Нейронные сети

Оглавление

Основы нейронных сетей

1.1 Введение

Нейронные сети применяются для распознавания изображений, распознавания речи, для перевода текстов, в робототехнике, для рекомендаций, поиска аномалий и так далее. В огромном количестве задач сейчас используются именно нейронные сети. Мы с вами поговорим о том, что это такое, как они работают, из каких частей состоят. Разберем основные архитектуры нейронных сетей, такие, как рекуррентные нейронные сети и сверточные нейронные сети. И коротко поговорим о современных архитектурах нейронных сетей.

Нейронные сети используются в огромном количестве областей, от автономных машин до, например, создания новых лекарств. Тем не менее очень часто они применяются как какие-то черные ящики. Мы просто подаем на вход данные и получаем какое-то предсказание. Внутри происходит магия, с таким подходом далеко не уедешь, и построить какую-то хорошую рабочую архитектуру, не понимая, как именно работает нейронная сеть, просто невозможно. Несмотря на то, что нейронные сети стали популярны только в последние 5-7-10 лет назад, они были изобретены и придуманы довольно давно, 60-50 лет назад они уже активно использовались. Например, Розенблатт, в 57-м году предложил модель перцептрона, которая стала прообразом современных нейронных сетей. Тем не менее они не стали популярны до какого-то недавнего момента. Почему?

Ученые исследовали продолжали работать над ними и в 80-х годах. Тем не менее особых успехов не добились. **Есть несколько причин, почему так произошло и почему нейронные сети стали популярны именно сейчас**. И ключевых три.

Первая - это **данные**. С появлением интернета и развитием технологий смартфонов, мы все с вами создаем огромное количество данных. Каждый день мы что-то пишем своим друзьям, пишем какие-то посты в соцсетях, создаем картинки, фотографируем, снимаем видео и так далее. Все это огромное качество данных, огромный массив данных, который может использоваться нейросетями для обучения. Нейросети - это такая модель, такие архитектуры, которым для своей работы просто необходимо использовать большое количество данных, и они стали хорошо работать с появлением большого количества данных - это логично.

Вторая причина - это **развитие вычислительных мощностей и ресурсов**. Если раньше какая-то задача на процессоры занимала недели и годы, то сейчас за несколько часов или дней можно обучить довольно сложную модель, и так как видеокарты развиваются постоянно, мы с вами можем все больше и больше считать и все лучше и лучше обучать нейронные сети.

И следствие всего текущего хайпа, развития отрасли и появления большого качества данных

и вычислительных ресурсов - это то, что у нас **появляется все больше и больше новых подходов; научные статьи появляются каждый день; огромное количество ученых вкладывает свои ресурсы и свои мозги в эту отрасль** - всё это несомненно двигает отрасль вперед.

Тем не менее не стоит считать нейронные сети какой-то серебряной пулей, которая решает любые задачи. При обучении и при использовании нейронных сетей существует довольно много сложностей и первая, наверное ключевая из них это то, что **модели нейронных сетей действительно сложно интерпретировать**, не всегда понятно, почему нейронная сеть делает то или иное предсказание. Почему мы классифицировали такой объект именно так. И существует довольно много методов борьбы с этим. Можно, например, в случае сверточных нейронных сетей сделать какие-то деконволюции, смотреть на attention и так далее. Тем не менее, в общем случае, нейронные сети действительно сложно интерпретировать и понять почему они так или иначе работают в отличие, например, от каких-то линейных моделей.

Как мы с вами уже говорили, нейронные сети требуют огромное количество данных. Чтобы обучить какую-то модель вам нужно собрать большой dataset, хорошо его разметить и подать на вход нейросети. Не всегда существует большое количество данных, и это определенная сложность.

И аналогичная проблема с вычислительными ресурсами. Скорее всего, вы не сможете запустить какую-то большую сложную модель просто у себя на ноутбуке. Нужен какой-то кластер или просто сервер с видеокартой, чтобы обучить вашу модель. И все равно обучение может занять продолжительное время: от нескольких дней, до даже недель. А так как архитектуры являются достаточно сложными, чтобы их настроить, чтобы подобрать какие-то параметры, нужно потратить много времени, и это тоже относится к определенным сложностями в работе с нейросетями. Даже если вы что-то настроили, не всегда модели заводятся, и нужно потратить время на исследование.

А так как отрасль новая и активно развивается, достаточно сложно следить за новинками, за какими-то новыми архитектурными моделями и поэтому порог, например, входа в отрасль достаточно высок, и все равно, несмотря на то, что появляются новые отличные фреймворки, которые позволяют на высоком уровне работать с нейросетями, нужно знать математику на определенном уровне и понимать как нейросети работают.

1.2 Что такое нейросеть

Очень часто в СМИ и в какой-то литературе считается, что нейронные сети, которые мы сейчас с вами будем изучать, являются аналогами того, как работает наш мозг - работают и построены по тому же самому принципу. На самом деле это, конечно же, не до конца правда. Несмотря на то, что нейронные сети изначально были вдохновлены именно тем, как работает наш мозг, сейчас аналогии достаточно мало. Очень часто они проводятся наоборот, в обратную сторону. Сначала открывается какой-то подход при обучении и тренировке нейронной сети, а потом оказывается, что в нашем мозге при определенных условиях все работает примерно так же.

Итак, нейронная сеть состоит из нейронов, и нейрон является главной концепцией, которую нужно сейчас понять. Что же такое нейрон? **Нейрон** — это вычислительная единица, которая

принимает какое-то количество значений, какой-то вектор, и просто возвращает число. По сути, нейронная сеть состоит из большого количества таких нейронов, которые передают информацию друг другу. Таким образом **нейронная сеть представляет собой просто супер позицию большого количества нелинейных функций**.

Итак, нейрон принимает на вход какой-то вектор, какие-то входные данные. Также **у нейрона существуют веса, которые настраиваются при обучении, которые обновляются при обучении**. Эти веса перемножаются с входными данными. Мы также можем добавить какое-то смещение, все это дело сложить и отправить в функцию активации. Таким образом мы поулчили какие-то входные данные, перемножили их с весами нейрона и на выходе вернули число. Это число может быть большим и тогда считается, что нейрон **активирован**, зажжен, и он уверен в том, что какой-то объект присутствует во входных данных. Нейрон может бы **не активирован**, значение может быть маленьким, и тогда считается, что какого-то объекта нет в наших данных.

