IAD_DEEP_LEARNING

Вопросы к контрольной работе
Современные методы машинного обучения
Майнор “Интеллектуальный анализ данных” НИУ ВШЭ

Люди с политоса

Осень 2020 – Зима 2021

Оглавление

Глава 1

Ответы на вопросы

1.1 Вопросы

1.1.1 Что такое полносвязный слой? Сколько у него может быть входов и выходов? Запишите формулу для выхода полносвязного слоя

Полносвязный слой (fully connected, FC) – линейная функция, которую мы применяем над входящими объектах. Цель слоя – классификация. На входе n чисел, на выходе m чисел.
$x_1 ... x_n$ - входы
$z_1 ... z_n$ - выходы
Соответственно, m линейных моделей (выходов), в каждой (n + 1) параметров
Мы сами решаем, сколько нейронов выходов должно быть. Например: 1000 признаков, надо 2 прогноза (eg рост и вес) = 1000 на вход, 2 на выход. Следующий слой (если он есть) тогда примет на вход два нейрона. Формула выхода: $z_j = \sum_{i=1}^{n}(w_{ij}x_i + b_j)$

1.1.2 Зачем в нейронных сетях нужны нелинейности? Приведите пример какой-нибудь нелинейной функции

По сути если мы просто накручиваем кучу линейных слоев, на выходе мы получим комбинацию линейных признаков. То есть просто набор линейных слоев ничем не лучше одного линейного слоя, с добавлением слоев мы будем получать линейную модель над линейными моделями. А мы хотели бы, чтобы каждый новый слой как-то улучшал модель + чтобы мы не ограничивались только линейной вщаимосвязью. Поэтому после каждого нелинейного слоя нужно применять нелинейность. Варианты нелинейностей:

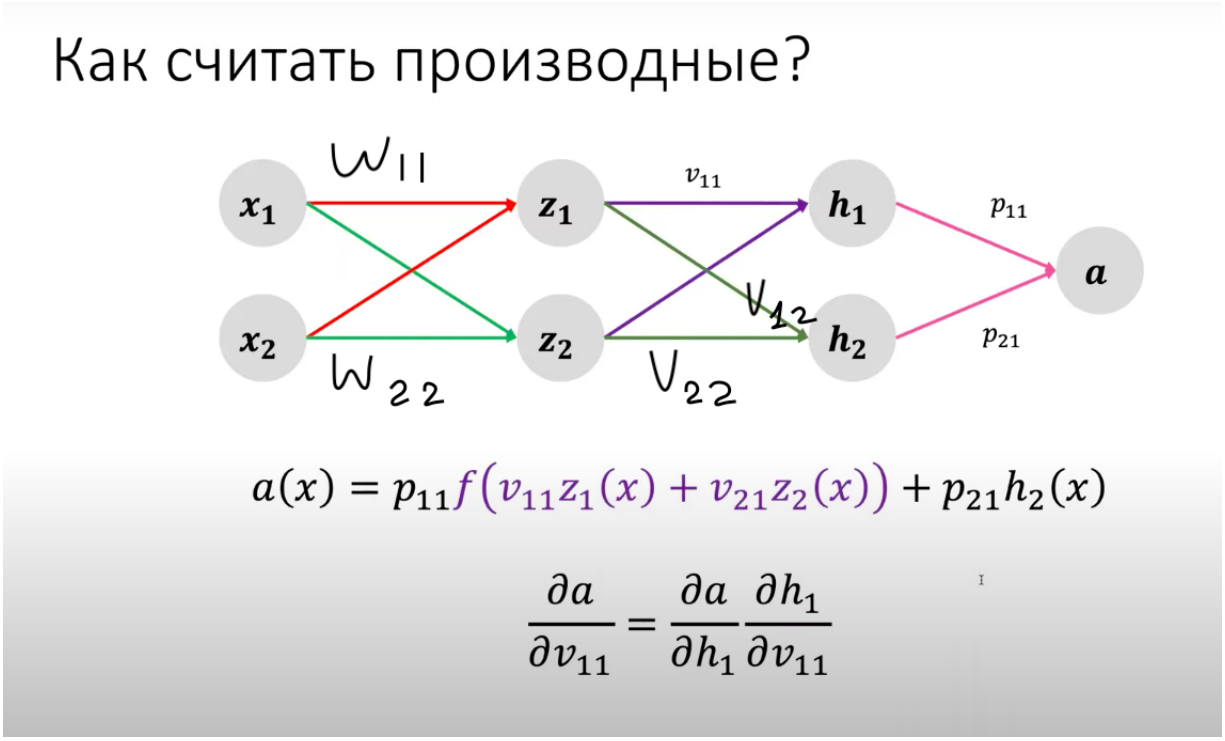
- Сигмоида – $f(x) = \frac{1}{1+exp(-x)}$
- ReLU – $f(x) = max(0, x)$. У ReLU есть модификаци типа Leaky ReLU ($f(x) = max(0, x) + \alpha * min(0,x)$)

1.1.3 Сколько нужно полносвязных слоёв в нейронной сети, чтобы она могла восстанавливать более сложные закономерности, чем линейная модель?

Когда мы говорим о полносвязной нейронной сети, то в них количество полносвязных слоев никак не улучшает модель (см. предыдущий вопрос). При этом, если в 2-ух слойной сети есть **хотя бы одна** нелинейность после полносвязного слоя, то мы уже получим достаточно высокую точность (Теорема Цыбулько: если взять любую непрерывную функцию, то можно подобрать такую 2-ух слойную НС, которая ее восстановит (приблизит) с заданной точностью). Нейронки поэтому очень мощные.

1.1.4 Типовая задача: дана нейронная сеть и указан конкретный параметр w (вес, соответствующий конкретному ребру). Показать, от каких других параметров будет зависеть производная функционала ошибки по w.

По идее надо посчитать производную выхода модели по этому весу, потому что она показывает, как меняется выход модели, есть мы как-то этот вес двигаем. Но если еще короче то от тех параметров, которые находятся на пути этого веса к выходу модели. Например:



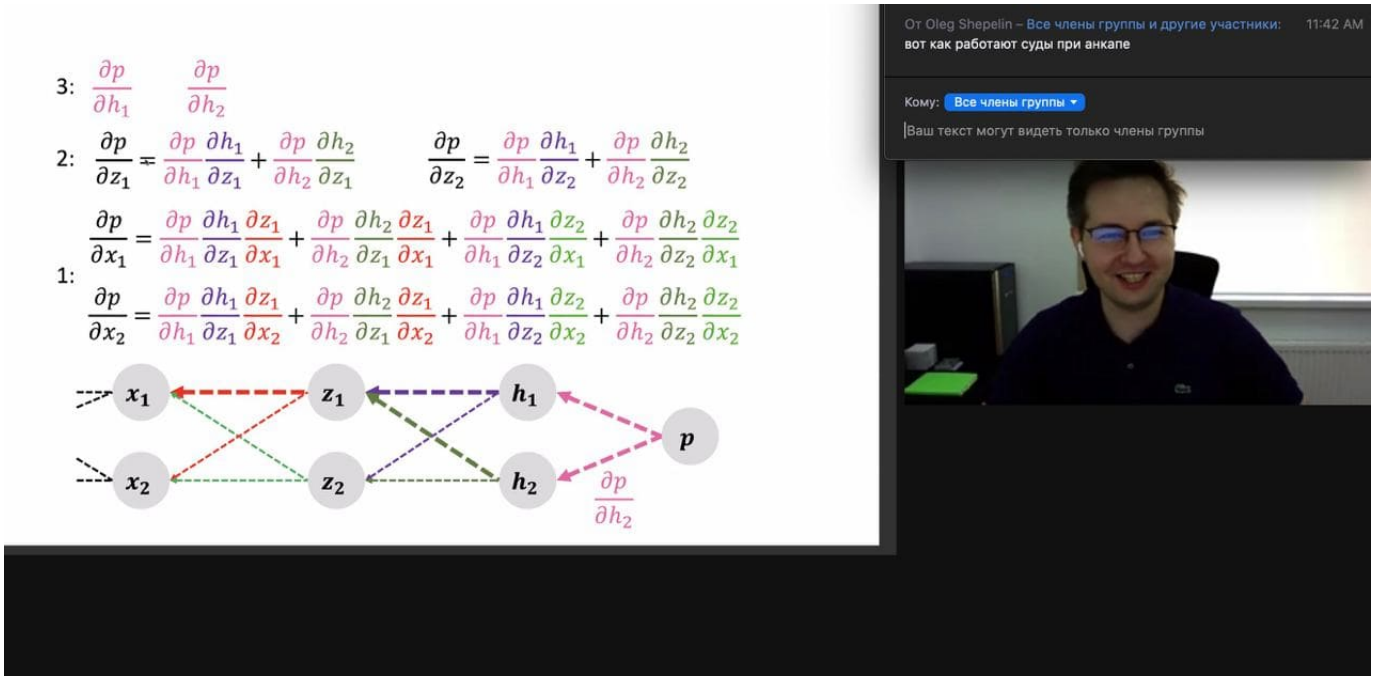
Здесь:

- производная ошибки по p_{11} зависит от h_1
- производная по v_{11} – от того, на сколько h_1 влияет на a и на сколько h_1 влияет на v_{11}
- по w_{11} – от влияния h_1 на a, h_1 на v_{11}, z_1 на w_{11}, h_2 на v_{12}, еще от p_{11}, p_{21}

и т. д.

В общем, правило такое: надо найти все пути из того, по чему дифференцируем к тому, что дифференцируем (в третьем, например, все пути от w_{11} до a). И на этих путях все ребра и параметры будут влиять на ее результат. (Чтобы точно посмотреть, надо перемножить все частные производные на этих двух путях и сложить пути). (На картинке p_{11}, конечно, тоже влияет на производную a по v_{11}, но в формуле не прописывается, потому что он как бы внутри производной a по h_1).

1.1.5 В чём идея метода обратного распространения ошибки? Для чего он нужен?



