Тренды в анализе данных. Статья 1

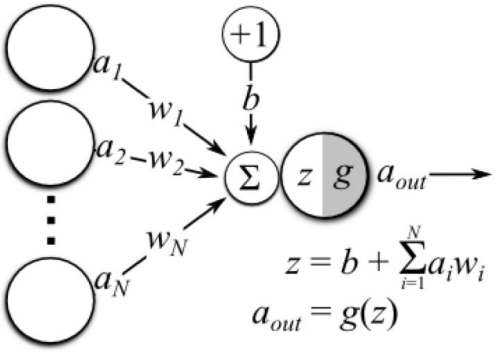
О цикле

Данный цикл статей основан на семинарских занятиях открытого курса Data Mining in Action по направлению “тренды в анализе данных”. На семинаре (и в статьях по нему) мы будем говорить о последних достижениях в области data science.
Как и говорилось на вступительной лекции, основная задача наших семинаров - это быстро перейти с того, что рассказывают в других курсах по глубокому обучению (например, здесь и здесь), к чему-то более сложному и новому, о чем нигде не рассказывают, по крайней мере так подробно.

Ну а на первом семинаре мы бы хотели провести быстрый обзор deep learning.

Вспомним про линейные модели

Пойдем с самого начала. Сейчас все, что мы рассматриваем будет на примере supervised-задач, где на основе некоторых данных нужно предсказать значение целевой переменной.
Вспомним стандартную линейную модель, которая пытается построить прямую, приближающую ответ. У нас есть n признаков, для которых нужно найти веса. Линейная регрессия строит ответ как линейную комбинацию признаков, в случае логистической регрессии к этому еще применяется функция сигмоиды.



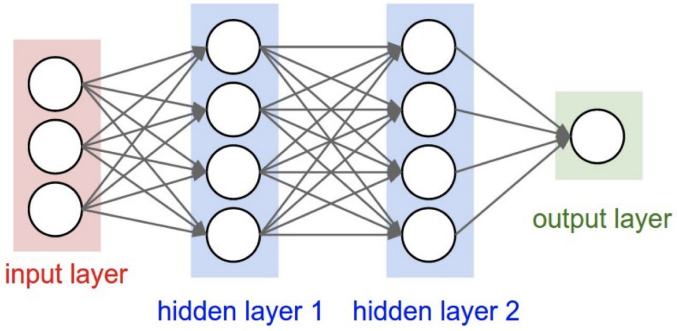
В чем минусы такой модели?
Не все можно предсказать линейной зависимостью. Эта модель весьма ограничена: если у нас в реальной жизни зависимость не линейная, то все очень плохо.

Какие плюсы?
Она интерпретируемая - можно по весу более-менее сказать, как тот или иной признак влияет на ответ. К тому же, она быстро учится и вычисляется. Если признаков очень много, то в первом приближении линейная модель - почти единственное, что может работать.

А что делать, если линейной зависимости в данных нет? Можно придумать какие-то признаки, относительно которых задача будет линейна. Самое простое - это взять попарные произведения признаков - то есть полиномиальную регрессию (это та же линейная регрессия, если взять признаки как попарные произведения или степени). Сразу же встает вопрос: можем ли мы автоматически, не перебирая все полиномы и степени, придумать хорошие признаки и автоматизировать подбор? Самый простой ответ - полносвязная нейронная сеть.

Полносвязная нейронная сеть

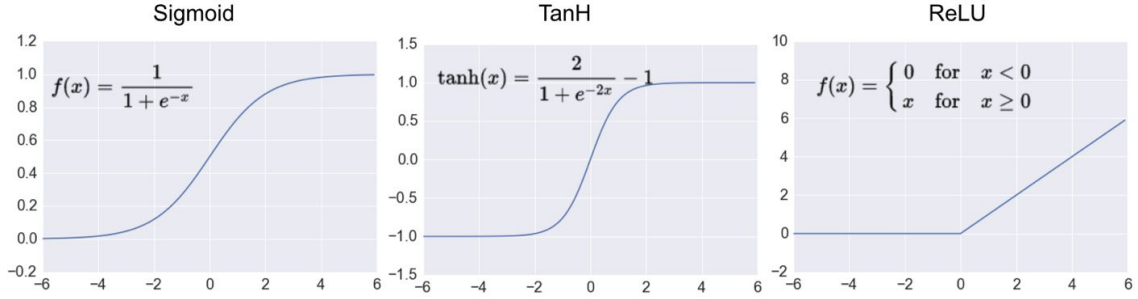
В чем идея? По сути это - та же линейная модель, но с некоторыми преобразованиями, которые добавляют нелинейности. Задача разбивается на две: первая - построить модель, которая будет по входным признакам строить хорошие признаки для нашей задачи, чтобы задача стала линейной. А вторая - это построение линейной регрессии.



Полносвязная сеть состоит из слоев (выделено одним цветом), которые содержат какое-то количество нейронов (кружочков).
Первый слой - входной: в нем содержатся сами признаки. К каждому нейрону следующего слоя идет стрелка от каждого нейрона из предыдущего слоя. Эти стрелки подразумевают домножение на какие-то разные веса и в итоге сложение (то есть, это - линейная комбинация нейронов с предыдущего слоя), в результате чего получается какое-то число. В каждом нейроне к получившемуся числу применяется нелинейная функция, которая называется функцией активации. И так процесс повторяется несколько раз, пока не придем к одному выходному числу.

Зачем нужна магическая нелинейная функция? Если бы у нас не было никакой нелинейности, то каждый нейрон второго скрытого слоя получался бы как линейная комбинация нейронов из первого слоя и так далее. В итоге вся модель делала бы какое-то линейное преобразование, и проблему линейности мы бы не решили.

Какие бывают функции активации?

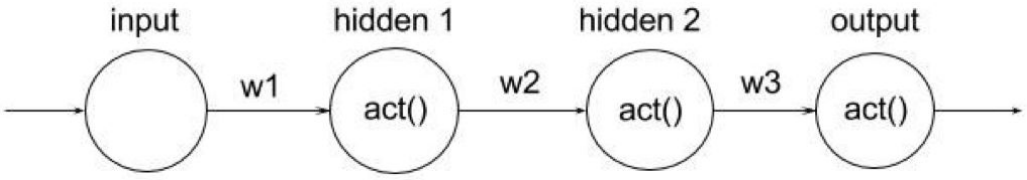


Первые две на графике относятся к классу сигмоид. Первая - логистическая функция, которая используется в логистической регрессии. Вторая - гиперболический тангенс. А последняя ReLU - типо срезки, то есть отсекающая все, что ниже нуля.
Вопрос: зачем разные функции, почему не ограничиться одной и радоваться?
На самом деле они представлены в каком-то историческом появлении. Сначала все пользовались сигмоидой (под сигмоидой обычно понимают именно логистическую функцию), так как логистическая регрессия появилась давно и хорошо работала.
Но тогда зачем же нужно что-то другое? У нее (сигмоиды) есть проблема - слишком большие куски (после 4 и до -4), где график почти горизонтальный, а значит, производная близка к нулю. Мы посмотрим, к каким проблемам это приводит, чуть позже.
Гиперболический тангенс немного смягчает проблему. А у ReLU с виду такой проблемы нет, по крайней мере, на положительной оси. По опыту он работает лучше в большинстве задач.
Обычно для ускорения вычислений внутри одного слоя все функции активации одинаковы. Между двумя слоями они могут быть разными, но стоит ли так делать - это уже отдельный вопрос.

