

Содержание

I Базовый поток Deep Learning School	5
1 Введение в нейронные сети. Часть 1. История развития Deep Learning	5
1.1 История глубинного обучения	5
1.2 Когда появился нейрон	5
1.3 Конференция психологов, математиков и кибернетиков	5
1.4 McCulloch и Pitt: первая искусственная нейронная сеть	5
1.5 Rosenblatt: перцептрон	5
1.6 Minsky и Papert: XOR problem, палки в колёса DL	6
1.7 Werbos: Backpropagation	6
1.8 Постепенное развитие в обществе	6
1.9 Утрата интереса к нейронным сетям	6
1.10 AlexNext - камбек нейронных сетей	6
2 Введение в нейронные сети. Часть 2. Механизм обратного распространения ошибки	8
2.1 Вспомним о логистической регрессии	8
2.2 Линейная неразделимость	8
2.3 Решение проблемы линейно неразделимой выборки	8
2.4 Функции	9
2.4.1 Сигмоида	9
2.4.2 Гиперболический тангенс	10
2.4.3 ReLU	10
2.4.4 Softplus	10
2.5 Структура нейронной сети	11
2.6 Примеры сетей	11
2.7 Backpropagation	12
2.7.1 Пример применения	12
2.7.2 Ещё один пример	12
2.8 Backpropagation в матричной форме	13
3 Введение в нейронные сети. Часть 3. Функции активации. Краткий обзор применений CNN и RNN	14
3.1 Функции активации	14
3.1.1 Сигмоида	14
3.1.2 Гиперболический тангенс	15
3.1.3 ReLU	15
3.1.4 Leaky ReLU	16
3.1.5 PReLU	16
3.1.6 ELU	16
3.2 Советы по выбору функции активации	17
3.3 Fancy neural networks	17
3.3.1 RNN - Recurrent Neural Networks	17
3.3.2 Свёрточные нейронные сети	17
4 Лекция. История развития сверточных нейронных сетей	18
4.1 ImageNet	18
4.2 HOG (Histogram of Oriented Gradients)	18
4.3 AlexNet	19

5 Лекция. Сверточные нейронные сети	21
5.1 MNIST	21
5.2 Первые мысли как решать задачу классификации картинок	21
5.3 Как перенести пространственную информацию в сеть?	21
5.4 Свёртка	21
5.4.1 Ядро	21
5.4.2 Padding	22
5.4.3 Stride	22
5.4.4 Свёртка цветных изображений	22
5.4.5 Фильтры	22
5.5 Применение фильтров в нейронных сетях	22
5.6 Real-world изображения, несколько слоёв свёртки	23
5.7 Функция активации для свёрточных сетей	24
5.8 Обучение свёрточной нейронной сети	24
6 Лекция. Пуллинг. Операция пулинга	25
6.1 Проблема больших карт активаций на выходе	25
6.2 Pooling	25
6.3 Куда вставлять слой polling'a?	26
6.4 Обзор слоёв AlexNet	26
7 Лекция. Задачи компьютерного зрения	27
7.1 Классификация, детекция, сегментация	27
7.1.1 Классификация	27
7.1.2 Детекция	27
7.1.3 Сегментация	27
7.2 Ещё применение задач CV	28
8 Лекция. Градиентная оптимизация в Deep Learning	29
8.1 Обучение нейронной сети	29
8.2 Stochastic Gradient Descent	29
8.3 SGD Momentum	30
8.4 SGD Nesterov Momentum	30
8.5 Adagrad	31
8.6 RMSProp	31
8.7 RMSProp normalization	31
9 Лекция. Регуляризация в Deep Learning	32
9.1 Пример переобучения	32
9.2 Нормировка данных	33
9.3 Почему нужно нормировать данные не только на входе?	33
9.4 Как раньше боролись с нормировкой данных внутри сети?	34
9.5 Batch-norm	34
9.6 Регуляризации	34
9.6.1 Weight decay	34
9.6.2 Dropout	35
9.6.3 Аугментация данных	35
10 Лекция. Архитектуры CNN	36
11 AlexNet	36
11.1 VGG	36
11.2 Затухание градиентов	37
11.3 Skip connection	37
11.4 ResNet	37
11.5 DenseNet	38
11.6 Bottleneck Block	39
11.7 Model ZOO	39
11.8 Inception (GoogleNet)	39

12 Лекция. Transfer Learning	40
12.1 Fine-tuning	40
12.2 Варианты как передать информацию из одной сети к другой	40
12.2.1 Domain Adaptation	41
12.3 Идеи решения задач Transfer Learning	41
12.3.1 Одноковое распределение слоёв	41
12.3.2 Ещё идеи - почитать статью	41
12.3.3 Архитектурное разделение	41
12.3.4 Extreme cases	41
II Продвинутый поток	42
13 Семантическая сегментация. Введение	42
13.1 Идея решения: Sliding Window	42
13.2 Fully-Conv network	42
13.3 CNN	42
14 Семантическая сегментация. Трюки: Deconvolution, Dilated Convolution	44
14.1 Замена upsampling	44
14.2 Варианты upsampling'a	44
14.2.1 Nearest Neighbours	44
14.2.2 Bed of nails	44
14.2.3 Bilinear	44
14.2.4 Max-unpooling	45
14.3 Dilated Convolutions	45
14.4 Multi-scale Context Aggregator	45
14.5 Pyramid pooling network	45
14.6 Семантическая сегментация. Архитектура UNet	46
14.7 Overlap-tile strategy	47
14.8 UNet: Loss	47
15 Лекция. Нейронная детекция объектов. Основы	49
15.1 Детекция объектов: постановка задачи	49
15.2 Датасеты	49
15.2.1 Датасет MS COCO 2017	49
15.2.2 Google Open Images	50
15.3 Метрики	50
15.3.1 Intersection over Union (Jaccard Index)	50
15.3.2 Positives and Negatives	50
15.4 Как решать bounding box?	51
16 Лекция. Нейронная детекция объектов. Двухстадийные нейросети	52
16.1 Классификация детекторов	52
16.2 Two-staged подход	52
17 Лекция. Нейронная детекция объектов. Одностадийные нейросети	53
18 Лекция. Генеративные модели, автоэнкодеры	54
18.1 Auto Encoder	55
18.1.1 Corruption denoising	56
18.1.2 Детекция аномалий	57
18.2 Variational Auto-Encoder	58
18.2.1 Loss	58
18.2.2 Дивергенция	59
18.2.2.1 KL-дивергенция	59
18.3 Сравнение латентных пространств AE и VAE	59
18.4 Пример обучения латентного пространства VAE о преобразовании собаки в птицу	60
18.5 Пример обучения латентного пространства VAE для получения весёлого человека	60
18.6 Лекция. Генеративные модели. Генеративно-состязательные сети	61
18.7 Идея GAN	62
18.7.1 Loss функция	62

18.8 Как мерить качество GAN моделей?	62
18.9 Немного трюков	62
III Вторая часть курса продвинутого потока	63
19 Введение в NLP	63
19.1 Что такое Natural Lnaguage Processing?	63

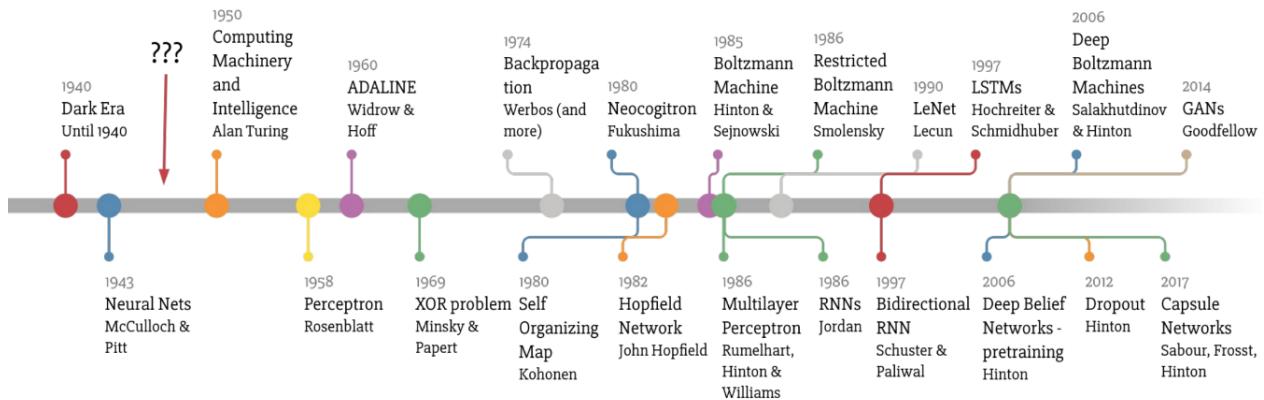
Часть I

Базовый поток Deep Learning School

1. Введение в нейронные сети. Часть 1. История развития Deep Learning

https://www.youtube.com/watch?v=ZfXpX8tMg-w&list=PL0Ks75aof3Th84kETS1Jq_ja-xqLtWov1&index=29ye

1.1. История глубинного обучения



1.2. Когда появился нейрон

До 40-х годов как будто бы ничего не происходило, но на самом деле это не так. Термин "нейрон" был введён в 1891 году. Его появление обязано аппарату раскрашивания различных клеток, который был изобретён Гольджи в 1873 году. Нейроны, как структурные единицы нервной ткани, получили ответственность за запоминание информации и выработку рефлексов, а также за агрегирование опыта и обратного подкрепления.

Также хочется сделать ремарку в сторону психологов: Уильям Джемс, один из самых известных профессоров психологии.

1.3. Конференция психологов, математиков и кибернетиков

В 1940-х годах в США собралась группа, состоящая преимущественно из психологов, математиков и кибернетиков и обсуждался вопрос: "Как создать искусственное сознание?" Уже тогда люди понимали, что с помощью использования математических методов можно будет создать цифровую копию (или просто искусственное сознание). Во многом по этому искусственные нейронные сети обязаны подходом и идеям к психологам.

1.4. McCulloch и Pitt: первая искусственная нейронная сеть

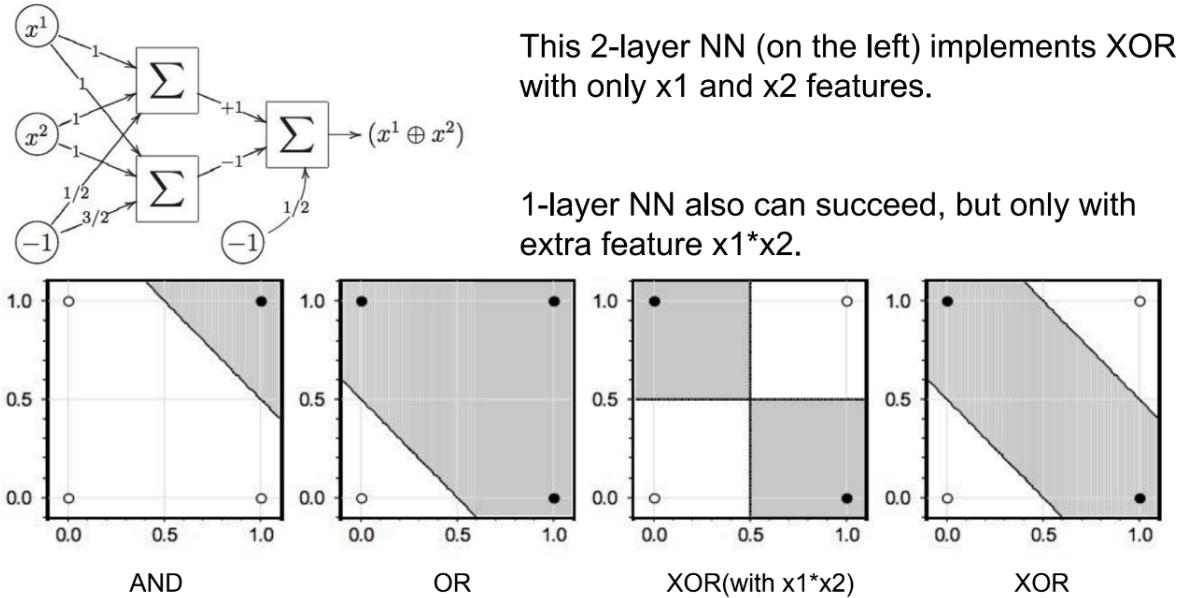
Первая веха, на которую стоит обратить внимание - 1943 год. Работа, представленная учителем и учеником: McCulloch и Pitt, они представили математическую модель искусственных нейронных сетей. Во многом это была прорывная работа, показывающая новый взгляд на происходящее. Не зря их модель разбирается практически в каждом курсе по нейронным сетям. Но есть несколько ремарок: Они ввели отличный понятийный аппарат, а во-вторых определили развитие области на несколько лет вперёд.

Но нейронные сети в целом сложная штука и их ждало много проблем и много задач.

1.5. Rosenblatt: перцептрон

В 1958 году Rosenblatt представил перцептрон в виде модели. Это математическая модель нейрона, которая на основании проб и ошибок пытается различать цвета, которые попадали на фотодатчик. По нынешним меркам это не очень захватывающе, но на 1958 год это было очень прорывной технологией. Как раз тогда примерно и начали создаваться методы компьютерного зрения.

1.6. Minsky и Papert: XOR problem, палки в колёса DL



Спрашивается: а как построить линейную разделяющую поверхность для XOR? Нельзя, точки неразделимы.

К этому есть два подхода: ввести новый признак (например признак $x_1 \times x_2$).

Второй подход - сделать второй слой, и результат этого видно на последней картинке.

Дальше идёт череда достаточно важных открытых.

1.7. Werbos: Backpropagation

Появление метода обратного распространения ошибки в 1974 году ознаменовало переход к новым сетям современного типа, где они стали каскадом каких-то дифференцируемых преобразований (дифференцируемость необходима, чтобы посчитать производные и обновить веса). После этого появляется, некоторого рода, эйфория, где люди думают: "Ура, нейронные сети обучаются, они хорошие, давайте их везде использовать".

1.8. Постепенное развитие в обществе

Спустя некоторое время возникает следующая ситуация: нейронные сети уже достаточно неплохо обучаются, появляются первые свёрточные нейронные сети, появляется LeNet (Lecun), появляются новые модели и возникает тонкий момент. Нейронные сети всё ещё немного маленькие, но всё ещё неплохо участвуют, а данных накоплено много. Человечество нашло модели, которые не переобучаются, то есть неплохо обобщают знание из обучающей выборки и при этом бьют в пух и прах многие модели.

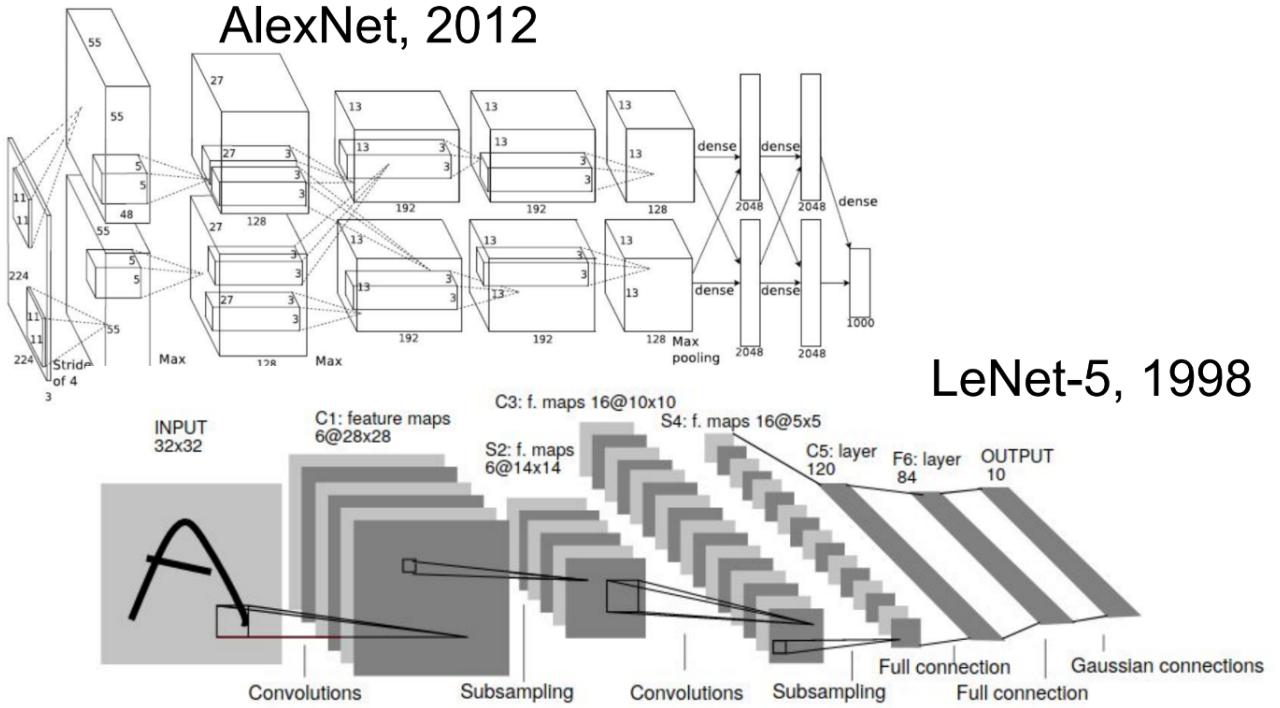
1.9. Утрата интереса к нейронным сетям

Постепенно получаются всё более и более сложные нейронные сети и выясняется, что переобучается они очень хорошо, лучше чем другие модели, и происходит это примерно в 2000-х годах. В это же время появляется другой подход обучения, основанный на градиентном бустинге. После этого интерес к нейронным сетям, на почве событий угасает на десять лет.

1.10. AlexNet - камбек нейронных сетей

Так происходит до 2011 года, пока на соревновании ImageNet нейронные сети не переворачивают всю ситуацию в свою пользу. Что произошло?

AlexNet, 2012



Появилась сеть AlexNet, которая победила с большим отрывом на соревновании ImageNet - соревнование по классификации изображений. AlexNet по своей структуре очень похожа на LeNet-5, которая была ещё в 1998 году, между ними почти 15 лет разницы, по структуре они похожи, но результат AlexNet был феерическим.

Почему AlexNet так взорвала мир нейросетей? К 2012-м году был накоплен большой объём данных, который позволял очень хорошо учиться и не переобучаться. Также нейронные сети по своей структуре хорошо подходят для работы с данными, обладающими внутренней связностью или структурой. Например изображение это набор пикселей, и информация состоит в том, как они расположены.

После этого и начался бум и было решено много задач с помощью машинного обучения.

2. Введение в нейронные сети. Часть 2. Механизм обратного распространения ошибки

https://www.youtube.com/watch?v=-yiq1DRX9K0&list=PL0Ks75aof3Th84kETS1Jq_ja-xqLtWov1&index=29

2.1. Вспомним о логистической регрессии

Как строить нейронные сети и как вообще говоря понять, что там происходит? Для начала, давайте начнём с простого и вернёмся в основам.

Вот у нас есть линейная модель: логистическая регрессия.

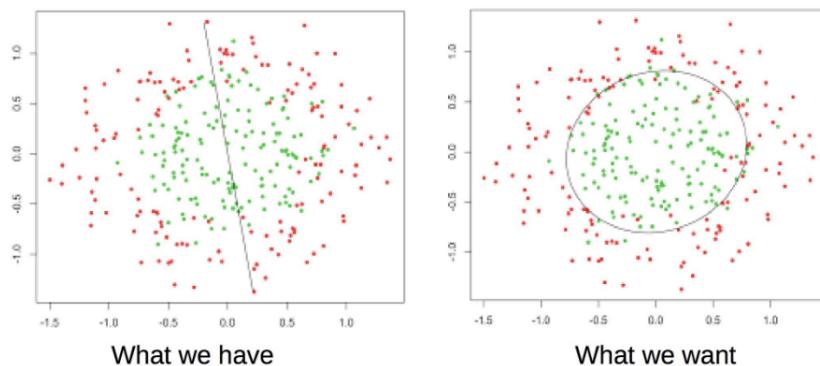
$$\mathbb{P}(y|x) = \sigma(\omega x + b)$$

$$L = - \sum_i y_i \log \mathbb{P}(y|x_i) + (1 - y_i) \log(1 - \mathbb{P}(y|x_i))$$

У нас есть некоторое признаковое описание x_i , затем мы производим линейное преобразование $\omega x + b$, применяем функцию активации - нелинейную сигмоиду. И получаем вероятность принадлежности объекта к каким-то классам. У нас есть функция потерь L - logloss или же кросс-энтропия. Мы можем оптимизировать параметры нашей модели, используя градиентные методы, чтобы подобрать оптимальные ω и b , чтобы решать задачу очень хорошо.

2.2. Линейная неразделимость

А что делать, когда у нас выборка линейно неразделима?



Решить эту задачу с помощью линейных моделей, той же самой логистической регрессии, не представляется возможным.

2.3. Решение проблемы линейно неразделимой выборки

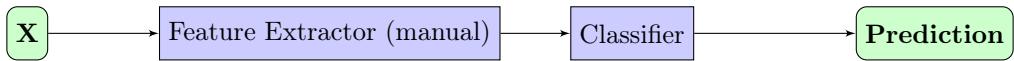
Подходов к решению этой задачи можно придумать много:

- Можно позвать деревья - они умеют строить нелинейную разделяющую поверхность, они это как-то сделают.
- Можно использовать SVM - использовать какое-нибудь ядро, которое позволит перейти в другое пространство и в нём выборка уже будет линейно разделима.
- Можно придумать новые признаки для наших данных, которое поможет разделить линейно.

Проблема в чём? Все подходы, помимо ансамбля деревьев требуют человеческого труда, человеческого гения, чтобы придумать новые признаки, подобрать ядро, и так далее. Это требует времени и сил. А когда мы говорим про построение эффективных и информативных моделей, нам бы хотелось максимально избавиться от какого-то труда по подбору оптимальной структуры и максимально это всё автоматизировать. Нейронные сети это как раз те механизмы и методы, которые позволили автоматизировать подбор признаковых пространств и построение информативных признаковых описаний наших объектов.

Как это работает?

Давайте ещё раз посмотрим на нашу логистическую регрессию:

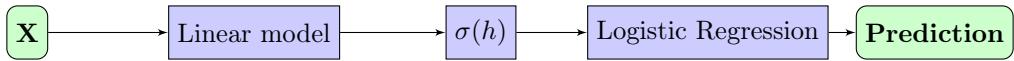


У нас есть, например, некоторое описание наших данных в каких-то признаках. Затем у нас, возможно, есть ручной отбор признаков. Потом берём все попарные между ними взаимодействия, и после этого строим нашу линейную модель. После этого предсказываем нашей моделью какой-то результат, обучаем параметры модели и так далее.

Но при этом отбор признаков происходит руками на основе наших решений. Это не очень хорошо автоматизируется. Давайте автоматизируем не только подбор параметров в классификаторе, но и подбор оптимальных признаков.

Как это можно сделать? Сделать это можно простым образом. Мы можем попросить какую-то модель, у которой есть параметры, генерировать нам признаки. Какую модель можно предложить?

Ну давайте это будет какая-то линейная модель:



Мы берём линейную модель: линрег. Применяем её к исходным данным. Затем нелинейная какая-то активация, сигмоида. После чего над этими признаками новыми применяем логистическую регрессию. А Зачем нам две линейные модели подряд? А давайте скажем, что линрег у нас отображает из 10 измерений в 50. То есть у нас есть 50 различных линейных комбинаций исходных 10 признаков. После чего мы помимо этого, ещё каждую линейную комбинацию покрываем сигмоидой сверху и она из, условного, \mathbb{R} переходит в $[0; 1]$. Зачем нам это нужно? А затем, что после этого у нас применяется логистическая регрессия, которая является линейной моделью. Линейная комбинация линейных комбинаций - есть линейная комбинация. Но так как мы с вами применили нелинейную сигмоиду, то комбинация нелинейная.

То же самое, что делал бы SVM с помощью другого ядра. Только в SVM мы подбираем ядра, а здесь функцию активации, но зато все остальные преобразования делаются моделью автоматически.

В итоге получается блок с сигмоидой, который имеет параметры, и от результата его работы зависит решение задачи. У нас логистическая регрессия может обучаться с помощью градиентного спуска, но при этом её результаты зависят от того, что пришло ей на вход. Каждый из блоков дифференцируем, а значит градиентные методы всё также работают, а значит могут найти не только значения параметров для лог рега, но и лин рега.

Для этого нам понадобится механизм обратного распространения ошибок.

2.4. Функции

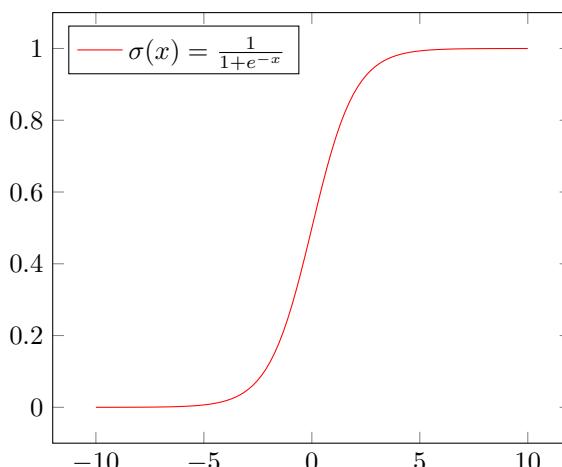
Допустим мы хотим построить нейронную сеть. Для этого нам понадобятся какие-то линейные преобразование, например $\omega x + b$. Во-вторых нам понадобятся нелинейные преобразования, чтобы из каскада линейных преобразований не получалось одно линейное преобразование.