По сути **нейронная сеть представляет собой просто большое количество нейронов, объединенных в слои**. Существует **входной** слой, какой-то **скрытый** слой и **выходной** слой. Все это слои нейронов, и все эти нейроны принимают входные какие-то параметры, входные значения, перемножают их со своими весами и возвращают число дальше. Таким образом у нас нейронная сеть может моделировать иерархическую структуру, которая, собственно, и представляет собой наш мир. У нас очень много объектов представляют собой какую-то иерархию - подобные зависимости может моделировать нейронная сеть — чем она, собственно, и занимается, - путем суперпозиции разных нелинейных функций.

Что это за нелинейные функции, функции активации в нейроне? После того, как мы перемножили входные данные с нашими весами, все это дело сложили и добавили какое-то смещение, мы отправляем полученные векторы или тензор в функцию активации. Функция активации применяется в каждом нейроне. Они бывают разные. Одной из первых, одной из основных, является функция активации **sigmoid**, которая позволяет нейрону вернуть значение от 0 до 1.

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

Таким образом у нас все нейроны возвращают число от 0 до 1, если 1 то нейрон уверен в том, что присутствует объект какой-то во входных данных, и он зажжен.

Еще одной важной функцией активации является **гиперболический тангенс**, который позволяет распределять значения от -1 до 1.

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

Функцией активации, которая получает все больше и больше распространения и используется в большом количестве моделей сейчас, является нелинейная функция **rectified linear unit (ReLU)**

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

1.3 Обучение нейросети

Итак, как мы с вами знаем, у наших нейронов, которые составляют нашу нейронную сеть, есть разные веса. Эти веса настраиваются при обучении таким образом, чтобы у нас нейронная сеть выдавала нужный нам результат. Как вы знаете, при обучении с учителем у нас существует набор меток, которыми размечены входные данные, и мы пытаемся настроить нашу нейронную сеть, нашу модель так, чтобы она выдавала желаемый нам результат на наших данных. Однако в случае, например, линейной модели, линейной регрессии мы просто берем градиент и прокидываем его всем нашим весам. Все довольно просто и понятно: мы идем обратно по градиенту и оптимизируем нашу функцию потерь.

В данном случае у нас есть нейронная сеть. Мы действительно можем сделать **forward pass** так называемый, пройти от входных данных к выходному результату и получить какой-то ответ. Также у нас есть функция потерь, и мы можем узнать, насколько наш ответ отличается от эталонного. Однако у нас есть наша функция, функция потерь, мы можем посчитать ее градиент, но **как нам обновить наши веса у наших нейронов**?

На помощь нам приходит основной и главный метод обучения нейронной сети на данный момент - это **метод обратного распространения ошибки**, который с помощью **chain rule** - производной нашей функции потерь - пробрасывает обратно в предыдущие слои значение ошибки при обучении. Мы не будем подробно останавливаться на том, как это конкретно работает, можете почитать об этом отдельно. Тем не менее важно знать, что у нас, как и в любых других моделях машинного обучения, существует функция потерь, существуют методы градиентной оптимизации, которые позволяют прокидывать нашу ошибку обратно к нейронам и настраивать веса нейронов так, чтобы мы получали нужный результат. И потом применяем нашу нейронную сеть уже на тестовой, квалификационной выборке или в продакшне.

Итак, в нейронных сетях также используются методы градиентной оптимизации, как и практически везде в машинном обучении, однако существует несколько модификаций, которые работают чуть лучше. А так как у нас пространство в нейронных сетях, в данных, к которым применяются нейронные сети, очень большое — оно может быть, например, разреженным, может быть просто большой размерности, существуют разные модификации градиентного спуска, например, **Momentum** или **Adam**, которые лучше использовать, которые быстрее и лучше сходятся. В качестве значения по умолчанию, в качестве дефолтного метода можете, например, всегда использовать **Adam** или **RMSProp** — он работает почти всегда быстрее, чем обычный градиентный спуск.

Итак, мы с вами обучаем нейронную сеть. Однако нейронная сеть — это не линейная регрессия, обучать ее не так-то просто, очень часто мы сталкиваемся с разными проблемами при обучении. У нас что-то не сходится, что-то переобучается, например. Как же нам это дело победить, как нам улучшить нашу нейронную сеть, ее работу, ее обучение?

Существует достаточно много подходов, и **основной подход к улучшению работы нейронной сети — это просто добавить в нее больше данных**. Нейронные сети — это такие модели, такие архитектуры, которые требуют большого количества данных. Чтобы настроить параметры наших нейронов, нам нужно подавать данные, нужно их размечать, нужно их готовить, так чтобы нейронная сеть могла обучиться.

Как вы знаете, **нейронная сеть является универсальным аппроксиматором**, и, в принципе, нейронные сети достаточной глубины, достаточного размера могут смоделировать любую функцию, то есть любое распределение можно выучить с помощью нейронной сети. Однако для этого требуется огромное количество данных и огромной мощности нейронная сеть. У нас таких мощностей, конечно, нет, но тем не менее, добавляя новые данные, мы можем улучшать качество нашей модели.

Мы можем не только добавлять просто данные, **мы можем наши данные как-то модифицировать**. Например, классическим подходом аугментации (добавления) каких-то новых данных в случае, например, картинок может быть добавление в нашу модель не только классических картинок, а картинок, повернутых на 180 градусов или повернутых под каким-то углом, картинок, например, к которым добавлен определенный шум. Таким образом, мы можем добавлять модифицированные данные в нашу модель и улучшать ее качество, улучшать качество генерализации модели.

Также с развитием области генеративных разных моделей мы можем брать наши данные, которые уже существуют, и **как-то используя генеративные модели, создавать новые данные**. И эти данные уже использовать для обучения, для дообучения нашей модели. Существует огромная область **Semi-supervised learning**, которая говорит именно об этом, что мы можем просто взять размеченные данные, часть неразмеченных данных, их доразметить или, например, просто создать какие-то новые данные, новый пример.