Оптимизация, обратное распространение ошибки

Теперь мы понимаем, как выглядит нейронная сеть и как она применяется к объекту. Но как находить и оптимизировать коэффициенты на стрелках, как подбирать параметры?

Для линейных моделей самое простое решение - это градиентный спуск. Для того, чтобы делать градиентный спуск, нужно знать градиент нашей функции потерь по параметрам модели. При этом предсказания у нас, в отличие от линейной модели, выглядят чуть более сложно.
Рассмотрим пример оптимизации нейросети с двумя скрытыми слоями и для простоты будем считать, что в каждом слое только один нейрон.



Если записать предсказание модели одной длинной формулой, то получается много уровней сложности и вложенности.

$$output = act(w3 * act(w2 * act(w1 * input)))$$

Вопрос: как считать градиент? Нам нужно уметь брать производные нашей функции потерь по w1, w2 и w3. В нашем случае, если каждый слой состоит из одного нейрона, то w1, w2, w3 - это просто числа. Нас спасает производная сложной функции, которая для первого веса, который глубже всего находится, выглядит вот так.

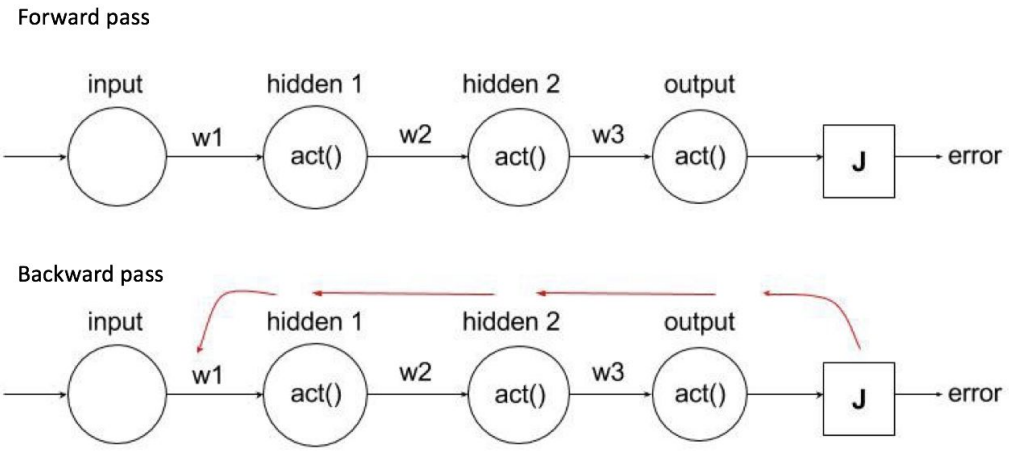
$$\frac{\partial}{\partial w1} output = \frac{\partial}{\partial hidden2} output * \frac{\partial}{\partial hidden1} hidden2 * \frac{\partial}{\partial w1} hidden1$$

В свою очередь, производная ошибки по первому весу выглядит следующим образом:

$$\frac{\partial error}{\partial w1} = \frac{\partial error}{\partial output} * \frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1} * \frac{\partial hidden1}{\partial w1}$$

Тут важно заметить, что производная нашей ошибки (функции потерь) по w1 зависит только от двух вещей:
1) - производная ошибки по выходу нашей нейронной сети,
2) - производная выхода по w1.

Чем это нас спасет? Тем, что, когда мы будем считать производную по w2, у нас будет уже на одно произведение меньше. А что дальше? Когда мы будем брать производную ошибки по w3 по самому дальнему весу, нам нужна будет производная ошибки по output и производная output по w3. При этом мы знаем, что и для w2, и для w1 нам производная ошибки по output нужна тоже. За счет этого мы ускоряем пересчет.



Таким образом, сначала мы берем некоторый объект и по нему делаем предсказание. Далее считаем ошибку и берем производную ошибки по предсказанию. После этого берем производную предсказания по w3, перемножаем и получаем производную ошибки по w3. Затем берем производную предсказания по второму скрытому слою, перемножаем на производную ошибки по предсказанию и получаем производную ошибки по второму скрытому слою. Теперь нам осталось просто посчитать производную второго скрытого слоя по w2, домножить на то, что было, и пойти дальше.

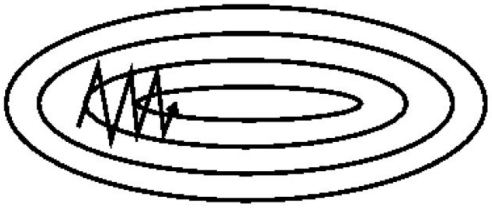
Считать градиент мы научились. Тогда можем написать простой градиентный спуск, который выглядит так:

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

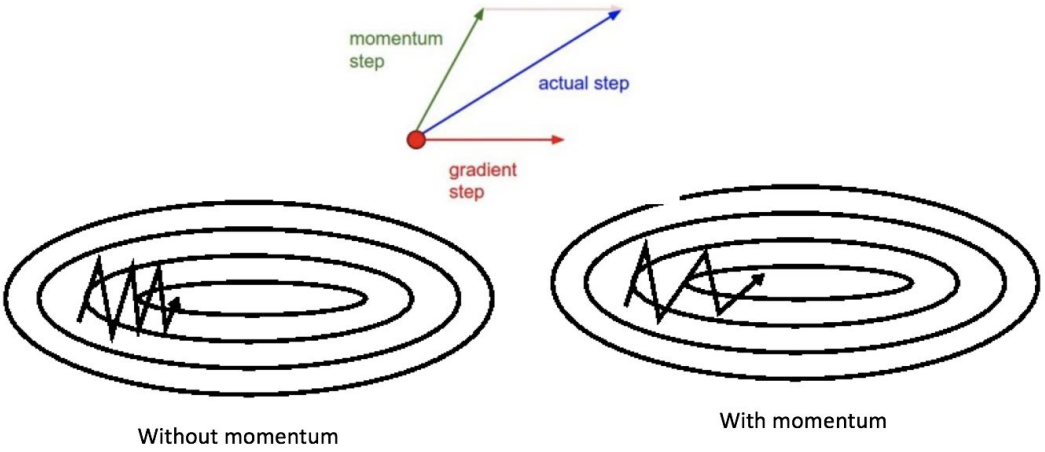
Вопрос: чем он плох? Мы просто несколько раз берем, считаем градиент нашей функции потерь по обучающей выборке и делаем градиентный шаг. Первое замечание: нам очень дорого считать производную по всем данным. Давайте тогда перейдем к стохастическому градиентному спуску и будем считаем ошибку по небольшому батчу (подвыборке).