Давайте разберём наши функции активации, которые и есть нелинейные преобразования.

2.4.1. Сигмоида

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

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

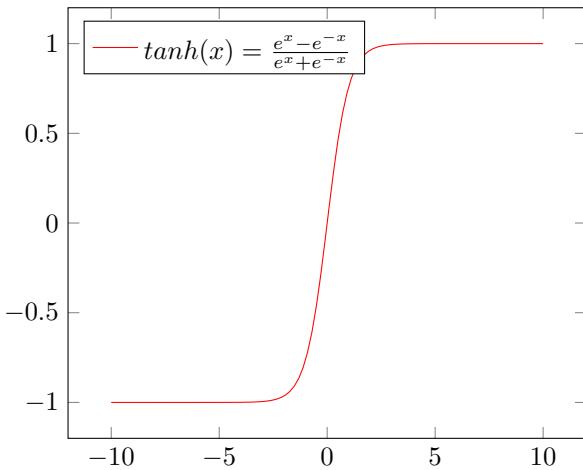


Она хороша тем, что переводит числа в отрезок $[0; 1]$.

2.4.2. Гиперболический тангенс

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

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

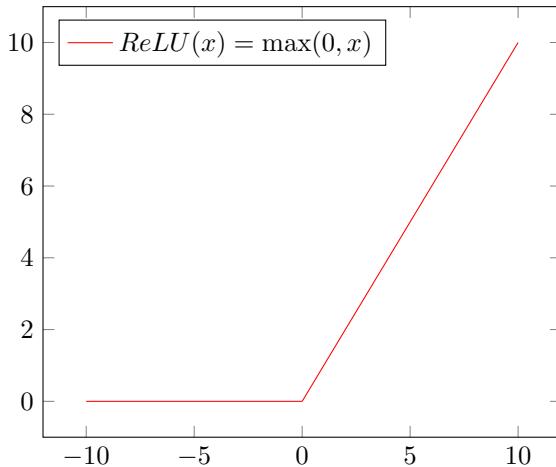


На первый взгляд от сигмоиды отличается мало чем, но на самом деле это отшкалированная сигмоида, но не от 0 до 1, а от -1 до 1.

2.4.3. ReLU

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

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



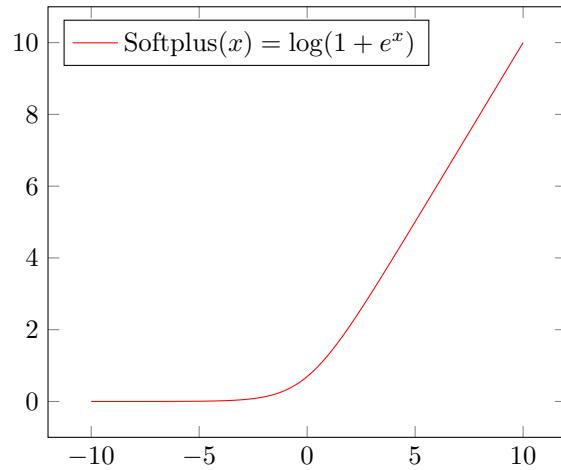
Внимательный слушатель предъявит то, что мы требовали дифференцируемые функции, а \max такой не является. В нуле производной нет.

Но нам нужно с вами, в каждой точке оценить градиент для наших параметров. Если посмотреть на ReLU, мы обнаружим, что при значениях $x > 0$ производная 1, при $x < 0$ производная 0. А в нуле мы можем сами определить производную. Гладкость функции при этом мы не требуем.

2.4.4. Softplus

$$\text{Softplus}(x) = \log(1 + e^x)$$

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



По сути решает проблему ReLU, определяя производную в нуле.

Функции активации нужны только для того, чтобы получить нелинейное преобразование над нашими признаками.

Что делать с ними дальше? Дальше строить с ними нейронную сеть.

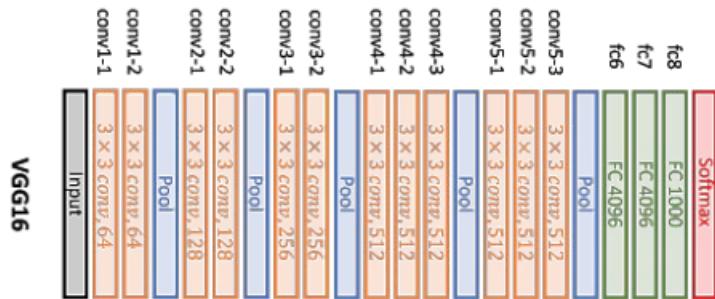
2.5. Структура нейронной сети

Введём понятие слоя. Слой - единица структуры нашей нейронной сети, включающее линейное преобразование, или которое может быть представлено в виде линейного. И как правило в слое есть функция активации. Свёртки можно представить как некоторое линейное преобразование, поэтому свёрточные слои тоже есть. Из слоёв и состоят сети.

Как правило ещё выделяют входной слой - просто исходное представление данных. Также выделяют выходной слой - итоговое представление данных: если задача классификации, то вероятность каждого класса, если задача регрессии - какое-то число или вектор из чисел, и так далее.

2.6. Примеры сетей

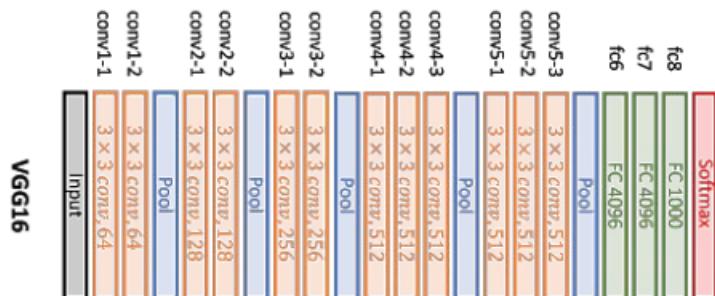
Вот простой пример VGG16:



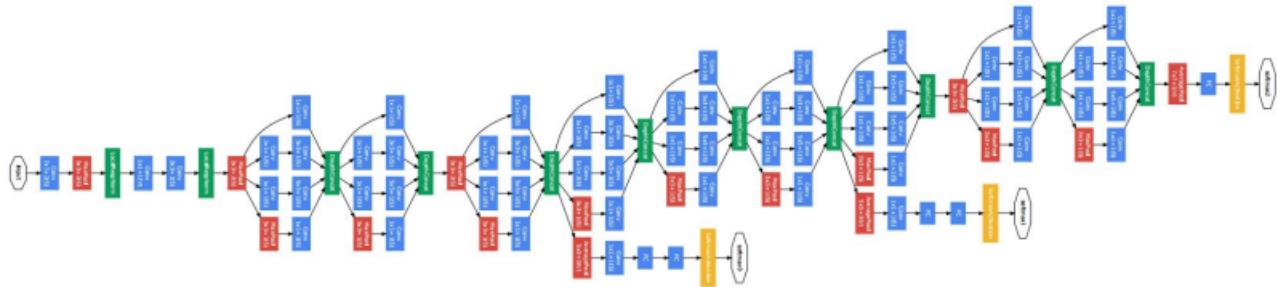
Эта сеть в 2014 показала отличный результат на ImageNet, значительно превосходящими результаты AlexNet. И здесь можно как раз увидеть слои сети.

У нас идёт несколько раз подряд несколько свёрточных слоёв и пулинг. Потом идёт три полносвязных слоя и в конце Softmax. Обучать такое вроде как понятно.

Но есть следующая версия VGG19, которая глубже:



А бывает ещё глубже, GoogleNet:



И все эти сети учатся с помощью обратного распространения ошибки. Как же это работает?

2.7. Backpropagation

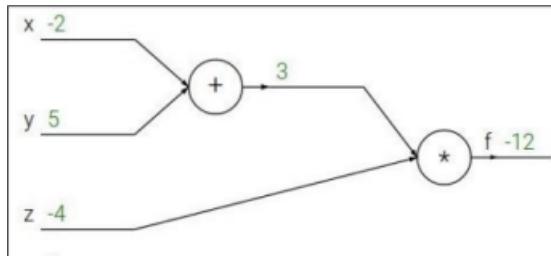
Backpropagation, на самом деле, является производной сложной функции, применение метода цепочек: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$

Вспомним про механизм градиентной оптимизации и зачем тут частная производная. Градиент - это направление, в котором функция растёт наиболее быстро. Если у нас есть функция потерь, то мы хотели бы знать направление наискорейшего убывания. Для этого нам и нужны все частные производные по параметрам. Посчитав их мы знаем направление градиента, а значит и антиградиента.

Если у нас есть некоторая функция потерь и мы хотим узнать, как нам нужно поменять значение x , нам нужно посчитать производную $\frac{\partial L}{\partial x}$. Но L напрямую не зависит от x , она зависит от x только через z - другую сущность, другое представление. Тогда мы можем посчитать $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$. А это нам нужно для того, чтобы посчитать градиент и знать в какую сторону нам двигать параметры, чтобы уменьшать функцию ошибки.

2.7.1. Пример применения

Давайте разберём простую функцию $f(x, y, z) = (x + y)z$, $x = -2, y = 5, z = -4$.



Давайте скажем, что $x + y$ объединяются в q , $\frac{\partial q}{\partial x} = \frac{\partial q}{\partial y} = 1$.

А $f = qz$, где $\frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$.

Теперь мы хотим найти $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$:

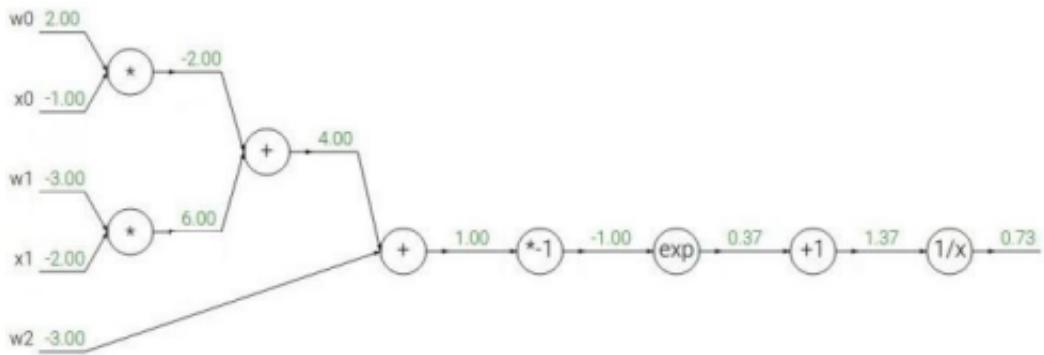
- $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = -4 \cdot 1$

- $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = -4 \cdot 1$

- $\frac{\partial f}{\partial z} = q = 3$

2.7.2. Ещё один пример

Зачем нам это нужно? Сделаем то же самое, но на более привычном примере, и воспользуемся теми же самыми методами, но для логистической регрессии: $f(w, x) = \frac{1}{1+e^{-(\omega_0 x_0 + \omega_1 x_1 + \omega_2)}}$.



Но здесь x - это данные, поменять мы их не можем, но вот производные по ω нам нужны. Давайте посчитаем производную по честному и через метод обратного распространения ошибки.

Блин, короче мне лень тихать - <https://youtu.be/-yiq1DRX9K0?t=999>

2.8. Backpropagation в матричной форме.

Пусть у нас y - вектор, и $y_i = f_i(x) = x_i$.

$$\text{Если мы хотим взять } \frac{\partial y}{\partial x}, \text{ то в матричном виде у нас это будет } \frac{\partial f}{\partial x} = \begin{bmatrix} \nabla f_1(x) \\ \nabla f_2(x) \\ \vdots \\ \nabla f_m(x) \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} \frac{\partial f_1(x)}{\partial x_2} \cdots \frac{\partial f_1(x)}{\partial x_n} \\ \frac{\partial f_2(x)}{\partial x_1} \frac{\partial f_2(x)}{\partial x_2} \cdots \frac{\partial f_2(x)}{\partial x_n} \\ \vdots \\ \frac{\partial f_m(x)}{\partial x_1} \frac{\partial f_m(x)}{\partial x_2} \cdots \frac{\partial f_m(x)}{\partial x_n} \end{bmatrix}$$

Нужно проговорить:

- Производная скаляра по скаляру: скаляр
- Производная скаляра по вектору: вектор
- Производная вектора по скаляру: вектор
- Производная вектора по вектору: матрица

Что теперь дальше делать? Правило обновлений точно такое же: $x_{t+1} = x_t - lr \cdot dx$.

3. Введение в нейронные сети. Часть 3. Функции активации. Краткий обзор применений CNN и RNN

https://www.youtube.com/watch?v=3F7rydcAa0w&list=PL0Ks75aof3Th84kETS1Jq_ja-xqLtWov1&index=31

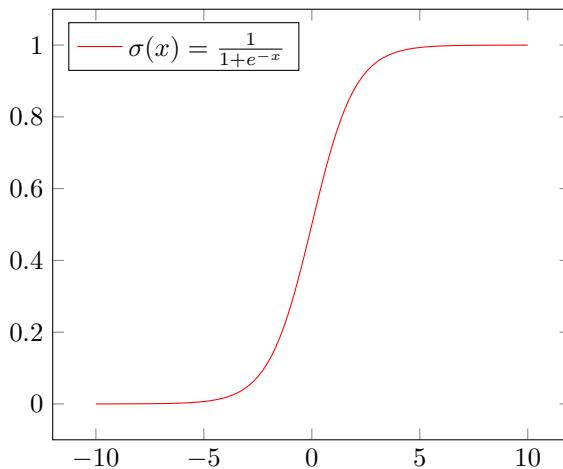
3.1. Функции активации

Про функции активации мы уже говорили - это те функции, которые делают нелинейные преобразования над нашими данными.

Нейронная сеть, по сути, каждый шаг строит новое признаковое представление наших данных. На каждом слое происходит ровно это. И чтобы у нас представления не были чередой линейных преобразований (которые могут склонуться в одно линейное преобразование), мы применяем функции активации. На самом деле их не спроста не так много и они все обладают своими свойствами.

3.1.1. Сигмоида

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
$$\sigma : \mathbb{R} \mapsto (0, 1)$$



Она неплохо интерпретируется: отображение из \mathbb{R} в $[0; 1]$, что можно интерпретировать как вероятность. Допустим её можно вывести как раз из метода максимального правдоподобия.

Плюсы:

- Неплохо интерпретируется. Можно считать вероятностью активации нейрона. Во многом эта функция и была вдохновлена нейронами биологическими по принципу их работы на первый взгляд - накапливают потенциал и активируются.
- Она дифференцируема

Минусы:

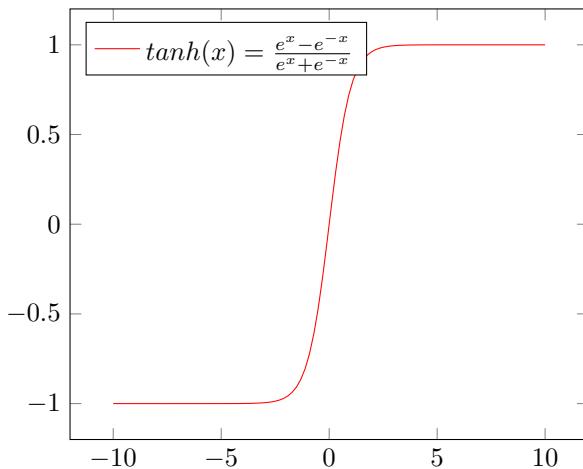
- Сигмоида обладает значительными хвостами, где производная будет ноль. Мы с вами считаем частную производную, и производная с хвостами для конкретной компоненты, будет почти ноль. Если справа от функции потерян шаг какая-то производная - эта производная умножилась на ноль, получили суммарно ноль и это называется затуханием градиентов.
- У сигмоиды не центрированный выход относительно нуля. Так как нейронные сети представляют собой каскад линейных моделей это плохо, так как линейные модели любят нормированные данные.
- Экспоненту сложно считать под капотом компьютера. При этом её нужно считать как при forward-pass'e, так и при backward-pass'e, так как производная экспоненты - экспонента.

Сигмоида, как и все функции активации применяется покомпонентно. Мы к каждой компоненте матрицы или вектора применяем сигмоиду, и ничего об остальных компонентах не знаем.

3.1.2. Гиперболический тангенс

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

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



В сравнении с сигмоидой - все те же плюсы и минусы, помимо того, что у нас данные отцентрированы, это естественно теперь плюс.

Но при этом ограничение на выходное значение очень важно, когда мы говорим про рекуррентные нейронные сети, там это используется. И те функции активации, которые мы разберём далее, не используются в рекуррентных нейронных сетях, потому что там функции активации должны иметь ограниченную область значений. Гиперболический тангенс используется чаще сигмоиды в рекуррентных нейронных сетях, и больше других функций активаций, в связи с рядом специфических ограничений.

3.1.3. ReLU

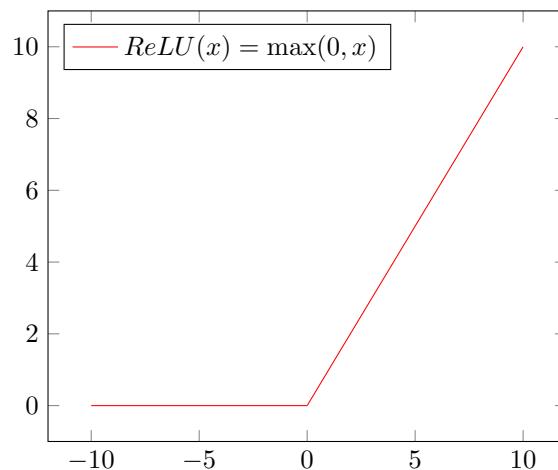
Как нам добиться того, чтобы наш градиент не затухал на хвостах, и нам было недорого посчитать производную?

Чтобы градиент не затухал - нужно, чтобы не было хвостов.

Как недорого посчитать производную - избавиться от экспоненты.

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

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



Плюсы:

- Производная справа нуля не затухает.
- Экспоненту считать не надо: у нас производная либо 0, либо 1.

Минусы:

- Выходное значение не центрировано.
- Слева на хвосте затухает градиент. Плохо.

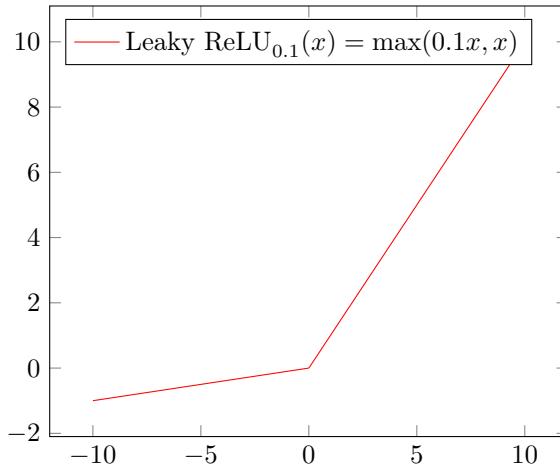
В нуле мы можем доопределить производную нулём, и тогда всё дифференцирование, которое нам необходимо - будет работать.

3.1.4. Leaky ReLU

Как починить, что у ReLU выход не центрированный? Давайте сделаем её чуть менее несимметричной:

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

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



Теперь выходное значение чуть более центрированное и градиент теперь не затухает, потому что слева от нуля тоже есть маленькая производная. Производная равна нулю теперь только в нуле. Красиво, удобно и работает.

3.1.5. PReLU

PReLU - Parametric ReLU. Теперь вместо того, чтобы выбирать коэффициент α - нейросеть сама будет его находить и это будет не гиперпараметр, а параметр (вес) для сети:

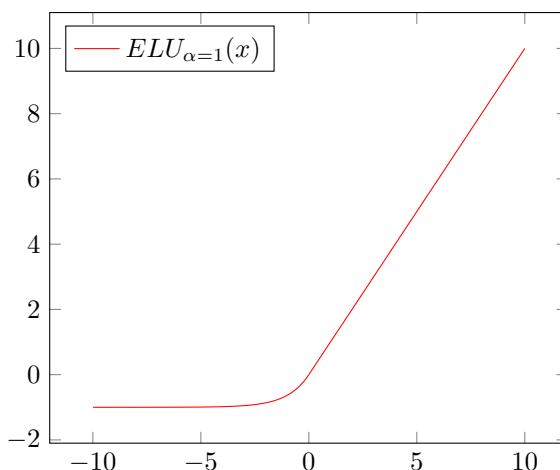
$$PReLU(x) = \max(\alpha x, x), \quad 0 < \alpha \ll 1$$

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

3.1.6. ELU

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

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



Содержит все преимущества ReLU, не затухает градиент и более отцентрирована к нулю, но теперь нужно считать экспоненту.

3.2. Советы по выбору функции активации

Если вы понимаете какая конкретно функция активации нужна - тогда всё хорошо.

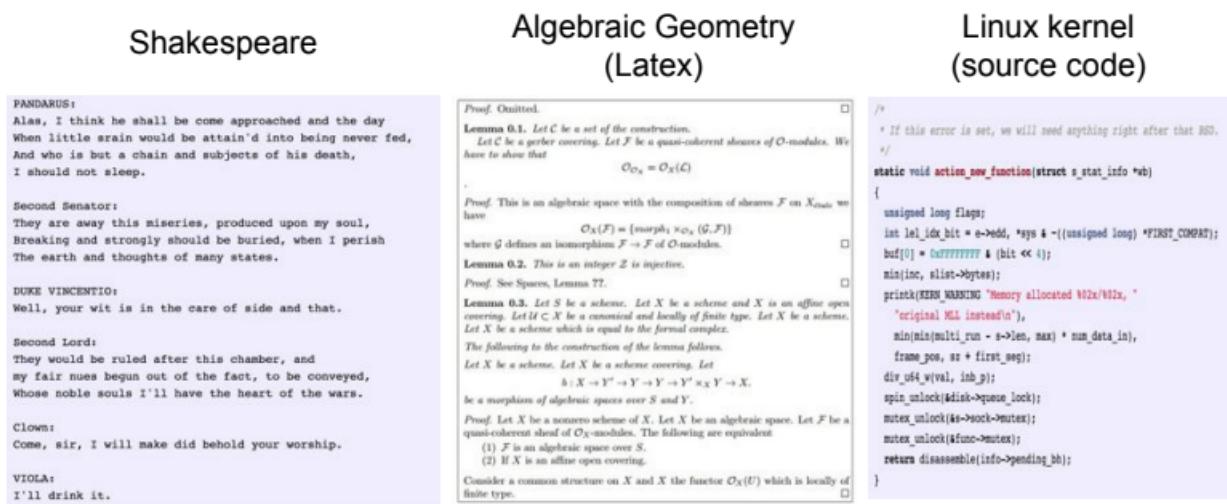
Если нет - в качестве бейзлайна используйте ReLU, если у вас нет ограничений на выход функции активации. Можно после этого попробовать LeakyReLU или ELU. Если нужно ограничение на выход - используйте tanh. Сигмоида используется только если надо что-то типа вероятности, в остальных случаях используйте тангенс.

3.3. Fancy neural networks

FNN - fancy только потому, что они чуть интереснее, чем FC.

3.3.1. RNN - Recurrent Neural Networks

Рекуррентные нейронные сети нужны для работы с текстами, и в целом с любыми последовательными.



Выше можно увидеть примеры текстов: пьеса Шекспира, математический текст в L^AT_EX и исходный код ядра Linux. Все эти тексты рекуррентные нейронные сети умеют генерировать, и конкретно эти писали нейронные сети. Понятное дело, это не оригинальные тексты, но они обладают похожим стилем с их оригиналами.

Рекуррентными нейронными сетями настолько хорошо генерировать текст, что в 2015 году они могли генерировать валидный с точки зрения интерпретатора или компилятора код.

3.3.2. Свёрточные нейронные сети

Про свёрточные нейронные сети хочется сказать две вещи:

- Именно с этих сетей началась современная история нейронных сетей на взлете.
- Они во многом похоже на то, как работает у нас зрительная кора. Это показывает, что свёрточные нейронные сети смогли уловить принципы, которые сделала природа в живых организмах.

В свёрточных нейронных сетях автоматически выучиваются ядра, которые самостоятельно учатся реагировать на объекты на изображении (палки, точки и так далее). Это во многом эмулирует то, что происходит в зрительной коре.

Статья о том, как производить атаки на свёрточные сети - <https://habr.com/ru/post/370541>.

