обучение нейронной сети. Наша лекция

Продолжим ноутбук от 5 раздела.

7.1. Построение архитектуры сети

Давайте наконец зададим архитектуру нейронной сети. Общий способ.

Нам понадобится подмодуль torch:

```
from torch import nn
```

Теперь зададим класс, который будет задавать нашу нейронную сеть:

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(in_features=28*28, out_features=64),
            nn.ReLU(),
            nn.Linear(in_features=64, out_features=64),
            nn.ReLU(),
            nn.Linear(in_features=64, out_features=10)
        )

    def forward(self, x):
        return self.model(x)
```

Посмотрим что будет, если создать объект такого класса:

```
model = MLP()
model
# MLP(
#   (model): Sequential(
#     (0): Linear(in_features=784, out_features=64, bias=True)
#     (1): ReLU()
#     (2): Linear(in_features=64, out_features=64, bias=True)
#     (3): ReLU()
#     (4): Linear(in_features=64, out_features=10, bias=True)
#   )
# )
```

Возьмём случайные данные:

```
x = torch.randn(64, 28*28)
y = model(x)
```

Посмотрим что лежит в y:

```
y.shape
# torch.Size([64, 10])
```

7.2. Как обучать модель

Давайте теперь ещё скажем как нашу модель обучать:

```
model = MLP()
```

```
for name, param in model.named_parameters():
    print(name, param.shape)
# model1.0.weight torch.Size([64, 784])
# model1.0.bias torch.Size([64])
# model1.2.weight torch.Size([64, 64])
# model1.2.bias torch.Size([64])
# model1.4.weight torch.Size([10, 64])
# model1.4.bias torch.Size([10])
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=1e-2, momentum=0.9)
criterion = nn.CrossEntropyLoss()
```

Теперь напишем наш TrainLoader и

```
for images, labels in train_loader:
    optimizer.zero_grad()
    logits = model(torch.flatten(images, start_dim=1))
    loss = criterion(logits, labels)
    loss.backward()
    optimezer.step()

print(loss.item())
```

Давайте теперь сделаем так, чтобы у нас было несколько эпох:

```
for epoch in range(1, num_epochs + 1):
    for images, labels in train_loader:
        optimizer.zero_grad()
        logits = model(torch.flatten(images, start_dim=1))
        loss = criterion(logits, labels)
        loss.backward()
        optimezer.step()

    print(loss.item())
```

Теперь сделаем ещё валидацию:

```
num_epochs = 10
for epoch in range(1, num_epochs + 1):
    for images, labels in train_loader:
        optimizer.zero_grad()
        logits = model(torch.flatten(images, start_dim=1))
        loss = criterion(logits, labels)
        loss.backward()
        optimezer.step()

    with torch.no_grad():
        for images, labels in test_loader:
            logits = model(torch.flatten(images, start_dim=1))
            loss = criterion(logits, labels)
```

with `torch.no_grad()` мы используем для того, чтобы не считать никакие градиенты и не тратить на это ресурсы. В инференсе нам градиенты не нужны. В новых версиях torch'a нужно использовать `with torch.inference_model()`

Давайте теперь на каждой эпохе ещё считать accuracy:

```
num_epochs = 10
for epoch in range(1, num_epochs + 1):
    for images, labels in train_loader:
        optimizer.zero_grad()
        logits = model(torch.flatten(images, start_dim=1))
        loss = criterion(logits, labels)
        loss.backward()
        optimezer.step()

    accuracy = (logits.argmax(dim=1) == labels).to(torch.float32).mean()

    with torch.no_grad():
        for images, labels in test_loader:
            logits = model(torch.flatten(images, start_dim=1))
            loss = criterion(logits, labels)

    accuracy = (logits.argmax(dim=1) == labels).to(torch.float32).mean()
```

7.3. Визуализация обучения

Но нам бы лучше наш loss всегда визуализировать. Давайте заведём массивы и будем в них складывать наши loss'ы, а потом визуализировать их. А также - будем переводить модель из режима обучения в режим инференса:

```
num_epochs = 10
train_losses, test_losses = [], []
train_accuracies, test_accuracies = [], []

for epoch in range(1, num_epochs + 1):
    model.train()
    running_loss, running_acc = 0.0, 0.0

    for images, labels in train_loader:
        optimizer.zero_grad()
        logits = model(torch.flatten(images, start_dim=1))
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

        accuracy = (logits.argmax(dim=1) == labels).to(torch.float32).sum()
        running_loss += loss.item() * images.shape[0]
        running_acc += accuracy

    train_losses.append(running_loss / len(train_loader.dataset))
    train_accuracies.append(running_acc / len(train_loader.dataset))

    running_loss, running_acc = 0.0, 0.0
    model.eval()
    with torch.no_grad():
        for images, labels in test_loader:
            logits = model(torch.flatten(images, start_dim=1))
            loss = criterion(logits, labels)

            accuracy = (logits.argmax(dim=1) == labels).to(torch.float32).mean()
            running_loss += loss.item() * images.shape[0]
            running_acc += accuracy

    test_losses.append(running_loss / len(test_loader.dataset))
    test_accuracies.append(running_acc / len(test_loader.dataset))
```

Напишем функцию, которая нам будет строить график потерь:

```
import seaborn as sns
import matplotlib.pyplot as plt
from IPython.display import clear_output
from tqdm.notebook import tqdm

sns.set_style('whitegrid')
plt.rcParams.update({'font.size': 15})

def plot_losses(train_losses, test_losses, train_accuracies, test_accuracies):
    clear_output()
    fix, axs = plt.subplots(1, 2, figsize=(13, 4))
    axs[0].plot(range(1, len(train_losses) + 1), train_losses, label='train')
    axs[0].plot(range(1, len(test_losses) + 1), test_losses, label='test')
    axs[0].set_ylabel('loss')

    axs[1].plot(range(1, len(train_accuracies) + 1), train_accuracies, label='train')
    axs[1].plot(range(1, len(test_accuracies) + 1), test_accuracies, label='test')
    axs[1].set_ylabel('accuracy')

    for ax in axs:
```

```
    ax.set_xlabel('epoch')
    ax.legend()

plt.show()
```

Теперь обучим нашу модель, отслеживая прогресс с помощью tqdm:

```
num_epochs = 10
train_losses, test_losses = [], []
train_accuracies, test_accuracies = [], []

for epoch in range(1, num_epochs + 1):
    model.train()
    running_loss, running_acc = 0.0, 0.0

    for images, labels in tqdm(train_loader, desc=f'Training-{epoch}/{num_epochs}'):
        optimizer.zero_grad()
        logits = model(torch.flatten(images, start_dim=1))
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

        accuracy = (logits.argmax(dim=1) == labels).to(torch.float32).sum()
        running_loss += loss.item() * images.shape[0]
        running_acc += accuracy

    train_losses.append(running_loss / len(train_loader.dataset))
    train_accuracies.append(running_acc / len(train_loader.dataset))

    running_loss, running_acc = 0.0, 0.0
    model.eval()
    with torch.no_grad():
        for images, labels in tqdm(test_loader, desc=f'Test-{epoch}/{num_epochs}'):
            logits = model(torch.flatten(images, start_dim=1))
            loss = criterion(logits, labels)

            accuracy = (logits.argmax(dim=1) == labels).to(torch.float32).mean()
            running_loss += loss.item() * images.shape[0]
            running_acc += accuracy

    test_losses.append(running_loss / len(test_loader.dataset))
    test_accuracies.append(running_acc / len(test_loader.dataset))

plot_losses(train_losses, test_losses, train_accuracies, test_accuracies)
```

Теперь нам нужно встроить Scheduler LR:

```
num_epochs = 10
model = MLP()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-2, momentum=0.9)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)
criterion = nn.CrossEntropyLoss()
train_losses, test_losses = [], []
train_accuracies, test_accuracies = [], []

for epoch in range(1, num_epochs + 1):
    model.train()
    running_loss, running_acc = 0.0, 0.0

    for images, labels in tqdm(train_loader, desc=f'Training-{epoch}/{num_epochs}'):
        optimizer.zero_grad()
        logits = model(torch.flatten(images, start_dim=1))
        loss = criterion(logits, labels)
        loss.backward()
```

```

optimezer.step()

accuracy = (logits.argmax(dim=1) == labels).to(torch.float32).sum()
running_loss += loss.item() * images.shape[0]
running_acc += accuracy

scheduler.step()
train_losses.append(running_loss / len(train_loader.dataset))
train_accuracies.append(running_acc / len(train_loader.dataset))

running_loss, running_acc = 0.0, 0.0
model.eval()
with torch.no_grad():
    for images, labels in tqdm(test_loader, desc=f'Test_{epoch}/{num_epochs}'):
        logits = model(torch.flatten(images, start_dim=1))
        loss = criterion(logits, labels)

        accuracy = (logits.argmax(dim=1) == labels).to(torch.float32).mean()
        running_loss += loss.item() * images.shape[0]
        running_acc += accuracy
    test_losses.append(running_loss / len(train_loader.dataset))
    test_accuracies.append(running_acc / len(train_loader.dataset))

plot_losses(train_losses, test_losses, train_accuracies, test_accuracies)

```

7.4. Сохранение нашей модели

У torch есть `torch.save`, который работает как pickle:

```

torch.save({
    'model_state': model.state_dict(),
    'optimizer_state': optimizer.state_dict(),
    'scheduler_state': scheduler.state_dict()
}, 'checkpoint.pt')

```

О модели у нас ничего загружено не будет, только веса. Как их применять - это уже проблема того, что будет использовать эту модель.

7.5. Загрузка модели

```
ckpt = torch.load('checkpoint.pt')
```

Если мы теперь захотим инициализировать нашу модель весами, которые сохранили, мы сделаем следующее:

```

model = MLP()
model.load_state_dict(ckpt['model_state'])
# <All keys matched successfully>

```

7.6. Обучение на GPU

```

!nvidia-smi
import torch

from torchvision.datasets import MNIST

train_set = MNIST('mnist', train=True, download=True)
test_set = MNIST('mnist', train=False, download=True)

import torchvision.transforms as T

```

```

transform = T.Compose([
    T.ToTensor(),
    T.Resize(28),
])

train_set = MNIST('mnist', transform=transform, train=True, download=True)
test_set = MNIST('mnist', transform=transform, train=False, download=True)

train_loader = DataLoader(train_set, batch_size=64, shuffle=True, num_workers=4, pin_memory=True)
test_loader = DataLoader(test_set, batch_size=64, shuffle=False, num_workers=4, pin_memory=True)

class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(in_features=28*28, out_features=64),
            nn.ReLU(),
            nn.Linear(in_features=64, out_features=64),
            nn.ReLU(),
            nn.Linear(in_features=64, out_features=10)
        )

    def forward(self, x):
        return self.model(x)

```

Для работы на GPU нужно задать device:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

Продолжаем, теперь создаём модель:

```

import seaborn as sns
import matplotlib.pyplot as plt
from IPython.display import clear_output
from tqdm.notebook import tqdm

sns.set_style('whitegrid')
plt.rcParams.update({'font.size': 15})

def plot_losses(train_losses, test_losses, train_accuracies, test_accuracies):
    clear_output()
    fix, axs = plt.subplots(1, 2, figsize=(13, 4))
    axs[0].plot(range(1, len(train_losses) + 1), train_losses, label='train')
    axs[0].plot(range(1, len(test_losses) + 1), test_losses, label='test')
    axs[0].set_ylabel('loss')

    axs[1].plot(range(1, len(train_accuracies) + 1), train_accuracies, label='train')
    axs[1].plot(range(1, len(test_accuracies) + 1), test_accuracies, label='test')
    axs[1].set_ylabel('accuracy')

    for ax in axs:
        ax.set_xlabel('epoch')
        ax.legend()

    plt.show()

num_epochs = 10
model = MLP().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-2, momentum=0.9)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)
criterion = nn.CrossEntropyLoss().to(device)
train_losses, test_losses = [], []

```

```

train_accuracies, test_accuracies = [], []

for epoch in range(1, num_epochs + 1):
    model.train()
    running_loss, running_acc = 0.0, 0.0

    for images, labels in tqdm(train_loader, desc=f'Training-{epoch}/{num_epochs}'):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        logits = model(torch.flatten(images, start_dim=1))
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

        accuracy = (logits.argmax(dim=1) == labels).detach().cpu().to(torch.float32).sum()
        running_loss += loss.item() * images.shape[0]
        running_acc += accuracy

    scheduler.step()
    train_losses.append(running_loss / len(train_loader.dataset))
    train_accuracies.append(running_acc / len(train_loader.dataset))

    running_loss, running_acc = 0.0, 0.0
    model.eval()
    with torch.no_grad():
        for images, labels in tqdm(test_loader, desc=f'Test-{epoch}/{num_epochs}'):
            images = images.to(device)
            labels = labels.to(device)
            logits = model(torch.flatten(images, start_dim=1))
            loss = criterion(logits, labels)

            accuracy = (logits.argmax(dim=1) == labels).detach().cpu().to(torch.float32).sum()
            running_loss += loss.item() * images.shape[0]
            running_acc += accuracy

        test_losses.append(running_loss / len(test_loader.dataset))
        test_accuracies.append(running_acc / len(test_loader.dataset))

plot_losses(train_losses, test_losses, train_accuracies, test_accuracies)

```