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

А можно ли сделать что-то еще? Чем плох наш стохастический градиентный спуск? Появляется проблема затухания градиентов. Когда мы с помощью обратного распространения ошибки считаем градиент, у нас (по старой формуле) производная по какому-то весу зависит от производных, которые идут в модели после вычислений. То есть, производная w1 зависит от производной hidden 2, output, hidden 1 и ошибки. А производная w2, в свою очередь, зависит от производной hidden 2, output и ошибки. И они перемножаются. Поэтому если они близки к нулю или слишком большие, то при перемножении они либо занулятся, либо взорвутся. Если в качестве функции активации у нас сигмоида, то, как мы знаем, у нее производная в большом отрезке оси близка к нулю. С другой стороны, если везде стоит ReLU, то будет обратная проблема: выходы слоев могут неограниченно расти в положительную сторону, и градиенты могут стать очень большими. И при небольшой ошибке, при небольшом изменении последних весов, веса, близкие к началу сети, будут меняться очень сильно.
Представьте, что вы пытаетесь оптимизировать функцию в духе 10x^2 + y^2 / 10. Линии уровня будут выглядеть как сплюснутые эллипсы, у которых по одной оси что-то в духе одной десятой, а по другой десять.

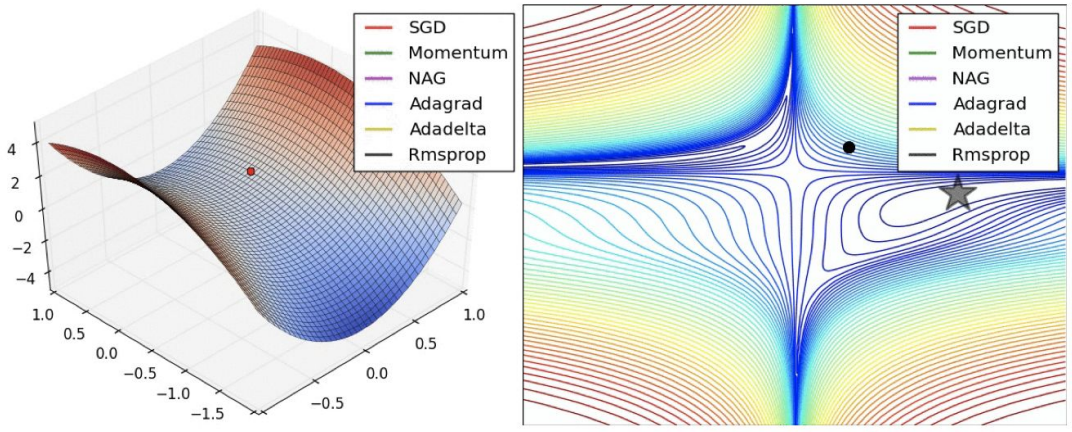


При этом градиент будет указывать 20x + y/5, то есть, смещение будет идти в основном по X. Это значит, что мы не будем двигаться к точке 0.0, где на самом деле оптимум, а нас будет все время по X проносить мимо этой точки. То есть, мы будем идти не по прямой к оптимуму, а какими-то зигзагами, как-то приближаясь. Это неоптимально.
Первое улучшение - моментум. Давайте добавим градиенту что-то вроде инерции. Будем идти на каждом шаге не по градиенту, а по, условно говоря, градиенту, усредненному с предыдущими шагами. Мы будем поддерживать скользящее среднее по градиенту с предыдущих итераций. И когда мы на новой итерации выберем новый батч, посчитаем по нему градиент, мы пойдем не по нему, а по среднему. Что нам это даст? По сути зигзаг: вверх, вниз, а среднее - по горизонтали почти хорошо ведет в сторону оптимума. Таким образом, колебаний станет меньше.



Могут быть проблемы: если мы приблизимся к точке локального оптимума, и градиент занулится, то из-за инерции мы будем идти дальше. Но это решение используется, в основном, в стохастическом градиенте, а он, скорее всего, нулю равен не будет.

Вывод такой: стохастический спуск - это хорошо. Однако плохо то, что он случайный. С этим хочется бороться. И есть куча различных методов для этого.



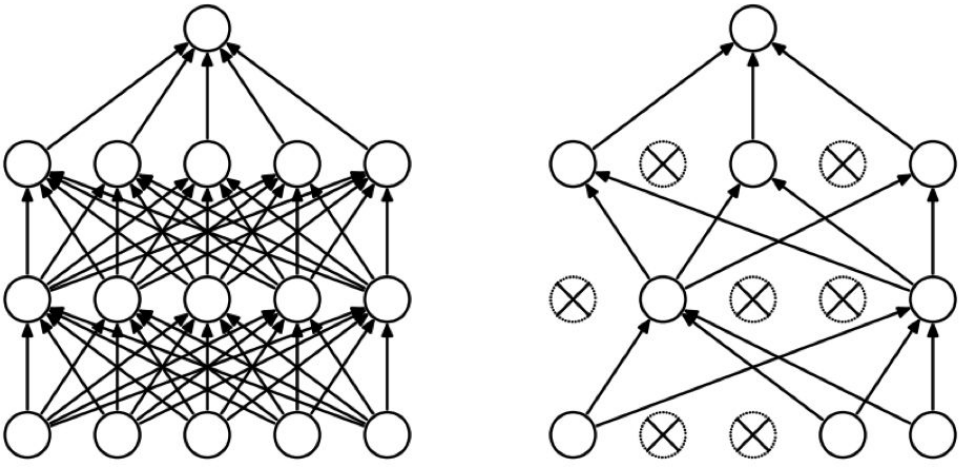
Здесь сравниваются несколько методов. В идеале мы хотели бы, чтобы наш метод оптимизации в силовых точках не застревал. То есть, например, когда по одной оси максимум, по другой - минимум.

Есть разные улучшения, чтобы справиться с этим: моментум, моментум Нестерова, Adagrad и Adadelta. Они пытаются бороться с той проблемой, что у некоторых параметров может быть своя длина оси эллипса. То есть, разные параметры важны по-разному, и нам хочется ослабить этот эффект. Что происходит: мы считаем среднюю норму градиента - какое в среднем мы проходим расстояние по шагам, и считаем сам средний градиент. Если точнее, то мы считаем среднюю норму градиента и средний квадрат компоненты по каждому параметру. Градиент будет такой же размерности, как и все

параметры. Можем подсчитать среднюю норму: то есть, то, насколько мы сдвигаемся в среднем. А можем посчитать для каждого параметра по отдельности средний квадрат той компоненты градиента, которая ему отвечает. Это будет среднее расстояние, которое проходит этот параметр. Чем оно больше, тем параметр важнее, так как это значит, что он чаще меняется и на большее расстояние.
То есть, понятно, что если нейронная сеть слишком большая, то она очень легко может переобучиться - просто запомнив все обучающие объекты, если слоев будет достаточно или если первый слой будет слишком широкий.