4. Лекция. История развития сверточных нейронных сетей

https://www.youtube.com/watch?v=Xq76hQHCkvQ&list=PL0Ks75aof3Th84kETS1Jq_ja-xqLtWov1&index=34

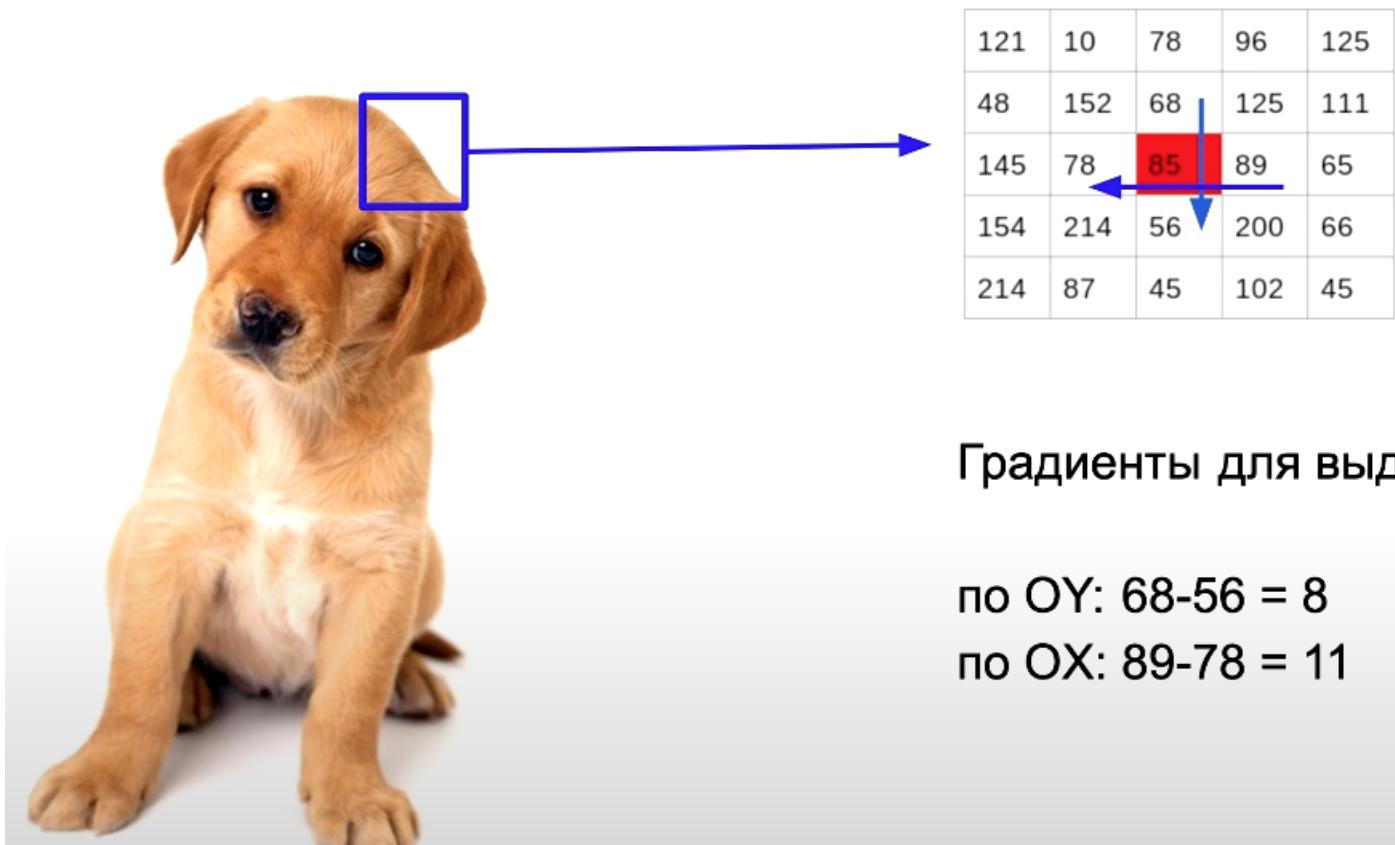
4.1. ImageNet

ImageNet - база данных изображений, поделённых на 1000 классов. В этой базе данных находится больше 14M изображений. Начала собираться в 2010 году и собирается по сей день с помощью крауд-сорсинга.

В 2010-м году стартовал конкурс по классификации изображений ImageNet. Нужно обучаться только на изображениях ImageNet и потом для новых картинок предсказать классы. В 2010-м и 2011-м году лучшие модели имели 28.2 и 25.8 процентов ошибок и были не нейронными сетями. Тогда ещё нейронные сети были слабые и не получится использовать нейронные сети для обработки изображений. Эти модели были на основе стандартного машинного обучения.

4.2. HOG (Histogram of Oriented Gradients)

Как можно без DL обрабатывать изображения?



Вот есть картинка. Давайте рассмотрим маленькую часть этой картинки (синий квадрат, 5×5 пикселей). Картинки в компьютере представляются матрицами чисел. У неё есть средний пиксель: 85. Давайте для этого пикселя посчитаем градиенты: горизонтальный и вертикальный. Для вертикального градиента - разница выше и ниже этого пикселя. Для горизонтального, соответственно, правого и левого.

Так посчитаем для всех пикселей картинки. Если мы возьмём пиксель на границе собаки и фона, такие градиенты будут очень большие, потому что они будут иметь совершенно разный цвет. А два соседних пикселя внутри собаки или два внутри фона - они будут иметь маленькие градиенты. Получается, что градиенты и пиксели определяют некоторую информацию об этом пикселе - находится ли пиксель на пересечении текстур или внутри однородного тела.

Для каждого пикселя выделим два градиента, и если, например, на картинке 200 пикселей, то градиентов будет 400. Вот эти 400 градиентов и будут фичами, которые есть у картинки, и дальше мы обучим нашу модель: случайный лес или лог рег.

Пайплайн:

- Берём картинку

- Выделяем фичи
- Обучаем наш классификатор

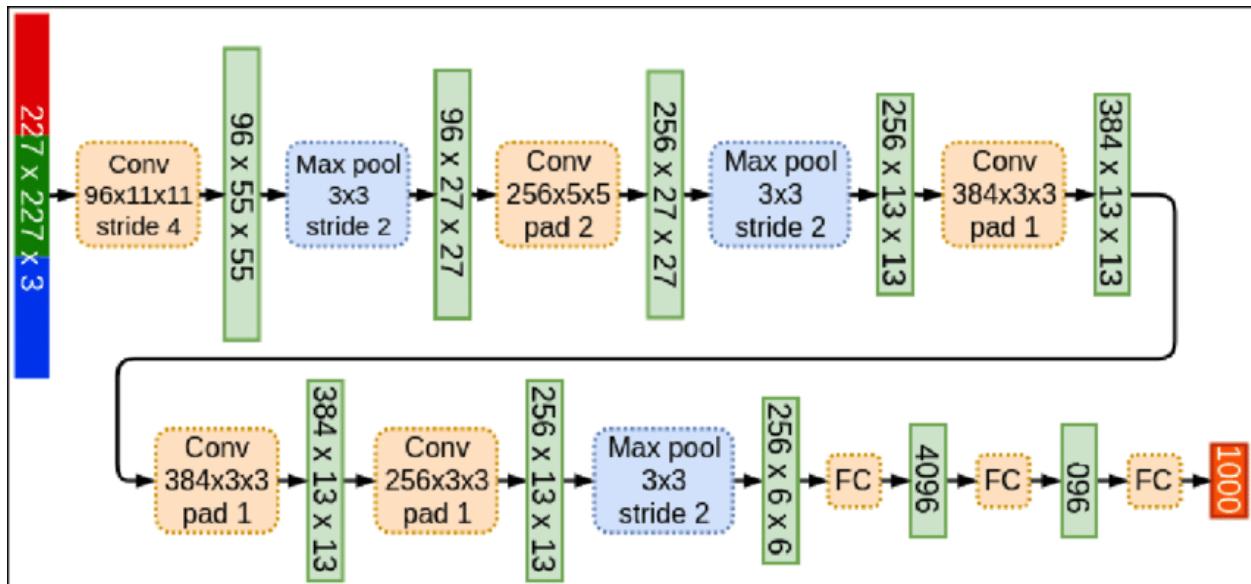
Сам процесс чуть сложнее, потому что есть ещё дополнительная обработка самой картинки, и фичи помимо градиентов, но это пример метода, который классифицировал изображения ещё до взрыва DL.

4.3. AlexNet

В 2012-м году в конкурсе ImageNet победила нейронная сеть AlexNet, и её процент ошибок составлял 15.3%, что практически в два раза меньше предыдущего победителя. Это была не просто нейронная сеть, а свёрточная нейронная сеть.

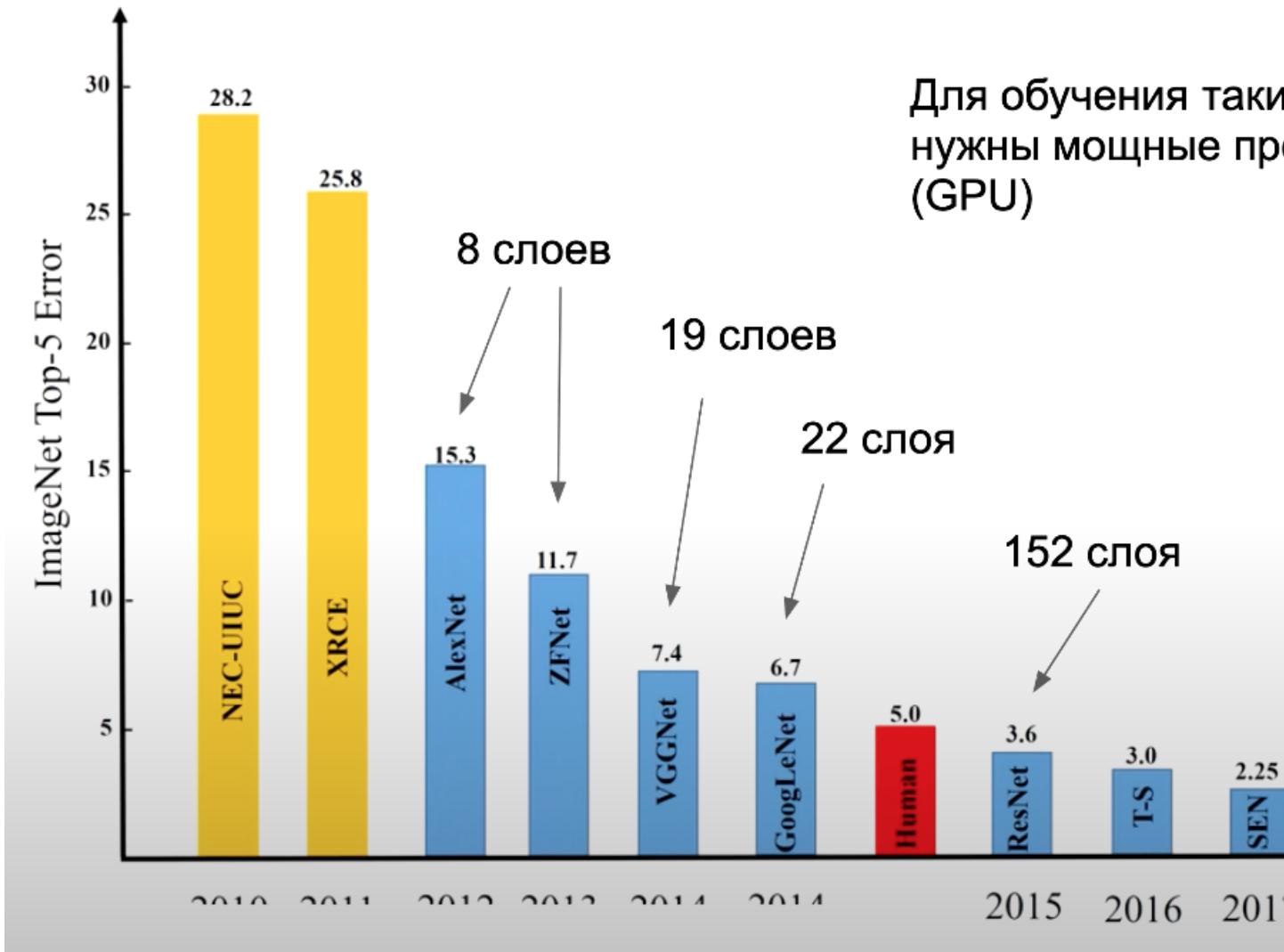
В этот момент люди и поняли, что нейронные сети обладают большим потенциалом в задачах с картинками, а тем более свёрточные нейронные сети. После этого начался бум свёрточных нейронных сетей.

Так устроена сама модель:



Ничего сложного, и такая архитектура выиграла ImageNet.

Когда начался бум сетей, все стали придумывать архитектуры сетей.



Перед 2015-м годом есть отметка Human в 5% ошибки - это значит, что человек допускает в среднем 5% ошибок на датасете ImageNet. После этого появился ResNet, который ошибается реже, чем человек. Это значит, что теперь человек ошибается в среднем чаще, чем нейронная сеть.

Также нужно заметить, что кол-во слоёв начало только увеличиваться - сети растут вглубь. Поэтому это и называется Deep Learning.

Такого прорыва не было бы, если бы не было прорыва в технологиях, потому что параметров очень много и на это нужны ресурсы. А помогли это обучать процессоры с GPU, потому что они производят вычисления быстрее, чем процессоры с CPU.

5. Лекция. Сверточные нейронные сети

https://www.youtube.com/watch?v=HpKGv-kYurk&list=PL0Ks75aof3Th84kETS1Jq_ja-xqLtWov1&index=35

5.1. MNIST

Давайте рассмотрим датасет MNIST. Это датасет рукописных цифр, поделённых на 10 классов: от 0 до 9. Каждая картинка в этом датасете чёрно-белая и размера 32×32 .

Чёрно-белая картинка в компьютере представляется двумерной матрицей чисел и каждое число принадлежит отрезку $[0; 255]$, которое выражает яркость пикселя: от 0 - самого чёрного, до 255 - самого белого.

5.2. Первые мысли как решать задачу классификации картинок

Как мы можем решать задачу классификации картинок из этого датасета с помощью полно связной сети? Мы могли бы взять картинку и растянуть её в вектор размерности 1024. Этот вектор мы и подадим на слой, и, получается, на входном слое будет 1024 вершины.

В конечном слое будет 10 нейронов, каждый из которых отвечает за 1 из 10 классов и говорит вероятность своего класса.

Какие тут есть минусы? У нас ломается расположение пикселей - мы теряем информацию об их взаимном расположении.

Также у нас получается очень большой первый слой - картинка маленькая, а вершин уже много. Что тогда будет с большими картинками? Представьте сколько будет весов при переходе ко второму слою.

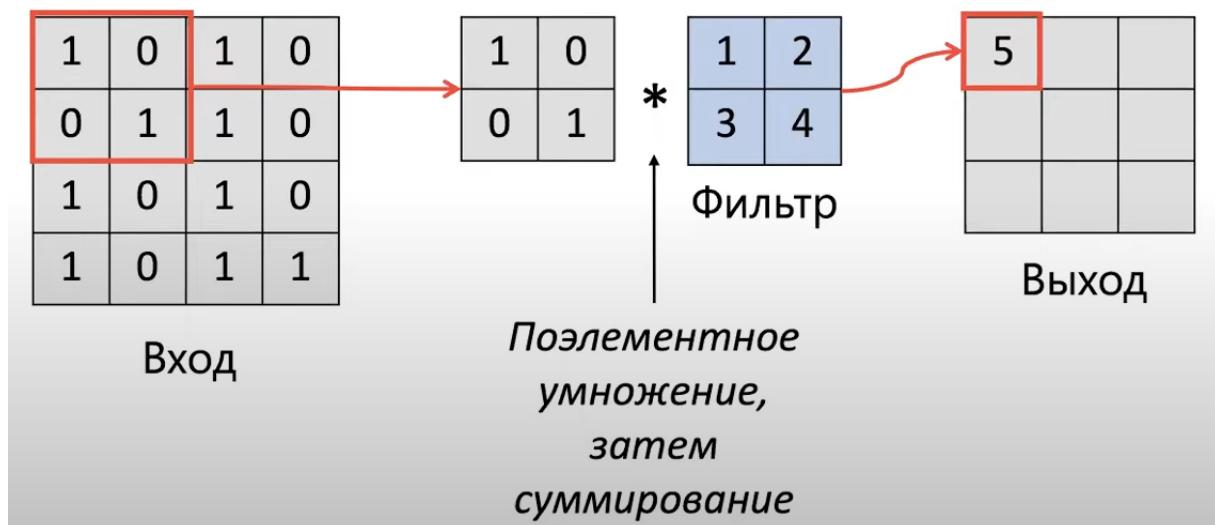
5.3. Как перенести пространственную информацию в сеть?

Что отличает "4" от "8"? Представим, что мы разделяем только на 2 класса, а не на 10. Для человека это очевидно - у четвёрки преимущественно горизонтальные и вертикальные линии, а у восьмёрки нет ни одной горизонтальной или вертикальной линии, зато есть изогнутые. Давайте как-нибудь перенесём эту идею на нейронную сеть.

5.4. Свёртка

5.4.1. Ядро

Введём понятие ядра (фильтра). Ядро - это квадратная матрица, размера $n \times n$. Этим ядром мы будем сворачивать наше изображение: с самого начала мы положим фильтр в нашу верхнюю левую часть изображения и сделаем поэлементное умножение - операцию свёртки.



То же самое мы делаем, свдинув фильтр вправо и таким образом мы проходим всю матрицу.

Таким образом мы свернём нашу картинку - итог называется свёрткой.

Заметим, что если фильтр был размера $n \times n$, а картинка $m \times m$, то итоговая картинка будет $(m - n + 1) \times (m - n + 1)$

5.4.2. Padding

Padding нужен для того, чтобы добавить рамку из нулей вокруг картинки, чтобы использовать те пиксели, которые не влезают в свёртку. То есть если мы используем padding 2 на картинке 32×32 и свёртке размером 3×3 , то итоговый размер картинки тоже будет 32×32 .

5.4.3. Stride

Перед этим мы двигали фильтр на 1 пиксел вправо или вниз. Если мы захотим пропускать (применять фильтр через 1 пиксель), то мы поставили бы параметр stride=2, он ровно за это и отвечает, на сколько пикселей мы каждый раз сдвигаемся.

5.4.4. Свёртка цветных изображений

Заметим, что мы рассматривали пока чёрно-белые картинки. А как обрабатывать цветные картинки? Цветные картинки представляются трёхмерными матрицами (RGB). Как сворачивать такие картинки?

Первый способ - использовать 3D свёртку. Ядро тоже будет не двухмерным, а трёхмерным. То есть ядро будет размерности не $n \times n$, а $n \times n \times 3$.

Есть второй способ как сворачивать цветную картинку. Можно поделить её по цветовым каналам на 3 двумерных картинки и мы получим свёртки по каждому из каналов.

5.4.5. Фильтры

Фильтры - способы реагирования на присутствие на изображении некоторых паттернов.

Допустим есть у нас 4-ка. Есть у нас фильтр такой:

$$\begin{pmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{pmatrix}$$

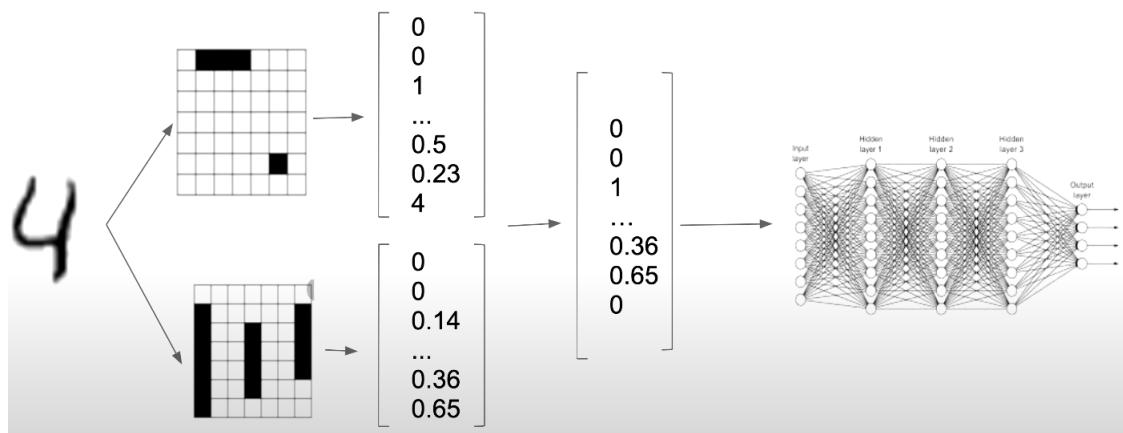
Такой фильтр будет искать вертикальные линии на изображении. При этом если нам попадётся просто монотонный участок изображения, то свёртка будет равна нулю.

То что вышло после применения фильтра ещё называют картой активации. И большие значения на карте активации показывают где "активировался фильтр".

Но не всегда фильтр показывает часть картинки изначальной. Карта активации просто показывает числа.

5.5. Применение фильтров в нейронных сетях

После применения фильтров мы их развернём в векторы, сконкатенируем и подадим дальше на полно связанный слой.



Мы сделали примерно то же, что делали люди до 2012-го года, только они подбирали ядра сами, а у нас ядра будут учиться сами и нам не нужно об этом заботиться - ещё одна автоматизация, о которой человеку не нужно заботиться и подбирать параметры самому.

5.6. Real-world изображения, несколько слоёв свёртки

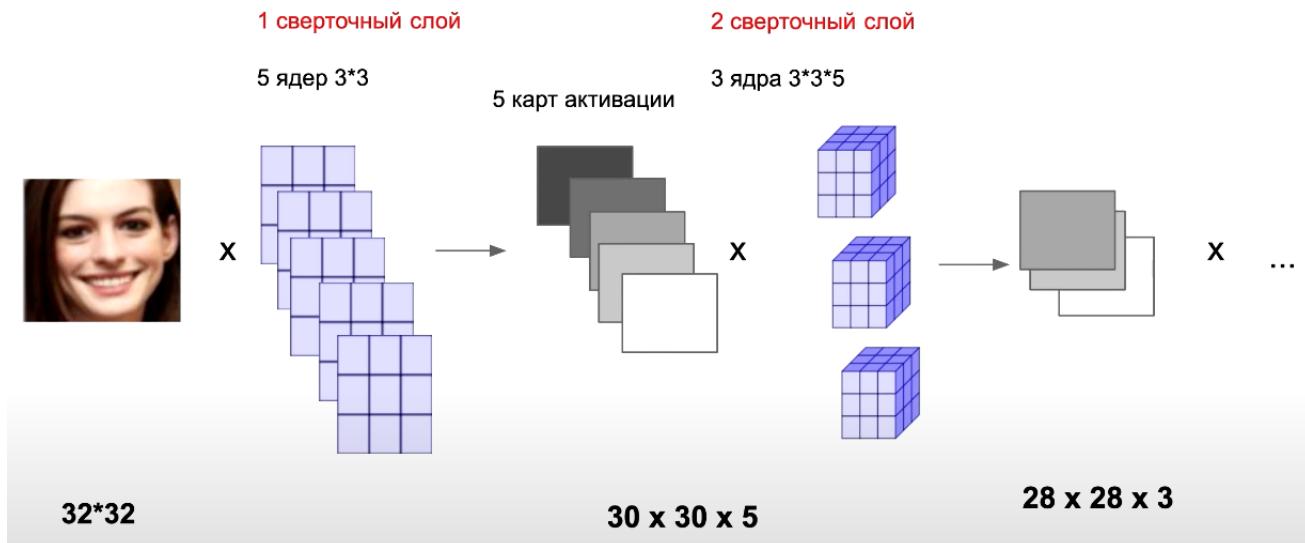
MNIST очень простая задача - и картинка чёрно-белая, и фигуры простые: мы можем сделать два фильтра, один из которых будет реагировать на горизонтальные, а другой на вертикальные прямые, и тогда уже нейронная сеть сможет отличать друг от друга некоторые числа.

Для реальных изображений одной операции свёртки не хватит, чтобы выделить всю уunjную информацию из изображения. У картинок очень сложная структура. Представьте насколько сложно по структуре лицо?



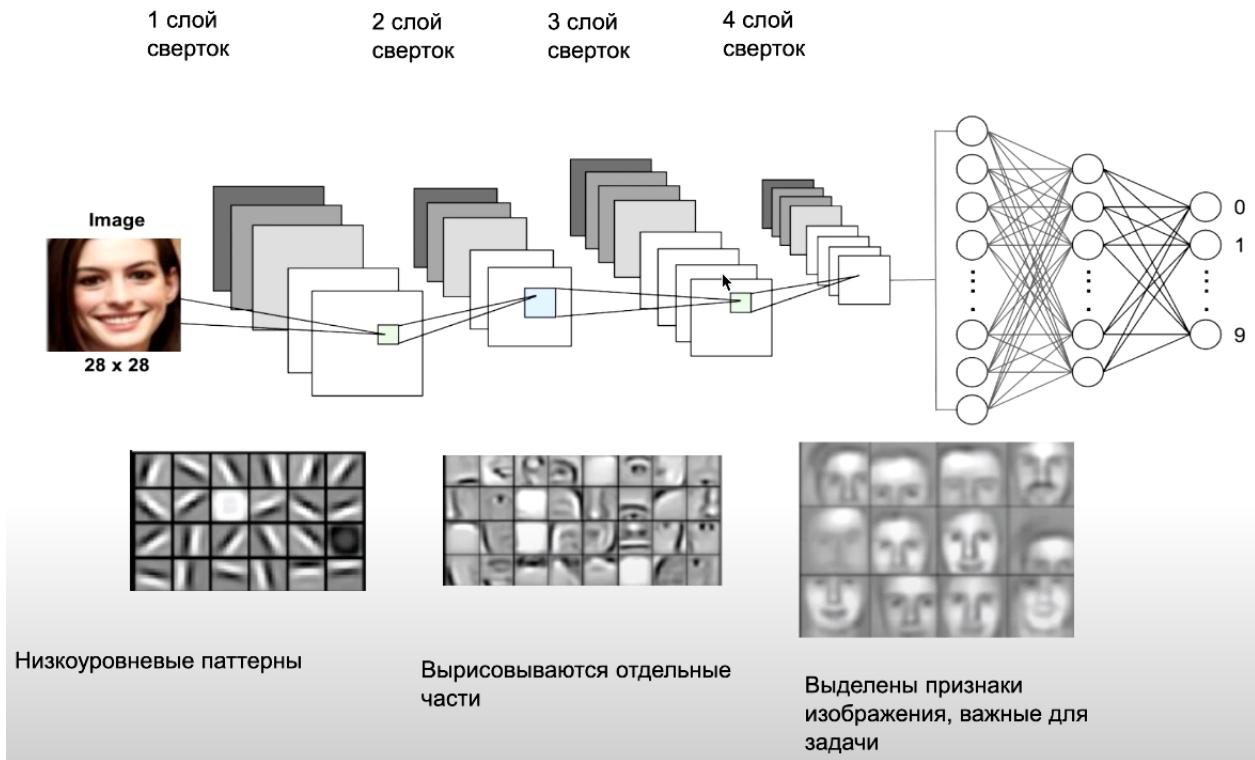
Если мы захотим различать людей друг с другом - это будет куда сложнее, ведь есть очень много каких сложных фичей, а вот простые, по типу вертикальных или горизонтальных фичей - есть на каждом лице.