Метод обратного распространения ошибки (backpropagation) — метод вычисления градиента, который используется при обновлении весов при наличии нескольких слоев.
Смысл: Во многие формулы входят одни и те же производные (например, на скрине выше - чтобы посчитать производную p по z_1 нужно посчитать 4 производные). В backprop каждая частная производная вычисляется один раз: мы как бы идём в обратную сторону по графу и считаем производные. Опять же на примере – производную $\frac{\delta p}{\delta h_1}$ мы посчиали на втором шаге, когда искали производную p по z_1. Эта же производная нам пригодится, когда мы будем считать p по z_2, p по x_2 и x_1. То есть вместо того, чтобы 4 раза считать одно и то же, мы посчиаем это значение 1 раз и будем использовать дальше, так как мы один раз уже прошли по этому конкретному ребру. Альтернативно мы могли бы считать градиенты по каждому слою, но тогда бы мы считали градиент для первого слоя и на 1-ом, и на 2-ом, и дальше до результирующего, так как выходы 1-ого слоя влияют напоследующие слои. Бэкпроп нас от этого спасает. + Сразу напрямую учитываем, как изменения на первом уровне результируются на выходе. Например, можем заметить, что чем больше $z_1(x)$, тем сильнее h_1 влияет на a.
Саммари: Обратное распространение ошибки заключается в том, чтобы последовательно менять веса нейронной сети, начиная с весов выходного нейрона. Значения весов будут меняться в сторону уменьшения ошибки. Используем для того, чтобы понять как сильно зависит функция от переменной (как итоговый a зависит от w - весами между слоями), по сути считаем градиент с конца.

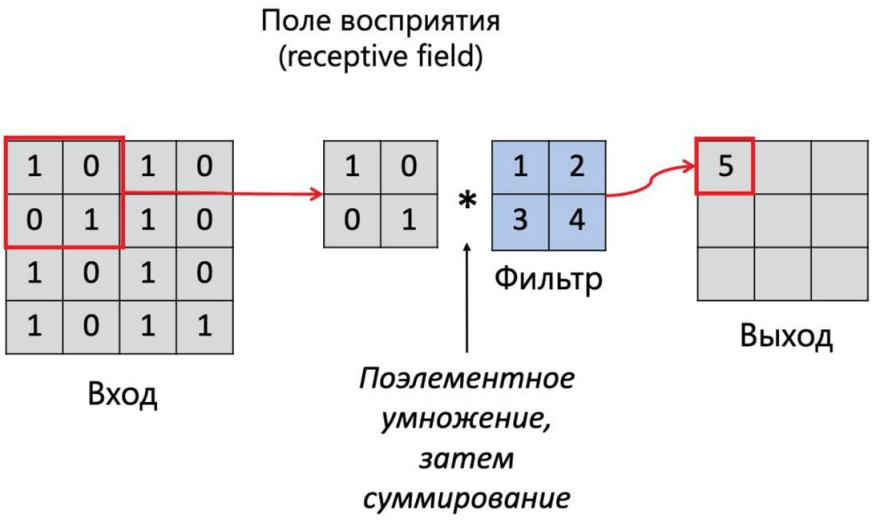
1.1.6 Почему применять полносвязные нейронные сети для работы с изображениями — плохая идея?

1. Очень много входных параметров, так как мы берем каждый нейрон на входе как отдельный параметр со своим весом. Наппример, в картинке 28x28 784 нейрона, если на первом полносвязном слое у нас 1000 нейронов, то всего мы получим (784 + 1)*1000 весов, то есть 785 000.
2. Очень легко могут переобучиться и подогнаться под выборку – по сути ПС не может научиться считывать какие-то паттерны и распознавать их на новых картинках, такая сеть

скорее будет запоминать значение пикселей в тренировочной выборке, а на тесте не сможет адекватно предсказывать. А лучше всего борется с переобучением как раз снижение параметров.
3. Не учитывает специфику изображений. Например, если у нас есть какой-то паттерн - кружок, и он на одной картинке внизу, а на другой – сверху, то ПС-сеть это сочтет вообще разными объектами. Или, например, если мы отзеркалим фотку собаки, для ПС-сети это будет совсем другая картинка, даже если она ей уже попадалась в тренировочном датасете.

1.1.7 Как устроена операция свёртки? Что такое фильтр? Какой смысл несёт комбинация свёртки и взятия максимального отклика?

Операция свёртки выявляет наличие на изображении паттерна, который задаётся фильтром. Чем сильнее на участке изображения выражен паттерн, тем больше будет значение свёртки. Результат свёртки изображения с фильтром — новое изображение. При чем, на последних сверточных слоях изображения со сверток становятся более осмысленными – например, мы можем на них видеть морды собак, лица людей и так далее. На первых шагах свертки скорее схватывают что-то более абстрактное, типа изгибов, поворотов, кругов, перепадов света и тени.
Процесс: на вход поступает картинка, по ней мы проходимя сверткой, помещая ее на разные участки картинки. Каждый пиксель картинки у нас представлен каким-то чиловым значением. Далее мы поэлементно умножаем пиксели данного участка с весами из фильтра, затем суммиуем, и получаем новое изображение, где каждый пиксель – отклик от фильтра. Если есть кусочек, похожий на фильтр, покажет максимальное значение. NB! свертка инвариантна к сдвигам



Фильтр задает паттерн, который надо найти на изначальном изображении (например, мордочка кошечки, которую мы будем прикладывать к пикселям и искать максимальное совпадение).
В чем смысл свертки + взятия макисмума: если мы прошлись каким-то фильтром по картинке, и где-то получили максимальное значение, значит, именно в этом курсе изображения наш паттерн. Тогда в каком бы месте картинки этот паттерн ни находился, максимум мы получим тот же. Отсюда инвариантность к смещению. На разных картирнках с этим паттерном мы также будем получать такие же или примерно похожие значения максимумов после свертки.

1.1.8 Что свёрточный слой принимает на вход и что даёт на выходе? Сколько у него входов и выходов? Запишите формулу для свёрточного слоя

Сверточный слой принимает на вход изображение – на 1-ом слое это исходное изображение, дальше свертки принимают на вход результаты предыдущих сверток. Соотвественно, число входов – это размер входного изображения (например, если изображение одноканальное 28x28, то на вход пойдет 784 пикселя), число выходов мы задаем сами, то есть мы сами решаем, сколько фильтров мы хотим применить к данному изображению, увеличить число каналов или уменьшить и т.д.
Формула сверточного слоя –

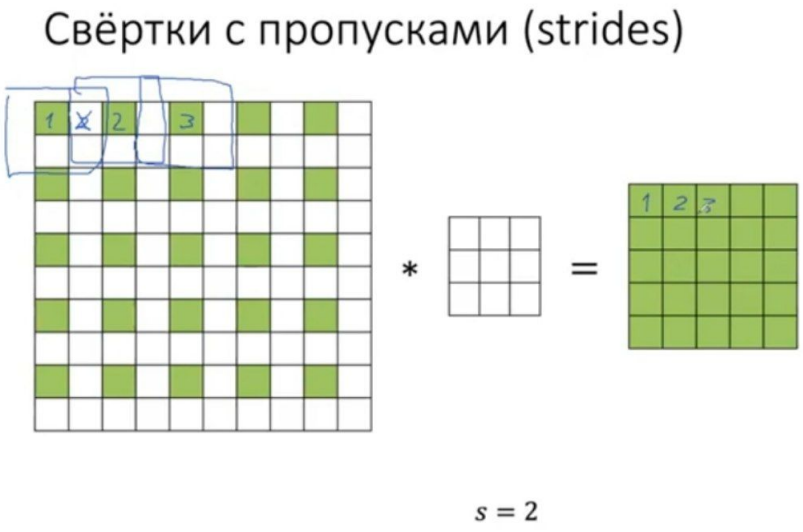
$$Im^{out}(x,y,t) = \sum_{i=-d}^{d} \sum_{j=-d}^{d} \sum_{c=1}^{C} k_t(i,j,c) Im^{in}(x+i, y+j, c)$$

Пытаемся разобраться в формуле. $Im^{out}(x, y, t)$ значит, что мы берем какую-то точку выходного изображения, расположенную в канале t (всего T каналов у выхода). d – это размер свертки. Соответственно, переменные i и j ходят от одного конца свертки до другого вверх и вниз. $k_t(i, j)$ значит, что мы берем t-ый фильтр (на одном слое мы можем задать много сверток-фильтров сразу). Из этого фильтра мы берем вес, который находится на позиции i,j. Например, если у нас свертка 3x3, то мы начнем подсчет с точки i = -1 и j =-1, то есть с верхнего правого конца. Дальше передвинемся в точку -1,0, то есть это вес фильтра посередине и сверху. $Im^{in}(x + i, y + j, c)$ значит, что мы берем исходное изображение, у которого C каналов (число исходных каналов изображение и число каналов на выходе отличается как правило). Переходим на канал с этого изображения. Берем точку с теми же координатами x и y, что у нас будут в результирующем изображении. От этой точки мы также отклоняемся на i и j, чтобы перемножать поэлементно на значения фильтра. После этого берем такой же пиксель на той же позиции, но уже в следующем канале входного изображения, и точно так же перемножаем со значениями фильтра. Потом эти все перемножения суммируются и получается пиксель в выходном изображении на определенном его канале.
Содераждательно: фильтр ездит по сколько угодно n-мерной квартинке, и на выходе будет давать нам 2-мерное изображение. Но фильтров в одном сверточном слое как правило много, T штук. Тогда каждый t-ый фильтр на выходе даст нам свой 2-мерный канал.
Фильтр фиксируется на какой-то точке ихсодного изображения, и за счет смещения на i и j сворачивает эту область вокруг пикселя в одно число. Это число и будет стоять в пикселе выходного изображения с теми же координатами.
Сколько всего параметров в сверточном слое? (2d + 1)(2d + 1)* c * t. Например, в RGB-изображении со свертками 3x3 и 10 фильтраму будеи $3^2 310 = 270$ параметров, это очень сильно меньше, чем было бы в подобной полносвязной сети. Сможем обучиться даже на маленькой выборке в 1000 картинок.