Dropout - борьба с переобучением

Как бороться с переобучением? Стандартный метод - это регуляризация. Можно добавить к функционалу ошибки или l1, или l2 по весам. Но в нейросетях придумали еще один хак: dropout. Во время обучения вы семплируете какой-то батч, а потом считаете по нему градиент. Иными словами, мы идем вперед, делаем предсказание, а после идем назад и считаем градиент.
Dropout предлагает нам действовать иначе. Когда мы делаем предсказание модели, то с некоторой вероятностью P мы зануляем выход из каждого нейрона. Во время прямого прохода часть нейронов нулевые, и в предсказании они не участвуют. Обычно dropout выставляется отдельно на каждый слой. Таким образом, вероятность внутри одного слоя одинакова, а в разных - не обязательно. Казалось бы, что это дает? Мы не позволяем нейросети явно записать какой-то объект в несколько нейронов, так как в какой-то момент они зануляются - и она не сможет ими воспользоваться.

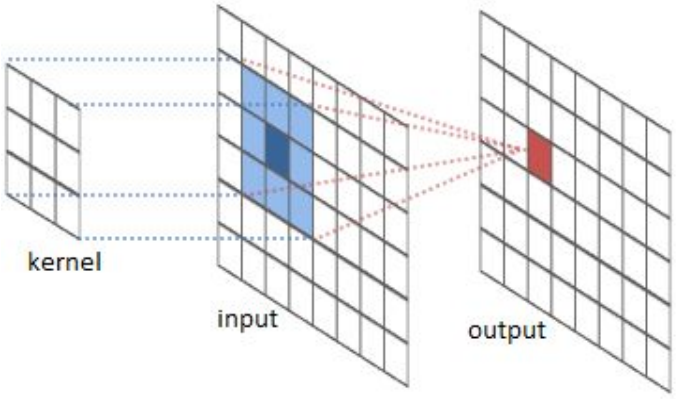


(a) Standard Neural Net (b) After applying dropout.

Комбинаторное объяснение: во время обучения мы случайно с равной вероятностью выбираем архитектуру нашей нейросети, просто выкидывая часть ребер. И понятно, что каждый набор без выброшенных нейронов представляет какую-то уникальную нейросеть. Сколько разных вариантов у нас будет? Например, с пятнадцатью нейронами - это 2^15 различных графов, которые могут появиться. Мы выбираем одну из сетей и учим ее на нашем батче. Тогда можно сказать, что итоговая нейросеть - это ансамбль из этого экспоненциального числа нейросетей. Такая процедура делается для каждого батча. Выброшенные нейроны обучаются потом, но при предсказании участвуют все. Разумеется, обучение обычно замедляется, но во сколько раз - это будет зависеть от вероятности занулить нейрон.
Обычно dropout работает получше, но ненамного. Какой метод выбрать - зависит от задачи, и чаще всего это решается эмпирически. l1 и l2 накладывают ограничения на веса, а dropout никаких ограничений не дает: просто добавляет шум, чтобы модель стала более устойчивой. В принципе, можно использовать и то, и другое вместе.
Вопрос - как подбирать вероятность P? Обычно стоит начать обучать модель без dropout, посмотреть, как она учится в течение какого-то времени и увидеть, что сеть сильно переобучается. После этого поставить вероятность 0.5 и заметить, например, что теперь модель недообучается. И таким образом подбирать параметр. Однако надо думать, сколько нейронов выкидывать из слоя: выкидывать 9 из 10 - это не очень хорошая идея.

Сверточные нейросети

Теперь перейдем к тому, как работать с изображениями. Вопрос: что нам мешает взять картинку просто засунуть ее в полносвязную сетку? Допустим, мы возьмем изображение 1024 x 768 и скажем, что мы хотим подать его в нейросеть и далее сделать слой из 32 нейронов. Во входном изображении будет порядка миллиона параметров. Далее, на первом слое все это придется умножить на 32, после чего сразу появится 32 миллиона параметров. Таким образом, мы во-первых получаем слишком много неинформативных параметров, а во-вторых не имеем никакой интерпретируемости. Кажется, что никто никогда не смотрит сразу на все изображение целиком. Когда вас спрашивают: это кошка или собака? Вы посмотрите на уши, хвост и нос - и уже тогда будете думать. Значит, нам тоже хочется научить смотреть модель на какие-то части, а не на все сразу, и уже по ним что-то говорить.
Поэтому введен еще один слой, называемый сверточным:



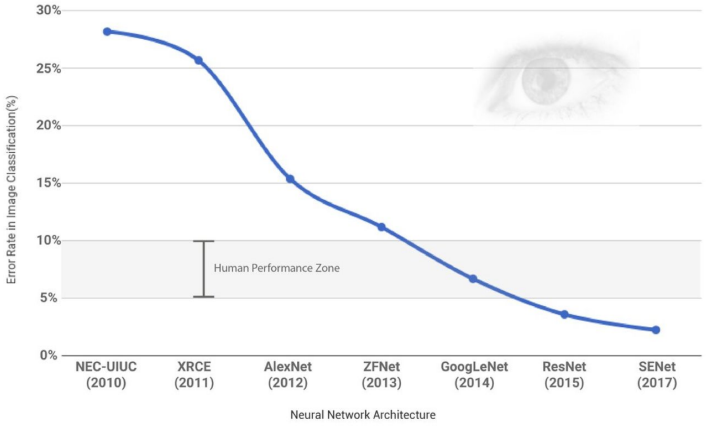
Что здесь происходит? По изображению бегает свертка - это матрица весов, в данном случае 3x3. Когда свертка накладывается на пиксели, то последние перемножаются с весами и складываются, в результате чего получается одно число. И так по картинке мы генерируем какую-то новую картинку. Если эти 9 чисел (ядро свертки) будут выглядеть примерно как ухо собаки, то, скорее всего, при перемножении на место в картинке, где находится ухо собаки, эта сумма будет максимальная. Мы учим фильтры, которые реагируют на какие-то части картинки. Сначала по умолчанию прикладываем к левому углу, считаем, а затем сдвигаем на один пиксель вправо. И так далее. В результате получаем очень плотное изображение. Да, можно сдвигаться больше, чем на один пиксель (это параметр называется страйд). Это дает нам меньше повторяющейся информации, а размер картинки эффективно сжимается.
Для некоторых задач удобно, чтобы на выходе сверточного слоя было такое же изображение, как и на входе. Тогда по бокам изображение заполняется нулями и таким образом немного увеличивается, чтобы результат был тот же. Ну и иногда, для разных странных задач, бывает полезно, чтобы размер свертки стал больше. Но, допустим, мы делаем обычную свертку. Мы уже поняли, что если страйд равен единице, то слишком много информации дублируется. Тогда мы можем сделать страйд побольше, а можем сделать слой пулинга. Мы берем нашу картинку и разбиваем ее на непересекающиеся прямоугольники, например, 2x2. После чего из каждого такого прямоугольника мы берем максимум. Другими словами, из четырех соседних применений свертки берется максимальный эффект. Тогда мы уменьшаем размер сразу в два раза по обоим измерениям, добавляем нелинейность и боремся с дублированием информации.