Чтобы решить эту проблему - нам понадобится несколько слоёв свёртки. Но как их между собой соединять?



Вот допустим у нас картинка была 32×32 . Если мы применим на неё 5 фильтров размера 3×3 и $\text{stride}=1$, то мы получим 5 матриц размера 30×30 . Это можно представить как картинку $30 \times 30 \times 5$ - по сути, картинка с пятью каналами. После этого мы сделаем ещё один свёрточный слой, где будет три ядра, размерностью $3 \times 3 \times 5$. После их применения мы получим три матрицы размером 28×28 (третье измерение пропало, потому что фильтр по последней координате был размера 5, как и наша картинка, что склонило одно измерение).

Тогда в общем случае у нас будет в самом начале несколько слоёв свёрток и примерно вот такие паттерны они будут изучать на каждом следующем слое:



После того, как последний свёрточный слой пройден - мы схлопываем матрицы в вектор и подаём на вход полно связной сети - классификатору.

При этом, если подать на вход нейронной сети какую-то картинку по центру из примера (просто ухо, нос и так далее), наш средний слой, который отвечает за нахождение этих паттернов - активируется больше всего.

А вот если подать на вход картинку чайника - средний слой вообще активируется и карта активации после среднего слоя будет очень маленькой по значениям.

Самые последние слои в таком случае будут реагировать, если на изображении полностью присутствовал паттерн лица. Если подать в качестве изображения чайник - последние слои активироваться не будут. А если подавать на вход картинку лица - последние слои отреагируют больше всего.

То что мы делаем несколько слоёв свёрток - это помогает находить высокогуровневые фичи. Каждая карта активации отвечает за то, насколько какие-то паттерны присутствуют на изначальной картинке. Следующие, после первого слоя свёрток - принимают на вход карты активации предыдущих слоёв и пытаются из более простых фичей составить более сложные: например из двух наклонных прямых и одной горизонтальной выделить нос.

5.7. Функция активации для свёрточных сетей

После слоя свёрточных сетей, как и после всех околовлинейных слоёв, нужно сделать функцию активации. Самая популярная и хорошо работающая функция активации промежуточных слоёв - ReLU (но не для последнего слоя - на последнем SoftMax, если у нас задача классификации).

5.8. Обучение свёрточной нейронной сети

А обучать свёрточную нейронную сеть мы будем с помощью cross-entropy.

А где же сами обучаемые параметры? Мы уже говорили, что это и будут сами ядра.

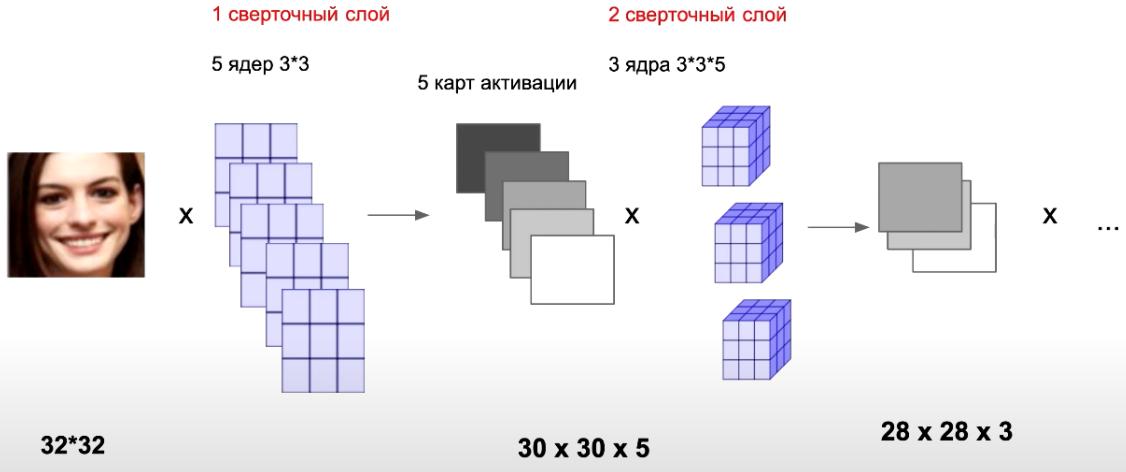
6. Лекция. Пулинг. Операция пулинга

https://www.youtube.com/watch?v=IxLuPHtZBTY&list=PL0Ks75aof3Th84kETS1Jq_ja-xqLtWov1&index=36

6.1. Проблема больших карт активаций на выходе

Пулинг - это техника уменьшения размерности карт активаций.

Если у нас была картинка 32×32 , мы применяем ядро 3×3 без паддинга и со stride=1, то мы получаем 5 карт активации размерностью 30×30 . Если мы к ним ещё применим 3 ядра размером $3 \times 3 \times 5$, то у нас получается три карты активации размером 28×28 .



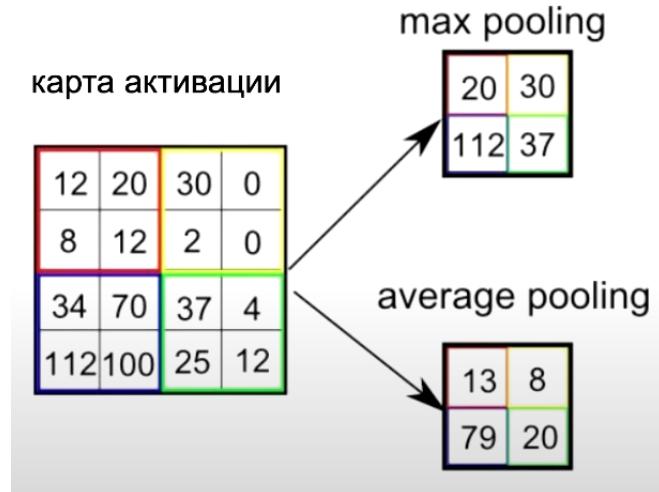
При таком применении фильтров у нас размерность картинки уменьшается, но не очень сильно: $(32 \times 32) \rightarrow (30 \times 30) \rightarrow (28 \times 28)$. Можно, конечно, регулировать размерность карт активаций с помощью stride'ов и padding'ов.

Но давайте предположим, что наше исходное изображение очень большое: 512×512 , чтобы уменьшить такую картинку нам понадобится много слоёв, значит много фильтров, значит много параметров, значит больше нужно обучать.

А зачем нам нужно, чтобы карты активации были маленькими? Выход свёрточных слоёв мы выдаём на вход полносвязной нейронной сети. Нам хочется уменьшить кол-во входных параметров на полносвязную сеть, чтобы уменьшить количество связей и весов, чтобы обучать было проще.

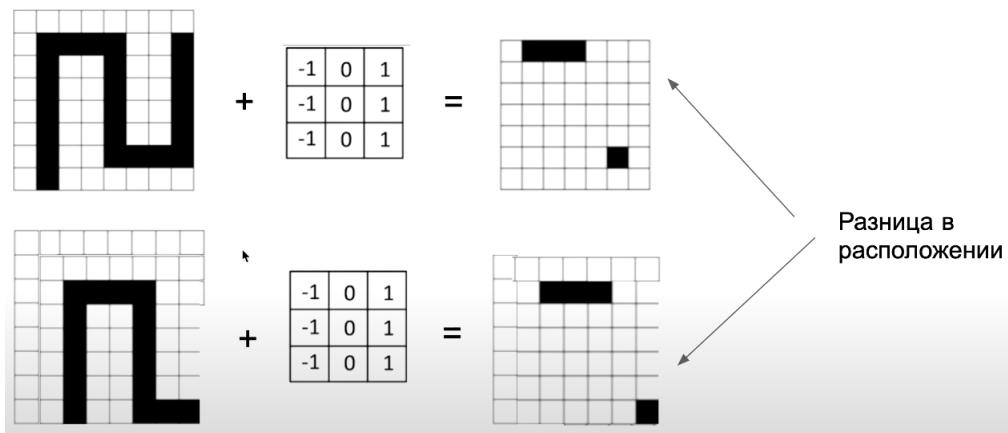
6.2. Pooling

Как работает pooling? Допустим у нас есть карта активации 4×4 . Давайте разобьём её на ячейки 2×2 . Дальше просто мы выбираем какой пулинг применять во всех ячейках (не в плане мы можем для разных ячеек применить разные функции пулинга, а одну на всех).



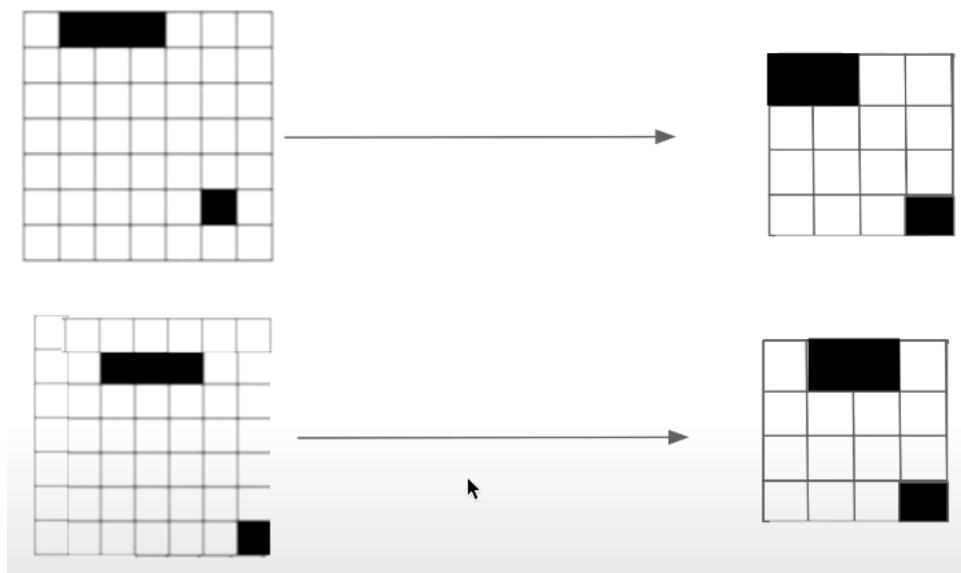
Просто уменьшение размерности - это не только то, для чего нам нужен pooling - мы могли бы просто ресайзить картинки и всё.

Ещё одна причина, почему нужно использовать пуллинг - пуллинг уменьшает чувствительность восприятия свёрток к положению объектов на картинке. Давайте рассмотрим пример:



Здесь у нас одна и та же картинка, просто на нижнем примере она сдвинута вправо-вниз, а карта активации получилась разная.

После применения MaxPooling'a 2×2 у нас получится следующая карта активации:



Карты активации после пуллинга из 6×6 теперь являются 4×4 . Но при этом, у нас осталась некоторая инвариантность такой же.

6.3. Куда вставлять слой polling'a?

В архитектуре наших сетей мы их будем вставлять после свёрточных слоёв (вместе с активацией) и дальше отдавать на свёрточный слой.

6.4. Обзор слоёв AlexNet

<https://youtu.be/IxLuPHtZBTY?t=782>

7. Лекция. Задачи компьютерного зрения

https://www.youtube.com/watch?v=3IPRcBIsgNA&list=PL0Ks75aof3Th84kETS1Jq_ja-xqLtWov1&index=37
Какие есть задачи в компьютерном зрении (CV)?

7.1. Классификация, детекция, сегментация

Они идут вместе, потому что их проще всего объяснить на одной картинке



7.1.1. Классификация

Классификация - нам дан пул картинок, и нужно определить к какому из заранее заявленных классов принадлежит каждая картинка. Пример - ImageNet.

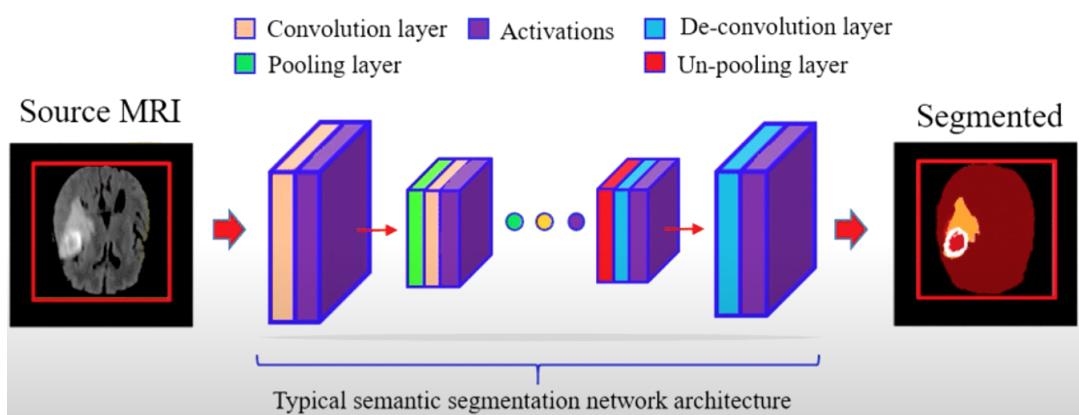
7.1.2. Детекция

Детекция - нам по картинке нужно понять, есть ли на ней какой либо объект, и если есть - обвести его в bounding box. Но это всё также обговорёный пул объектов - найти что-то новое, на чём мы не обучались - мы не можем.

7.1.3. Сегментация

Мы хотим отделить объекты на картинке друг от друга, покрасить их как-то, выделив что вот это - это один объект. Но выделять мы хотим не всё - например, на картинке видно, что мы покрасили только человека и животных. При этом, нужно отметить, что справа рука, которая обнимает собаку, принадлежит человеку и сетка это поняла. При этом разные собаки тоже покрашены в разные цвета.

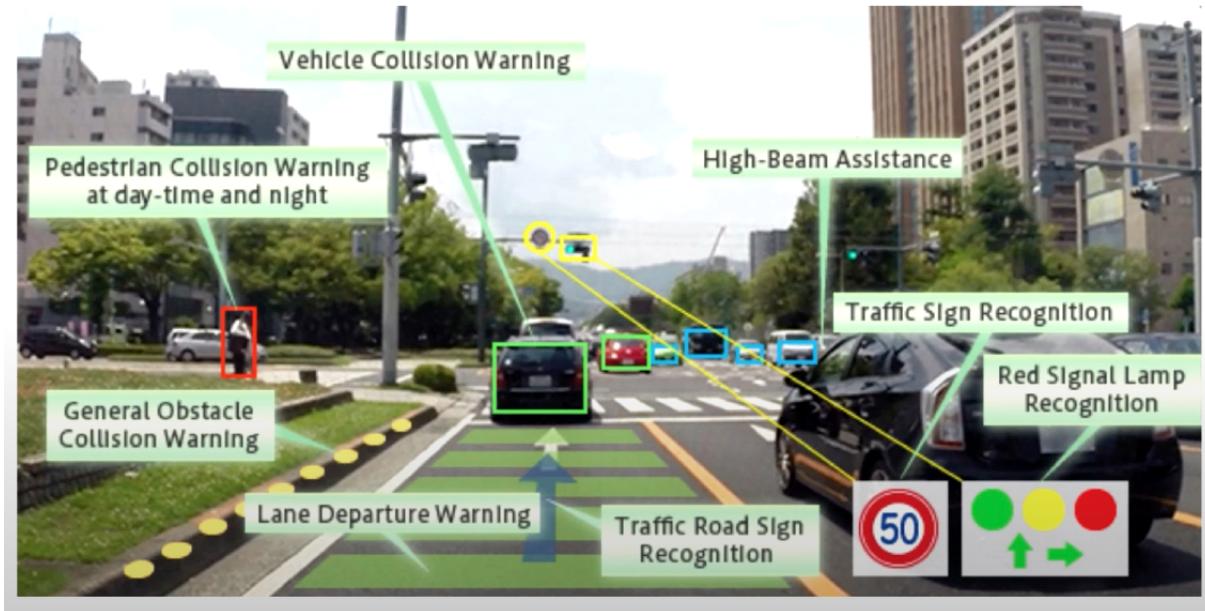
Сегментация - важная задача для медицины. Сейчас в эру машинного обучения, эру Deep Learning'a, они проникают везде. Пишутся даже статьи, которые направлены на медицинскую тематику. В медицине одна из самых больших задач - сегментация. Пример:



У нас есть слева МРТ снимок мозга человека. У человека возможно есть опухоль, а возможно нет. Хочется обучить такую нейронную сеть, которая покрасила бы этот снимок в поражённые ткани, и в непоражённые.

Также стандартная ситуация - сегментировать клетки под микроскопом. Их много и они раздлены стенками, но нужно сделать это много и точно.

7.2. Ещё применение задач CV



Очень важны задачи CV в беспилотных автомобилях. Беспилотные автомобили очень сложная технология в целом. Он смотрит вокруг камерами и лидарами. Остановимся на камерах: какие задачи для них нужно решать? Во-первых - нужно решать задачу детекции. Нужно детектировать машины, чтобы не врезаться в них, пешеходов - чтобы не сбить, дорожные знаки и светофоры - чтобы соблюдать ПДД.

Когда мы задетектировали знак - нам нужно вырезать картинку с ним и понять какой это знак - это уже задача классификации. Точно также и с детекцией светофора, а потом распознаванием его статуса.

Также есть интересная задача, которая называется поиском по изображениям: иногда GPS плохо работает и не может определить где беспилотник работает. Для беспилотника это чревато плохими последствиями. Беспилотник может сфотографировать местность вокруг и попробовать поискать в базе данных фотографий города похожую местность: так действительно работает. Можно по фото из переднего стекла машины понять, где она сейчас находится.

Также есть задача оценки положения: помимо того, что мы задетектировали машину, нужно понять насколько она от нас далеко и нужно ли тормозить.

Ещё задача CV - OCR (Optical Character Recognition) - по фотографии текста распознать - что это за текст. Тут же и улучшение качества фотографий документов, сканов, чеков. Эта задача на стыке NLP и CV, но CV в ней используется.

Ещё одна задача - Video Understanding. Нужно понимать то, что происходит на видео или на фото. Нейронная сеть должна выдать текст связный описывающий что происходит на фото, и показать откуда она эту информацию взяла. Это снова задача на стыке CV и NLP, но CV в ней присутствует в полной мере.

Также могут быть субтитры к видео - например, по разнаванию движения рта. Или просто добавить описание: что человек делает на этом видео - Action Description.

Также есть задача Pose Estimation - по видео человека понять как движутся суставы человека. Сама по себе задача непонятно зачем нужна, но чаще всего это средняя точка для решения других задач CV.

Ещё пример задач CV - GANs - генерация новых картинок по уже существующей.

Примером может быть Style transfer - вы хотите нанести на картинку какой-то стиль.

Также есть генерация/дополнение изображений - по какому-то эскизу дополнить картинку.

Где используется Pose Estimation? Когда мы хотим перенести на видео движения одного человека на другого человека. <https://youtube.com/watch?v=PCBTZh41Ris>

8. Лекция. Градиентная оптимизация в Deep Learning

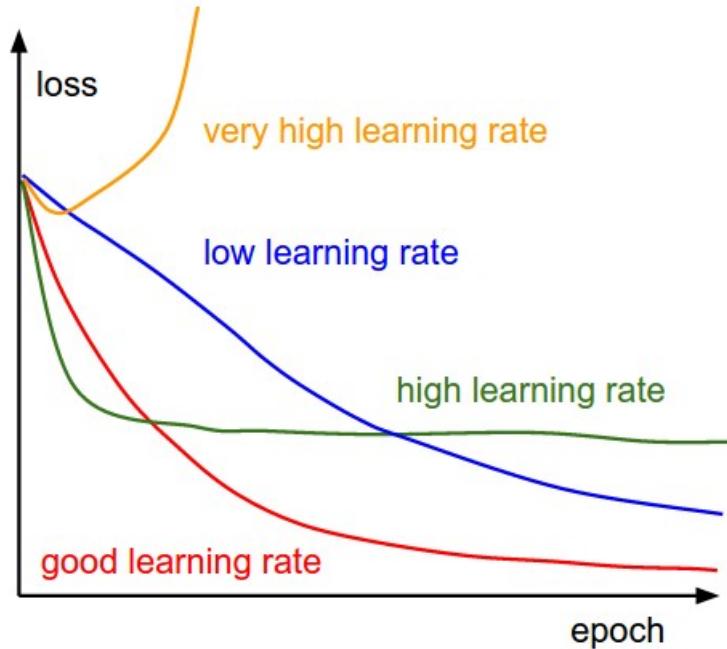
https://www.youtube.com/watch?v=6CvpM00-DB4&list=PL0Ks75aof3Th84kETS1Jq_ja-xqLtWov1&index=41

8.1. Обучение нейронной сети

Мы точно также обучаем нейронную сеть градиентным спуском. Но мы не будем никогда обучать по одному объекту, будем обучать только по батчу (обычно берут размера степени двойки).

Ну и как обычно $x_{t+1} = x_t - lr \cdot dx$.

График обучения выглядит обычно так:



При этом для самого спуска у нас тоже есть разные оптимайзеры. Они отличаются только по тому, как на основе градиента перейти в новую координату.

Вспомним, что learning rate — по сути, тоже параметр для модели. При этом у нашей модели есть очень много параметров, и брать один и тот же lr — это очень странно. Подбор lr для каждого параметра по отдельности тоже плохо — параметров много и мы, по сути, столько же хотим найти для lr.

8.2. Stochastic Gradient Descent

Мы берём батч размера B , считаем loss:

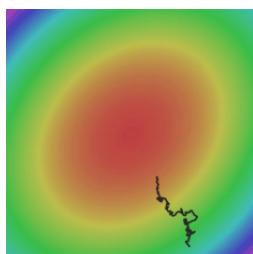
$$L(W) = \frac{1}{B} \sum_{i=1}^B L_i(x_i, y_i, W)$$

Затем дифференцируем по параметрам нашей модели и ищем средний градиент:

$$\nabla_W L(W) = \frac{1}{B} \sum_{i=1}^B \nabla_W L_i(x_i, y_i, W)$$

Ну а дальше делаем шаг как всегда.

Вот пример того, как мы можем идти:



Мы видим, что путь какой-то очень шумный. Почему? Потому что у нас шумная оценка градиента на каждом шаге: мы используем маленько подмножество объектов.

Один из способов починить шумный градиент - взять огромную выборку объектов. Градиент скорее всего будет устойчивым, но считать это будет долго и неэффективно. А среди малых количеств объектов могут быть шумовые, мы получаем большую ошибку и градиент, и потому отходим от нужного пути.

Также наши объекты могут быть разной природы: например, мы классифицируем любые растения, а на очередном батче получили только грибы.

8.3. SGD Momentum

Momentum - это как раз идея как починить минусы SGD. Мы будем смотреть на предыдущие градиенты и не сразу их забывать.

Если мы представим весь процесс как шарик, который катится вниз - ему нужна инерция, чтобы двигаться плавно, а не рывками. Когда он скатывается куда-нибудь, он не сразу снизу останавливается, а сначала ещё чуть-чуть едет в горку.

Формула выглядит так:

$$v_{t+1} = \rho v_t + \nabla_W f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

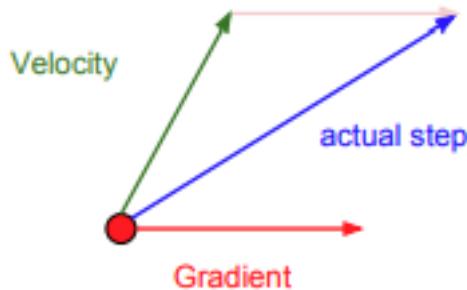
где v - это и есть инерция (v от слова velocity).

8.4. SGD Nesterov Momentum

Можно научить наш градиентный спуск заглядывать в будущее. Что? Как?

Как работала инерция в SGD Momentum в визуальном плане?

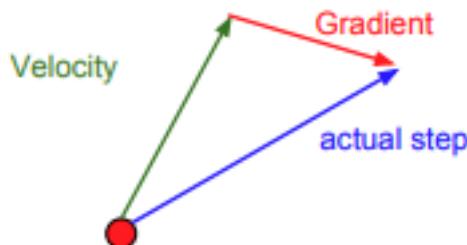
Momentum update:



Но что если сейчас инерция и так очень большая?

После подсчёта градиента модель из-за инерции смещается очень сильно куда-то в другое место и градиент не актуален. Давайте тогда сначала сдвигать нашу точку по инерции, а потом уже считать градиент?

Теперь наше передвижение выглядит по другому:



Картинки похожи, но стоит обратить внимание в какой момент мы считаем градиент.

Доработка очень маленькая, но работает лучше во многих случаях.