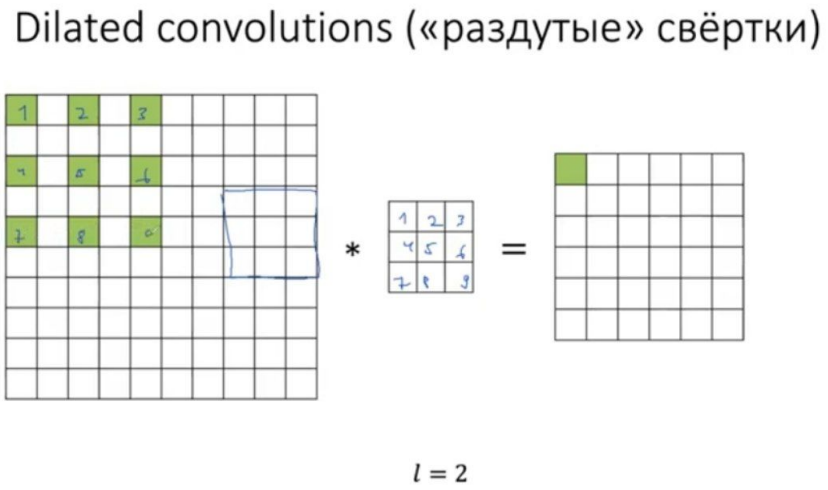
1.1.9 Что такое поле восприятия (receptive field)?

Поле восприятия (receptive field) – сколько пикселей из исходного изображения видит фильтр. Пример: картинка 9x9, а фильтр 3x3. В идеале мы бы хотели, чтобы на последних слоях фильтры видели крупные куски исходной картинки, так мы сможем найти более крупные сложные паттерны, которые маленькие фильтры могли пропустить. Например, маленький фильтр видит колесо/лобовое стекло/капот, а фильтр на последних слоях видит машину целиком. Как увеличить RF? Можно наслаивать свертки – но тогда будет медленно расти. Можно просто сделать сразу большой фильтр, но это очень много параметров, переобучимся. Тогда есть три способа быстро увеличить RF без СМС и регистрации.
1) рассматриваем картинку блоками (pooling) – исходную картинку делим на блоки по размеру фильтра и в каждом считаем максимум, значение каждого элмента куска заменяем

на максимальное значение. Потом дублирующиеся пиксели в куске можно удалить – тогда изображение сократится в n раз (где n размер фильтра) (pooling + subsampling);
2) свертки с пропусками (strides) – перескакиваем пиксели в изначальном изображении, то есть ставим свертку с центром не в каждом фильтре, а через n пикселей.



3) раздутые свертки (dilated convolutions) – фильтр раздуваем пропусками.



1.1.10 Типовая задача: дана нейронная сеть, состоящая из двух свёрточных слоёв. Для каждого слоя известен размер свёртки (например, 3x3) и пропуск (stride — например, 2). Посчитаейте размер поля восприятия для каждой позиции после применения двух свёрточных слоёв

Вот отсюда: The receptive field (RF) l_k of layer k is:

$$l_k = l_{k-1} + ((f_k - 1) * \prod_{i=1}^{k-1} s_i)$$

where l_{k-1} is the receptive field of layer $k-1$, f_k is the filter size (height or width, but assuming they are the same here), and s_i is the stride of layer i.

Если следовать этой формуле, то RF после второго слоя будет равен 7 (формула работает для слоя 2+).

```
00 01 02 03 04 05 06
10 11 12 13 14 15 16
20 21 22 23 24 25 26
30 31 32 33 34 35 36
40 41 42 43 44 45 46
50 51 52 53 54 55 56
60 61 62 63 64 65 66
```

Если более содержательно подходить к задаче: например, мы поместили нашу свертку в центр картинки (пиксель номер 33). Так как свертка имеет размер 3x3, она покроет пиксели 22, 23, 24, 32, 33, 34, 42, 43, 44. На первом сверточном слое RF будет 3x3.

Мы проходимя так сеткой по всему изображению. Поскольку stride равен 2, мы поместим нашу свертку в пиксели 11, 13, 15, 31, 33, 35, 51, 53, 55. То есть на выходе мы получим фильтр уже с 9 ячейками (ниже квадратами разного цвета показано, где окажется светка и какой участок картинки она видит):

А вот так будет выглядеть итоговый фильтр:

```
11 13 15
31 33 35
51 53 55
```

Далее мы проходимся сверткой по тому фильтру, который мы уже получили. Если мы снова поместим свертку посередине, в пиксель 33, то мы ей покроем все пиксели от 11 до 51. Если посмотреть на картинку выше, то пиксели от 11 до 55 после применения сверток видели всю картинку – она вся оказалась закрашена квадратами. Эта исходная картинка имеет размер 7x7. Соответственно, после второго сверточного слоя наш пиксель 33 будет иметь RF равный 7x7.

Пример взят отсюда.

1.1.11 Как устроен max-pooling слой? Для чего он нужен? Как он влияет на размер поля восприятия?

Max-pooling – это слой без обучаемых парметров, который призван сжать нашу картинку, сохранив в ней максимум ннформации. Для этого мы разбиваем ее на сегменты (например, берем куски 2x2), внутри каждого сегмента выбираем пиксель с максимальным значением, и оставляем только его. Картинка тогда сократится в n раз, где n – размер сегментов, который мы выборали. Содержательно это значит, что мы учитываем самый яркий отклик внутри куска, самые яркие пиксели. Тогда в целом мы сможем уловить паттерн, который есть на этом участке картинки, даже выкниув не особо информативные яркие пиксели.

Все это позволяет увеличить Receptive Field, так как в последующем слое каждый пиксель будет "отвечать"за целый сегмент. Например, при сегменте 2x2 один пиксель во втором слое будет видеть фргамент 2x2 исходного изображения, если же мы повторим pooling, то RF будет 4x4.

1.1.12 Для чего делают выравнивание (padding) в свёрточных слоях? Какие способы выравнивания вы знаете?

1. Применение операции свертки уменьшает изображение. Ввиду того, например, что могут быть проблемы со считываением пикселей по краям изображения – в них нельзя поставить центр свертки, эти пиксели тогда в выходное изображение не попадут. В связи с этим в сверточных слоях используется дополнение изображения padding). Выходы с предыдущего слоя дополняются пикселями так, чтобы после свертки сохранился размер исходного изображения.
Почему вообще уменьшение размера изображения плохо? 1. Для pooling хорошо бы четный размер изображения (т.к. чаще всего pooling 2x2, изображение сокращается в 2 раза. Хорошо бы тогда, чтоб у нас были четные высота и ширина). 2. valid mode: если паттерн стоит с краю, то нам очень сложно поставить сертку в этот край так, чтоб весь паттерн влез и мы его распознали. То есть пиксели на краях без паддинга не будут оказывать большого влияния на результат.

Способы выравнивания:
1) Zero-padding – бавим по границам нули так, чтобы посчитанная после этого свертка в valid mode давала изображение такого же размера, как и исходное (есть риск, что модель научится понимать, где края изображения, какие-то фильтры обучатся чисто на то, что обычно по краям картинок, а нам бы хотелось, чтоб фильтры были инварианты к положению паттерна на картинке).
2) Reflection padding – зеркальное отражение. Берем край картнки и зеркалим его в обратном направлении. Не получится находить края изображения, но теперь модель может начать находить зеркальные отражения и подбирать фильтры под них.
3) Replication padding – пиксель на границе равен ближайшему пикселю из изображния. Но тогда могут получиться какие-то константыне области. Модель может настроиться на паттерны из-за такого паддинга.

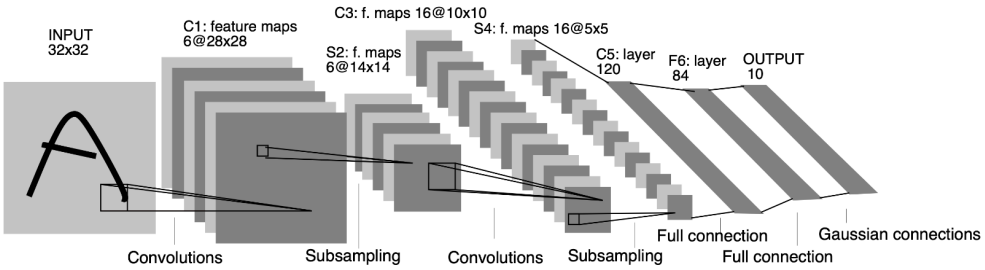
Короче из-за падинга в любом случае можем обучиться как-то не так на края.

Резюме:
- Паддинг позволяет контролировать размер выходных изображений
- Паддинг позволяет учитывать даже объекты на краях
- Разные типа паддингов допускают разные способы переобучения под края

1.1.13 Как обычно устроены свёрточные сети для задачи классификации (или регрессии) изображений? Как осуществляется переход от свёрточных слоёв к полносвязным?

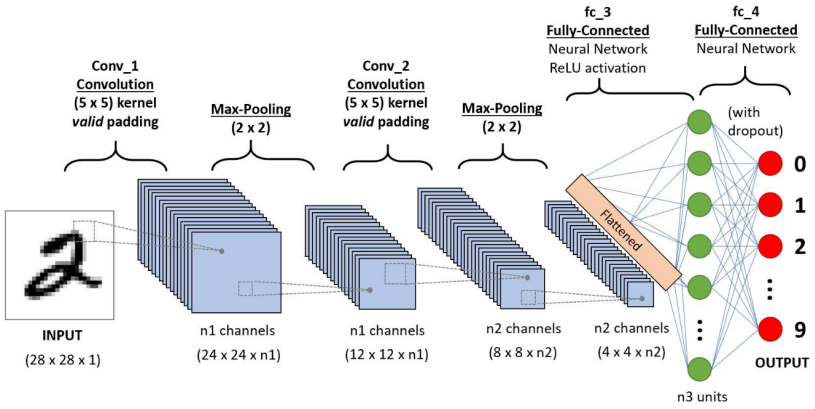
• Последовательноеи применение комбинаций вида «сверточной слой» → нелинейность → poolling или сверточной слой → нелинейность
• Выпрямление (flattering) выхода очередного слоя. По сути наши изображения – n-мерные матрицы, которые мы можем вытянуть в один тензор.
• Серия полносвязных слоев

LeNet (1998)



http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

Типичная архитектура



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

1.1.14 Как работают стохастический и mini-batch градиентный спуск?

Стохастический градиентный спуск мы проходили в прошлом году, надеюсь, базовую идею все помнят)))

Классический ГС:

$$w^t = w^{t-1} - \eta \nabla Q(w^{t-1})$$

Здесь $Q(w) = \frac{1}{l}\sum(< w,x > -y_i^2)$, то есть для подсчета градиента для одного шага мы будем считать сумму по всей выборке (а у нас тут огромные выборки...).

В классической версии останавливаемся если:

$$||w_t - w_{t-1} \leq \varepsilon||$$

Для НН у нас миллионы параметров – нам нужно посчитать разницу между этими миллионами объектов и сохранить параметры сразу на 2-ух шагах, то есть очень плохо. Так что в случае НН мы смотрим не на это, а когда ошибка на тестовой выборке перестает падать/начинает расти.