Типичная нейронная сеть выглядит следующим образом. У нас есть какая-то картинка, которая на самом деле - три матрицы RGB. Мы делаем необходимое количество сверток, потом пулинг, и снова свертки - и так далее. После этого наше изображение становится небольшого размера, и тогда мы вытягиваем его

в вектор, после чего можем работать с ним полносвязными сетями, которые могут уже выдать нам, например, кошка это или собака. Страйды лучше пулинга тем, что с ними требуется меньше операций для вычисления. Но если страйды не очень большие, то картинка не становится намного меньше. Соответственно пользуясь только этим изображение сжать сложно. А если сделать страйды очень большими, то на что-то мы можем не посмотреть. Обычно их используют для того, чтобы сделать изображение больше (или есть какие-то проблемы с вычислениями).
Это была базовая структура, которая чаще всего работает относительно хорошо.

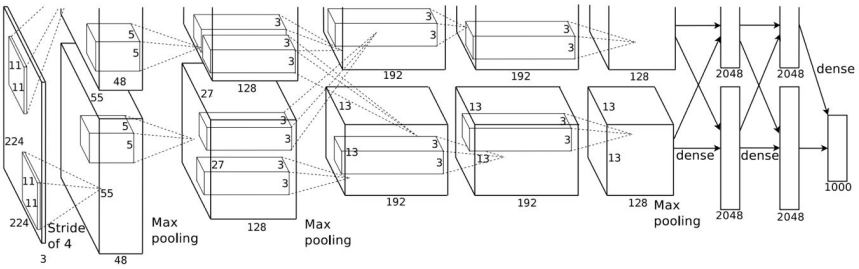
Архитектуры нейросетей с ImageNet

Теперь мы поговорим о том, как нейросети развивались и какие к ним придумывали хаки. Основная причина, благодаря чему развиваются сверточные нейросети уже много лет - это соревнование imageNET. В нем больше миллиона картинок размера около 1024x1024, тысячи классов (например, 30 различных пород собак). Соревнование заключается в том, чтобы лучше всего решить задачу классификации. И с каждым годом ошибка моделей становилась все меньше. Мы рассмотрим последние пять сеток.



Alexnet

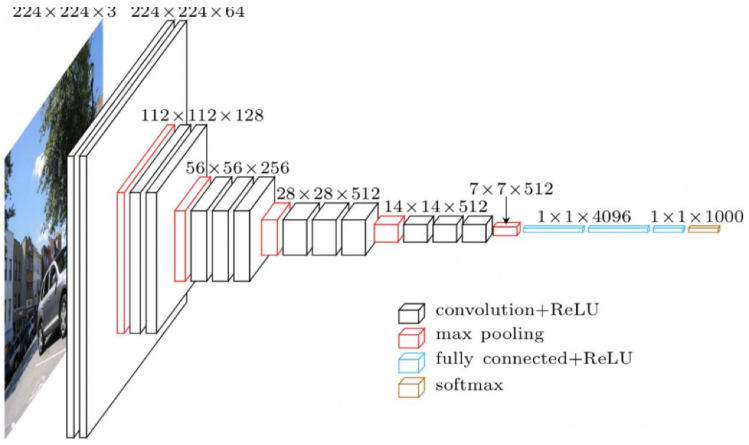
Это одна из первых больших нейронных сетей, и появилась она в 2012. При этом уже в 2014 году качество нейронной сети стало лучше среднего качества человека на той же задаче. В AlexNet в целом, ничего необычного: свертки, пулинг, свертки, пулинг и полносвязные слои. Но примечательно то, что это - одна из первых сеток, которая училась на видео-карте, однако по своему размеру на одну видео карту не влезла.



Поэтому она была как бы разрезана и считалась на двух видео-картах: нижняя часть - на одной, а верхняя - на другой. И они по большей части между собой информацией не обменивались, кроме того, где на картинке явно указана связь между верхом и низом.

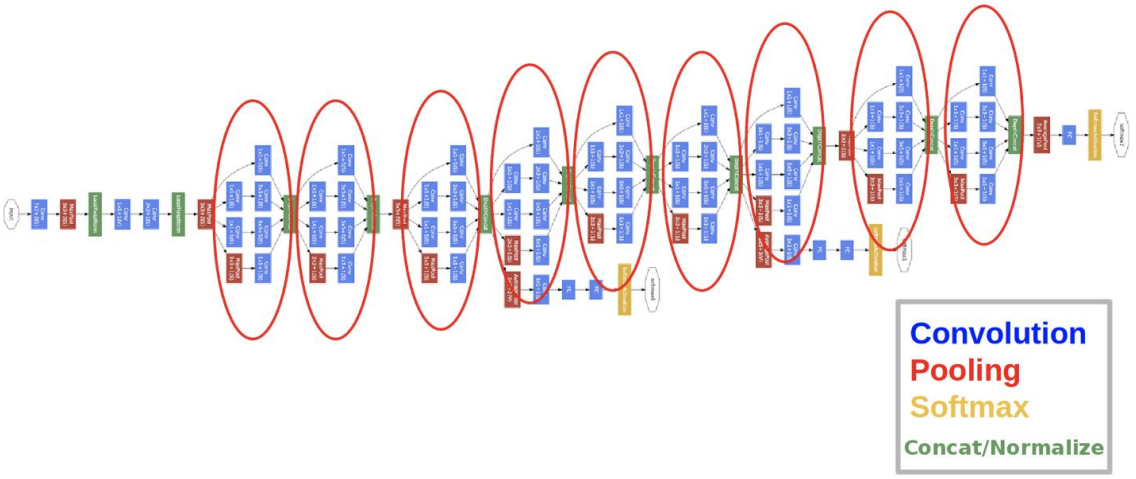
VGG16

Следующая сетка **VGG16** - тоже ничего особо примечательного. Уже чуть больше сверток и пулингов:



GoogleNet

Первая действительно интересная - это **GoogleNet сеть**. Она примечательна тем, что тут уже не такой обычный подход:

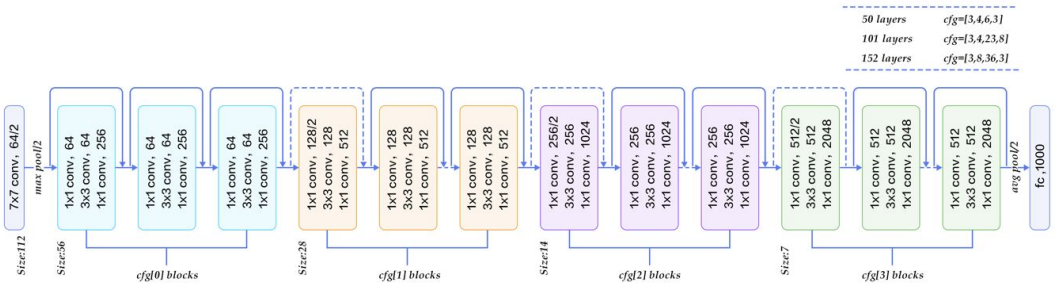


Сама сеть состоит из повторяющихся слоев, но внутри себя они более сложные. Кроме того, в ней есть три выхода, отмеченных желтым цветом. Последний (справа) выдает ответ на всю задачу, а два (снизу) - сделаны, чтобы выдавать ответы классификации но не для финального решения. Дело в том, что у сети много слоев, вследствии чего возникла проблема с затуханием градиента: по сути, она не училась. Сеть в трех местах делает предсказание нашего ответа. Таким образом, мы можем сначала обучить левую часть до первого предсказания: и там все хорошо, так как она не очень глубокая, а потом доучивать подлиннее - и так далее. Это первый предложенный способ, как бороться с затуханием градиентов. В итоге промежуточные ответы лишь вспомогательные, чтобы помогать учиться весам. Различие с обычным подходом в том, что раньше нам пришлось бы учить большую сеть с рандомными весами, а здесь предлагается учить сеть, где левая часть уже почти обучена.