Ну и итоговый подсчёт переходов выглядит как:

$$v_{t+1} = \rho v_t - \nabla_W f(x_t + \rho v_t)$$

$$x_{t+1} = x_t + \alpha v_{t+1}$$

8.5. Adagrad

Помните у нас была идея сделать для каждого признака свой lr для каждого параметра? Из каких здравых вещей мы это можем сделать? Давайте предположим, что у нас по одним признаком у нас градиенты меняются сильно, а по другим меняются слабо или не меняются. Если градиенты меняются слишком быстро - то скорее всего lr большой и нам нужно его понизить. Если меняется очень медленно - скорее всего наоборот lr маленький и его нужно повысить.

Давайте сделаем какой-то кеш, мы накапливаем квадратный градиент, чтобы понять его амплитуду. То есть кеш будет только неотрицательный:

$$\text{cache}_{t+1} = \text{cache}_t + (\nabla f(x_t))^2$$

$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$

То есть, мы тут ещё и по компонентам меняем нашу точку.

Но при этом есть одна проблема - кеш неубывает. Шаг за шагом понижается lr. Значит, скорее всего мы может остановиться в обучении ещё до того, как придём в нужную точку.

8.6. RMSProp

Та же самая идея, что у Adagrad, но наш cache будет затухать, чтобы не расти постоянно:

$$\text{cache}_{t+1} = \beta \text{cache}_t + (1 - \beta)(\nabla f(x_t))^2$$

$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$

Тогда мы можем учитывать значения кешей в предыдущие моменты времени, но и не забывать о текущем значении градиента.

8.7. RMSProp normalization

Давайте объединим идеи про инерцию и кеши:

$$v_{t+1} = \gamma v_t + (1 - \gamma) \nabla f(x_t)$$

$$\text{cache}_{t+1} = \beta \text{cache}_t + (1 - \beta)(\nabla f(x_t))^2$$

$$x_{t+1} = x_t - \alpha \frac{v_{t+1}}{\text{cache}_{t+1} + \varepsilon}$$

9. Лекция. Регуляризация в Deep Learning

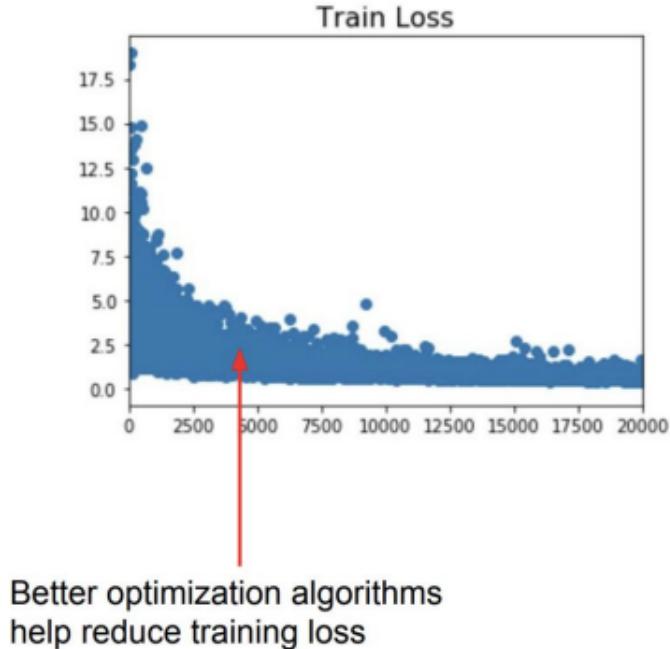
https://www.youtube.com/watch?v=x72-oUjv1ew&list=PL0Ks75aof3Th84kETSlJq_ja-xqLtWov1&index=42

Регуляризация - это техника, которая помогает более эффективно использовать вашу модель, чуть более умным способом внедрить понимание в работу вашей модели.

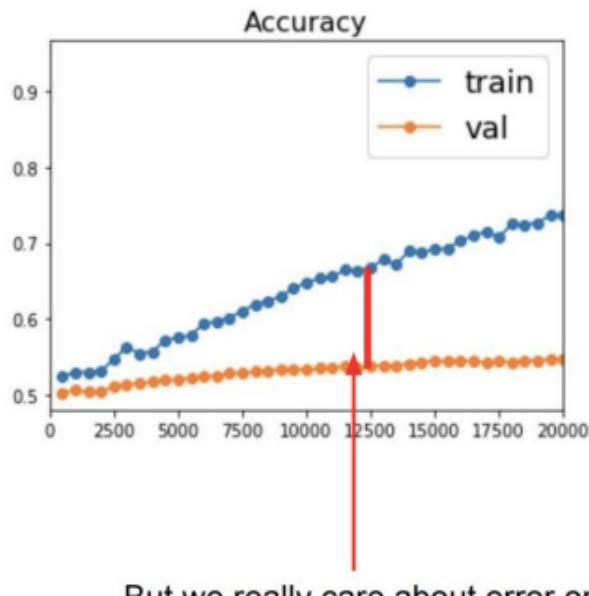
Как можно использовать регуляризацию помимо похожих на L_1 и L_2 норм?

9.1. Пример переобучения

Допустим у нас есть вот такой график loss'а для нашего train:



Но давайте взглянем как у нас выглядит accuracy для train и val выборки:



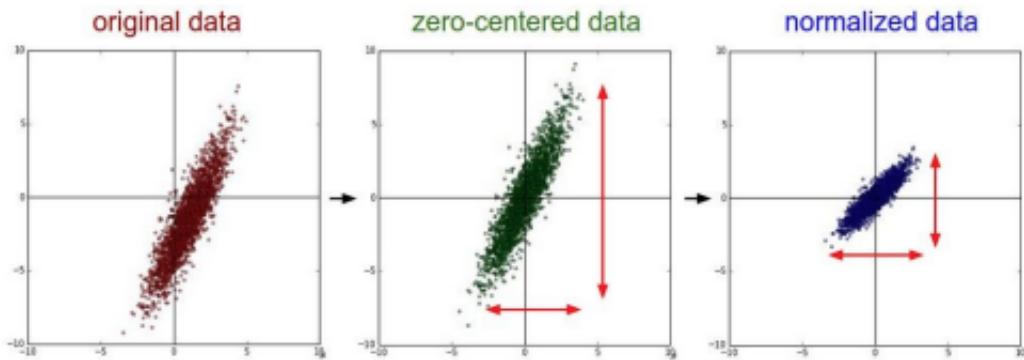
Обычно между ними должен быть небольшой гап, но на данном графике мы видим, что train явно увеивается от val, и это один из сигналов переобучения.

Как же нам добиться того, чтобы наша модель не переобучалась? Для этого нам и нужна визуализация, подготовка данных и подходящая архитектура.

Как пример, если взять MNIST и обучить VGG19, скорее всего это всё переобучится достаточно быстро. У модели огромная способность к запоминанию, и она скорее запомнит датасет, чем что-то выучит.

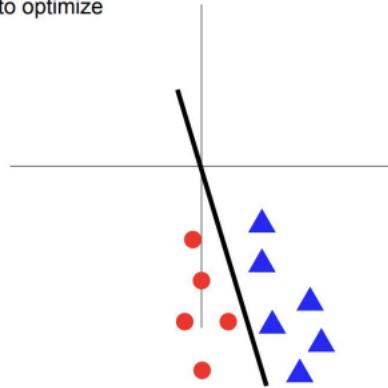
9.2. Нормировка данных

Давайте посмотрим на то, что можно с нашими данными сделать. Допустим у нас есть данные из \mathbb{R}^2 :

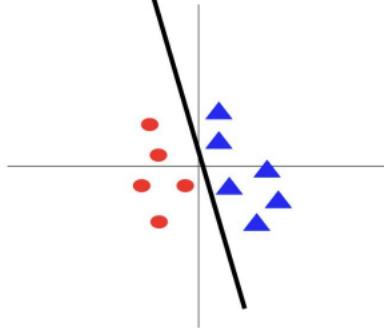


У непредобработанных данных мы видим, что данные смещены относительно среднего и по одной координате, и по другой. Чем это может быть плохо? Допустим мы используем линейную модель для бинарной классификации:

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize

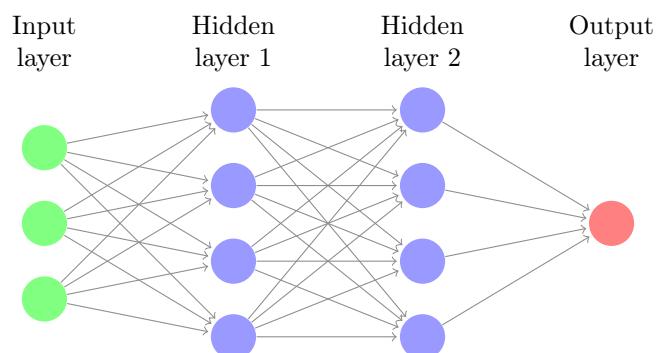


Когда мы данные не подготавливаем - у нас малейший поворот нашей разделяющей прямой очень сильно влияет на предсказания. В случае, когда данные нормализованы, таких последствий не происходит.

Давайте просто нормализуем данные - вычтем среднее и поделим на дисперсию. Вроде как похоже на стандартное нормально распределение.

9.3. Почему нужно нормировать данные не только на входе?

Линейная модель не хочет получать на вход не отнормированные данные. Но что у нас происходит в нейронной сети? Допустим у нас есть такая нейронная сеть:



Давайте посмотрим на второй скрытый слой: мы с вами берём нашу функцию потерь, сначала обновляя параметры второго скрытого слоя, потом первого скрытого слоя. Что же можно сказать? С точки зрения второго скрытого слоя - ему на вход приходит какое-то описание данных. Второй скрытый слой понятия не имеет, что он второй, а не первый. Мы учитывали что ему пришли на вход и обновили веса. Дальше мы

пробрасываем градиенты на следующий слой. Мы обновили то что мы пробрасываем на второй слой. На первом слое данные тоже поменяются. Второй слой этого не знает и это плохо. Одновременно обновлять всё мы не можем.

9.4. Как раньше боролись с нормировкой данных внутри сети?

Раньше просто брали и понижали lr до очень маленьких значений. Значения менялись очень по маленькому и слои не замечали изменения данных с других слоёв.

А другие методы? Отнормировать данные на input слой мы можем, а вот дальше мы никак не можем - сеть сама что-то делает с весами.

9.5. Batch-norm

Кто нам мешает нормализовывать буквально каждый вход для линейных моделей? никто не мешает.

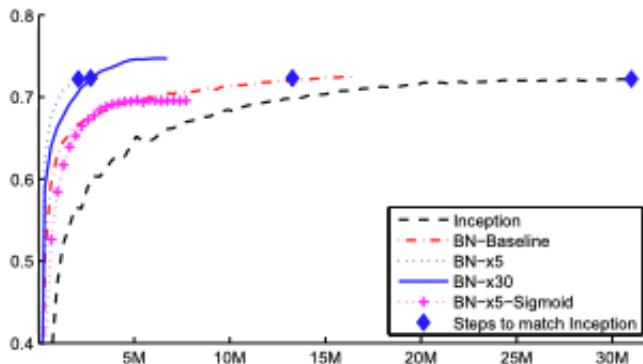
При этом, нам нужно нормировать текущее среднее и дисперсию по всем объектам, чтобы у нас не сломался инференс. Тогда к обычным подсчётам среднего и дисперсии по батчу нужно добавить ещё

$$\mu_i = \alpha \cdot \text{mean}_{\text{batch}} + (1 - \alpha) \mu_{i-1}$$

$$\sigma_i^2 = \alpha \cdot \text{variance}_{\text{batch}} + (1 - \alpha) \cdot \sigma_{i-1}^2$$

Как делается в оригинальной статье? В оригинальной статье авторы ещё обучают параметры γ и β , которые нужны, чтобы итоговый \hat{x}_i переделать как $\gamma\hat{x}_i + \beta$, потому что может в среднем и дисперсии была какая-то информация, а отнормировав, мы эту информацию потеряли. Но это преобразование обучаемо.

Вот как batchnorm улучшает обучение:



А вот оригинальная статья по batchnorm - <https://arxiv.org/pdf/1502.03167.pdf>

9.6. Регуляризации

Но batchnorm решил только проблему с тем, что мы учились очень долго, а как решить проблему переобучения?

9.6.1. Weight decay

С некоторыми регуляризациями мы уже знакомы:

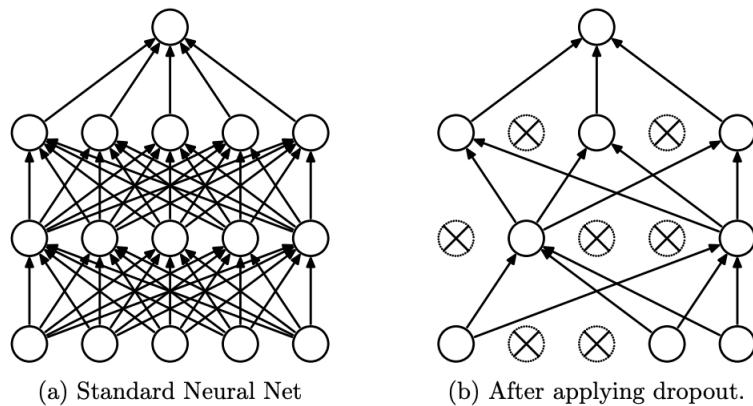
$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W)$$

где роль $R(W)$ у нас может играть

- L1 регуляризация: $R(W) = \|W\|_1$
- L2 регуляризация: $R(W) = \|W\|_2^2$
- Elastic Net ($L1 + L2$): $R(W) = \beta \|W\|_2^2 + \alpha \|W\|_1$

9.6.2. Dropout

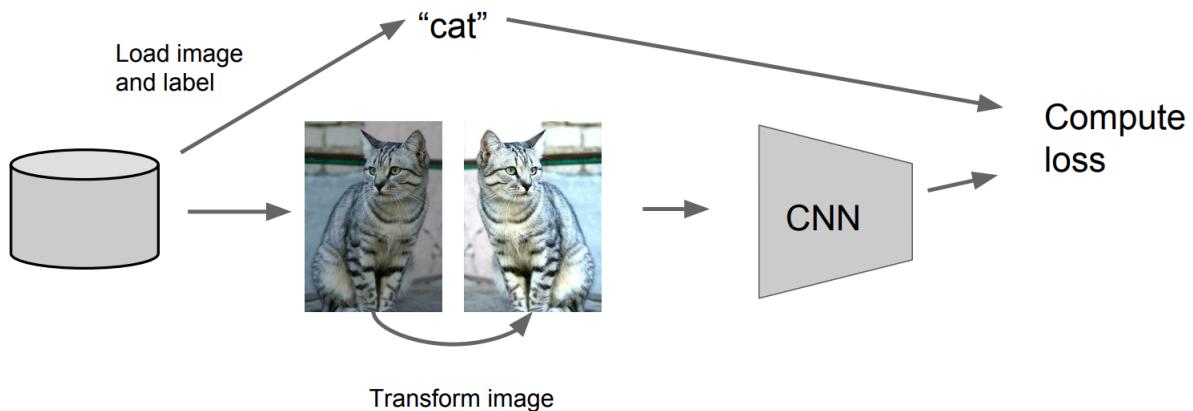
Классическая техника, которая предлагает - а давайте случайным образом занулять значения и работать на частично-признаковом описании. Мы не выкидываем признаки, а зануляем, эти значения не вкладываются в предсказание, поэтому они помечены крестиком:



Вы можете спросить: а зачем такая техника, если мы часть информации теряем? Да, теряем, но мы делаем это осмысленно. Давайте представим такую ситуацию: у нас есть группы работников, которые готовят блюдо. Одни режут огурцы, другие жарят картошку, и так далее. Представьте, что один работник заболел и люди не могут приготовить итоговое блюдо - это плохо. Всё зависит ото всех и выпадение всего одного члена его ломает. Нам нужно, чтобы люди умели замещать друг друга и процесс не ломался. Примерно то это и делает Dropout. Это делает модель устойчивой к аномалиям и странным значениям.

9.6.3. Аугментация данных

Это регуляризация на уровне данных: например, на уровне классификации изображений, мы хотим, чтобы отражённый и в другой цветовой температуре котик был всё ещё котиком:



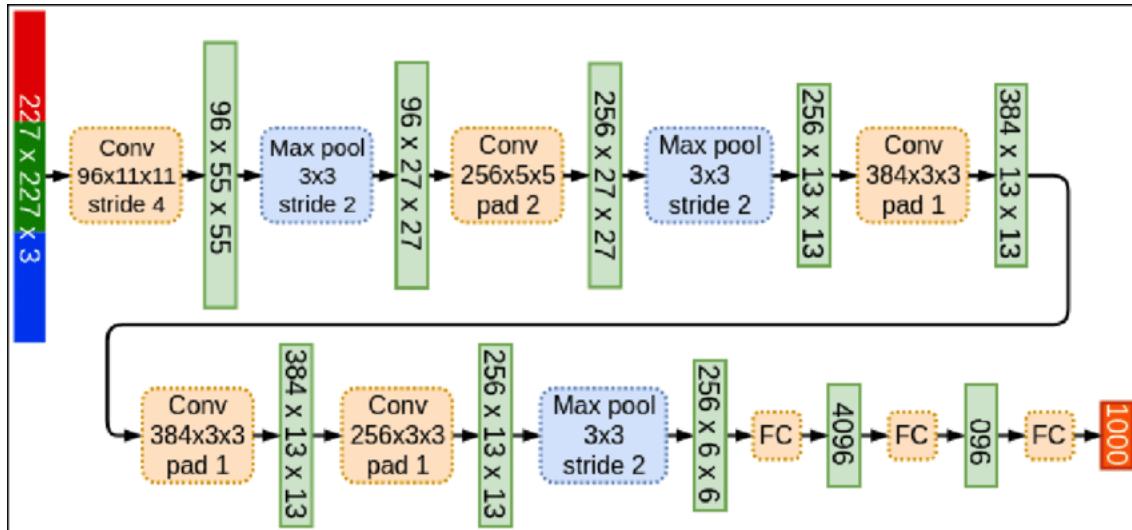
То есть, аугментация - это изменение данных, которое не меняет целевую переменную. С точки зрения сети картинка будет другой, потому что цвета будут другими, но учиться будет по другому.

10. Лекция. Архитектуры CNN

https://www.youtube.com/watch?v=TcUPuKpIlhQ&list=PL0Ks75aof3Th84kETS1Jq_ja-xqLtWov1&index=45

11. AlexNet

Это первая нейросеть, которая выиграла конкурс ImageNet. До этого побеждали только решения без нейросетей. Сама сеть простая:



Давайте обратим внимание вот на что: AlexNet принимает на вход цветное изображение 227×227 . После первого свёрточного слоя карта активации сильно уменьшится до $55 \times 55 \times 96$. Это происходит потому, что в первом слое используются очень большие фильтры 11×11 и $\text{stride}=4$.

Второй слой тоже довольно сильно понижает размерность карт активации до 27×27 , но их получается 256 и это достигается размерами фильтров 5×5 .

3, 4 и 5 слои никак не меняют размеры карт активации, потому что фильтры у них 1×1 . Какой же у них смысл? Такие слои могут использоваться для манипуляции количества карт активации. Также это можно использовать для дополнительной нелинейности - после свёртки идёт нелинейная функция.

На последнем свёрточном слое мы получим 256 карт активации размерностью 13×13 - вот их мы разворачиваем в векторы и отправляем на вход FC слою.

Прогоняя ещё через один FC слой и делаем выход на 1000 нейронов.

Но что же сделало AlexNet таким крутым, что AlexNet победил, а остальные архитектуры нет?

- Сделали функцию активации между слоями ReLU, а не tanh
- Параллельно учили на нескольких GPU. Это не относится напрямую к архитектуре, но относится к тому, сколько экспериментов можно успеть провести и сколько архитектур попробовать
- Аугментация данных. Тоже не относится к архитектуре, но помогает обучаться лучше

11.1. VGG

Через 2 года была представлена VGG:



VGG тоже обычная свёрточная сеть. В чём её отличие от AlexNet?

- Она глубже, больше слоёв.
- Свёрточные слои имеют фильтры 3×3

Почему иметь фильтры размерности 3×3 лучше, чем 11×11 ? Если у вас фильтры размерности 11×11 , то это значит, когда фильтр проходится по картинке - он собирает информацию по большой части картинки. Этот свёрточный слой не может реагировать на маленькие паттерны в квадратике 3×3 или 4×4 . Поэтому свёртки такие маленькие.

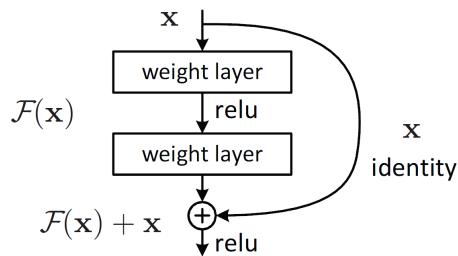
Но есть несколько вариантов VGG, и в разных VGG разное количество FC слоёв.

11.2. Затухание градиентов

Люди после AlexNet и перехода к VGG решили, что чем больше слоёв, тем сеть лучше - разные слои выделяют разного рода сложность информации о картинке. Почему мы просто не бахнем больше слоёв? Потому что тогда мы столкнёмся с проблемой затухания градиента, про которую мы уже говорили при использовании сигмоиды.

11.3. Skip connection

Для решения этой проблемы придумали skip connection:



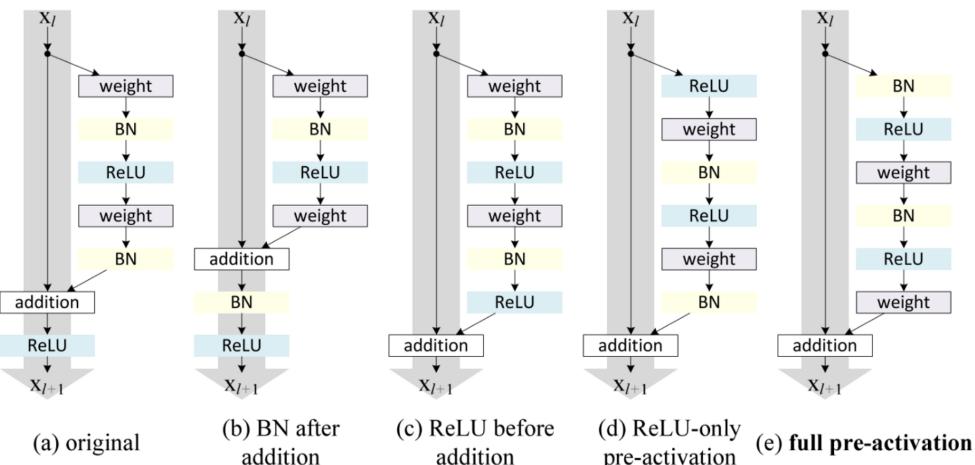
Когда мы считали back propagation для i -го слоя, мы использовали только $i + 1$ -й слой. А в skip connection мы пропускаем иногда некоторые слои. Тогда дальние слои будут связаны со слоями, где градиент ещё не затух. При этом у skip connection нет веса - мы просто передаём какой-то слой не домножая его абсолютно никак вперёд.

Также плюс этого метода в том, что он позволяет протолкнуть информацию с передних слоёв в задние.

Сам метод используется чаще всего в CNN.

11.4. ResNet

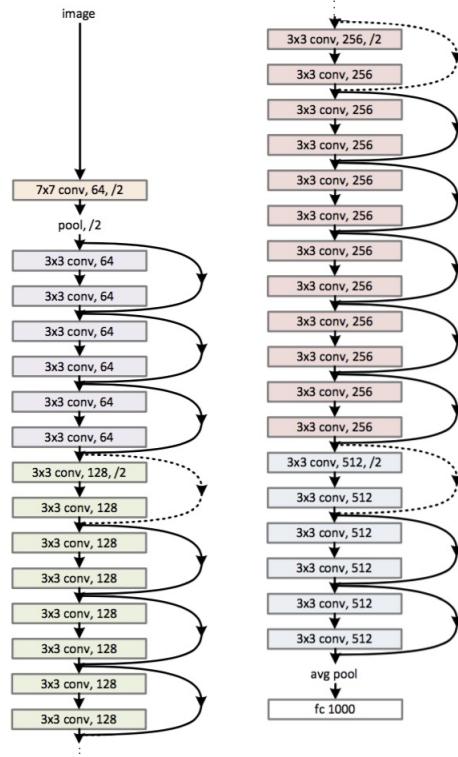
ResNet как раз и поместил в себя skip connection'ы:



Только в самой сети это называется Residual connection из-за сложности внутренней реализации этих ответвлений сети.

Вот как выглядит сам ResNet34:

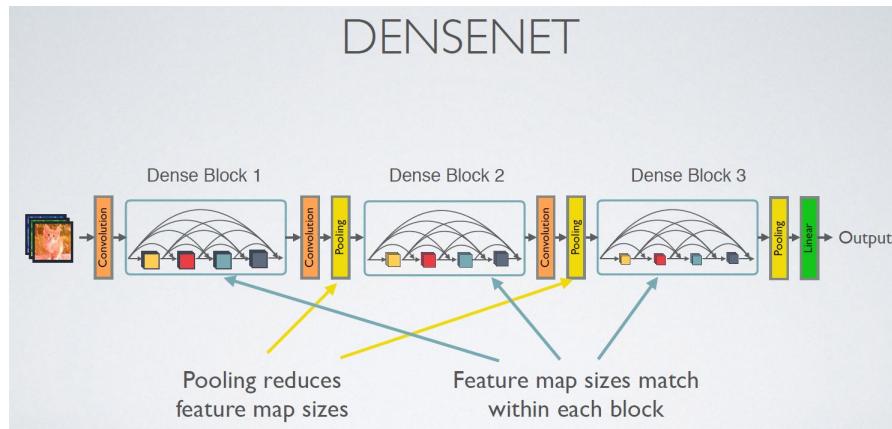
34-layer residual



Самая обычная CNN сеть, только со skip connection'ами. После каждого conv слоя идёт batchnorm слой, и в каждом двух блоках ReLU.