Существует также много достаточно методов, которые позволяют улучшать качество работы нейронной сети и улучшать качество ее обучения. С некоторыми из них вы уже знакомы, например, **L1/L2 Регуляризация** позволяет добавлять дополнительные ограничения, дополнительные контрейнты на веса нейронов таким образом, чтобы они не были слишком большими, например.

Также одним из, наверное, самых важных, самых известных подходов к регуляризации является **Dropout**. Как вы знаете, у нас нейронная сеть состоит из нейронов, из разных слоев. Так вот, Dropout говорит о том, что мы при обучении можем выкидывать какие-то нейроны, например, на каждом слое мы можем выкидывать 50% случайных нейронов. И таким образом, наша нейронная сеть будет обучаться лучше. Достаточно странно: у нас нейронная сеть, например, состоит из тысячи нейронов, мы каждый раз выкидываем просто 500 из них, то есть как бы мы уменьшаем мощность нашей сети. Тем не менее, все равно все работает. Почему же так происходит? На самом деле тут есть несколько интуиций, и например, одна из них — при обучении таким образом, когда у нас каждый нейрон может быть в какой-то момент выключен, нейроны учатся не обуславливаться друг на друга. Они пытаются действительно выучивать реальные данные, а не зависеть от своих предшественников. А еще одна интуиция, которая может быть даже понятнее, это то, что у нас по сути, **когда мы обучаем нейронную сеть, где каждый нейрон может быть выкинут в какой-то момент, мы обучаем ансамблевую модель**. Как вы знаете, ансамблевые модели работают лучше. Если мы обучаем нейронную сети с Dropout, то по сути мы обучаем большое количество нейронных сетей с разными нейронами. Таким образом, при инференсе, при предсказании, при использовании нашей нейронной сети мы как бы усредняем ансамбль большого количества нейронных сетей. Таким образом, наша нейронная сеть лучше обучается, лучше работает.

Еще одним механизмом, которые стал популярен совсем недавно и применяется сейчас практически везде, является **Batch normalization**. Как вы знаете, входные данные нашей сети, нашей модели нужно нормализовать, чтобы данные были примерно одного порядка и чтобы с ними мо-

дели было легче работать. Так вот оказывается, данные нормализовать можно не только перед обучением, но и во время, например, между батчами. Об этом, собственно, и Batch normalization.

Ну и самый простой метод регуляризации так называемой - это **Early stopping, или ранняя остановка**. Он говорит о том, что в принципе, когда вы обучаете вашу модель, в какой-то момент можно остановиться, можно дальше не продолжать. Почему можно дальше не продолжать? Например, потому что падает качество на тестовой выборке. Вы обучаетесь, у вас качество на train улучшается, а на тесте внезапно начало падать, то есть у вас ваша модель переобучается. В этот момент вы можете остановиться или откатиться чуть назад и сказать, что вот, собственно, эти параметры, вот эти настроенные веса мы будем дальше использовать, нам можно дальше не обучаться.

Архитектуры нейронных сетей

2.1 Свёрточные сети

Давайте поговорим про **глубокое обучение, или про deep learning**, про которое так много пишут и говорят в последнее время. Что же это такое? Чем отличается от классических нейронных сетей, с которыми мы с вами уже работали?

На самом деле, глубокое обучение — это просто набор техник или методов по работе с теми самыми нейронными сетями, а **глубокой нейронной сетью** можно назвать любую сеть с большим количеством скрытых слоев (на самом деле, достаточно обычно двух, чтобы назвать сеть глубокой).

Итак, поговорим про **сверточные нейронные сети, или про convolutional neural networks**. Они были предложены Яном Лекуном в конце прошлого века и **призваны облегчить и улучшить работу с изображениями**.

Допустим, у нас стоит задача классификации изображения, то есть у нас есть какая-то картинка. Мы пытаемся понять, что на ней нарисовано: кошка, собака, слон, носорог или велосипед. Мы можем решать эту задачу с помощью обычной нейронной сети, с помощью feed forward нейронной сети, и ставить в соответствие каждому пикселю на входном слое какой-то нейрон. И последовательно пропускать эти данные.

Однако с ростом изображения, с ростом количества пикселей у нас будет расти количество нейронов. Значит, будет расти количество параметров нейронной сети последовательно, что не всегда здорово. Также в такой простой архитектуре явно никак не учитывается локальность, то есть то, что пиксели, которые находятся рядом друг с другом, обычно сильно связаны. Например, соседние пиксели в изображении высокого разрешения обычно примерно одного цвета. И в нашем изображении, в разных частях нашей картинки, могут быть очень похожие объекты, очень похожие текстуры, очень похожие формы.

Чтобы как раз строить и отлавливать иерархию таких форм и объектов, и были предложены сверточные нейронные сети. Сверточная нейронная сеть — это обычная нейронная сеть с новыми типами слоев. Эти слои сверточные и pooling слои. Как раз про них мы и поговорим.

Начнем со **сверточного слоя**. Сверточный слой представляет собой какой-то набор фильтров. Будем называть их фильтрами, также они называются, например, **features или kernel** — ядра, и они как раз непосредственно и применяются к нашему изображению с помощью операции свертки. Наш фильтр — это по сути просто матрица с числами. Именно эта матрица является параметром нашего сверточного слоя, именно она и будет настраиваться, обучаться в процессе оптимизации

нашей нейронной сети.

Итак, наша матрица, наш фильтр последовательно с каким-то шагом бежит по изображению и применяется, применяется с помощью операции скалярного произведения, то есть **dot product** двух матриц. У нас получается какое-то число. Таким образом наши фильтры последовательно применяются друг за другом к нашему изображению, получая какую-то **матрицу активации**. И при обучении нейронной сети наши фильтры постепенно, постепенно выучивают определенные формы, текстуры, фигуры или более сложные объекты.

Важно понимать, что даже обычное изображение обычно состоит не только из матрицы и пикселей, но и также у него есть какая-то глубина, например, количество каналов может быть три — RGB. Таким образом, наши фильтры могут быть не просто матрицей, а, например, тензором и так далее. Существуют также 3D свертки и более сложные моменты. Однако здесь все так же будет просто. У нас просто есть какой-то фильтр, и он с помощью dot product применяется к нашему изображению. И у нас получается какая-то матрица активации.