ResNet

Следующая идея - **ResNet**. Они пытаются явно решить проблему затухания градиентов. Каким образом? Стандартная проблема заключается в том, что градиенты предыдущего слоя зависят от последующего. В таким случае наша задача - избавиться от того, что они постоянно перемножаются на числа, близкие к нулю. Давайте сделаем какие-то преобразования, чтобы градиенты не перемножались на это число. Например, научимся делать так, чтобы часть градиентов пропускалась. В обычном случае мы берем картинку, берем свертки, пулинг и так далее. А теперь мы взяли изображение, сделали пулинг два раза и получили картинку. Однако мы хотим изображение исходного размера и будем использовать для этого страйды. Далее к исходной картинке прибавляем поэлементно выходную картинку (то есть, складываем их). Так как

преобразование выглядит как функция от входного изображения + входное изображение, то у градиента появляется единичка. Вся их сеть выглядит так:



Мы повторяем такие блоки очень много раз. Эта модель победила в 2015 году на imageNet, суммарно в ней было 152 слоя, в то время как у предыдущей гугловской модели - порядка двадцати.

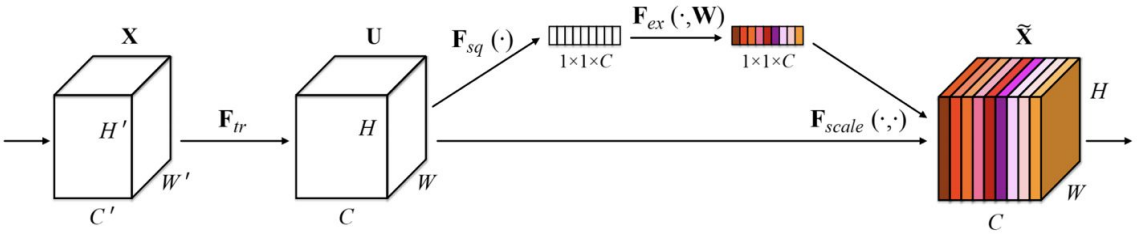
Кроме этого, у ResNet был еще один хак - BatchNorm. Это как раз попытка как-то нормализовать наши данные. В чем суть? Известно, что всякие линейные модели хорошо работают, когда данные из нормального распределения. Почему? Если вы оптимизируете MSE, это эквивалентно тому, что вы максимизируете правдоподобие модели в предположении того, что у нее гауссовское распределение. То есть, таргет зависит от входных переменных по закону нормального распределения. А что такое модель максимума правдоподобия? Это когда мы пытаемся максимизировать правдоподобие или его логарифм. Логарифм для экспоненты - это и будет ровно квадратичная форма с какими-то коэффициентами. Другими словами, линейные модели для регрессии любят нормальные данные.

Идея была в том, что можно попробовать выходы всех нейронов тоже сделать нормальными, по крайней мере, сделать им нулевое среднее и единичную дисперсию. И для этого мы делаем отдельный слой BatchNorm. Когда к вам приходит батч на него, вы считаете по нему выборочное среднее и дисперсию и нормируете его. Теперь на выходе нормальный батч. Но при этом внутри слоя есть среднее, скользящее среднее и средняя дисперсия. И вы обновляете эти средние с помощью выборочного среднего и выбранной дисперсии вашего батча. Для чего? Когда приходит новый объект для тестирования, то вы уже не по нему считаете среднюю дисперсию (так как это - один объект), а используете те, которые считали внутри вашего слоя. Задача следующая: выходы всех слоев должны быть похожи на нормальное распределение. Можно было бы нормировать входные данные, однако это не дает гарантии, что внутри все будет нормально, потому что если используете ReLU, то получается, скорее всего, что-то не слишком похожее на нормальное распределение. А BatchNorm делает распределение ближе к нормальному и улучшает сходимость по градиентам.

Но есть и проблемы. Если нет возможности гонять большой батч параллельно в нейросети, то средняя дисперсия по выборке получается очень шумная (то есть, например, считать по двум объектам - плохо). Есть некоторые попытки бороться с этим (BatchRenormalization), однако это все равно не спасает.

SeNet

Последняя сеть которая выиграла соревнование, называется **SeNet**. В ней используется нечто вроде механизма внимания, который сейчас активно используется почти везде, но особенно в машинном переводе. В чем идея? Вот мы берем результат применения наших сверток. Это трехмерная матрица CxHxV. Давайте найдем максимальный отклик каждого фильтра. Каждый фильтр возвращает одну картинку. В таком случае, давайте найдем максимум в этой картинке и скажем, что это его максимальный отклик. По CxHxV мы берем максимум по двум измерениям - ширине и высоте, в результате чего получаем по одному числу - верхний вектор.



Далее, к нему применяем полносвязную сеть, которая делает такой же вектор, но от 0 до 1. Таким образом, мы делаем нейронную сеть с сигмоидой на выходе. Получили C чисел. Каждый слой, который соответствует своей C, домножаем на соответствующее число из этого результата. Итого: мы взяли картинку, применили к ней 64 свертки и получили 64 новых картинки. Но каждая картинка - двумерная матрица.

Теперь вопрос: для нашей картинки какие из сверток были самые правильные? Охарактеризуем результат каждой свертки максимум по всему изображению и дальше сделаем нейросеть, которая будет говорить, какие из свертки самые важные, то есть, скажет, на что домнажать (от 0 до 1). Так мы позволяем нейросети, в зависимости от изображения, смотреть на какие-то определенные результаты сверток. Например, для одного изображения левые свертки нам не нужны, а правые - понадобятся. Это позволяет ей обращать внимание на какие-то отдельные куски. Дальше они берут слой из ResNet и сверху над ним делают свою идею с вниманием, таким образом побеждая. Причем у них нет никакой специальной функции потерь для оптимизации внимания. Они оптимизируют только точность модели, но дают ей возможность выучить внутри себя маленькую сетку, которая будет акцентировать внимание на

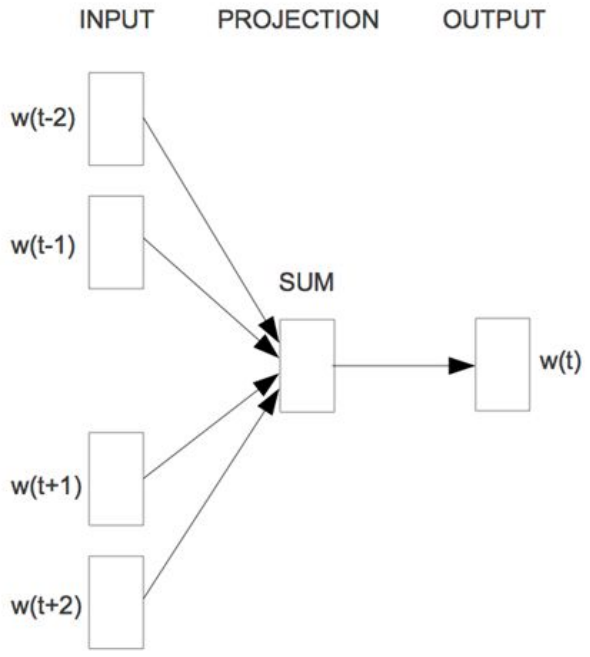
отдельных фильтрах. Соответственно, все параметры учатся одновременно только ради Log loss.