Стохастический ГС: На каждом шаге вместо вычисления градиента по всем объектам вычисляем градиент для случайного объекта.

$$w_t = w_{t-1} - \eta_t \nabla L(y_{i_t}, a(xi_t)$$

Плюсы: быстрее вычисления на каждом шаге – не надо суммировать что-то по всем объектам, лучше подходит для больших выборок и динамического обучения;
Минусы: хотя он быстрее на каждом шаге, сходится он может медленнее

Проблема со стохастическим градиентным спуском: мы хорошо идем в сторону минимума, но в конце нас начинает слишком сильно колбасить - слишком сильно шагаем, перескакиваем минимум, как это исправить?
Решение: Пусть у нас длина шага зависит от номера итерации, пусть она будет убывать

1. Начальное приближение: w^0
2. Повторять, каждый раз выбирая случайный объект i_t:

$$w^t = w^{t-1} - \eta_t \nabla L\left(y_{i_t}, a\left(x_{i_t}\right)\right)$$

3. Останавливаемся, если ошибка на тестовой выборке перестаёт убывать

Стохастический градиентный спуск ценен, потому что можно обрабатывать больший объем данных (в памяти хранится только один объект) → можем обучаться на огромной выборке

Замечания:
• Оценка по одному объекту несмещённая - то есть в среднем мы идём в правильную сторону.
• Точка оптимума – точка, где наименьшая ошибка по всей выборке в среднем. Но не на каждом объекте в отдельности! То есть в том минимуме, куда мы придем на SGD, градиенты все равно не будут нулевыми, какая-то ошибка останется.
• Поэтому важно, чтобы длина шага стремилась к нулю, чтоб мы не ушли от точки минимума по выборке.
• Сходимость к глобальному минимуму гарантируется только для выпуклых функций, а нейросети не такие

Обычно длину шага подбирают так, чтобы она не уменьшалась слишком быстро.

Mini-batch SGD:

Берем случайных m объектов, по ним считаем градиент ошибки и усредняем. Улучшаем не по одному объекту, а по m.

$$w^t = w^{t-1} - \eta_t \frac{1}{m} \sum \nabla L(y_{t,j}, a(x_{t,j}))$$

где $x_{t,j}$ - объект j из батча, сформированного на шаге t

Замечания:
• Размер пакета — обычно порядка десятков или сотни
• Имеет смысл брать степень двойки (память выделяется как степень двойки, более оптимально используем) - лучше всего размер от 2 до 32
• Возможно, делает оценку градиента более стабильной, но не факт
• Вычислительно почти так же эффективен, как шаг по градиенту одного объекта — за счёт векторизации

1.1.15 Для чего нужен метод инерции (momentum)? Как он работает? Запишите формулы

Метод инерции (momentum) позволяет решить проблему с градиентным спуском, поскольку могут быть проблемы со сходимостью: если линии уровня функции вытянутые, то очень сильно колбасит, если шаг маленький, то идем ооооооочень медленно.
Momentum – инициализируем нулем, потом записываем туда интерцию. По сути усредняем направление по всем предыдущим шагам. При изменении параметров на подстройку параметра в текущей эпохе оказывает влияние подстройки параметров в предыдущие эпохи. Как будто шарик, который катится в сторону минимума, очень тяжёлый, его сложнее сбить с этого спуска в какую-то иную сторону. Направление на новом шаге – среднее между предыдущими шагами градиентом в новой точке. α – то, на сколько мы хотим учитывать прошлые шаги.
$h_t = \alpha h_{t-1} + \eta_t \nabla Q(w^{t-1})$

$w^t = w^{t-1} - h_t$
α - параметр затухания (на следующем шаге двигаемся во столько раз медленнее)
h_t - инерция, усредненное направление движения

Nesterov Momentum (усложнение: использует идею "заглядывания вперёд используя производную не в текущей точке, а в следующей (если бы мы продолжали двигаться в этом же направлении без изменений)):
$h_t = \alpha h_{t-1} + \eta_t \nabla Q(w^{t-1} - \alpha h_{t-1})$
αh_{t-1} - куда попадем на следующем шаге

1.1.16 Для чего нужен метод AdaGrad? Как он работает? Запишите формулы

Adagrad – алгоритм, который подстраивают learning rate в зависимости от обучения, которое уже было осуществлено на предыдущих шагах для данного веса при параметра.

Например, у нас в данных может быть некая популярная категория, где мы часто встречали единицы и часто обновляли вес для такого one-hot параметра, а может быть не популярная, где единицы встречаются редко, и для которой мы редко обновляем веса. Тогда не логично для двух таких параметров подгонть веса с однаковым шагом. Некоторые веса уже близки

к своим локальным минимумам, тогда по этим координатам нужно двигаться медленнее, а другие веса ещё только в середине, значит их можно менять гораздо быстрее. Иначе обучим веса для параметров с разным качеством. Также у данных может быть разный масштаб – где-то лично шагать медленно по единицам, где-то можно шагнуть на 1000 сразу.

$$G_j^t = G_j^t + (\nabla Q(w^{t-1}))_j^2$$
$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{G_j^t + \varepsilon}}(\nabla Q(w^{t-1}))_j)$$

j – это номер параметра, по которому мы берем градиент. Эпсилон мы добавляем в дробь, чтобы не разделить на 0 при первых итерациях. Соотвественно, при обновлении веса мы умножаем на частную производную ошибки по w_j (берем градиент и достаем его j-ое число). Так как G_j^t стоит в знаменателе, если мы уже достаточно много шагали по этому параметру, learning rate будет низким и наоброт.
Но G_j^t постоянно растет, можем не успеть дойти до минимума.
RMSProp: $G_j^t = G_j^t + (1-\alpha)(\nabla Q(w^{t-1}))_j^2$. Умножаем на некую альфа $< 1 \rightarrow$ затухание предыдущих шагов, тогда скорость будет зависеть от того, как мы недавно шагали, а не вообще из всех шагов.
Adam – учитываем и инерцию, и то, сколько уже шагали по параметру.

1.1.17 Опишите, как работает метод Dropout: что он делает на этапе обучения нейронной сети и на этапе применения

На этапе обучения на каждом шаге градиентного спуска Dropout зануляет один или несколько нейронов. Единственный гиперпараметр этого слоя – p, то есть вероятность зануления.
Dropout слой генерирует n бинарных случайных величин (число нейронов прошлом слое), каждую с вероятностью p. Потом домножаем покоординатгно выходы прошлого слоя на эти 0 и 1.
Выглядит это как $d(x) = \frac{1}{p}m \times x$. Зачем делить на p? Потому что из-за зануления меняется число нейронов (например, 4 нейрона с единичными весами давали нам выход 4, а если один занулим он станет 3). При делении на p мы сохраняем масштаб. Такой вариант называется inverted dropuot.
На этапе применения $d(x) = x$, слой работает просто как единичный слой, через него проходят сигналы без изменения. В оригинале не было нормировки на этапе обучения, но было домножение на p в ходе применения. Вариант выше лучше, потому что здесь мы на применении ничего не делаем, и сеть работает быстрее.
Зачем это нужно? Боремся с переобучением. Требуем, чтобы даже выкидывая часть нейроов мы давали более или меее нормальное предсказание. Если сломать человеку руку он научится норм пользоваться другой, если сломать нейросети нейрон....

1.1.18 В чём заключается BatchNorm? Какую проблему он пытается решать?

Проблема:
В нейронной сети каждый слой обучается на выходах предыдущих слоёв
Если слой в начале сильно меняется (меняется его масштаб) то все следующие слои надо передылывать (internal covariance shift в нейронках)

Идея: преобразовывать выходы слоёв так, чтобы они гарантированно имели фиксированное распределение (не дает слоям выдавать разные масштабы прогнозов)

Batch normalization (нормализация по батчам):
• Реализуется как отдельный слой (после полносвязного слоя)

• Вычисляется для текущего батча. Считаем по выходам слоя среднее и стандартное отклонение. Далее машстабируем выходы - вычитаем среднее и делим на стандартное отклонение. И потом доставляем бета и умножаем на гамма – эти параметры обучаются на гриадиентном спуске. Тогда в итоге выходы ПС имеют среднее бетаj и стандартное отклонение гамма j.

По текущему батчу считаем:
Среднее: $\mu_b = \frac{1}{n}\sum_{j=1}^{n} x_{b,j}$
Дисперсию: $\sigma_b^2 = \frac{1}{n}\sum_{j=1}^{n}(x_{b,j} - \mu_b)^2$

Масштабируем выходы полносвязного слоя:

$\tilde{x}_{b,j} = \frac{(x_{b,j}-\mu_{b,j})}{\sqrt{\sigma_{b,j}^2+\epsilon}}$

Берем житый выход из него вычитаем его среднее и делим на стандартное отклонение. Модифицируем выходы житого нейрона так, чтобы было нулевое среднее и единичное станартное отклонение.

$z_{b,j} = \gamma_j \tilde{x}_{b,j} + \beta_j$
Умножаем маштабированный выход на некоторый коэффициент гамма и бета, это параметры, которые обучаются, т.е. градиентным спуском их нужно оптимизировать.
В итоге выходы этого слоя будут иметь среднее β_j и стандартное отклонение γ_j, независимо от того, что было до батчнорма. Мы имеем нужное нам среднее и нужное стандартное отклонение и подбираем их так, чтобы они были оптимальными.

На этапе применения мы делаем тоже самое, но в качестве среднего и дисперсии (μ_B и δ_B^2) используем показатели по всем батчам.

• Реализуется как отдельный слой
• Вычисляется для текущего батча
• Оценим среднее и дисперсию каждой компоненты входного вектора:

$$\mu_B = \frac{1}{n}\sum_{j=1}^{n} x_{B,j}$$

покоординатно

$$\sigma_B^2 = \frac{1}{n}\sum_{j=1}^{n}\left(x_{B,j} - \mu_B\right)^2$$

• Отмасштабируем все выходы:

$$\tilde{x}_{B,j} = \frac{x_{B,j} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

• Зададим нужные нам среднее и дисперсию:

$$z_{B,j} = \gamma \tilde{x}_{B,j} + \beta$$

обучаемые параметры

Batch Normalization

Во время применения нейронной сети:

- Те же самые формулы, но вместо μ_B и σ_B^2 используем их средние значения по всем батчам

Замечания:

• Позволяет увеличить длину шага в градиентном спуске, ускорять обучение

• Не факт, что действительно устраняет covariance shift, то есть сдвиг масштабов выходов. Некоторые слои все еще сильно меняют распределение выходов. Почему это открытый вопрос.