Также мы можем наш фильтр применять не с каким-то шагом один, а с более сложным шагом или добавлять какой-то **padding** к нашему изображению, то есть дополнять его по краям нулями, чтобы у нас что-то лучше работало. Наши фильтры могут быть различного размера. **В зависимости от размера нашего фильтра мы можем смотреть на разную область**, на разное количество пикселей на нашем изображении, на разное количество данных. Таким образом, постепенно двигаясь от входного слоя, мы можем улучшать, увеличивать или уменьшать область, которую наш фильтр видит.

Итак, при обучении нейронной сети мы настраиваем наши фильтры, наши ядра так, чтобы они распознавали определенные объекты. Что интересно, таким образом на первых слоях нашей нейронной сети обычно при обучении выучиваются какие-то простые моменты: цвета, текстуры либо просто линии или углы. Чем дальше мы движемся по нашей нейронной сети, тем более сложные иерархические сущности выучиваются, что логично, примерно так и работает наш мозг. Мы для начала выучиваем на нейронной сети какие-то просто линии, потом, например, в случае задачи классификации человеческих лиц мы можем выучивать глаза, рты и более сложные компоненты нашего лица. И на последних слоях у нас уже настоящие объекты. Это могут быть какие-то лица, автомобили или слоны. Таким образом у нас **строится иерархическая структура нашего изображения**. И наши фильтры используются в разных частях, для того чтобы распознавать различные части изображения, чтобы распознавать различные объекты, которые встречаются повсеместно.

Итак, и второй слой, про который мы поговорим, второй слой, который часто используется в сверточных нейронных сетях, более простой — это **pooling слой, или слой субдискретизации**, который по сути делает **downsampling, то есть уменьшение размера изображения**. Например, самый простой - max pooling - просто может уменьшить в два раза наше изображение, находя в каждом квадрате, в каждой матрице какое-то максимальное значение, максимальное число пикселя и оставляя именно его. Таким образом, мы просто уменьшаем качество изображения. Это очень полезно для того, чтобы уменьшать количество параметров нашей модели.

Однако в последнее время очень часто эти самые pooling слои заменяются разными трюками, например, можно делать вместо pooling слоя просто какую-то свертку с более большим шагом или можно, например, делать свертки даже один на один.

Итак, важно знать, что сверточные сети — это по сути обычные сети просто с новыми типами слоев. Сверточные слои выучивают фильтры, которые распознают различные объекты наших изображений. Также очень часто в конце сверточной нейронной сети добавляются **fully connected слои**, то есть обычная нейронная сеть, таким образом, чтобы потом, например, решить задачу классификации.

2.2 Рекуррентные сети

Теперь поговорим про **рекуррентные нейронные сети, или recurrent neural network**, которые позволяют работать с последовательностями данных, например с текстом, с видео или со звуком.

Допустим, если вы читаете какой-то роман, то, как вы воспринимаете слово, на которое вы сейчас смотрите, напрямую зависит от того, что вы прочитали до этого, зависит от предыдущего слова, от того, что было на предыдущей странице. Вы воспринимаете информацию в контексте.

Однако обычно нейронные сети никак не позволяют запоминать какой-то контекст, никак не позволяют хранить информацию с предыдущих шагов — они просто делают предсказания. Также они не очень хорошо предназначены, для того чтобы анализировать информацию последовательную, которая может быть неопределенной длины, как, например, какой-то текст.

Рекуррентные нейронные сети как раз позволяют вам хранить контекст, хранить информацию с предыдущих шагов. Например, если мы анализируем текст, нам на вход может подать какое-то слово, мы выдаем предсказание и запоминаем какую-то информацию про это слово. И именно эта информация, это состояние, этот state, передается потом, когда мы делаем следующее предсказание. Следующее предсказание мы делаем уже, учитывая информацию о предыдущем. То есть нам на вход поступает не только следующее слово, но и информация с предыдущего шага и так далее. Таким образом мы строим как бы цепочку, которая последовательно учитывает предыдущие шаги. Так и работают рекуррентные нейронные сети. Обучаются они тоже с помощью **backpropagation**: с помощью модификации **backpropagation through time**, которая как раз разворачивает все эти шаги.

Есть несколько важных модификаций к рекуррентным нейронным сетям, одна из которых называется **LSTM, или Long Short-Term Memory**. Они добавляют новые блоки в нашу рекуррентную нейронную сеть, чтобы она лучше запоминала разные состояния. Как вы можете догадаться, если мы читаем какой-то длинный текст с помощью нашей рекуррентной нейронной сети, постепенно мы, так как передаем информацию с предыдущих шагов, так или иначе забудем информацию о том, что было достаточно далеко. И рекуррентная нейронная сеть действительно страдает от такой проблемы — очень сложно помнить что-то, что было больше, например, 30 шагов назад.

На помощь могут прийти как раз **LSTM-блоки**, которые немного усложняют наше состояние — это уже не просто вектор, это уже чуть более сложный аппарат. Мы не будем подробно останавливаться на том, как это работает. Важно знать, что это то же самое состояние, только оно немного более сложно устроено. У нас есть определенные гейты, которые отвечают за то, стоит ли забывать какие-то данные, которые нам приходят на вход, стоит ли их, может быть, помнить, с каким весом их стоит учесть и так далее. Таким образом, эти LSTM-блоки тоже обучаются. Мы настраиваемся запоминать определенные вещи в нашем тексте. Например, если это слово с

заглавной буквы, возможно, стоит его запомнить, потому что это имя какого-то главного героя и так далее.

2.3 Современные архитектуры

Давайте обсудим несколько подходов и идей к тому, как современные модели строятся и в каких направлениях думают современные исследователи и ученые.

Первая модель, о которой мы с вами поговорим, - это **Autoencoder**. Autoencoder известны достаточно давно, однако они часто применяются и сейчас как часть каких-то более сложных моделей, и, в принципе, здесь есть несколько важных идей, которые нужно знать.

Autoencoder можно представлять себе как модель, которая состоит из двух нейронных сетей. Эти нейронные сети могут быть как обычными, так и сверточными, например, в зависимости от данных, с которыми мы работаем.