Обработка текста

Теперь можем поговорить, как работать с текстами. Основная проблема заключается в следующем: как переделать в признаки картинки и числа - вполне очевидно. Однако, как извлекать признаки из текста - уже не так понятно.

Bag-of-words - это первое приближение, что можно делать с текстом: сказать, что текст - это мешок слов, и нас интересует: есть ли в тексте слово или нет и, если есть, то сколько раз оно там встречается. Тогда текст можно описать длинным вектором, где длина - число слов в словаре, и для каждого слова проставлено, например, 0 или 1 - встречается слово или нет. Или числовая характеристика - сколько раз это слово встречается в тексте.

Кроме этого, есть всякие эвристики сверху: можно подумать и решить, что те слова, которые встречаются везде, нам не таки интересны, они мусорные, их можно исключить. Это называется tf idf. Но на самом деле разных эвристик здесь много.



Bag-of-words

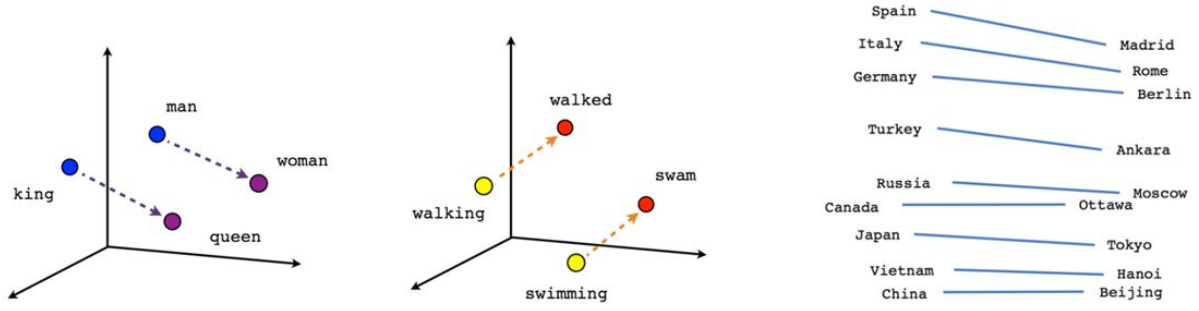
Давайте скажем, что каждому слову мы сопоставляем какой-то вектор фиксированной длины - 400 чисел, например. Тогда любой текст мы можем представить в виде такой матрицы, где по одной оси будет размерность 400 - длина вектора слова, а по другой оси - длина текста в словах. И дальше будем работать с такой длинной матрицей.

Самое часто встречаемое про тексты в нейросетях - word2vec. Идея достаточно простая, но при этом интересная. Мы хотим как-то определить эти вектора для слов. Давайте возьмем и сделаем вспомогательную сеть, которая будет по контексту слова (два слова до и два слова после) предсказывать само слово в контексте.

Архитектура следующая: мы берем четыре слова, для каждого из которых есть свой вектор, записанный в словаре. Затем мы складываем их, делаем один простой полносвязный слой и получаем много вероятностей. Наконец, мы учим это все на большом корпусе текста (из википедии, например).

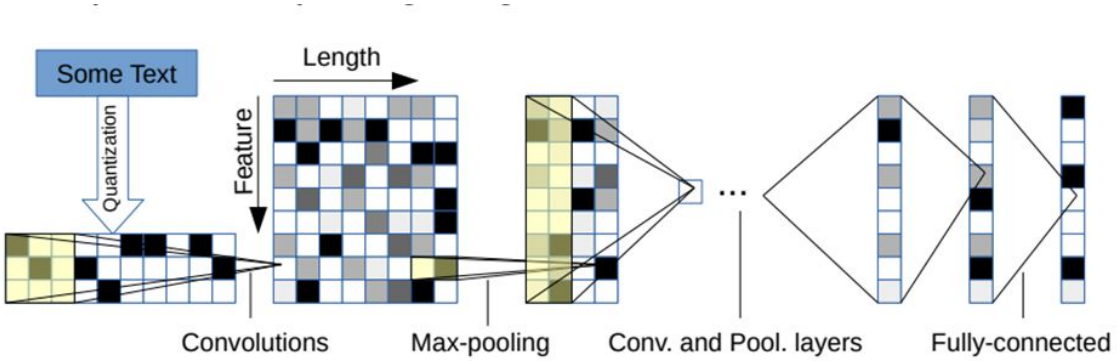
Второй подход - это предсказать по слову контекст. Мы хотим для каждого слова или контекста слов сделать такие вектора, чтобы по этим векторам можно было найти слово из контекста. То есть, мы получаем, что если у нас два слова часто встречаются в одинаковом контексте, то их вектора должны быть близки. И мы каждому слову хотим поставить в соответствие какой-то вектор. Из каких соображений его ставить? Если одно слово можно заменить на другое, и при этом смысл особо не поменяется, то эти слова должны быть близки по этим векторам. В таком случае мы можем взять много текстов и попытаться предсказать по контексту искомое слово. Тогда, если два слова встречаются в одном контексте, вектора у них получатся одинаковые, а если в разном - то разные. Слова из одного контекста обычно рядом, если их кластеризовать.

Если нарисовать вектор от слова “мужчина” до слова “женщина” и от слова “король” до слова “королева”, они получаются одинаковы. Соответственно, можно из “женщины” вычесть “мужчину” и получить какой-то вектор, который отвечает, условно говоря, за смену пола.



Male-Female Verb tense Country-Capital

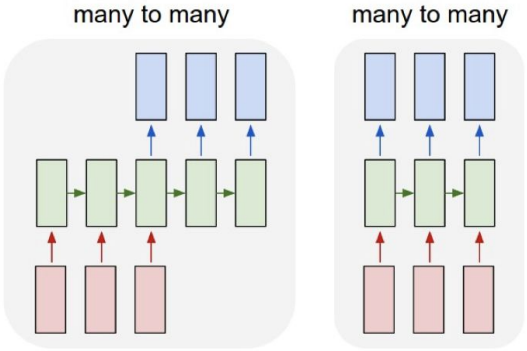
Сейчас с текстами делают что-то более современное: его разбивают на слова или символы, и которым ставится в соответствие какой-то вектор. Из этих векторов мы получаем большую матрицу, с которой можем работать, как с картинкой. Она не будет особо разряженной, так как это не bag-of-words, и мы можем взять размер вектора заведомо меньше, чем число всех слов или букв. Далее, подход такой же: свертки, пулинг и так далее.