11.5. DenseNet

В этой сети тоже используются skip connection'ы, но каждый новый слой получает на вход все предыдущие слои. Но давайте такое назовём Dense блоком, тогда сеть выглядит так:



Можно заметить, что между Dense блоками используются conv слой + пуллинг - это используется для того, чтобы подогнать размерность с одного dense блока на другой.

Плюсы архитектуры:

- Skip connection'ы очень сильно помогают не затухать градиентам
- Количество слоёв и параметров не очень много
- Conv слои выделяют более разнообразные фичи, потому что принимают больше фичей
- Суммарно эти факторы говорят о малом переобучении - параметров мало и можно обращать внимание на разные фичи

11.6. Bottleneck Block

В torch есть обычные блоки, а есть bottleneck блоки. Они используются чтобы уменьшить размерность карт активаций и уменьшить количество параметров, которые требуются для обработки картинок. Bottleneck блоки - это как прокаченные pooling'и.

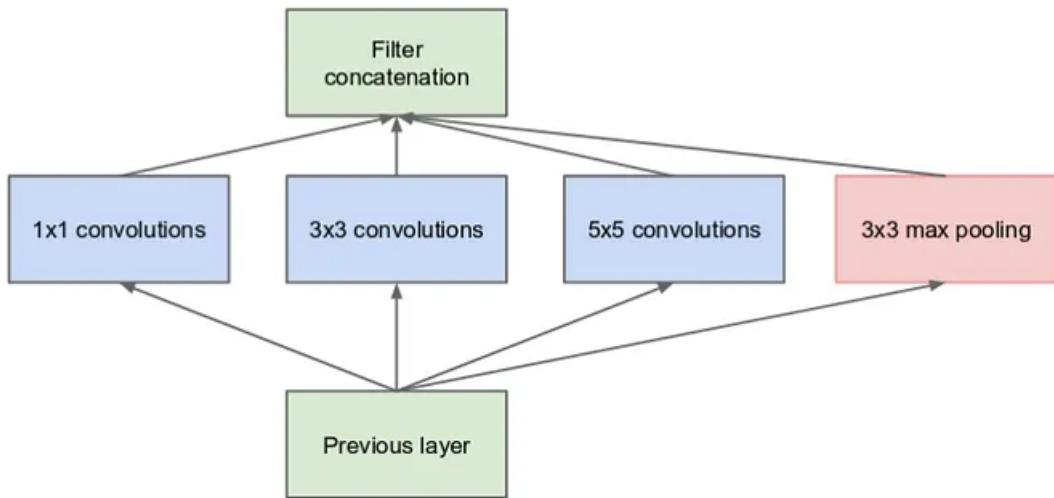
11.7. Model ZOO

В torchvision куда больше архитектур, чем то что мы обсудили.

Давайте обсудим MobileNet. Когда нейронные сети снова вскочили - люди начали решать огромную кучу задач с помощью них. Это всё хорошо, но у нас, например, есть оффлайн перевод текста на телефоне. Для того, чтобы это делать с помощью нейронной сети - нужно нейросеть скачать на мобильник и прогнать предложение через сеть. Если сеть большая - прогон будет занимать долгое время, потому что железо мобильных телефонов очень сильно уступает ПК. При этом проблема во времени критична, потому что мы скорее всего не захотим ждать 5 минут, чтобы у нас перевёлся текст. Нужно адаптировать нейронные сети под мобильное устройство. MobileNet - одна из удачных нейронных сетей, которая сделала это. MobileNet использует легковесные convolution'ы.

11.8. Inception (GoogleNet)

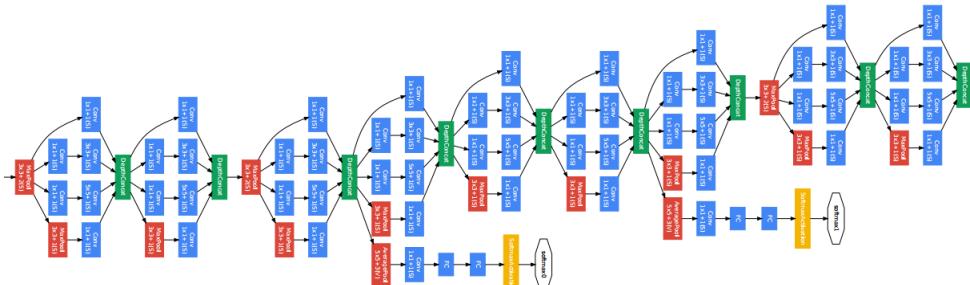
Мы почему-то всегда делали так, что conv слои шли друг за другом, и максимум использовали skip connection'ы, чтобы передавать информацию к верхним слоям. Давайте на одном слое делать несколько conv слоёв (как бы это парадоксально не звучало):



(a) Inception module, naïve version

Все эти conv слои будут выделять разную информацию: кто-то очень низкоуровневую, кто-то очень локальную, кто-то менее локальную.

Также - в качестве борьбы с затуханием градиентов - давайте делать предсказание не один раз, а несколько, в разных частях сети:



Жёлтые блоки на картинке - это и есть наши классификаторы.

12. Лекция. Transfer Learning

https://www.youtube.com/watch?v=oLpRESo27Zw&list=PL0Ks75aof3Th84kETS1Jq_ja-xqltWov1&index=46

Часто датасет под какую-либо задачу для обучения сети содержать мало объектов. Если обучать сеть на этом датасете с нуля, то сеть переобучится.

Пример: медицинские датасеты изображений опухолей, машинный перевод с малораспространённых языков (татарский) и так далее.

Как работает человеческий мозг в этой ситуации? Человек никогда не обучается под какую-либо задачу с нуля. Допустим что какой-то человек знает русский язык и в детстве ещё выучил английский язык. Если этот человек захочет учить испанский язык, то это сделать будет проще, чем человеку, который не знает испанского: у испанского и английского похожая грамматика, конструкции и слова.

Давайте эту идею, что мы обучились делать одну задачу и теперь пытаемся сделать другую задачу - перенесём на нейронную сеть.

Допустим у нас есть одна нейронная сеть, которая обучилась на первом датасете. Также у нас есть вторая нейронная сеть, которая хочет обучиться на втором датасете. Мы хотим использовать знания первой нейросети, для обучения второй.

Эта идея и называется transfer-learning'ом.

12.1. Fine-tuning

Допустим у нас есть ImageNet, но в нём мало фотографий диких животных. Но задачу распознавания диких животных мы всё ещё хотим решить. При этом у нас есть уже одна сеть, которая обучена на ImageNet.

Обучать сеть с нуля не очень хорошо - потому что она бы переобучилась. Давайте сделаем то же самое, что мы делали бы, если бы обучали сеть с нуля, но инициализируем не рандомными значениями, а весами сети, которая была предобучена на ImageNet. После этого дообучаем сеть датасетами диких животных.

Когда наша сеть обучается на ImageNet - она понимает какую-то информацию о картинках. А учитывая, что какие-то животные были среди картинок ImageNet, то наша сеть поймёт и что-то о животных. Когда мы будем обучать её на датасете диких животных - всё что ей останется, это изучить какую-то специфическую информацию о диких животных. Маленького датасета с животными достаточно, чтобы изучить какую-то малую часть информации.

Какая есть проблема? У сети, которая обучалась на ImageNet в конце 1000 нейронов, а в нашем новом - 10 классов, и поэтому просто взять и дообучить нейросеть мы не сможем. Тогда давайте после обучения сети на ImageNet выкинем последний слой и заменим его на новый слой, где будет 10 выходящих нейронов, а не 1000, и он будет инициализирован случайными весами.

На самом деле так тоже будет переобучение. Мы просто поменяли инициализацию сети перед обучением - весов всё ещё много в моменте обучения, а картинок мало. Решение - заморозка слоёв. Это означает, что какие-то слои в процессе обучения не будут обучаться и изменение весов произойдет не будет. Таким образом - мы можем заморозить первые слои сети и дообучить только несколько последних слоёв сети. Тогда обучаться будет меньше параметров и сеть не переобучится.

12.2. Варианты как передать информацию из одной сети к другой

Давайте поговорим о классификации вариантов transfer-learning'a и классификации проблем transfer-learning'a.

Есть три понятия, которые похожи между собой:

- Transfer Learning
- Domain Adaptation
- Multi-task Learning

Что же такое Transfer learning?



В нашем примере Source Domain это ImageNet, Target Domain - это датасет с дикими животными.

Первый вариант задач Transfer Learning'a - это когда не совпадают пространства двух доменов: как пример работа NLP в Source Domain на одном языке, и на другом в Target Domain.

12.2.1. Domain Adaptation

Второй вариант задач - когда распределения признаков в двух доменах разные. Допустим один датасет - ревью фильмов на кинопоиске, а второй датасет - ревью на мобильные приложения в русском GooglePlay. Пространство признаков одинаковые - ревью на русском языке. При этом - распределение признаков разные. Ревью на кинопоиске больше содержат слов "кино" "фильм" "режиссёр" а ревью на мобильные приложения больше содержат слов "мобилка" "связь" и что-нибудь подобное. То есть распределение использования слов разное, и распределение признаков тоже разные.

Третий вариант - не совпадают лейблы, условное распределение лейблов при условии доменов не совпадают. Пример: есть датасет людей, одних и тех же. Но при этом в первой задаче мы определяли пол человека, а во второй задаче определяем расу человека. Можно собрать в кучу и разобрать на две разные идеи - два разных домена и две разные задачи.

Четвёртый вариант - деление на Supervised и Unsupervised - то есть, есть ли лейблы у Target Domain.

12.3. Идеи решения задач Transfer Learning

Когда модель обучается - она учится выделять из объектов inter-domain и intra-domain информацию. Если на примере распознавания лиц:

- Inter-domain - расположение элементов на лице, выражение лица
- Intra-domain - цвет лица, широта глаз

Иными словами: inter-domain - информация, которая сеть выделяет из данных, которая верна для source domain и target domain. Intra-domain информация - которая характерна именно для датасета, на котором мы обучаемся: если мы обучаемся распознавать лица конкретной расы, то это будет цвет лица или широта глаз, например.

Звучит клёво, если мы будем уметь извлекать из первой модели в transfer learning'е то, как она выражает inter-domain фичи и давать эту информацию второй модели. Одну идею мы уже рассматривали - fine-tuning.

12.3.1. Однаковое распределение слоёв

Это идея, где на target-domain'е нет лейблов - unsupervised learning. Давайте будем с помощью некоторого loss'a, чтобы на выходе после некоторых слоёв сети распределение для картинки из source domain и target domain было одинаковое.

12.3.2. Ещё идеи - почитать статью

<https://arxiv.org/pdf/1802.03601.pdf>

12.3.3. Архитектурное разделение

Давайте будем делить архитектурно на две части, и одну часть будем заставлять учить информацию, которая характерна для конкретного domain, а вторую часть учить на то, что характерно для обоих domain.

<https://arxiv.org/pdf/1608.06019.pdf>

12.3.4. Extreme cases

Extreme cases - какие-то нестандартные сеттинги. Например, у нас есть сеть, обученная на большом датасете (и хорошо обученная), и мы хотим использовать информацию из этой сети для обучения новой сети на маленьком датасете.

Но проблема в том, что у нас нет доступа к весам этой большой сети и её архитектуре, мы можем получать только эмбеддинги. По сути - это blackbox. Как здесь сделать transfer learning? С помощью какого-то loss и выхода нашей сети мы можем сделать так, чтобы выход нашей сети на конкретной картинке совпадал с выходом большой сети. Таким образом, наша сеть косвенно будет учиться у большой сети и приобретёт некоторую информацию.

Часть II

Продвинутый поток

13. Семантическая сегментация. Введение

<https://www.youtube.com/watch?v=tIqndofykgc&list=PL0Ks75aof3Tiru-Uv0vYmXzD1tU0NrR8V&index=47>

Про задачу сегментации мы уже говорили, давайте введём формальности: на выходе мы для каждого пикселя будем выдавать вероятность его цвета, принадлежность к какому-либо виду объектов.

13.1. Идея решения: Sliding Window

Давайте для пикселя возьмём окно размеров 5, и тогда получится окошечко 10×10 пикселей. Для каждого пикселя мы получим такое окно и будем пропускать через какой-нибудь классификатор.

Какие есть минусы у этого решения? Это очень долго считать: нам нужно сделать очень много проходов по сетке классификации для одной картинки. Качество такой сетки тоже будет не очень хорошее, потому что в этом маленьком окошке может быть очень трудно что-либо распознать, а если расширять окошко - то мы можем наоборот лишние пиксели захватить или сложно будет понять к какому классу принадлежит пиксель.

13.2. Fully-Conv network

Мы поняли, что нам нужен output такого же размера, как и input. Давайте просто сделаем несколько conv слоём, где размеры карт активации будут совпадать с размером картинки.

Для того, чтобы сделать предсказание - у нас на последнем слое должно быть C карт активаций, где C - количество классов.

Какие здесь проблемы? Это очень дорого, потому что conv слои большие, и значит у нас будет много параметров. Делать меньшие слои - мы не увидим паттерны, которые будут нужны для сегментации.

Тогда нам нужно какие-то образом сжимать информацию о картинке перед построением сегментации и снижать количество параметров.

13.3. CNN

Давайте возьмём VGG, ResNet, Inception или ещё какую-нибудь сетку для классификации. Возьмём картинку с коровами и прогоним через сетку и выкинем какие-то FC слои. Мы получим карту активации, которая меньше, чем наша изначальная картинка. Давайте просто upsampling'ем её. Это значит - по-

высим её resolution. Например из матрицы $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ можно сделать матрицу $\begin{pmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{pmatrix}$. В принципе

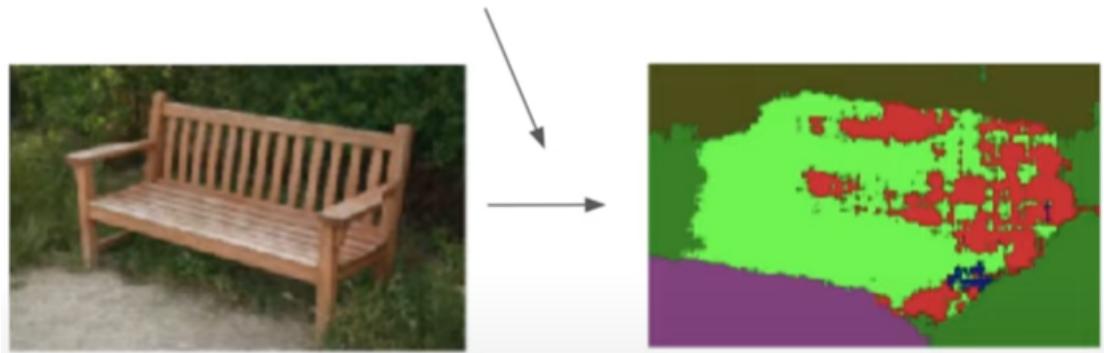
это работает. И просто по итогу учим loss, чтобы карта сегментации раскрывалась в сегментируемую картинку.

Минусы такого решения:

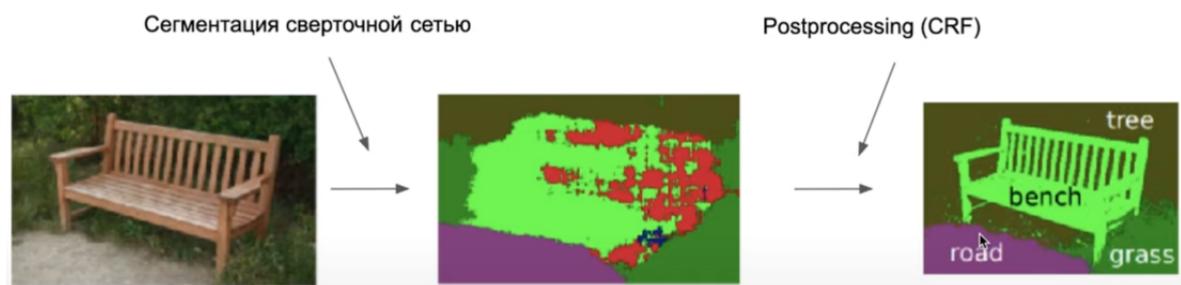
- Сегментация будет не точной из-за того, что мы делаем upsampling.
- Downsampling и большой stride разрушают пространственную информацию
- Scale Variability - это не о том, что объект может быть в разных частях картинки, а о том, что объект может быть разного размера на картинке.

Вот пример того, как это работает в чистом виде:

Сегментация сверточной сетью



Но вот если сделать Post processing, то у нас сегментация становится в разы лучше:

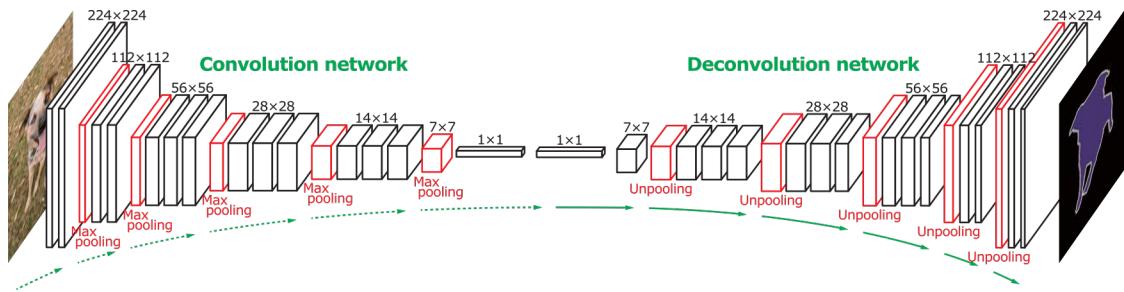


14. Семантическая сегментация. Трюки: Deconvolution, Dilated Convolution

<https://www.youtube.com/watch?v=K73tZxH9nvE&list=PL0Ks75aof3Tiru-Uv0vYmXzD1tU0NrR8V&index=48>

14.1. Замена upsampling

Давайте заменим upsampling - будем не просто пытаться как-то ресайзнууть картинку, а чуть более умное решение:



После получения финальной фиче карты в CNN мы не будем делать upsampling, а сделаем ещё одно обучение для новой части сети - deconvolution nn. С одной стороны кажется, что в этой новой сети мы будем разворачивать картинку аналогично тому, как мы её сворачивали: и да, и нет.

На самом деле такие слои называются Transposed Convolution. Работают они так: сначала сделаем upsampling такого вида:

$$\begin{pmatrix} 2 & 5 \\ 4 & 13 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 13 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 3 & 1 & 2 \\ 2 & 0 & 0 \\ 5 & 4 & 7 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 4 & 0 \\ 18 & 127 & 57 \\ 0 & 8 & 0 \end{pmatrix}$$

И это upsampling со stride=2. С 2-ки нужно сдвинуться на 2 вправо, чтобы получить 5-ку. После этого происходит операция свёртки.

На самом деле, с какой-то стороны transposed convolution - это upsampling + convolution.

14.2. Варианты upsampling'a

14.2.1. Nearest Neighbours

$$\begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix} \rightarrow \begin{pmatrix} 10 & 10 & 20 & 20 \\ 10 & 10 & 20 & 20 \\ 30 & 30 & 40 & 40 \\ 30 & 30 & 40 & 40 \end{pmatrix}$$

14.2.2. Bed of nails

$$\begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix} \rightarrow \begin{pmatrix} 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ 30 & 0 & 40 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

14.2.3. Bilinear

$$\begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix} \rightarrow \begin{pmatrix} 10 & 12 & 17 & 20 \\ 15 & 17 & 22 & 25 \\ 25 & 27 & 32 & 35 \\ 30 & 32 & 37 & 40 \end{pmatrix}$$

14.2.4. Max-unpooling

Тут подход состоит в том, что чтобы сделать unpooling - нужно знать как мы делали pooling в самом начале сети. Допустим у нас происходил такой pooling при переходе от матрицы 4×4 к матрице 2×2 :

$$\begin{pmatrix} 1 & 2 & 6 & 3 \\ 3 & 5 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 7 & 3 & 4 & 8 \end{pmatrix} \rightarrow \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

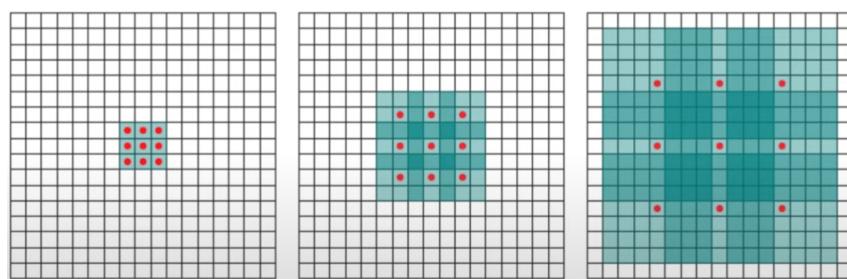
Тогда при операции unpooling'а мы будем получать следующие матрицы:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 \end{pmatrix}$$

При этом нужно уточнить, что сама deconvolution сетка не обязана быть симметричной с convolution частью.

14.3. Dilated Convolutions

Теперь мы upsampl'им сам фильтр: $y_i = \sum_{k=1}^K x_{ii+rk} w_k$. У этого решения есть преимущества в виде того, что мы маленьким фильтром можем захватить большое пространство картинки:



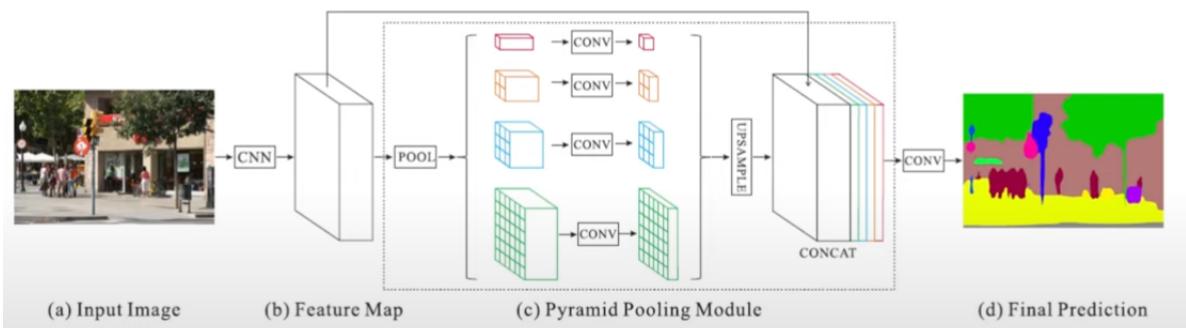
14.4. Multi-scale Context Aggregator

Давайте предположим, что есть две фотографии: где пёсель большой, и где пёсель маленький. Так вот наши dilated convolution как раз могут искать пёсика в разных размерностях.

Но чтобы точно что-то сделать с размерностями картинки - мы можем отправлять картинку в разные части сети.

14.5. Pyramid pooling network

Но есть другая идея как побороть разные размеры на картинке. Мы уже видели про параллельные conv слоя, а теперь у нас будет 4 параллельных pooling слоя.



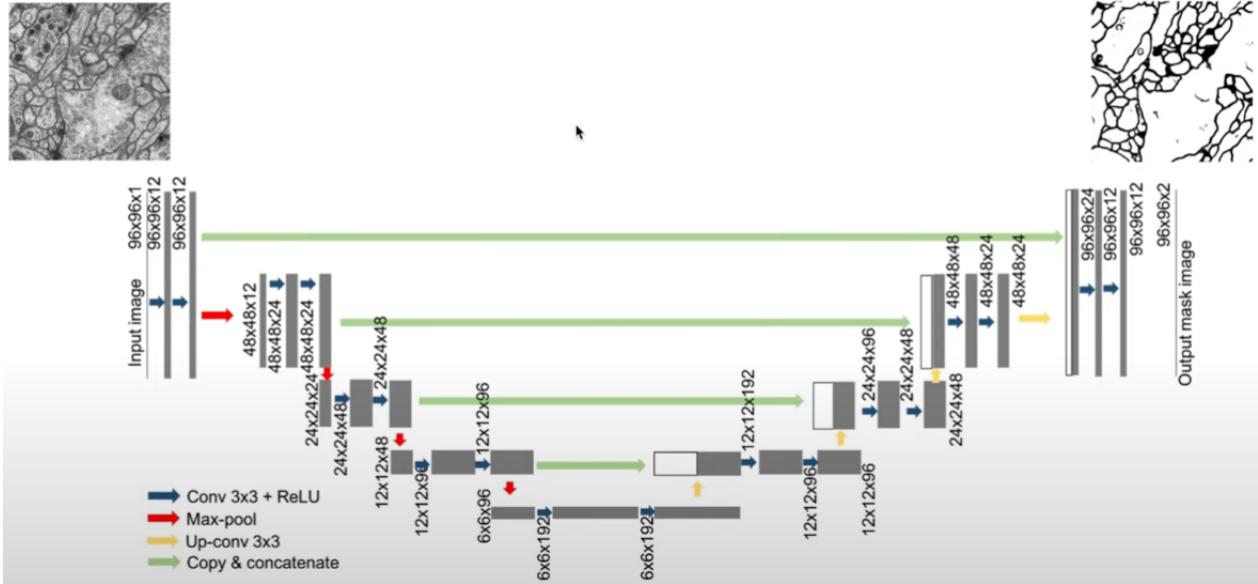
У нас будет четыре карты активации: $1 \times 1, 2 \times 2, 3 \times 3, 6 \times 6$. Все эти карты активаций мы прогоняем через conv слой 1×1 и дальше upsampling.