Допустим, у нас на входе есть какая то картинка, мы эту картинку отправляем в **encoder**, таким образом, чтобы наш encoder преобразовал нашу картинку, наши входные данные в какое-то скрытое представление, латентное. Причем так, чтобы вот это скрытое представление было меньшей размерности. Таким образом мы как бы сжимается наши данные с помощью encoder. А вторая сеть, которая называется **decoder**, эти данные разжимает, причем разжимает так, чтобы наши данные восстановились максимально точно, таким образом наша функция потерь как раз смотрит на то, чтобы наши данные восстановились точно без потерь. Причем важно заметить, что у нас здесь нет никакой разметки, это **unsupervized задача**. Что же здесь полезного и как потом это можно использовать?

Во-первых, с помощью такого Autoencoder используя нашу сеть, которая называется encoder мы можем сжимать наши данные, причем так как мы сжимаем так, чтобы эти данные потом можно было восстановить у нас в нашем латентном слове остается то самое-самое важное представление, которого достаточно для того, чтобы описать наши данные, это представление и можно использовать для того чтобы потом, например, обучать какие-то новые модели.

Также Autoencoder часто расширяется, и навешиваются дополнительные ограничения на латентный слой. Например, мы можем делать так, чтобы наш латентный слой представлял собой какое-то распределение, или можно пытаться строить какую-то арифметику - в общем как-то с этим работать. А так как в этом латентном слое содержится самая нужная выжимка информации, которая есть в данных, это очень удобно и часто полезно делать.

Следующая архитектура, про которую мы с вами поговорим это **Generative Adversarial Network (GAN)** (достаточно сложно перевести на русский, это состязательные сети какие-то) - это генеративная модель, которая позволяет генерировать новые примеры, новые данные.

Можно рассмотреть эту архитектуру с помощью такой мини-максной игры. Допустим, у нас есть фальшивомонетчики и есть полицейские. Фальшивомонетчики последовательно пытаются делать фальшивые монеты, а полицейские их ловить. Таким образом, если у нас есть какой-то фальшивомонетчик, который только пришел в дело, он вначале создает какие-то слабые фальшивые монеты, которые очень легко полицейские отлавливают. Тем не менее, наблюдая за тем, как полицейские отлавливают его монеты, фальшивомонетчик все лучше и лучше создает свои фейковые ненастоящие монеты, таким образом чтобы полицейские их все хуже и хуже распознавали.

Полицейские в свою очередь тоже смотрят на то как монета эволюционирует и постепенно находят новые трюки и новые способы ловить нашего фальшивомонетчика. Они друг с другом соревнуются. Наш фальшивомонетчик пытается создавать новые объекты так, чтобы полицейский не мог его поймать, чтобы он считал, что монета, которая лежит перед ним всегда реальная, несмотря на то, что эта монета могла бы придти от фальшивомонетчика. Примерно так и работают GAN-ы или Generative Adversarial Network.

У нас **есть две нейронные сети, которые друг с другом соперничают**. Одна сеть называется **генератором** - эта сеть генерирует данные, а есть сеть **дискриминатор**, которая пытается отличить эти данные от настоящих. Таким образом мы на вход нашему генератору подаем какой-то шум, можем подавать что-то еще, это в принципе неважно. Важно знать, что наш генератор пытается создать новый объект, а наш дискриминатор получает на вход данные от нашего генератора и данные из реального мира, например, это могут быть картинки: сгенерированные картинки и реальные картинки из dataset, и пытается понять, какая из этих картинок реальная, а какая из этих картинок сгенерирована генератором. Таким образом, наш дискриминатор в начале хорошо отличает реальные картинки от нереальных, потому что генератор для начала необучен и плохо генерирует. Однако, тем не менее постепенно соревнуясь наш генератор все лучше и лучше пытается обманывать наш дискриминатор и в идеале у него это должно получиться, и наш дискриминатор должен перестать отличать реальные примеры от сгенерированных.

Существует огромное количество модификаций, несмотря на то, что этой идее всего 3-4 года, есть очень большое количество моделей, которые используют в своей основе эту архитектуру. Это очень важная модель, которая применяется сейчас очень много где.

Следующим подходом, о котором хотелось бы поговорить, является **Seq2Seq** модели для рекуррентных нейронных сетей.

Допустим у нас есть задача анализа настроения в тексте. У нас, например, есть какое-то количество твитов про какое-то событие, и мы хотим понять: положительно или отрицательно они заряжены, то есть у нас есть какой-то дискретный ответ: 0 или 1. То есть у нас есть рекуррентная сеть, которая анализирует наш текст и выдает 0 или 1. Таких задач достаточно много.

Однако множество задач, которые решают рекуррентные нейронные сети, в качестве ответа представляют собой какую-то последовательность данных. Например, нам нужно произвести машинный перевод. У нас есть текст на русском языке - мы хотим перевести на английский, или у нас есть какой-то вопрос - мы хотим на него ответить. В таком случае в качестве результата работы нашей сети, нашей рекуррентной нейронной сети, потому что мы имеем дело с последовательностью, и является какая-то опять же последовательность символов, последовательность слов, и мы хотим это как-то моделировать с помощью нашей архитектуры, и на помощь нам приходит архитектура Seq2Seq. В таком случае у нас обычно есть какая-то сеть, которая называется энкодером, которая анализирует нашу входную последовательность и имбедит её в какое-то внутреннее представление, потом разворачивает последовательно с помощью рекуррентной сети, например, в перевод или в ответ.

Здесь существует огромное количество модификаций, мы можем добавлять какой-то attention. То есть например, какую-то еще одну сеть, которая выучилась смотреть на наши предложения и находить там самый важный момент, например какие-то слова с заглавной буквы и так далее. Мы можем разворачивать не из нашего внутреннего представления, а начинать разворачивать чуть раньше, можем по-разному передавать разные слои друг другу. В общем очень много модифи-

каций, однако важно представлять себе, что **рекуррентные нейронные сети часто именно Seq2Seq, то есть у нас часто на входе есть последовательность данных и на выходе есть последовательность данных**.