Например, на вход подается текст, и наша модель первым слоем берет какую-то букву и ставит в соответствие какой-то случайный вектор. Тогда получается матрица размера N x M, где N - размер вектора для одного символа, а M - количество слов в тексте. В word2vec похожесть векторов измеряют по расстоянию между ними, например, с помощью l2 нормы или косинусного расстояния. И выводы о том, что лучше, а что хуже - по большей части, эмпирические, а не математические.

Работа с последовательностями

Наконец, поговорим про последовательности. Кстати, это имеет косвенное отношение и к текстам. До этого мы всегда работали, с некоторыми оговорками, что объекты у нас одинакового размера. К примеру, “картинка 32x32”, “текст на 100 чисел”. Но давайте поговорим о том, что будет в случае, если вход у нас - это последовательность, и мы хотим по нему что-то предсказать или, наоборот, по одному объекту предсказать какую-то последовательность.

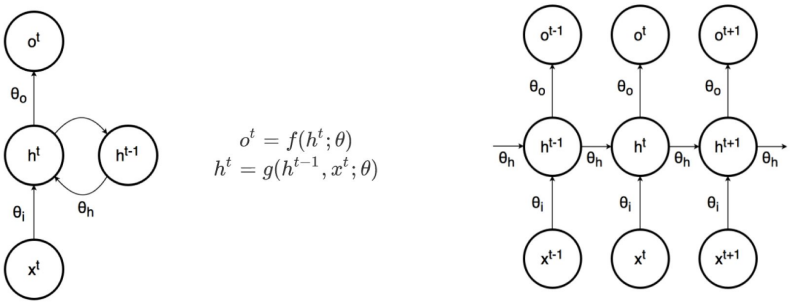


Чем отличаются эти два подхода? В них обоих мы предсказываем последовательность по последовательности. Однако в левом подходе подразумевается, что мы можем сначала посмотреть на всю последовательность и уже после этого делать предсказания, а в правом - мы должны делать предсказания в режиме реального времени.

Рекуррентные сети.

Самое простое, что мы могли бы сделать, когда у нас есть объект нефиксированного размера, это следующее: давайте скажем, что мы ограничиваемся 100 символами, и будем что-то добавлять или отрезать, если у нас получается не ровно 100. Однако это тоже не очень хорошая идея, так как у нас есть некоторая память стандартного размера, и мы можем много потерять. Идею с памятью хочется как-то изменить и сделать ее автоматической. Другими словами, мы не можем хранить бесконечный контекст: нам нужна какая-то память фиксированного размера, но мы не хотим терять много информации.

Вот есть объект X, и x_t - это его значение в момент t. Если мы работаем с текстом, то это x_t - это символ, если например с видео, то x_t - это некоторый кадр. У нас внутри сети есть какая-то память - вектор фиксированного размера, который как-то отвечает за все, что эта сеть увидела в предыдущие моменты времени. Допустим, приходит новый момент времени. Как в этой ситуации работает мозг человека? Когда человек увидел что-то новое, он сначала обновляет свою память, добавляя новое, и после этого делает какой-то вывод на основе всего опыта. Тут что-то похожее.



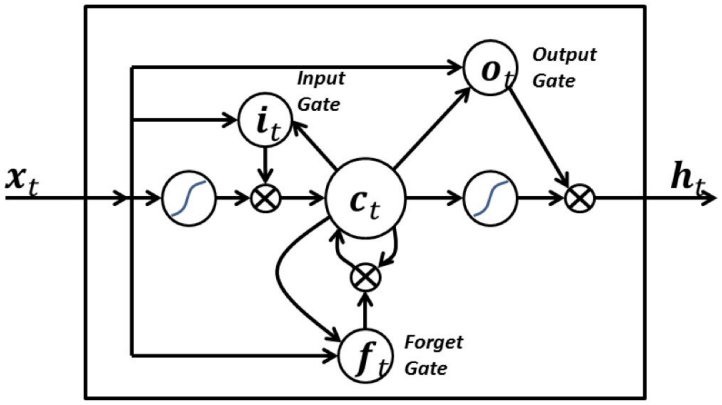
К нам пришла новая информация, и с помощью этой информации x_t и с помощью предыдущего состояния h_{t-1} мы получаем текущее состояние h_t, по которому и генерируем предсказание. Когда мы что-то обновляем или делаем предсказания - это полносвязные слои. Таким образом, у нас есть полносвязный слой, который принимает два входа - предыдущее состояние и текущий вход x_t, и по ним получает новое состояние, а еще полносвязный слой, который по текущему состоянию возвращает ответ.

Вот у нас пришел на вход x_0. Мы обновили скрытое состояние (изначально, к примеру, нулевое) и сделали предсказание, после чего передали скрытое состояние дальше, взяли новый объект и сделали то же самое. То есть, для каждого объекта нашу рекуррентную сеть мы можем развернуть в последовательность простых полносвязных сетей. Во время обучения

рекуррентная нейронная сеть так и делает. Мы даем батч или один объект фиксированной длины, и она разворачивает рекуррентность в длинную полносвязную сеть - и обучает ее. И так происходит отдельно для каждого батча. Если есть длинные последовательности (например, 1000), то в процессе обучения градиенты будут перемножаться, и все проблемы с затуханием или взрывом градиентов будут во много раз больше.

Архитектура LSTM

Улучшение рекуррентных сетей - это LSTM. В них попытались еще сильнее приблизиться к тому, как работает мозг человека, добавив несколько механизмов внимания.



К нам приходит новая информация, а дальше у нас есть отдельный модуль, который определяет по новой информации, на что из нее нужно обратить внимание, то есть, это сигмоида, которая перемножается поэлементно. Еще у нас есть отдельный модуль, что из памяти модели нужно забыть - то есть, еще одна сигмоида. Таким образом, мы добавили два гейта, а третий гейт говорит, что нам из нового состояния нужно выдать на выход. Можно не заморачиваться с пониманием и сказать проще: мы просто добавили несколько механизмов внимания, чтобы с градиентами все было лучше, обосновав это тем, что это похоже на то, как работает мозг человека - что-то игнорирует из входа и что-то забывает из памяти.

В качестве примера применения: есть сеть, которая научилась генерировать LATEX код, который даже почти компилируется. И он выдает какую-то ерунду, которая похожа на типичный учебник математики с теоремами, формулами и так далее.

Еще можно провести следующий эксперимент: последовательно делать классификацию. Изображение подавать на вход не как матрицу 10x10, а преобразовать в вектор длины 100, и рассматривать как последовательность. Таким образом, задачу классификации картинок рассматривать как задачу по последовательности вернуть число.

Дополнительная литература:

Convolutional Neural Networks for Visual Recognition
Usage of activations
Neural Networks & The Backpropagation Algorithm, Explained
An overview of gradient descent optimization algorithms
Dropout: A Simple Way to Prevent Neural Networks from Overfitting
A technical report on convolution arithmetic in the context of deep learning
An Intuitive Explanation of Convolutional Neural Networks
Image Classification
The Unreasonable Effectiveness of Recurrent Neural Networks
Recurrent Neural Network
Multiple Object Recognition with Visual Attention
DRAW: A Recurrent Neural Network For Image Generation