14.6. Семантическая сегментация. Архитектура UNet

<https://www.youtube.com/watch?v=yEuIV5FsRM&list=PL0Ks75aof3Tiru-Uv0vYmXzD1tU0NrR8V&index=49>

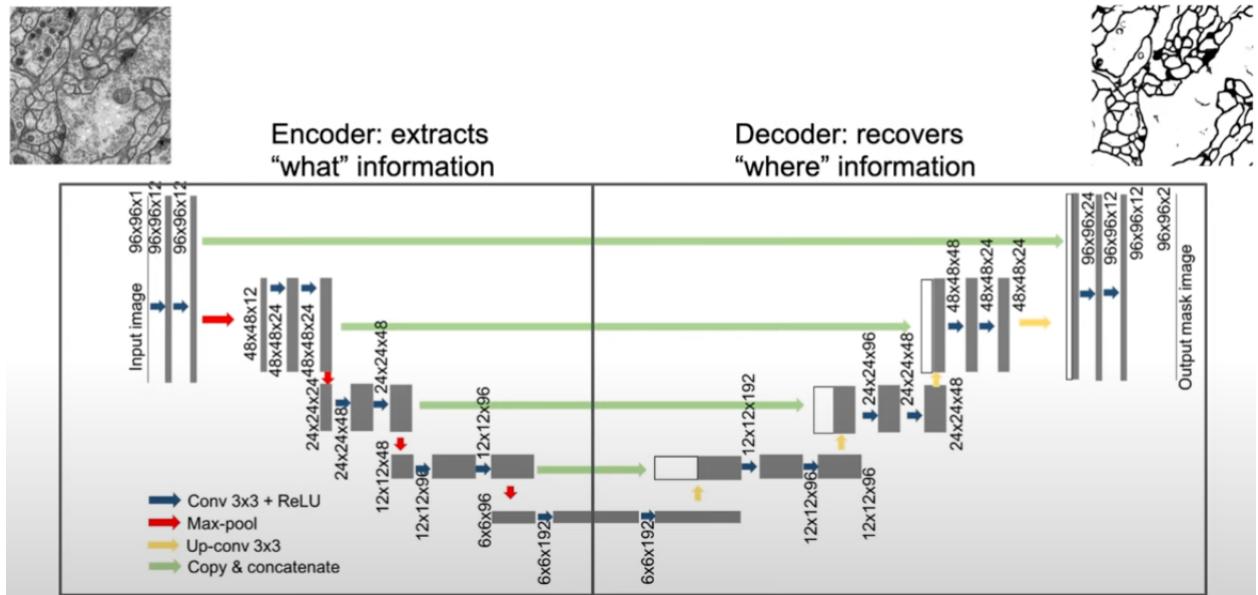
UNet - архитектура для задачи сегментации, предложенная в 2015-м году и конкретно для медицинских целей. Чем отличается задача сегментации медицинских изображений от обычной задачи сегментации?

- В сегментации медицинских изображений очень важно точно сегментировать очень маленькие объекты.
- Обычно медицинская сегментация - бинарная.



Если не учитывать skip-connection'ы - то это архитектура из прошлого видео. Но вся суть в них. В правой части сети у нас будут конкетанериоваться фиче мапы от upsampling'a и те, которые были получены в conv сети. И вот в отличие от любой deconvolution сети, эта сеть должна быть симметричной. Почему эта симметричная информация помогает UNet?

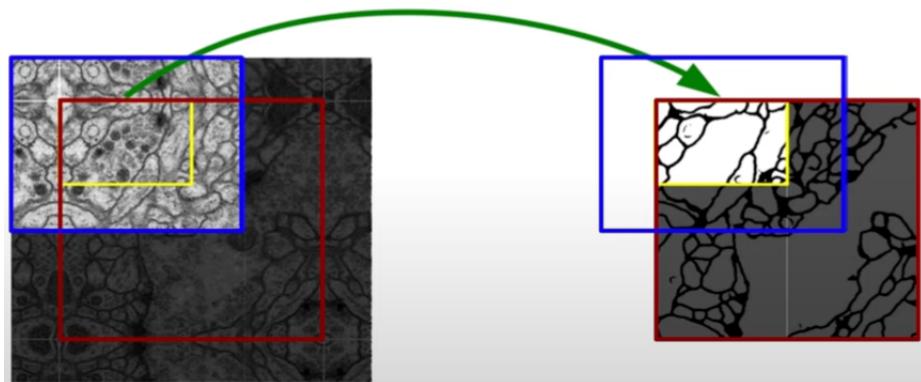
Давайте обозначим эти две части сети как encoder и decoder:



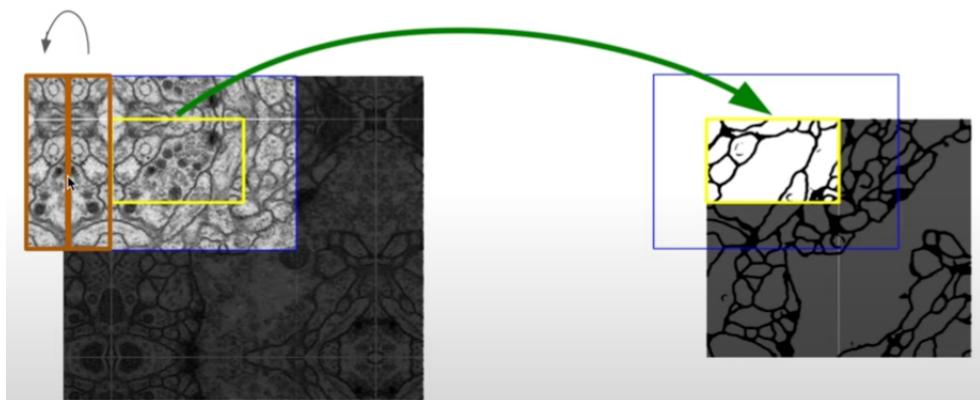
То есть в энкодере мы вычленяем информацию из картинки, а с помощью skip-connection'ов в декодере пытаемся найти место для этой информации. Это помогает бороться с проблемой, что при сжатии и расжатии картинки в conv и deconv слоях информация теряется.

14.7. Overlap-tile strategy

В UNet используются сополнительные слои без паддингов, поэтому сегментация на выходе может быть получена только для внутренней области изображения, то, что указано красным на картинке:

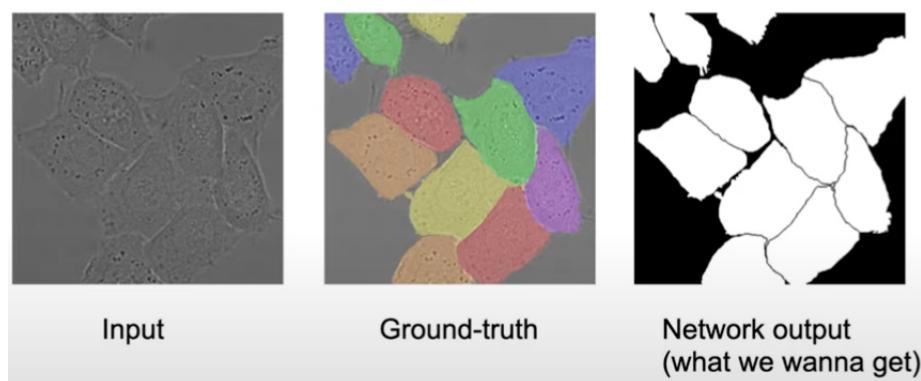


Решили эту проблему тем, что отзеркалили эту часть, которая не влезает и этим самым расширили картинку:

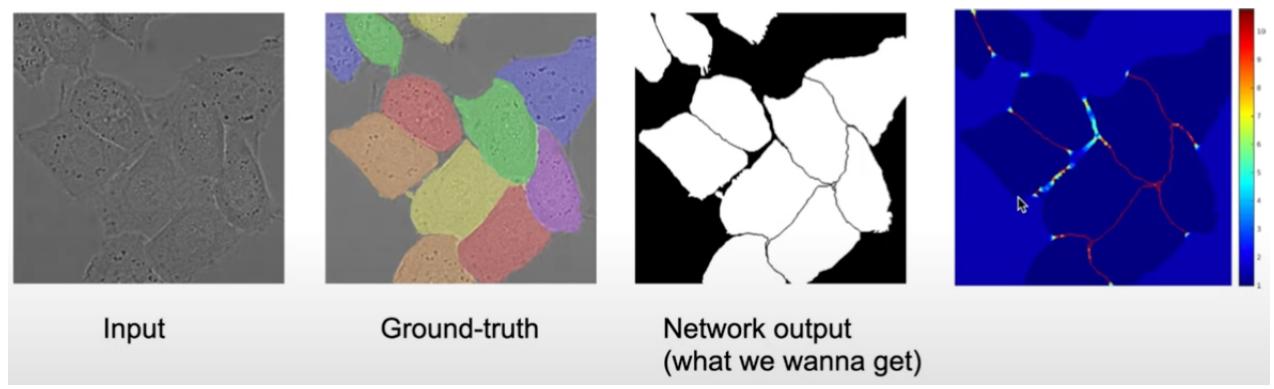


14.8. UNet: Loss

У UNet есть ещё одна проблема: в медицинских задачах нужно сегментировать очень тонкие линии, примерно так:



Loss у сегментации считается для каждого пикселя. Но обучаясь сетка будет акцентировать внимание на фон и внутренность клеток, и будет мало обращать внимание на очень тонкое разделение между клетками. Для этого в UNet'е сделали веса для пикселей:



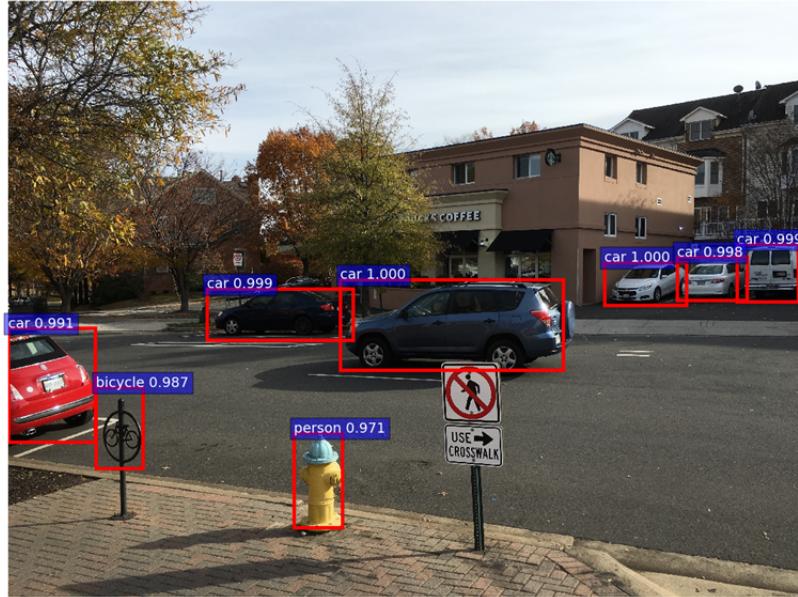
То есть красные пиксели будут давать большую ошибку, чем тёмно-синие.

15. Лекция. Нейронная детекция объектов. Основы

<https://www.youtube.com/watch?v=Y4JvVOaZWsU&list=PL0Ks75aof3Tiru-Uv0vYmXzD1tU0NrR8V&index=51>

15.1. Детекция объектов: постановка задачи

У нас есть картинка и мы хотим на неё сделать bounding-box на каждый объект - прямоугольники, которые показывают где находится объект:



Почему такие объекты, а не маски? Маски потом тоже придумали, но изначально исследователям показалось сначала проще предсказывать такое предсказание в виде прямоугольника, и размечать такие данные тоже намного проще.

Сами Bound-Box'ы задаются следующими полями:

- Верхний левый угол (x_{tl}, y_{tl})
- Нижний правый угол (x_{br}, y_{br})
- Какой класс находится в bound-box
- Уверенность в детекции - вероятность.

Получается всего 6 чисел.

15.2. Датасеты

15.2.1. Датасет MS COCO 2017

Содержит:

- 80 классов для детекции.
- На train 120k изображений $\sim 18\text{GB}$
- На val 5k изображений $\sim 1\text{GB}$
- На test 40k изображений $\sim 6\text{GB}$

Изображения содержат разные размеры и качества картинок.

15.2.2. Google Open Images

Содержит:

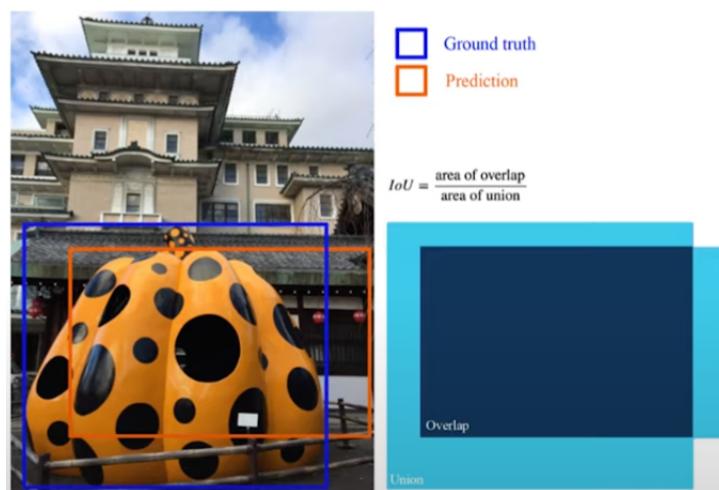
- 600 классов
- 1,743M картинок на train
- 14,6M боксов на train
- 41k картинок на val
- 304k боксов на val
- 125k картинок на test
- 937k боксов на test

15.3. Метрики

15.3.1. Intersection over Union (Jaccard Index)

У нас есть два прямоугольника - ответ и наше предсказание (y_i, p_i), тогда метрика считается как

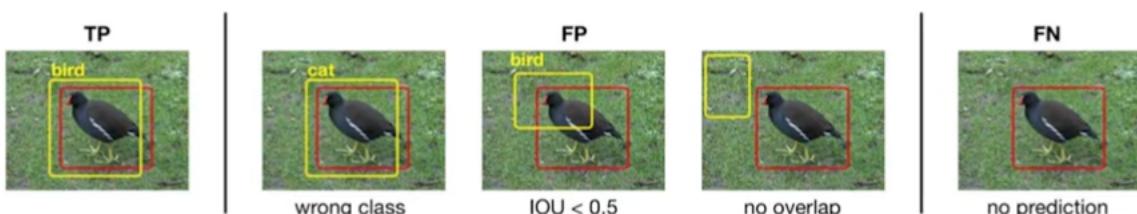
$$\frac{|y_i \cap p_i|}{|y_i \cup p_i|}$$



15.3.2. Positives and Negatives

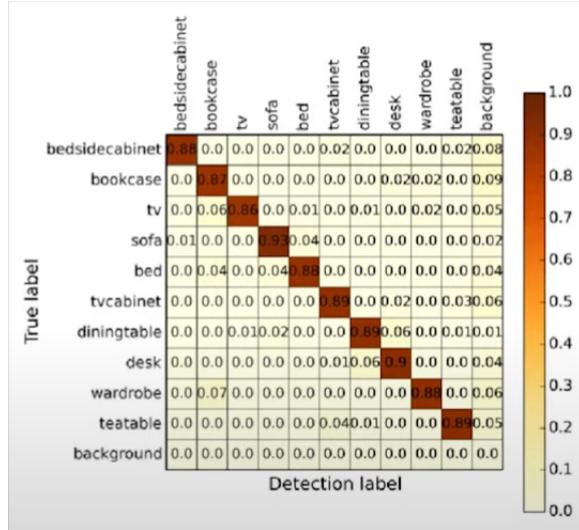
При этом ещё нужно ввести что в этом случае мы считаем True Positive итд.

- True Positive - у нас $\text{UoI} \geq 0.5$, мы верно угадали класс.
- False Positive - мы определили неверный класс или $\text{UoI} < 0.5$
- Если мы вообще не определили объект - False Positive



True negative непонятно как считать, но для Precision и Recall нам его и не нужно.

Когда классов много - неплохо выводить confusion matrix:



15.4. Как решать bounding box?

Давайте просто перебирать окна разного размера, перемещая их по картинке, а оставляем в итоге те box'ы, которые имеют высокий скор.

16. Лекция. Нейронная детекция объектов. Двухстадийные нейросети

<https://www.youtube.com/watch?v=WrKl7GHWiIA&list=PL0Ks75aof3Tiru-Uv0vYmXzD1tU0NrR8V&index=52>

16.1. Классификация детекторов

Детекторы делятся на три вида:

- Two-Stage (Proposal-based)

- RCNN
 - Fast RCNN
 - Faster RCNN
 - Mask RCNN

- One-Stage

- SSD
 - YOLOv3
 - DSOD
 - DFBNet

- Points-based

- CenterNet
 - CornerNet
 - ExtremeNet

16.2. Two-staged подход

Состоит из двух этапов (как не странно)

1. Генерация самих боксов
2. Приводим боксы к одному размеру (обычно относится к первому этапу)
3. По каждому боксу с помощью CNN предсказать класс внутри бокса
4. В пост-процессе объединяем некоторые боксы

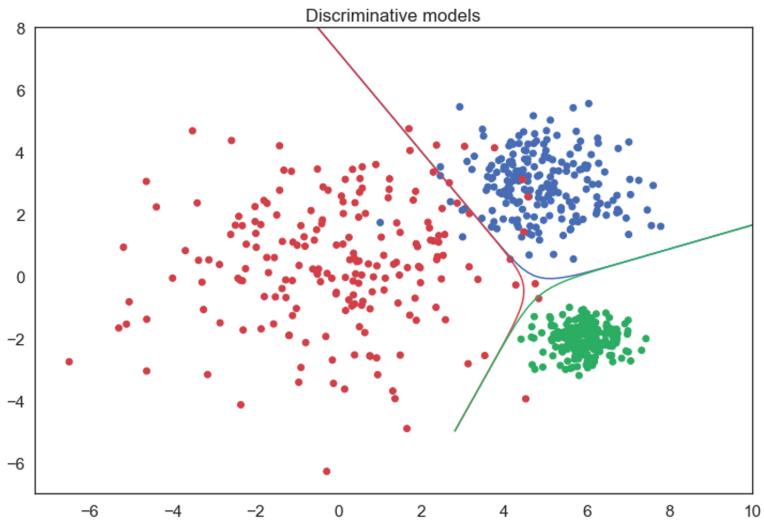
17. Лекция. Нейронная детекция объектов. Одностадийные нейросети

<https://www.youtube.com/watch?v=OPK63uqAQLs&list=PL0Ks75aof3Tiru-Uv0vYmXzD1tU0NrR8V&index=53>
TODO: затухать

18. Лекция. Генеративные модели, автоэнкодеры

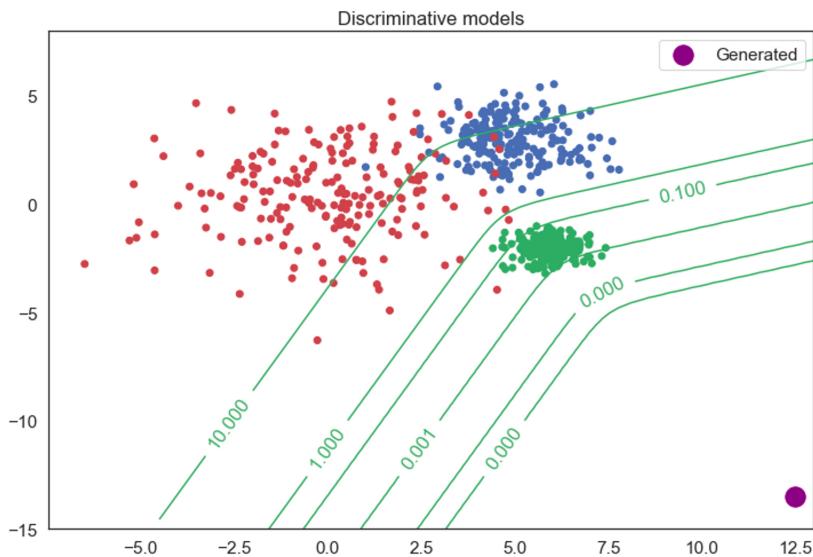
<https://www.youtube.com/watch?v=6qVfC7P9dEc&list=PL0Ks75aof3Tiru-Uv0vYmXzD1tU0NrR8V&index=57>

Все изученные ранее модели были дискриминативные: с помощью них можно было выучить распределение $\mathbb{P}(y|x)$. Теперь же мы хотим уметь решать немного другую задачу. Допустим у нас есть многоклассовая классификация:



Мы получили разделяющие поверхности. А теперь мы хотим попытаться сгенерировать не изображение, а точки какого-то класса. Что нужно, чтобы сгенерировать изображение из зелёного класса?

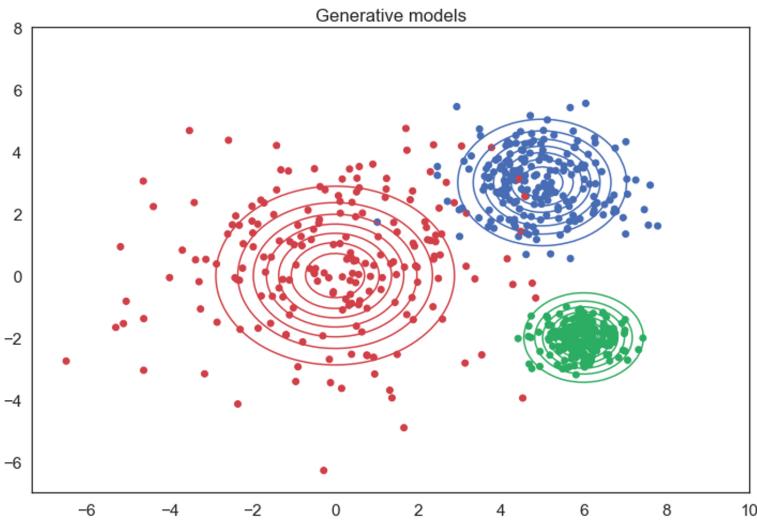
Давайте возьмём $\log(\mathbb{P}(y|x))$ и для градиентного спуска добавим минус перед логарифмом. Мы будем спускаться по этой поверхности $-\log(\mathbb{P}(y|x))$. Если мы так сделаем - мы сгенерируем фиолетовую точку.



Ну и эта точка является просто шумом, хоть и попадает в поверхность зелёного класса, а в терминах изображения это было бы непонятно что.

Как мы тогда хотим спускаться по такой поверхности, чтобы сгенерировать объект, который будет удовлетворять распределению объектов наших данных?

Такие модели как раз называются генеративными. Если у нас есть какая-то такая штука и мы будем спускаться в центр нашего распределения, то у нас, скорее всего, получится какая-то такая штука:



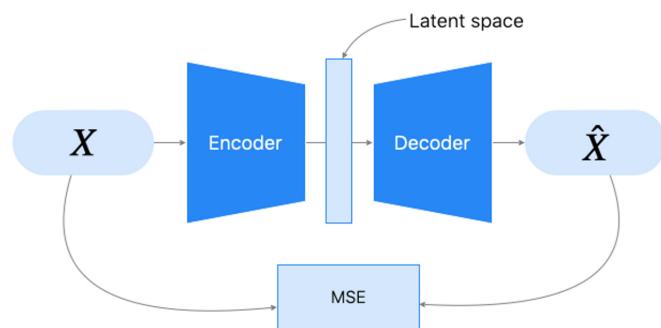
Условно - захотим красную точку - сгенерируем её в красном центре.

Какая есть проблема? Очевидно, здесь выборка какая-то игрушечная. Здесь данные распределены просто - двухмерные и с явным центром. Картинки размером 28×28 уже становятся не такими очевидными, что уж говорить про картинки большего размера.

Предположим, что есть функция f такая, что величина $f(X)$ будет распределена нормально, причём как правило можно взять $f : \mathbb{R}^n \mapsto \mathbb{R}^m$. То есть, условно - мы берём большое изображение и уменьшаем его. Мы сейчас будем учить такую функцию. Такое пространство размерности m называется латентным. Для генерации нам необходимо найти и другую функцию g , такую, чтобы $g(f(x))$ снова была распределена как исходные данные.

18.1. Auto Encoder

Мы берём изображения (например чисел), пропускаем через encoder, потом пытаемся восстановить с помощью decoder'a и смотрим на разницу объектов:



Мы получим какую-то модель, которая будет хорошо кодировать нашу модель в меньшей размерности. Сам AutoEncoder выглядит примерно так:

```

class AE(nn.Module):
    def __init__(self, inp_dim, hidden_dim):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(inp_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, hidden_dim)
        )
        self.decoder = nn.Sequential(
            nn.Linear(hidden_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, inp_dim)
        )

    def forward(self, x):
        shapes = x.shape
        x = x.view(x.size(0), -1)
        return self.decoder(self.encoder(x)).view(*shapes)

    def encode(self, x):
        x = x.view(x.size(0), -1)
        return self.encoder(x)

```

На вход мы подаём входную размерность и размерность, которую мы хотим получить у латентного пространства.