Модификации и улучшения этой идеи: доведением ее до такого экстрима можно представлять себе **Differentiable Neural Computer (DNC)** от компании Deep Mind, который был опубликован в журнале Nature.

Здесь у нас есть сеть, которая работает с ячейками памяти. То есть, у нас существует такой супер meta-attention, то есть у нас есть какая-то область памяти, с которой может оперировать наша нейронная сеть. Таким образом, при работе с данными, мы можем в эту память записывать, мы можем оттуда считывать, мы также можем смотреть, за тем как, например, в каком порядке мы записываем наши данные в эту память.

Важно здесь понимать, что у нас нейронные сети уже, не обычные какие-то сети с которыми мы с вами работали. То есть у нас в обычной нейронной сети вся информация обычно хранится в наших параметрах, в наших весах, нашей модели, там пропускаются градиенты и так далее или, например, у нас есть наши модули, где хранится какой-то скрытый state. **В DNC есть действительно настоящая память, которая выделяется, которая аллоциируется, которую можно освобождать, в которой можно писать, эту память можно перезаписывать**.

Следующей архитектурой о которой хотелось бы поговорить, которая совершенно не связана с предыдущими, является **ResNet**. Как вы наверняка знаете, существует соревнование ImageNet, которые проводятся с 2010 года, именно там в 2012 году был представлен AlexNet от Алекса Грижевски, который показал очень хорошее качество на своих задачах, который как раз и запустил бум нейронных сетей, которые мы с вами сейчас наблюдаем. И с тех пор каждый год улучшается качество работы сетей на этом dataset. Представляются новые архитектуры, где-то есть какие-то новые идеи, где-то просто увеличивается количество слоев. Хотелось бы разобрать одну из таких моделей, которая была представлена как раз на этом соревновании, которая называется ResNet.

И тут есть одна интересная важная идея, которая опять же используется очень много где. Обычно, когда мы смотрим на нейронную сеть мы, себе представляем поток данных, как просто какое-то последовательное преобразование посредством применения каких-то нелинейных функций, какой-то нормализации и так далее.

ResNet предложил дополнительно к такому потоку данных еще добавлять какой-то connection, который обходит эти преобразования напрямую. Таким образом, у нас существуют **res блоки**, который представляет собой преобразования: сверточные, нормализацию, функцию активации, однако параллельно с ней идет просто поток данных, которые никак не преобразовывались. Таким образом, в очень многих задачах сейчас существует несколько потоков данных так, что у нас один поток преобразуется с помощью каких-то операторов, а другой поток остается неизменным. Таким образом, мы можем **сохранять какую-то важную информацию из исходного слоя**.

Последняя архитектура, о которой хотелось бы поговорить это **Actor Critic** модели и архитектуры из области Reinforcement Learning.

Здесь у нас **Reinforcement Learning** - у нас есть, какая-то среда, у нас есть какой-то агент, который в этой среде работает. И здесь опять же существует огромное количество направлений в исследованиях.

Интересно обсудить такой подход, когда у нас опять же существуют две модели, две нейронные сети, которые друг другу помогают. Одна из моделей называется **актором**, другая - **критиком**. Актор пытается посмотреть на среду и пытается выбрать какое-то действие, а критик пытается эти действия критиковать или смотреть на то, как у нас распределяется ценность различных состояний в нашем пространстве. Таким образом, мы пытаемся выбрать действия. Актор у нас пытается найти какую-то политику, которую нужно придерживаться для того, чтобы максимально выгодно в этом окружении действовать. А критик смотрит на ценность различных состояний и тоже ему в этом помогает.

2.4 Введение в TensorFlow

Поговорим о том, как применять Python для обучения и для создания своих нейронных сетей. Сейчас существует несколько популярных фреймворков для решения таких задач. Мы с вами будем работать с *TensorFlow*.

Огромной популярностью в последнее время пользуются также *PyTorch*, который немного архитектурно по-другому устроен. Однако, принцип абсолютно тот же. Если вы понимаете, как работают нейронные сети, выучить соответствующий фреймворк *TensorFlow*, либо *PyTorch*, либо какой-то еще - это задача одной-двух недель. У них немного отличается синтаксис и парадигма, однако, в сущности, это те же самые математические модели, с которыми мы с вами познакомились.

Итак, давайте приступим.

```
import tensorflow as tf

hello = tf.constant('Hello world!')
```

Далее мы **запускаем сессию**.

```
sess = tf.Session()

result = sess.run(hello)
print(result)

sess.close()
```

Что же такое сессия? Все вычисления в *TensorFlow* происходят в рамках какого-то вычислительного графа. В этом графе есть определенный оператор и есть данные. Операторы передают данные друг другу. И таким образом у нас получаются вычисления. Вычисления в этом графе происходят в рамках определенной сессии. Эту сессию мы и создаем. Точно так же можно сделать только **с помощью стандартного контекстного менеджера**.

```
with tf.Session() as sess:
    result = sess.run(hello)
    print(result)
```

Давайте попробуем добавить немного параметров в наш вычислительный граф.

```
a = tf.constant(2)
b = tf.constant(3)

c = tf.constant([1, 2, 3, 4])
d = tf.constant([2, 3, 4, 5])
```

Воспользуемся контекстным менеджером и попробуем просто вывести наши переменные. Однако просто выводить переменные не так интересно. Нам интересно производить какие-то вычислительные операции на нашем графе. Именно это мы и сделаем.

```
with tf.Session() as sess:
    print('a = {}, b = {}, c = {}, d = {}\\n'.format(
        sess.run(a), sess.run(b),
        sess.run(c), sess.run(d)
    ))

    print('a + b = {}\n'
          'a * b = {}'.format(sess.run(a + b), sess.run(a * b)))

    print('c + d = {}\n'
          'c * d = {}'.format(sess.run(c + d), sess.run(c * d)))
```

Итак, в *TensorFlow* есть вычислительный граф, в который мы можем записывать определенные данные и добавлять операторы. Вычисляя наш граф, вычисляя определенный оператор, мы получаем результат.