• Но батчнорм дочно повышает сходимость и дает более качественные результаты.

1.1.19 Для чего нужны аугментации данных? Приведите примеры того, как можно аугментировать изображения

1. Для предотвращения переобучения (во всех смыслах и понятиях). Например, если у нас попугаи всегда слева на картинке, НС просто запомнит, что нужно искать слева, и попугая справа не узнает.
2. Аугментация – это бесплатное расширение обучающей выборки

Примеры:

1. Рандомное изменение цвета изображения (RGBShift)
2. Рандомный контраст (RandomContrast)
3. Разные развороты изображения (любой)
4. Можно повырезать из картинки патчи, чтоб часть оказывалась скрыта
5. Блюр
6. Добавить шум – adversal attacks, которые не доступны глазу человека, но НС из-за этого вообще по-другому воспринимате картинку

1.1.20 В чём основные отличия между архитектурами AlexNet и VGG?

AlexNet (2012):

1. Блоьшие свертки – 11x11
2. 60 лямов параметров было – то есть очень много, мб из-за больших сверток с кучей параметров не так хорошо обучались

VGG (2014):

1. 5 вариантов НС, различаются глубиной слоев (от 11 до 19)
2. Только маленькие сверстки 3x3
3. Меньше параметров на одном слою из-за маленьких сверток, но зато куча слоев и каналов. В варианте E 144 ляма
4. Более сложна инициализация (сначала обучается вариант A со случайными начальными весами, потом этими обученными весами из A инициализируются веса в более глубоких сетках»

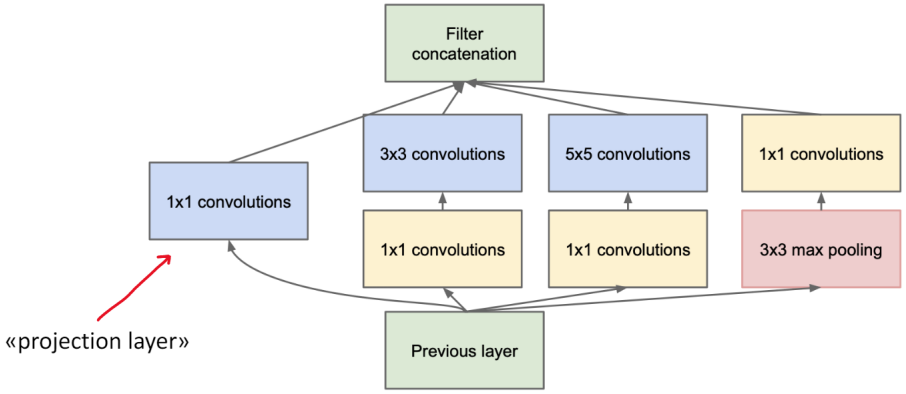
В обоих – использование Dropout, ReLU, аугментация, градиентный спуск с momentum.

1.1.21 Опишите идею Inception module в GoogLeNet

FUN FACT AND MEME:
«We will focus on an efficient deep neural network architecture for computer vision, code named Inception, which derives its name from (. . .) the famous “we need to go deeper” internet meme.»

Основная суть:
• Вообще это своеобразный слой - комбинация нескольких слоев (у них 1 на 1, 3 на 3, а также 5 на 5) с их объединением их один выходной вектор, которые образуют вход следующей стадии (что дальше идет в архитектуре)
• Сначала делается 1 на 1 сверстка перед нанесением другого слоя, который как раз используется для уменьшения размерности (как метод главных компонент – сохраняем всю информацию данных, но сжимаем их размергность с помощью линейного преобразования).
• Параллельный слой Max Pooling, который предоставляет еще одну возможность начальному слою
• To understand the importance of the inception layer’s structure, the author calls on the Hebbian principle from human learning. This says that “neurons that fire together, wire together”. The author suggests that when creating a subsequent layer in a deep learning model, one should pay attention to the learnings of the previous layer.
• Suppose, for example, a layer in our deep learning model has learned to focus on individual parts of a face. The next layer of the network would probably focus on the overall face in the image to identify the different objects present there. Now to actually do this, the layer should have the appropriate filter sizes to detect different objects.
• This is where the inception layer comes to the fore. It allows the internal layers to pick and choose which filter size will be relevant to learn the required information. So even if the size of the face in the image is different (as seen in the images below), the layer works accordingly to recognize the face. For the first image, it would probably take a higher filter size, while it’ll take a lower one for the second image.

GoogLeNet (2014)



(b) Inception module with dimensionality reduction

свёртки делаются с паддингом!

http://arxiv.org/abs/1409.4842

1.1.22 Что такое residual connections? Для чего они нужны?

Проблема:

Это необходимый элемент построения ResNet модели. Базовая идея состоит в следующем: в остальных типах нейронных сетей каждый следующий слой, пройдя всякие там пробразования (maxpool, relu и т.д) в итоге переходит только в следующий слой. При residual connections каждый выход слоя сохраняется и влияет не только на следующий слой, но и подается на вход другим слоям, следующем далее в сети.

Берем обычную сверточную сеть и добавляем больше слоев. По идее если мы говорим о тренировочной выборке добавление большего количества слоев (читай параметров) должно привести в конечном счете к переобучению модели, а значит и большему качеству на тестовой выборке. ЭТОГО ОДНАКО НЕ ПРОИСХОДИТ. Ошибка на тренировочной выборке выше. Что-то мешает им подогнаться под данные.

Объяснение:

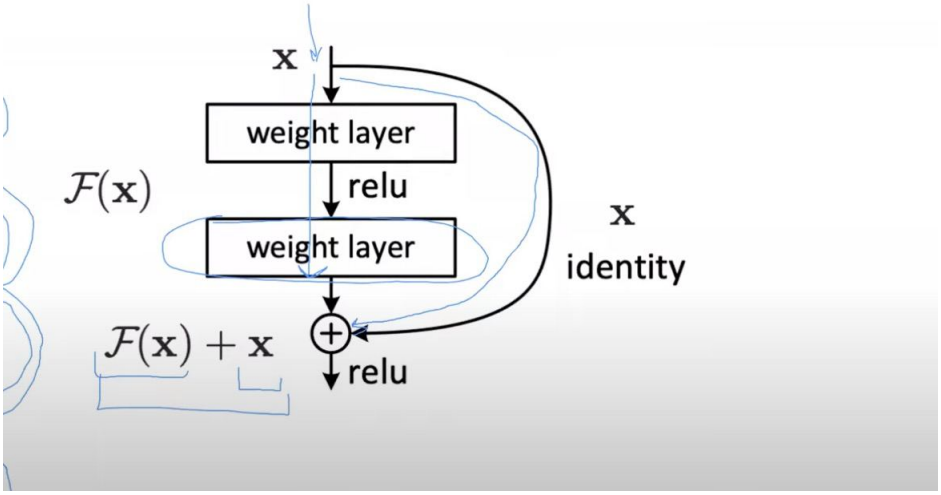
с лекции - при обратном распространении ошибки во время обучении градиент теряется.

Почему residual:

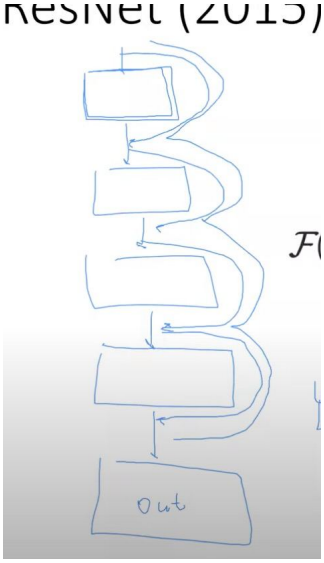
Потому что в такого рода архитектуре мы иначе определяем задачу, которую решают сверточные слои. Предположим, что у нас есть настоящее распределение параметра x - H(x). В данной архитектуре мы на каждой слой подаем x, в то время как модель пытается выучить R(x)= H(x) -x, то есть отклонение от обучающей выборки

Решение:

skip connections. Давайте мои сигналы (значения на предыдущем слое) будут передаваться не только на следующие слои, но и мимо них. Это значит, что выход слоя у меня будет записываться как выход предыдущего слоя + выход предпредыдущего слоя. То есть мой сигнал проходит одновременно через какие-то преобразования (сверточные сети), но также и идет в обход и попадает тоже в наш слой.

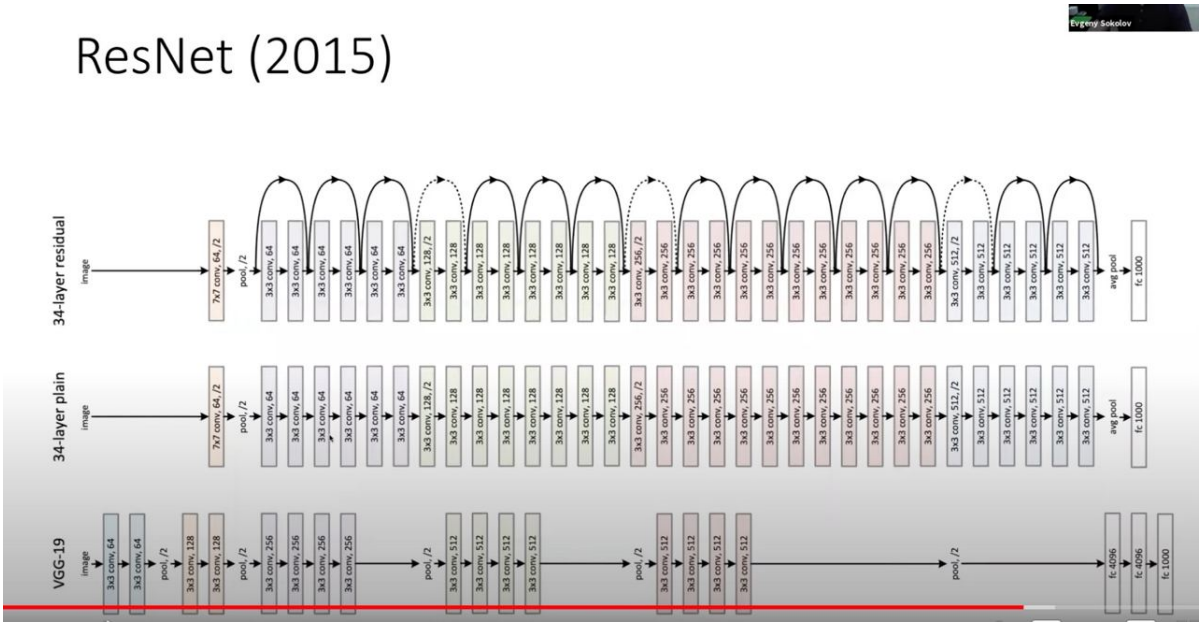


Это означает, что входной сигнал, который я подал сети в самом начале обучения, проходя через skip connections, и может в принципе никак не преобразовываться. Тогда наши сверточные слои учат не основной сигнал, но только добавки. В то время как основный сигнал это то, что мы первоначально мы подали на вход модели.