Если мы посмотрим на то, что получается у модели, учитывая, что модель не знает ничего о y - только сами признаки, мы увидим, что между собой эти числа в латентном пространстве различимы:



18.1.1. Corruption denoising

Для чего ещё можно использовать автоэнкодеры? Представьте мы передаём сигналы через провод, который допускает ошибки - у нас получается шумное изображение, а мы хотим обратно получить исходное. Вот у нас есть линия, в которой мы берём точки и добавляем к ним шум, а после этого пытаемся зашумлённые точки восстановить в те, которыми они были.

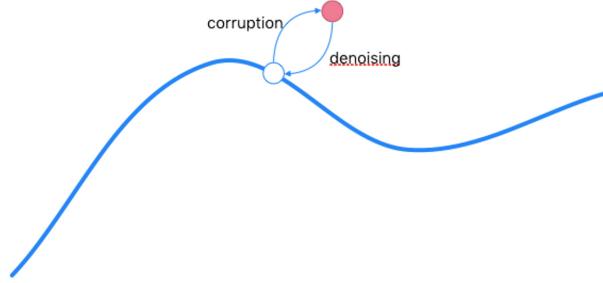
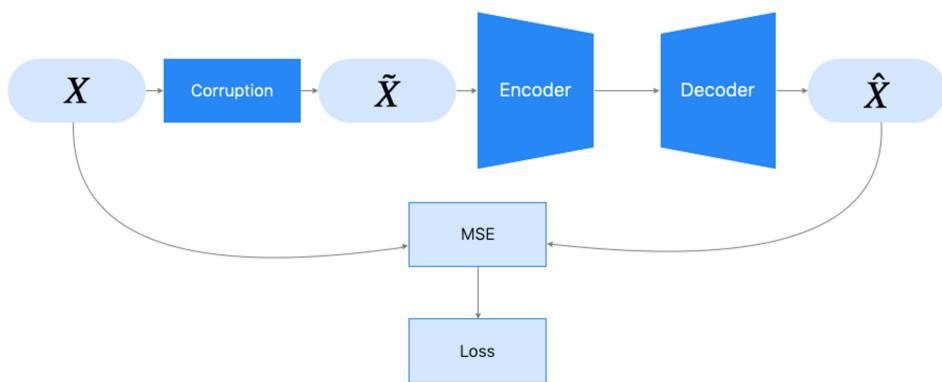
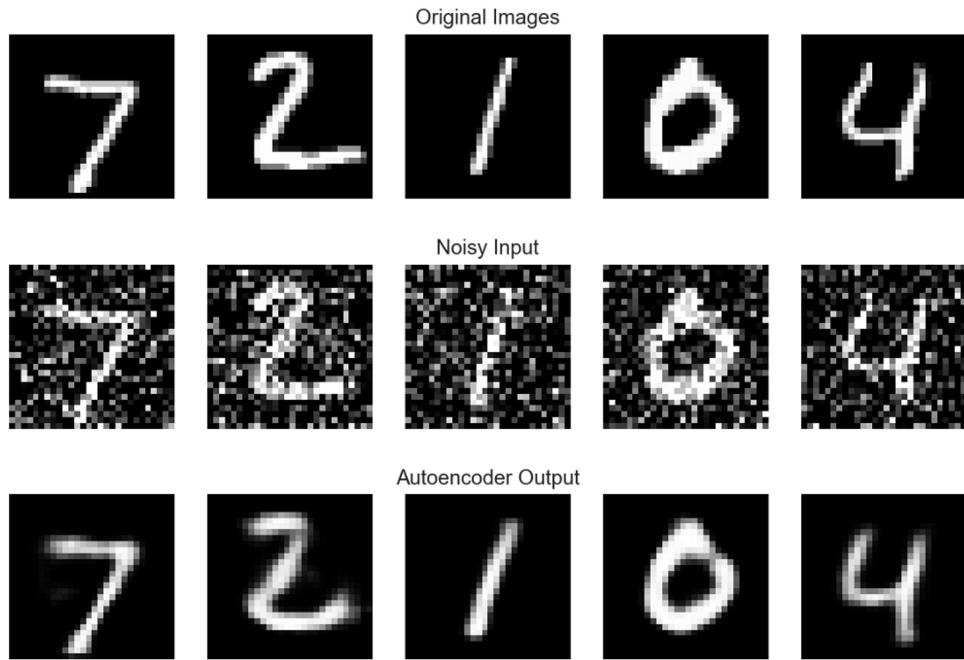


Схема получается очень похожей на просто auto-encoder, за исключение того, что добавилась логика зашумления точки:



Давайте попробуем прогнать MNIST через auto-encoder:

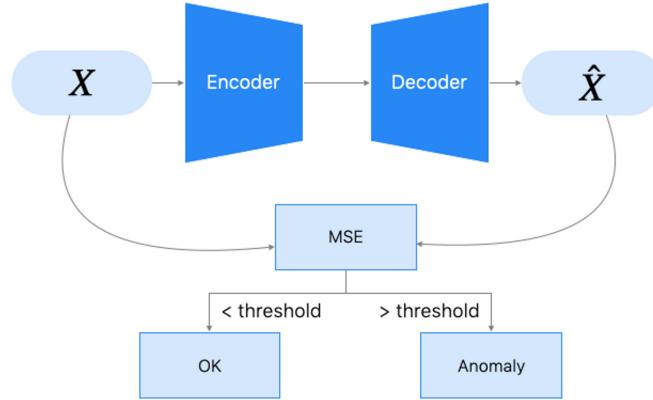


Видно, что изображения похожи между собой. Саму эту операцию можно делать не только с картинками или числами, но со всем чем хотим: например для звука тоже можно применять.

18.1.2. Детекция аномалий

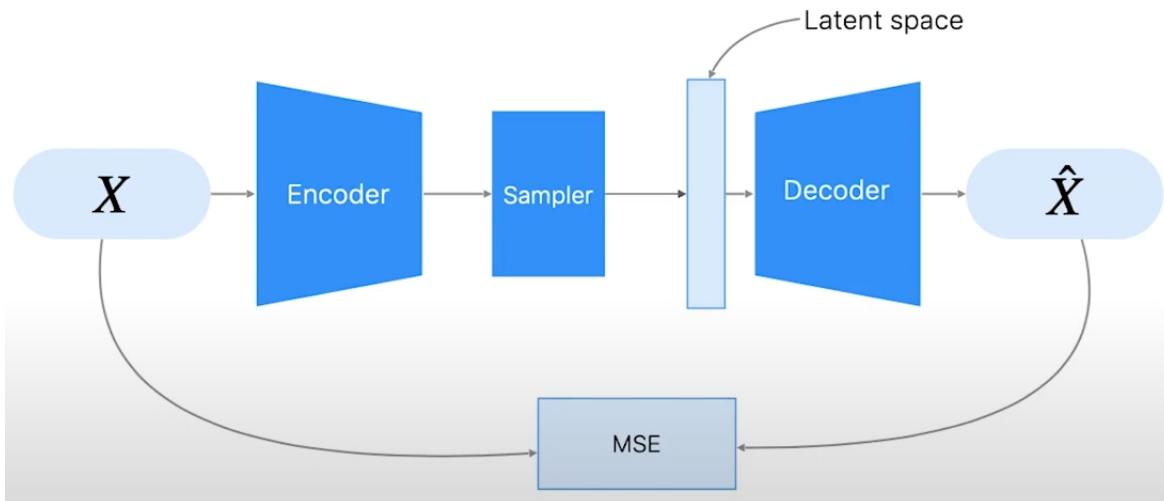
Допустим у нас есть модель, чтобы классифицировать собачек и кошечек. А тут приходит изображение морского котика, что делать? Давайте возьмём auto-encoder и обучим его на собачках и котиках. После

обучения мы получим encoder и decoder, который хорошо кодирует и декодирует только изображения с собчками и котиками. Если нам на вход придет изображение морского котика, то разность между тем, что поступило на вход auto-encoder'у и вышло - будет большая. Давайте смотреть на эту разность и решать - аномалия это или нет:



18.2. Variational Auto-Encoder

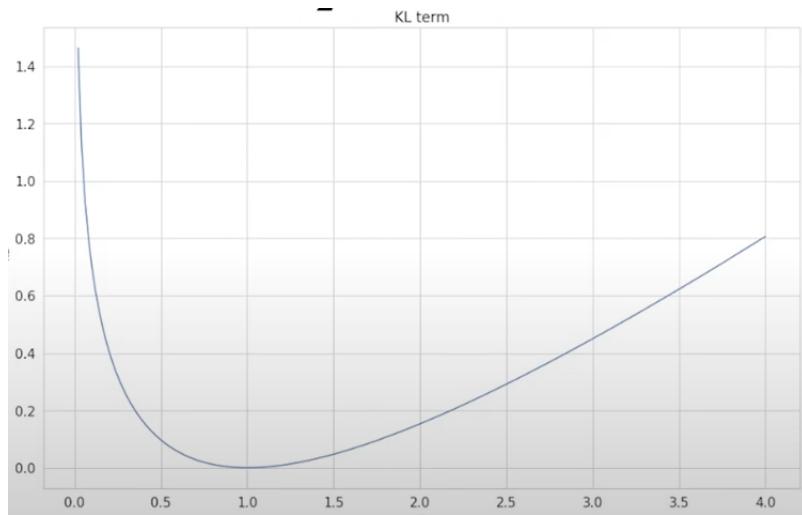
Отличие от auto-encoder'a в том, что вместо того, чтобы encoder выдавал какой-то вектор, то здесь encoder выдаст параметры какого-то распределения (допустим нормального) по каждой компоненте: $\{\mu_i, \sigma_2^i\}$. Эти параметры мы засовываем в sampler. С помощью этих параметров мы можем выбрать случайную точку из этого распределения, и вот эта точка будет располагаться в латентном пространстве. То есть объект, которых входит в VAE генерирует не одну точку, а пространство, из которого точка будет браться. После этого мы эту точку пускаем в декодер и получаем выходной объект.



18.2.1. Loss

Для генерирования данных нам нужно знать из которого распределения нам семплировать. Наверное, нам хочется, чтобы это распределение было хорошо известно, и чтобы семплер не нужно было руками реализовывать и он был готов. Самым распространённым распределением является многомерное нормальное распределение. Мы же ещё хотим, чтобы \mathbb{E} было 0, а также с каким-то стандартным отклонением 1 по каждой компоненте. Мы хотим, чтобы все μ_i и σ_i больше всего подходили под эти параметры. Предлагается такая функция:

$$L_{KL} = \frac{1}{2}(\sigma_i - \log \sigma_i - 1 + \mu_i^2)$$



Как мы видим, минимум достигается в единице, чего мы и хотели. А зачем такая сложная формула? Дело в том, что эта функция взята не спроста.

18.2.2. Дивергенция

Как померить меру схожести между двумя распределениями? Используется дивергенция: $D(p||q)$ для распределений p и q

Свойства дивергенции

- $D(p||p) = 0$
- $D(p||q) \geq 0$
- при $p \neq q$ у нас $D(p||q) \neq D(q||p)$
- $D(p||q) + D(q||h) > D(p||h)$

18.2.2.1. KL-дивергенция Одной из самых часто использующихся дивергенций является *KL*-дивергенция:

$$KL(p||q) = - \sum_{x \in X} p(x) \log \left(\frac{q(x)}{p(x)} \right)$$

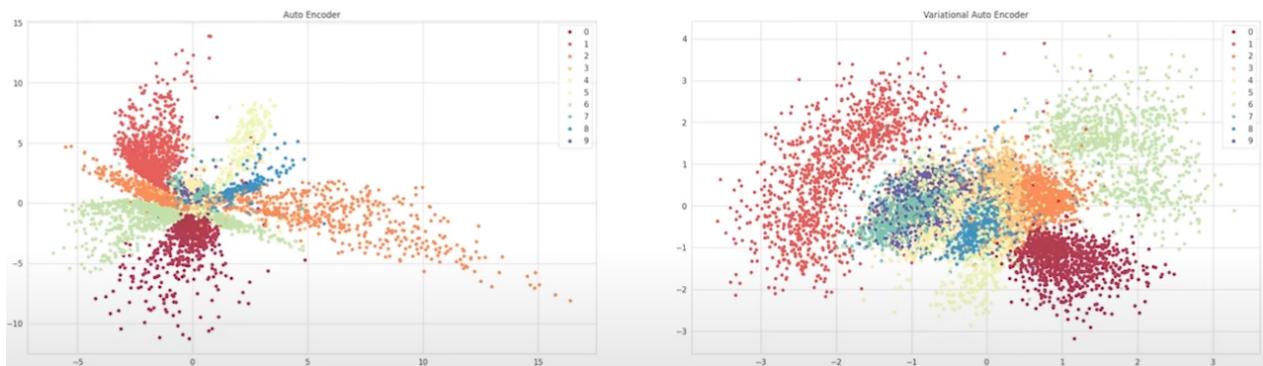
Таким образом, вместо обычного loss для auto-encoder'a, мы добавим ещё loss для того, как мы кодируем данные в латентном пространстве.

Тогда итоговый Loss у нас будет:

$$L = L_{\text{rec}} + \beta L_{\text{KL}}$$

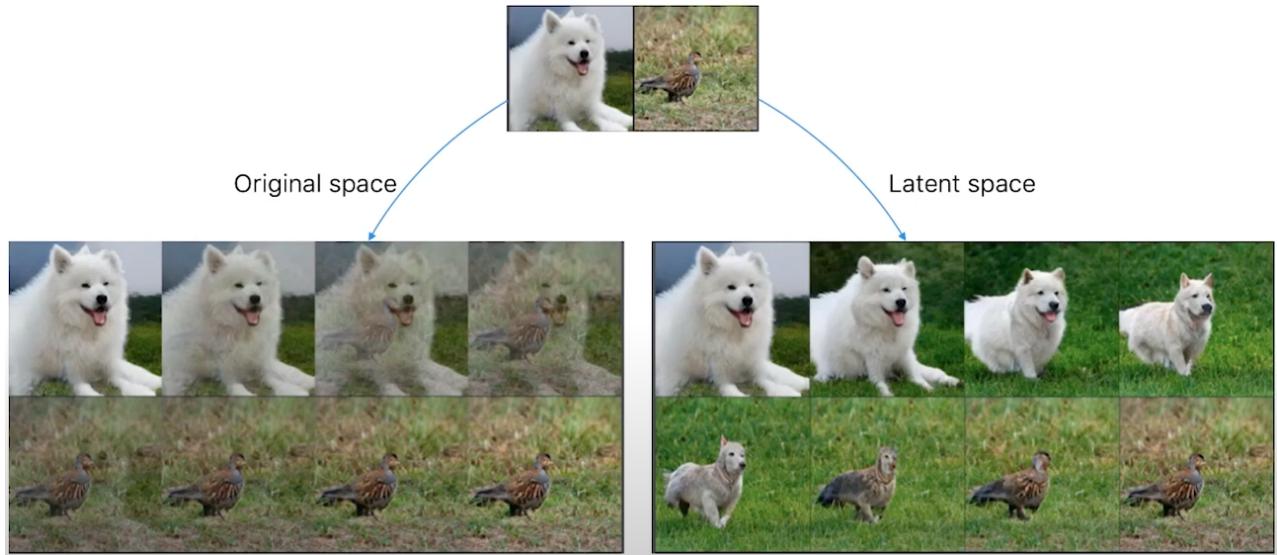
18.3. Сравнение латентных пространств AE и VAE

AE сжимает всё плюс минус в одну точку, хоть и различимые данные, но классы находятся близко. В VAE мы в loss учли, чтобы попытаться раздвинуть наши данные:



18.4. Пример обучения латентного пространства VAE о преобразовании собаки в птицу

Давайте обучим VAE и попытаемся из собачки получить птичку.



Слева представлено если мы будем сочетать λ исходной картинки и $(1 - \lambda)$ конечной картинки. Это явно не то, что мы хотим получить, потому что собака просто исчезает и появляется птица. И это даже не лежит в каком-то пространстве адекватных картинок, которые можно было бы назвать фотографиями.

А вот то, что получилось справа - вполне то что мы хотели получить. Собака именно трансформируется в птицу и картинки вполне нормальные. Качество перехода куда лучше.

18.5. Пример обучения латентного пространства VAE для получения весёлого человека

У нас есть датасет с двумя лейблами для объектов: фото людей, которые грустят, и которые улыбаются. Давайте обучим VAE и пропустим сначала все картинки с улыбкой и усредним точки. Эта точка, грубо говоря, представляет из себя латентное представление людей, которые улыбаются. То же самое мы можем сделать и с людьми, которые грустят. Давайте вычтем одно из другого - и мы получим вектор - то что значит улыбка, по сути, в латентном пространстве. Тогда мы можем прибавлять этот вектор в латентном пространстве тем людям, которые не улыбаются и в выходном изображении они будут улыбаться.

18.6. Лекция. Генеративные модели. Генеративно-состязательные сети

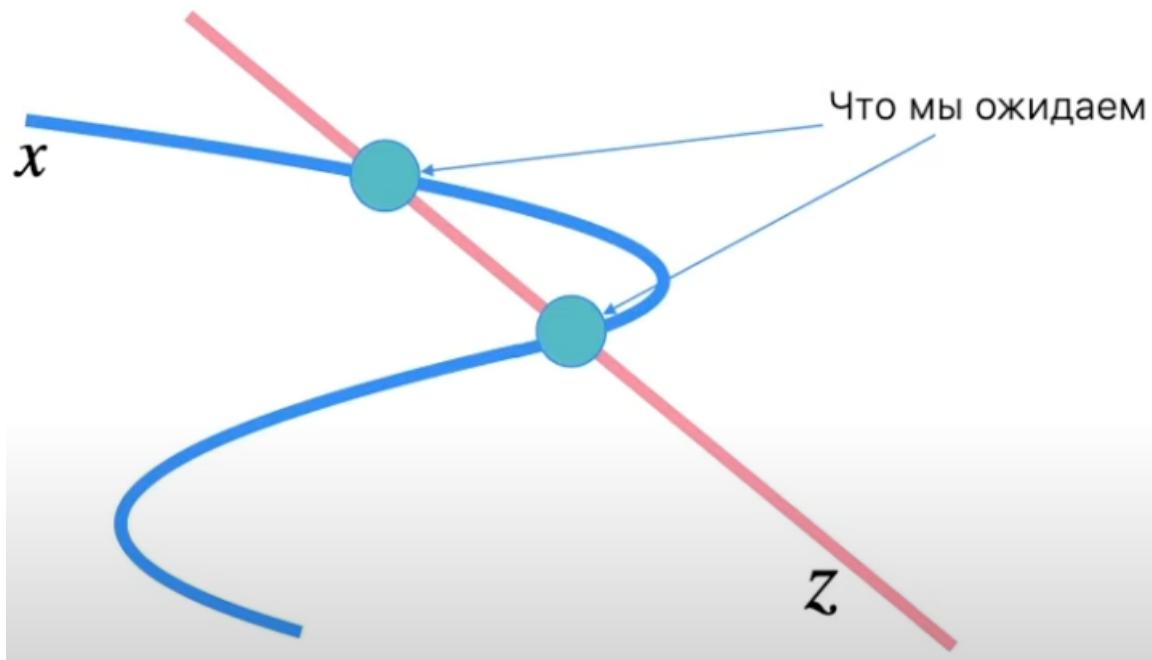
<https://www.youtube.com/watch?v=An20D0E0ctc&list=PL0Ks75aof3Tiru-Uv0vYmXzD1tU0NrR8V&index=60>

Давайте посмотрим что происходит с лицами при пропускании через VAE:



Что с ними не так? Самое очевидное - они все мутные. Почему так получается? В целом лица норм отобразились, а вот фон очень сильно размылся. При этом качество генерации не то что мы хотели бы получить. Что пошло не так?

Как мы уже говорили - латентное пространство имеет размерность меньше, чем исходное. Представим, что у нас есть какая-то функция, которая говорит о x и z - это наше латентное пространство:



Мы уже говорили, что когда мы обучаем VAE - мы не берём информацию о целевой переменной. Тогда бея объект, VAE никак не учитывает информацию о нём (например - эти две точки могли быть '1' и '7' на картинке, они похожи, но для них есть лейблы). VAE нам выдаст точку где-то посередине между этими двух покрашенных для минимизации MSE. Это можно сравнить с тем, что если 0 это орёл, а 1 это решка, модель нам скажет, что выпадет 0.5, потому что она минимизирует MSE.

В данном случае у нас получилось, что VAE хорошо закодировало лицо - мы можем различить объекты между собой. Но при этом VAE плохо закодировал фон, и во время декодирования декодер пытается предсказать средний фон. Из-за этого и происходит смазанность.

18.7. Идея GAN

Давайте делать по другому. Мы не будем учить encoder в пространство латентное, мы избавимся от этой части и будем обучать только decoder и отвечать на запросы вида "хорошая получилась генерация?". Это можно ещё обговорить как "Картинка получилась похожей на то, что находится в датасете".

18.7.1. Loss функция

$$L = \mathbb{E}_x[\log(D(x))] + \mathbb{E}_z[\log(1 - D(G(z)))]$$

где:

- $\mathbb{E}_x[\log(D(x))]$ - \mathbb{P} (картинка хорошая)
- $\mathbb{E}_z[\log(1 - D(G(z)))]$ - $1 - \mathbb{P}$ (картинка хорошая)

То есть генератор пытается показать, что картинка хорошая, а дискриминатор сказать, что она плохая. Получается две нейросети пытаются спорить между собой. Поэтому это и называется соревновательными нейросетями. На практике по батчу мы берём такую функцию:

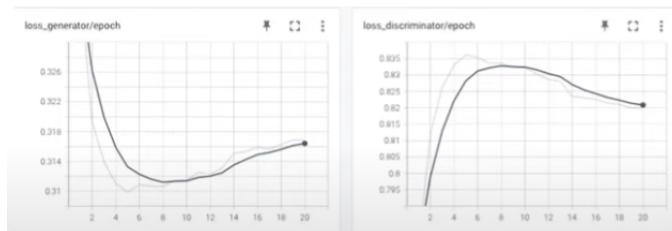
$$L = \frac{1}{B} \sum_{i \in B} [\log(D(x_i)) + [\log(1 - D(G(z_i)))]]$$

Давайте рассмотрим одну итерацию нашего алгоритма:

- Семплируем из латентного пространства - получаем выходное изображение
- Говорим что то, что сгенерировано генератором - у этого лейбл 0
- Считаем ВСЁ и обновляем веса дискриминатора (генератора не трогаем)
- Генерируем новое изображение и инвертируем его метку в лейбл НЕсгенерированного изображения
- Считаем ВСЁ и обновляем веса генератора (дискриминатор не трогаем)

Таким образом, эта гонка заставляет генератор делать такие картинки, которые дискриминатор пропустит.

При этом нужно отдельно поговорить про loss. Сначала генератор очень глупый и не знает что нравится дискриминатору. Но затем loss дискриминатора начинает расти, потому что генератор понимает всё лучше и лучше какие картинки генерировать.



Но при этом мы теперь по loss не можем понять обучилась наша модель или нет - нам нужны новые метрики.

18.8. Как мерить качество GAN модели?

- Глазами - ну очевидно, что человек может ± распознать норм перед ним картинка или нет
- Frechet Inception Distance:

$$FID(x, g) = \|\mu_x - \mu_g\|_2^2 + Tr \left(\sum_x + \sum_g -2 \left(\sum_x \sum_g \right)^{\frac{1}{2}} \right)$$

- Leave-one-out 1-NN - суть в том, что нас не волнует как они распределены, а мы берём случайное изображение с KNN neig=1. Если классификатор может отличить изображения] - плохо. Если не может - значит мы хорошо генерируем.

18.9. Немного трюков

Помимо того, что мы делали лейблы 0 и 1 для сгенерированных и истинных данных, мы добавим некоторый шум в эти лейблы.

дотехать

Часть III

Вторая часть курса продвинутого потока

19. Введение в NLP

19.1. Что такое Natural Language Processing?

Это система лингвистики и ML. Если проводить параллели с CV - то нам нужно понимать как работать с картинками, какие операции с ними возможны, какая обработка иногда нужна. Точно также нужно понимать как работать с текстами. Поэтому необходимо понимать что делать с текстами.

Если говорить про структуру - сначала идёт анализ текста, потом генерация текста:



В понимании текста нам важна морфология, синтаксис, семантика и прагматика. Для чего нам это нужно? Для начала - текст это слова. Понимая их форму и связь мы можем понимать взаимодействие между ними. После понимания взаимодействия мы можем перейти к пониманию значения некоторых текстов.

Если мы понимаем в качестве чего, почему и для какой задачи мы используем используя прагматику - мы уже используем смысл.

Исходя из этого мы уже можем генерировать текст, но по разному - исходя из наших целей. На вход нам подаётся текст, мы его как-то обрабатываем, опираясь на подчасти языка, и после этого мы выделяем некоторые фрагменты в тексте. Выделяем мы это для того, чтобы выделить важное. Выделив важное мы выделяем мысль в тексте.

Вот у нас есть текст и нам нужно произвести с ним много каких этапов. Для понимания текста нам нужно произвести некоторые шаги. Что нам нужно будет сделать?

- Токенизация
- Нормализация слова
 - Стемминг
 - Лемматизация
- Удаление слов
 - Стоп-слова
 - Неинформационные слова, шаблоны