Еще один важный тип, с которым нужно познакомиться, - это *placeholder*. Если в случае константы у нас сразу задано значение и оно никогда не меняется, потому что это константа, то *placeholder* говорит о том, что нам нужно добавить в наш вычислительный граф определенный объект, который потом получит свое значение - в момент выполнения нашего графа.

```
a = tf.placeholder(tf.int16)
b = tf.placeholder(tf.int16)

add = tf.add(a, b)
mul = tf.multiply(a, b)

with tf.Session() as sess:
    writer = tf.summary.FileWriter('logs', sess.graph)
    # > tensorboard --logdir logs/

    print('a + b = {}'.format(sess.run(add, feed_dict={a: 3, b: 1})))
    print('a * b = {}'.format(sess.run(mul, feed_dict={a: 7, b: 8})))
```

Также, обратите внимание, мы делаем дополнительную переменную *writer*, в которой записывается какая-то информация о нашей сессии, о нашем вычислительном графе. Вы можете посмотреть информацию о вычислительном графе с помощью программы *TensorBoard*. Запустите програм-

му *TensorBoard* в консоли и вы увидите, как выглядит ваш вычислительный граф. Вы можете с ним производить определенные операции, добавлять туда дополнительные значения и в принципе смотреть, как у вас выглядит ваш вычислительный граф, чтобы, например, отладить его работу.

Однако, конечно же, просто складывать и перемножать числа не так интересно. Давайте решим более сложную задачу. А конкретно, вначале разберемся с линейной регрессией. Так как, **вычислительный граф позволяет вам работать с разными сущностями - это просто парадигма вычисления, где у вас есть какие-то операторы, есть данные и вычисляется результат, то мы можем, конечно, не только обучать нейронные сети, а делать в принципе абсолютно все что угодно**.

Давайте импортируем уже знакомые вам библиотеки для визуализации и создадим задачи регрессии.

```
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

from sklearn.datasets import make_regression

n_samples = 42

x_train, y_train = make_regression(
    n_samples=n_samples, n_features=1,
    noise=15, random_state=7
)
x_train = (x_train - x_train.mean()) / x_train.std()
y_train = (y_train - y_train.mean()) / y_train.std()
```

Давайте попробуем вывести

```
print(x_train[:5])
```

Это просто числа, с каким-то шумом. Они позволят нам провести определенную прямую в этих точках.

Будем обучать с помощью градиентного спуска с определенным *learning_rate*, минимизируя функцию потерь, которую мы определилим ниже.

```
X = tf.placeholder('float')
Y = tf.placeholder('float')

W = tf.Variable(np.random.randn(), name='weight')
b = tf.Variable(np.random.randn(), name='bias')

prediction = tf.add(tf.multiply(X, W), b)

learning_rate = tf.placeholder(tf.float32, shape=[])
```

```
cost = tf.reduce_sum(tf.pow(prediction - Y, 2)) / n_samples
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Давайте проинициализируем глобальные переменные. Это нужно всегда делать для того, чтобы, например, наши *variable* приняли параметры, которые мы им указали.

```
init = tf.global_variables_initializer()
```

И осталось только **запустить нашу сессию и обучать нашу модель**.

```
epochs = 1000
sess = tf.Session()
sess.run(init)

lr = 0.1
for epoch in range(epochs):
    for (x_batch, y_batch) in zip(x_train, y_train):
        sess.run(optimizer, feed_dict={X: x_batch, Y: y_batch, learning_rate: lr})

    if epoch % 100 == 0:
        lr /= 2
        c = sess.run(cost, feed_dict={X: x_train, Y: y_train})
        print('Epoch #{}: cost = {}'.format(epoch, c))
```

Давайте запустим нашу модель и, как вы видите, у нас пошло вычисление.

Итак, еще раз, у нас есть сессия и есть наш вычислительный граф. Чтобы нам обучить нашу модель, мы запускаем наш оптимизационный алгоритм на функции потерь. Функция потерь определена так, что минимизирует нашу квадратичную ошибку предсказания. Предсказание, в свою очередь, строится на основе линейной модели так, как мы с вами разбирали ранее.

Итак, наша модель обучилась. Давайте **посмотрим, как же хорошо она описала наши данные**.

```
plt.plot(x_train, y_train, 'ro', label='Original data')
plt.plot(x_train, sess.run(W) * x_train + sess.run(b), label='Fitted line')
plt.legend()
plt.show()
```

Как видите, наша линия достаточно неплохо описала данные - это, действительно, уже обученная линейная регрессия.

Давайте не забудем закрыть сессию.

```
sess.close()
```

Таким образом *TensoFrlow* позволяет вам описать вычислительный граф, или Computational Graph, в который вы можете добавить данные и добавить какие-то операторы. Эти операторы могут передавать данные друг другу. Операторы могут зависеть друг от друга. Таким образом, вы строите вычислительный граф, который приходит к определенному результату. Чтобы запустить этот граф вам нужно создать сессию и запустить граф в рамках сессии. Таким образом, вы можете обучать вашу модель.

2.5 Классификация изображений на Tensorflow

Продолжим знакомство с *Tensorflow* и обсудим, как с помощью этой библиотеки решать задачу классификации изображений с помощью нейронной сети.

Загрузим dataset mnist. Mnist представляет из себя набор рукописных номеров от 0 до 9. Именно их мы и будем классифицировать.

```
import matplotlib.pyplot as plt
import tensorflow as tf

from tensorflow.examples.tutorials.mnist import input_data

%matplotlib inline

mnist = input_data.read_data_sets('/tmp/data/', one_hot=True)
```

Давайте посмотрим как выглядят наши данные.

```
image = mnist.train.images[7].reshape([28, 28]);
plt.gray()
plt.imshow(image)
```

Это просто картинки 28x28 черно-белые. Мы будем пытаться их распознавать с помощью нейронной сети.

Давайте **определим некоторые параметры нашей модели**. У нас будет нейронная сеть из двух скрытых слоев.

```
learning_rate = 0.1
epochs = 1000
batch_size = 128

n_hidden_1 = 256
n_hidden_2 = 256
num_input = 784 # 28 x 28
num_classes = 10

X = tf.placeholder('float', [None, num_input])
Y = tf.placeholder('float', [None, num_classes])
```

Давайте **определим параметры нашей нейронной сети, ее конфигурацию**. Как вы знаете, нейронная сеть представляет из себя какое-то количество нейронов, которые передают друг другу информацию. Один нейрон обладает набором весов и, получая информацию на вход, он перемножает полученный вход с весами, прибавляет bias turn и отправляет это все дальше.