Давайте я позволю своему сигналу проходить неизмененным мимо слоев.

Пример построения модели с skip connections



В самом низу у нас VGG с 19 слоями. Хотим сделать глубже, сначала увеличиваем количество слоев. Но мы уже знаем, что такая модель будет иметь худшее качество нежели модель с 16 слоями на трейне. Мы переходим к третьей картинке, где мы гоняем первоначальный сигнал через дуги. Модель начинает лучше обучаться, остается проблема переобучения, но проблему недообучения решается с помощью skip connections.

1.1.23 В чём заключается идея переноса знаний (transfer learning) между нейронными сетями? Как можно построить свёрточную нейросеть для решения задачи с небольшой обучающей выборкой, если есть уже обученная свёрточная сеть для похожей задачи?

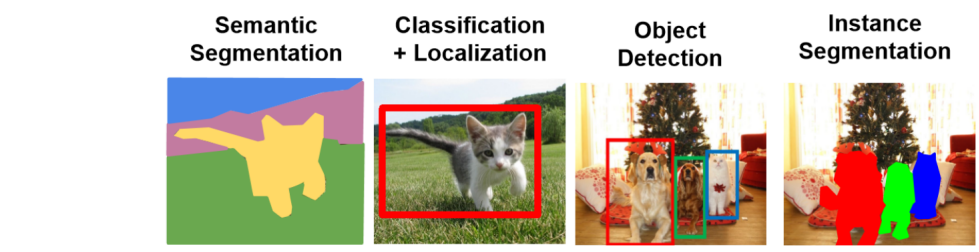
Суть transfer learning:
Благодаря ImageNet мы смогли тестировать разные алгоритмы, а собирать новую такую выборку (а тем более для каждой задачи) - можно просто повеситься (как по силам, так и по деньгам). Поэтому мы можем на самом деле уже использовать хайповые алгоритмы для совсем других задач и классификаций (однако чем более непохожая задача, тем больше слоев нужно переобучать)

Как использовать в итоге для новой задачи с небольшой выборкой?
Мы берем просто модель из другой задачи (даже популярные сетки с их особенностями, конечно), а затем заменяем последний на слой с нужным числом выходов и обучаем только его (это как обучение линейной модели).

Если данных много, то обучаем уже несколько слоев до выхода, лучше с LR пониже.

1.1.24 В чём заключается задача семантической сегментации изображения? Что является объектом и ответом в этой задаче?

Задача: правильно определить локализацию объектов на изображении (границы его). Нам не просто интересно, есть ли на картинке кот, нам нужно понять, где он именно на картинке. **Данные:** изображения и их корректная сегментация. **Объекты** – пиксели изображения, **ответы** – класс каждого пикселя.



1.1.25 Как в задаче семантической сегментации вычисляется качество решения? Опишите, как вычисляется мера Жаккара для каждого класса

Метрики качества

- Попиксельная доля верных ответов:

$$L(y,a) = \frac{1}{n}\sum_{i=1}^{n}[y_i = a_i]$$

- Мера Жаккара (считается отдельно для каждого класса):

$$J_k(y,a) = \frac{\sum_{i=1}^{n}[y_i = k][a_i = k]}{\sum_{i=1}^{n}\max([y_i = k],[a_i = k])}$$

(можно усреднить по всем классам)

1.1.26 Опишите, как устроена категориальная кросс-энтропия, которая применяется для обучения моделей семантической сегментации

Функция потерь

- Для одного изображения:

$$L(y,a) = \sum_{i=1}^{n}\sum_{k=1}^{K}[y_i = k]\log a_{ik}$$

Функция потерь

- Для одного изображения (categorical cross-entropy, CCE):

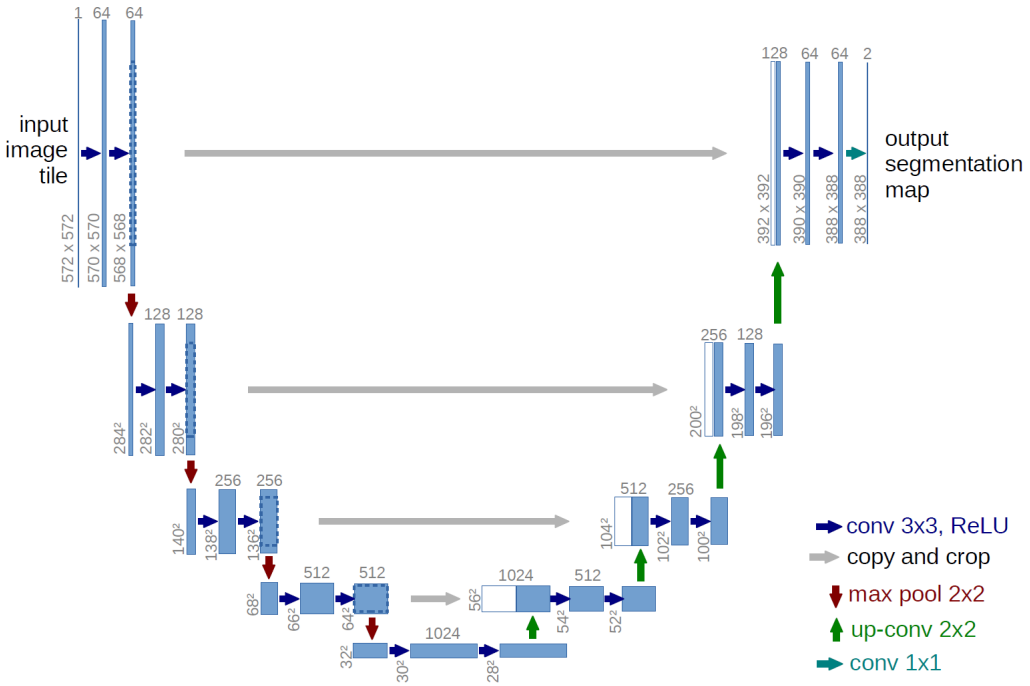
$$L(y,a) = \sum_{i=1}^{n}\sum_{k=1}^{K}[y_i = k]\log a_{ik}$$

- Если модель в i-м пикселе выдаёт какие-то числа $b_{i1}, \ldots, b_{iK}$, то их можно превратить в вероятности через softmax:

$$a_{ik} = \frac{\exp(b_{ik})}{\sum_{m=1}^{K}\exp(b_{im})}$$

Обучаем, по сути, на версию лог-лосса для многоклассовой классификации: суммируем по всем пикселям и по всем классам индикаторы того, что пиксель относится к какому-то классу и умножаем на вероятность того, что пиксель относится к этому классу. Это все максимизируем. Требуем, чтобы модель выдавала как можно большую вероятность для правильного класса этого пикселя.

1.1.27 Опишите архитектуру U-Net (как она в общем выглядит и какое у неё главное отличие от fully convolutional архитектур)



Ну если вкратце, то сначала это просто полносвязная сверточная сеть в левой части буквы U, где мы удваиваем количество каналов признаков три раза, а потом в правой части мы начинаем уменьшать количество каналов. При этом в правой части, когда мы расширяем карту признаков, мы также добавляем напрямую признаки из левой части буквы U. То есть мы не просто останавливаемся где-то внизу, как при обычной задаче классификации, а разворачиваем это все.

1.1.28 В чём заключается задача детекции объектов? Что является объектом и ответом? Какие способы измерения качества решения вы знаете?

Сначала мы находим прямоугольник, содержащий в себе объект, потом мы классифицируем то что оказалось внутри прямоугольника.

Задача детекции объектов – определить, в каком прямоугольнике находится искомый объект класса k (машины).

Модель выдаеет для класса k список прямоугольников с уверенностями (уверенность P в том, что в данном прямоугольнике на картинке, во-первых, есть объект, во-вторых, этот объект принадлежит классу k). Принцип Maximum Suppression состоит в том, что мы сначала сортируем прямоугольники по уверенности, а потом удаляем все прямоугольники, которые сильно пересекаются с нашим.

Объект: изображение с объектами

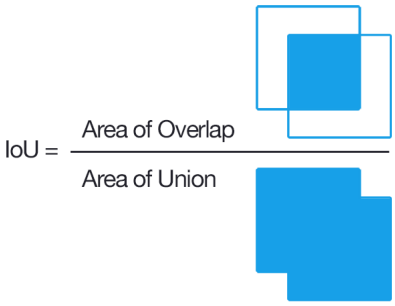
Ответ: список прямоугольников в порядке убывания уверенности

Порог t – порог, после которого мы считаем прямоугольнки корректным (т.е. в каком случае мы засчитываем прямоугольник, пересекающийся с нашим, в качестве верного).

Метрики качества

Если говорить про качество есть метрика IoU – Intersection over Unioun, которая считается по простой формуле:

$$IoU = \frac{overlap}{union}$$



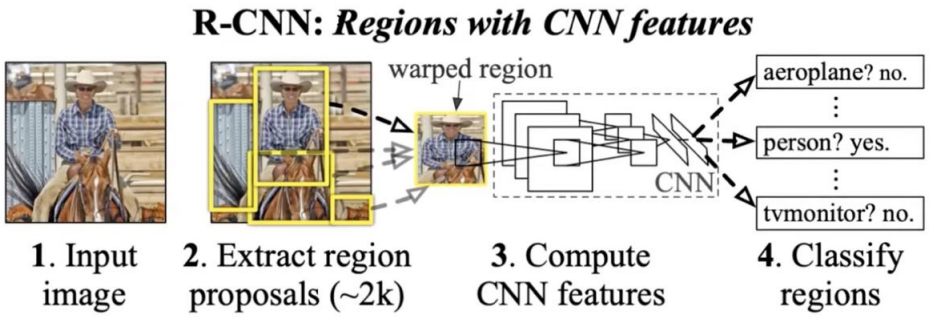
Дальше выдаются метки TP, FP, FN:

1. IOU $>$ 0.5: TP – класс определен верно;
2. $0.5 \geq IOU$: FP
3. $IOU > 0.5$: FN это если мы неправильно определели класс

1.1.29 Опишите идею модели R-CNN для детекции объектов

Это модель в класс моделей двухшаговой детекции объектов (1 шаг - найти много кандидатов: прямоугольников, в которых есть объекты, 2 шаг - классифицировать объекты в этих прямоугольниках)

Как устроена R-CNN:



Мы делим картинку на прямоугольники, в которых предположительно есть какие-то объекты. (Это делается из статей дедов, в которых деды придумали, как накинуть на картинку прямоугольники, в которых что-то есть). Этих прямоугольников примерно 2000 -region proposals (или кандидаты). Я беру каждый прямоугольник (кандидата) и использую на нем AlexNet (сверточная нейронная сеть).

Сеть дает мне вероятность того, что один из 1000 классов ImageNet, мне это не нужно, поэтому я беру выход предпоследнего слоя и, используя линейный классификатор (лог регрессия или svm) учу модель предсказывать вероятность мне нужных классов (тех, которые мы пытаемся детектировать на картинке)

Чем плох подход:
будет плохая скорость - 3 блока в задаче, которые учатся независимо.
ЕЩЕ:

R-CNN

Проблемы:
- Не end-to-end
- Генерация кандидатов может быть очень сложной
- Свёрточная сеть практически не настраивается под данные
- Долго (много признаков, много классификаторов)

1.1.30 В чём идея модели Fast R-CNN? Что она предсказывает?

- Прямоугольники все еще генерируются независимо, но этапы обучения нейросети (выделения признаков) и классификации объединяются.

- Сначала мы применяем к правильному ответу механизм генерации кандидатов и генерируем новый прямоугольник-"кандидат".

- Из последнего сверточного слоя сети вырезаем подобласть тензора, которая соотвествует прямоугольнику-"кандидату".

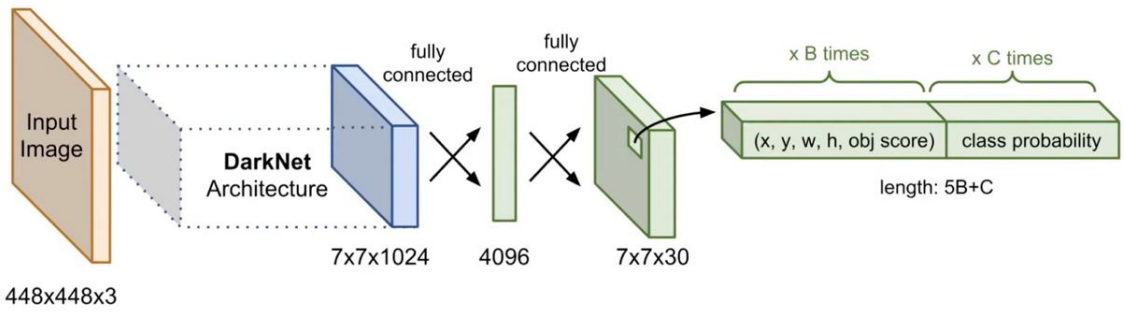
- Потом двумя FC предсказываем две вещи:
1) вектор с предсказанными классами (все объекты, которые есть на картинке),
2) 4 числа, которые отвечают за то, на сколько нужно сдвинуть каждую сторону прямоугольника, чтобы его лучше подогнать. Это и есть обучение модели. То есть модель предсказывает корректировки.

- И классификаторы, и сверточную часть (которая обучает кандидатов) мы обучаем совместно на большой выборке. То есть loss состоит из ошибки классификатора + ошибки предсказаний корректировки, и именно эту сумму мы будем минимизировать. Поэтому это end-to-end модель.

То есть, по сути мы дообучаем всю сеть, но генерация кандидатов занимает много времени.

1.1.31 Как работает модель YOLO для одношаговой детекции объектов?

Структура сети:

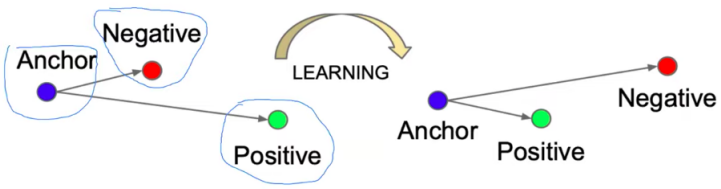


в чем идея одношаговой детекции: Разбиваю картинку на некоторое количество прямоугольников: их s*s штук.

Прогоняем это дело через сверточную архитектуру, в итоге получаем сверточный слой. Мы говорим, что в каждом прямоугольнике я могу детектировать всего B объектов (не больше,обычно 2). Для каждого прямоугольника я буду предсказывать некоторое количество параметров - координаты объекта, вероятность того, что в прямоугольнике есть объект, и вероятность того, что в этом прямоугольнике ПРИ УСЛОВИИ, что в этом прямоугольнике есть объект.

1.1.32 Запишите формулу для триплетной функции потерь. Объясните, почему она хорошо подходит для задачи идентификации

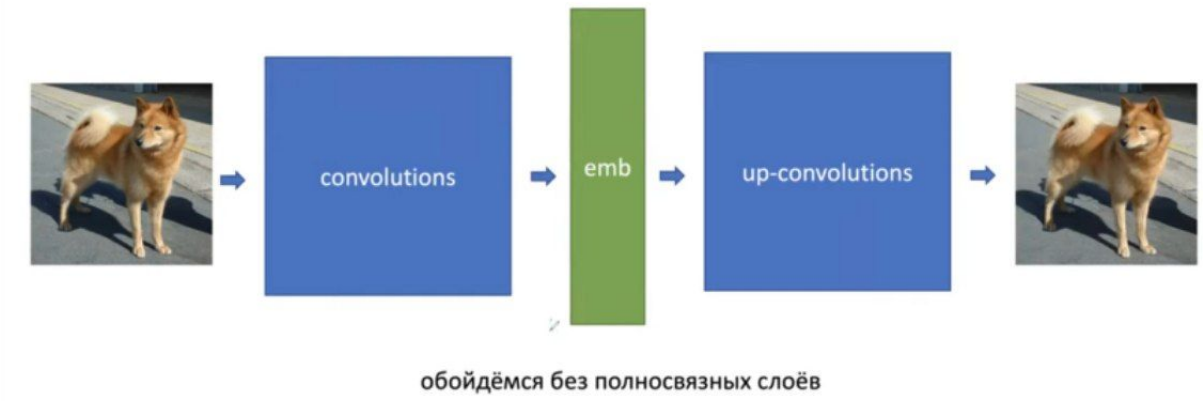
Мы выбираем некоторый батч с картинками и обучаем на триплетный лосс, который требует, чтобы в векторе, который мы сделали из этого батча, фотки одного человека оказались близко, а разных – далеко (ну и соответственно надо батчи собирать специфически). Прикол в том, что благодаря такому обучению, сеть начинает узнавать человека с разных ракурсов. Формула вот, ее минимизируем:



$$\sum_{i}^{N}\left[\|f(x_i^a)-f(x_i^p)\|_2^2-\|f(x_i^a)-f(x_i^n)\|_2^2+\alpha\right]_+$$

1.1.33 Как устроены автокодировщики? На какой функционал они обучаются?

Автокодировки нам нужны для обучения без учителя – когда у нас нет разметки данных с готовыми правильными ответами. Нам нужно, чтобы похожие изображения имели похожие представления (их вектора находились рядом). Это достигается, если мы из вектора картинки оказываемся в состоянии +- хорошо восстановить саму картинку.



Автокодировщики обучаются на следующий функционал:
$\frac{1}{l}\sum L(x_i, g(f(x_i))) \to \min$, где

L – это расстояние между изображениями (например, попиксельно измеряем евклидово расстояние),
g – это декодер,
f – это кодировщик
Есть также идея perceptual loss. Например, нам не особо хочется хорошо восстаналивать фон картинки, это не самое главное. Тогда можем требовать, чтоб у похожих картинок были схожи свертки на последних слоях, там, где нейроны уже находят осмысленные паттерны (например, морды собак)

1.1.34 В чём идея denoising autoencoders?

Обычные автокодировщики легко переобучаются. Идея этого автоинкодера в том, что мы добавляем случайный шум к нашим картинкам, кодируем их, раскодируем и требуем, чтобы они выдавали картинки уже без шума.

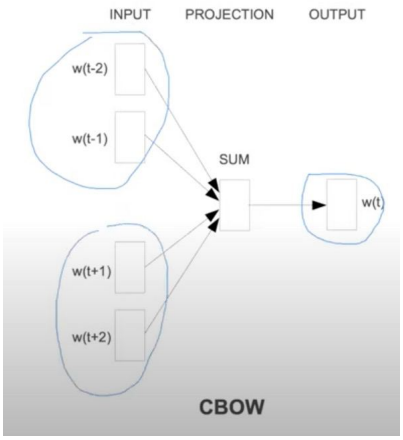
1.1.35 Опишите идею word2vec. Что требуется от обучаемых в ней представлений слов? Для чего нужен negative sampling?

Общая идея word2vec:

Обучим вектор числовой для каждого слова из словаря,заведенного для нашей обучаемой выборки. Это вектор может содержать любые числа и быть размерностью тоже любой, но одинаковой для всех чисел.