```
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'output': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
```

```
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'output': tf.Variable(tf.random_normal([num_classes]))
}
```

Таким образом мы определили конфигурацию нашей сети - наши веса. Мы их случайно инициализировали, и потом наша нейронная сеть при обучении будет их как-то изменять, чтобы получать корректные предсказания.

Давайте определим функцию *network*, которая будет в нашем вычислительном графе как раз оперировать.

```
def network(x):
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    output_layer = tf.matmul(layer_2, weights['output']) + biases['output']\n",

    return output_layer
```

Определили такую функцию - именно она и будет нашей нейронной сетью. Именно ее мы будем обучать. Однако, как же мы ее будем обучать? **Нам нужна какая-то функция потерь. Давайте ее определим**.

Мы будем здесь минимизировать кросс-энтропию, потому что мы будем решать задачу логистической регрессии. То есть мы будем пытаться сделать так, чтобы наше распределение над классами, которое мы предсказали, максимально соответствовало реальному распределению.

```
logits = network(X)

loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits_v2(
        logits=logits, labels=Y
    )
)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train = optimizer.minimize(loss)

correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

init = tf.global_variables_initializer()
```

Давайте начнем **обучать нашу нейронную сеть**. Конечно, мы инициализируем сессию, инициализируем все наши переменные со случайными значениями, начальными и будем в цикле обучать нашу нейронную сеть по эпохам.

```
with tf.Session() as sess:
    sess.run(init)
```

```
for epoch in range(epochs):
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    sess.run(train, feed_dict={X: batch_x, Y: batch_y})

    if epoch % 50 == 0:
        train_accuracy = sess.run(
            accuracy,
            feed_dict={
                X: mnist.train.images,
                Y: mnist.train.labels
            }
        )

        print('Epoch #{}: train accuracy = {}'.format(epoch, train_accuracy))

print('Test accuracy = {}'.format(
    sess.run(
        accuracy,
        feed_dict={
            X: mnist.test.images,
            Y: mnist.test.labels
        }
    )
))
```

Давайте запустим наше обучение. У нас пошло обучение. Давайте пока пробежимся еще раз и посмотрим, что здесь происходит. У нас в цикле по эпохам мы бежим и каждый раз делаем одно и то же. Мы достаем следующий *batch*, то есть следующий набор объектов, отправляем их в наш вычислительный граф. Вычислительный граф у нас тренируется с помощью объекта *train*, объект *train*, соответствует оптимизатору *AdamOptimizer*, который минимизирует функцию потерь. Для того, чтобы нашу функцию потерь минимизировать, мы должны передать наши данные, прокинуть их сквозь наш вычислительный граф и получить какое-то значение. Чтобы посмотреть, как хорошо наш вычислительный граф, наша модель работает, мы вычисляем метрику *accuracy* и передаем ее тоже в запуск сессии.

Посмотрим, обучалась ли наша модель. Да! Как видите у нас тестовая accuracy 80%, что достаточно неплохо. В принципе с другой инициализацией, если бы мы перезапустили бы наше обучение нейронной сети, скорее всего мы могли бы получить даже больше. Однако, и для начала это уже неплохо.

Существует, также дополнительная надстройка над *TensorFlow*, которая называется *Keras*. *Keras* изначально был отдельный библиотекой, однако сейчас он встроен в *TensorFlow* и позволяет производить операции над графами *TensorFlow* в более упрощенном виде, то есть в более простом синтаксисе. Очень часто это бывает полезно. *Keras* появился в версии 1.4 и если у вас версия *TensorFlow* больше, вы можете использовать эту надстройку для того, чтобы чуть-чуть попроще строить базовые нейронные сети. Давайте посмотрим как это делается.

Будем решать ту же самую задачу, тоже будем работать с mnist - этот dataset тоже есть в *Keras*. Мы здесь загрузим наши данные, немного их преобразуем и будем работать с ними точно так же, как работали с предыдущей задачей.

```
batch_size = 128
num_classes = 10
epochs = 2

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255

print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
```

Однако, обучение нейронной сети и то, как мы специфицируем ее конфигурацию, ее архитектуру, здесь немного отличается. У нас наша нейронная сеть будет последовательностью определенных слоев, как, собственно, все нейронные сети.

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(num_classes, activation='softmax'))
```

И как вы видите, мы просто последовательно добавляем какие-то слои. Мы не определяем функцию *network*, мы не пишем какой-то класс. Мы просто последовательно добавляем уже описанные слои из нашей библиотеки - и часто это бывает удобно, потому что мы просто хотим построить как лего, какую-то определенную архитектуру.

Давайте запустим нашу модель, и для того чтобы ее скомпилировать и запустить, нам уже не нужно писать какие-то циклы, нам не нужно инициализировать сессии, все достаточно просто, все намного проще. **Для начала скомпилируем нашу модель, то есть зададим параметры этой модели и сохраним ее**.

Мы будем минимизировать кросс-энтропию, как и делали в предыдущий раз, будем использовать оптимизатор *Adam* и будем оптимизировать метрику *accuracy*.

```
model.compile(
    loss='categorical_crossentropy',
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy']
)

_ = model.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test)
)
```

Обратите внимание, что здесь мы видим уже интерфейс, который позволяет нам в более приятной форме смотреть на прогресс. И в принципе *Keras* - это как раз та самая библиотека, которая вам **позволит строить базовые модели, чуть более просто** чем *TansorFlow*.

Однако, *TansorFlow* **намного более гибкий и позволит вам чуть глубже погрузиться в архитектуру нейронной сети и изменить определенные моменты**: надстроить, написать свои собственные какие-то функции активации и так далее.

Давайте посмотрим на *accuracy* на тесте.

```
loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', loss)
print('Test accuracy:', accuracy)
```

Тестовая *accuracy* получилась здесь намного лучше - 0.97. Потому что у нас чуть более сложная нейронная сеть, и, возможно, мы чуть лучше инициализировали параметры.