Требования к обучаемым представлениям слов:

1) Self supervision - возьмем кучу текстов, и обучим с помощью них такую модель, что если мы удалим часть слов из текста, модель могла их восстановить. То есть по векторам соседних слов можно узнать, какое слово было удаленov - Contionous bag of words



2) Обратная первому пункту идея. Хочу с помощью вектора среднего слова хорошо предсказывать векторные представления слов, следующих до него и после него - Skip-gram

Skip-gram model

- Вероятность встретить слово w_O рядом со словом w_I:

$$p(w_O|w_I) = \frac{\exp(\langle v'_{w_O}, v_{w_I} \rangle)}{\sum_{w \in W} \exp(\langle v'_w, v_{w_I} \rangle)}$$

- W — словарь
- v_w — «центральное» представление слова
- v'_w — «контекстное» представление слова

На картиночке выше показано, как работатьс моделью и ворого пункта. Есть какие-то два слова w_0 и w_1. Наша модель подбирает 2 вектора для каждого слова w: v'_{w_0} и v_{w_1}, такие что числитель (экспонента скалярного произведения чисел) будет максимальна. Почему два: в одном случае слово является контекстным словом, в другой центральным. Знаменатель это нормировка, которая делает так, что выполняется условию согласно которому

$\sum_{w \in W} P(w|w_I) = 1$, где W -все возможные слова.

суть обучения - бегаем по тексту. Берем слово в качество центрального и предсказываю вероятность встретить слова, находящиеся рядом с ним

$$\sum_{i=1}^{n} \sum_{\substack{-c \le j \le c \\ j \ne 0}} \log p(w_{i+j}|w_i) \to \max$$

i - позиция центрального слова, с помощью которого мы предсказываем соседние слова.

Дальше мы пробигаем по словам из контекста на с вперед, на с назад и предсказываю для них вероятности при условии данного центрального слова (то есть вероятность встретить их по соседству) (там еще можно суммировать по всем текстам) - и мы эту вероятность максимирируем. Замечание: j - то, насколько мы отступаем от центрального слова (не больше с и не меньше (-с), гиперпараметр)

Зачем negative sampling: То, что было представлено выше, очень долго обучать. Хотим быстрее обучать представления. Будем требовать а) чтобы сигмоида из скалярного произведения слова в контексте и центрального слова была больше (мы больше ничего не требуем про слова, которых нет в контексте этого слова - раньше мы настраивали вероятность для всех слов из текста сразу так, что для близких слов мы требовали больших вероятностей и маленьких для остальных). Так не круто, поэтому мы добавляем еще одно слагаемое, который будет содержать случайные слова из словаря, требуем, чтобы для этих случайных слов сигмоида от скалярного произведения этих слов с нашим центральным словом была как можно меньше. Тем самым, мы избавились от нормировки, но при этом мы пытаемся занизить вероятности для всех остальных.

Negative sampling

$$p(w_O|w_I) = \log \sigma(\langle v'_{w_O}, v_{w_I} \rangle) + \sum_{i=1}^{k} \log \sigma(-\langle v'_{w_i}, v_{w_I} \rangle)$$

- w_i — случайно выбранные слова
- Слово w генерируется с вероятностью $P(w)$ — шумовое распределение
- $P(w) = \frac{U(w)^{\frac{3}{4}}}{\sum_{v \in W} U(v)^{\frac{3}{4}}}$, $U(v)$ — частота слова v в корпусе текстов

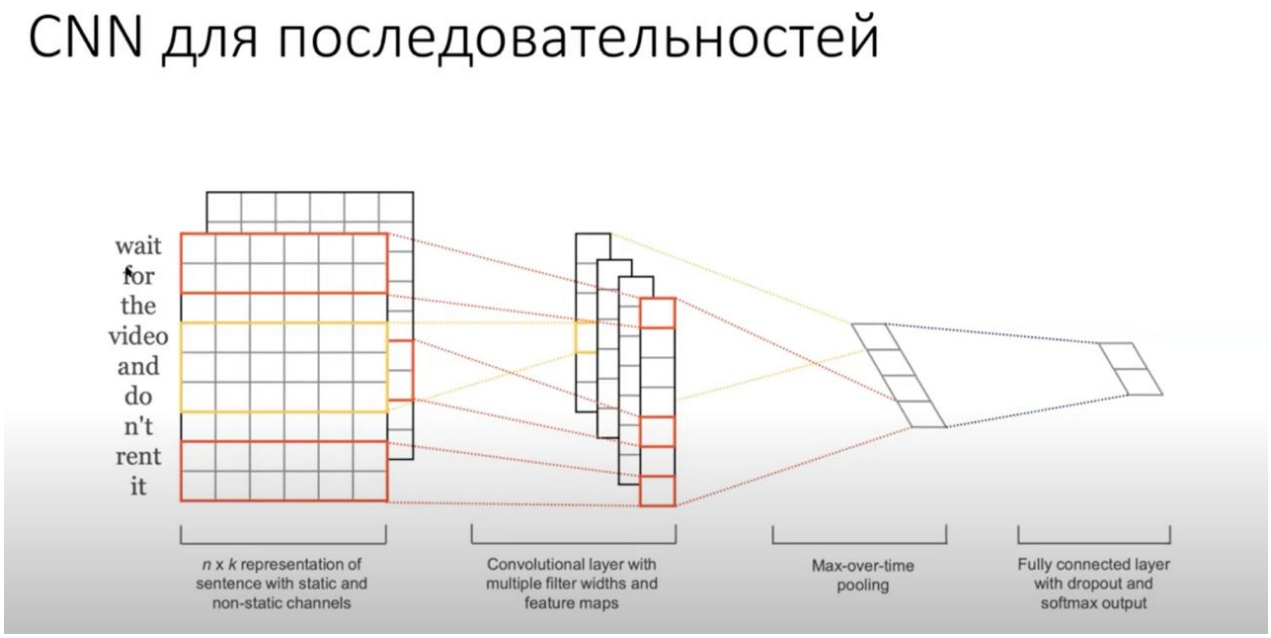
1.1.36 Опишите модель FastText

Каждое слово - мешок подслов (наборов символов). В этом мешке будет храниться (<слово>,<сл,сло,лов,ово,во>), то есть слово + набор подряд идущих 3-символьных сочетаний (скобочки тоже включаем). Каждое слово - набор токенов t_1.....

Представления векторные будем учить для токенов, а не для слов. В тоже самое время

вектор самого слова - сумма векторов, обученных для токенов из мешка этого слова. Это позволяет учесть опечатки и неточности в написании предложений и текстов, различия в верхнем и нижнем регистре. Также мы увеличиваем выборку. Качество получается больше.

1.1.37 Как можно использовать свёрточные нейронные сети для классификации или регрессии на текстовых данных? Как в них могут использоваться представления из word2vec или других методов?



Представляем каждое слово вектором из 6 (в данном случае чисел).Каждая строка - представление слов. Количество слов= количество строк. Применим множество сверток, каждая свертка будет одномерной. Как применяем? Ну вот берем подряд два идущих слова (wait, for) и сворачиваем с определенным фильтром. Дальше будем ходить этим фильтром по всем словам нашего датасета. В итоге будет получаться вытянутый одномерный вектор. Замечание: сами свертки не должны быть фиксированной длины, они могут браться разные. Важно, что внутри каждой из сверток количество слов, которые мы пропускаем через фильтр фиксировано и одинаково. Дальше я беру максимум по всему тексту для каждого фильтра и получаю количество чисел, равное количеству фильтров. Это пример реализации одного сверточного слоя. Наконец, после чего использую полносвязный слой и предсказываем классы или регрессию (уровень тональности, например)

Как в них могут использоваться word2vec?

Вопрос в том, как мы можем получить векторы:

1) CNN-rand - случано инициализирую векторы для каждого из слов

2) CNN-static - вектора из word2vec

3) CNN-non-static - вектора из word2vec и дообучаю их с помощью сверток (то есть они меняются вместе с обучением модели, чтобы повышать качество модели)

Свертками мы читаем научные статьи :)

1.1.38 Опишите модель простейшей рекуррентной нейронной сети

Реккурентные сети имитируют последовательное прочтение текста. Мы читаем текст последовательно и постепенно всё лучше понимаем, о чём он.

Последовательность: x_1,x_2...x_n - любая (например, набор слов). Они закодированы векторами. Читаем слева направо

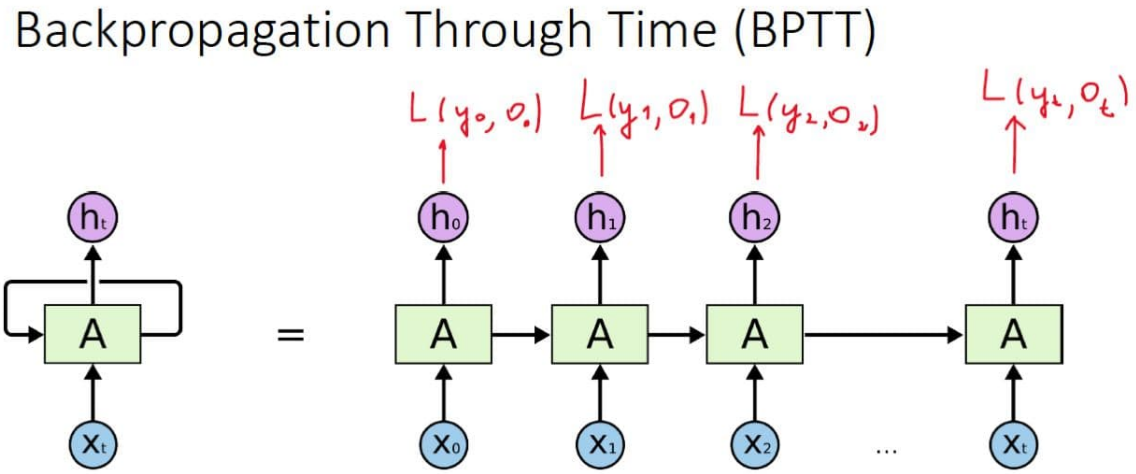
h_t — накопленная информация после чтения t элементов (вектор).

$h_t = f(W_{xh}x_t + W_{hh}h_{t-1})$, где

W_{hx} – это некая обучаемая матрица, ее мы умножаем на вектор t-ого словма (на выходе также будет вектор). W_{hh} – это предыдущее скрытое состояние.

Если хотим выдавать что-то на каждом шаге (например, если мы хотим предсмказывать вследующее слово по уже написанному куску текста):
$O_t = f_o(W_{ho}h_t$, где t – это сколько символов уже было написано.

Развёрстка RNN. Backpropagation Through Time (BPTT).



Берем x0, получаем скрытое состояние h0.
Берем x1, получаем скрытое состояние h1.
И вот так постепенно читаем текст и обновляем скрытые состояния, в них копится информация. Верим, что к моменту t вектор будет содержать всю информацию до него.
Можно делать многослойные RNN и уходить вглубь.