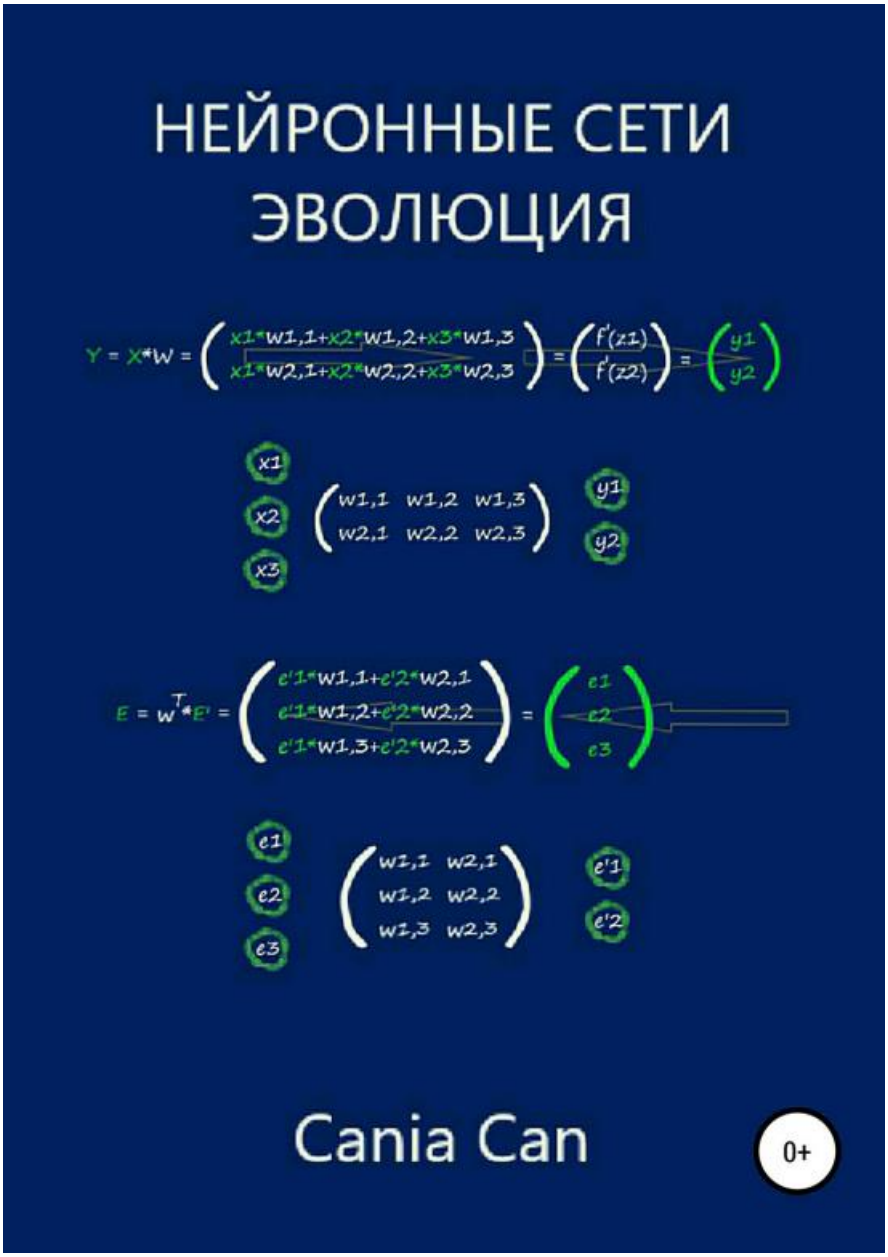
Каниа Алексеевич Кан
Нейронный сети. Эволюция



SelfPub; 2018

Аннотация

Эта книга предназначена для всех, кто хочет разобраться в том, как устроены нейронные сети. Для тех читателей, кто хочет сам научиться программировать нейронные сети, без использования специализированных библиотек машинного обучения. Книга предоставляет возможность с нуля разобраться в сути работы искусственных нейронов и нейронных сетей, математических идей, лежащих в их основе, где от вас не требуется никаких специальных знаний, не выходящих за пределы школьного курса в области математики.

Пролог

Технология искусственных нейронных сетей

С течением времени, по сегодняшний день, человечество сделало не мало для того чтоб приспособится самому и приспособить окружающий мир под себя. Было сделано немало научных открытий, инженерных изобретений, на основе которых создавались целые отрасли промышленности (машиностроение, энергетика, цифровые технологии и т.д.), которые значительно облегчили жизнь людей.

Но двигаясь вперед, все более актуален вопрос эффективного управления созданным хозяйством. Сегодня, для того чтобы человечеству хватило ресурса для освоения нового и развития уже созданного, требуется новые технологии, которые могли бы справиться с поставленной задачей более эффективно, значительно облегчая и даже заменяя труд людей.

Одной из таких технологий призвана стать – технология искусственных нейронных сетей, идея которой заключается в том, чтобы максимально близко смоделировать работу человеческой нейронной системы, так же эффективно обучаться и исправлять ошибки. Можно сказать, что главная особенность ИНС – способность самостоятельно обучаться и действуя на основании предыдущего опыта, с каждым разом делать все меньше ошибок.

Как пример применения ИНС, можно привести сферу охранного видеонаблюдения – где система искусственных нейронных сетей распознаёт присутствие людей в ненадлежащих зонах, забытые вещи, идентифицируя по лицу личность человека, отпечаткам пальцев и т.д. Ну а об автопилоте в автомобиле думаю наслышаны все, уже сегодня они колесят на просторах дорог в разных странах, пускай хоть и пока в качестве эксперимента, но это уже реальность! Конечно же, это далеко не всё чем ограничиваются искусственные нейронные сети. Их возможности поистине безграничны. Многие эксперты в сфере технологий, называют технологию ИНС – одной из ключевых технологий будущего.

Введение

Цель книги. Для кого она предназначена

Цель книги – объяснить, как устроены и работают нейронные сети, на простом и понятном, даже для школьника старших классов, языке!

Эта книга предназначена для всех, кто хочет разобраться в том, как устроены нейронные сети. Для тех читателей, кто хочет сам научиться программировать нейронные сети, без использования специализированных библиотек машинного обучения.

Книга предоставляет возможность с нуля разобраться в сути работы искусственных нейронов и нейронных сетей, математических идей, лежащих в их основе, где от вас не требуется никаких специальных знаний в области математики, не выходящих за пределы школьного курса.

Для улучшения восприятия информации, в книге сознательно избегается терминология, как например – персептроны, конволюция и так далее, так как приоритетом в данной книге является понимание принципа работы искусственных нейронных сетей, а не заучивание терминов.

Что мы будем делать

Самым разумным подходом для понимания развития технологии искусственных нейронных сетей, будет её биологическая интерпретация. Которая конечно же, в этой книге, не будет претендовать на историческую достоверность.

Мы будем представлять рождение и изменение искусственных нейронов и их взаимодействие между собой – по аналогии с эволюцией биологических видов. Как и в природе, движение от простейших организмов к более сложным, мы начнем с рождения

простейшего искусственного нейрона (с одним входом и выходом), используя для его работы (жизнедеятельности), простейший математический аппарат.

В дальнейшем, задачи, которые должен решать искусственный нейрон, будут становится сложнее. Для их решения нам потребуется эволюционировать наш созданный нейрон. Добавляя новые входы, или убирая ненужные, наподобие того, как это происходит в живой природе (например – отрастание или отмирание некоторых частей тела), вместе с тем модифицируя его математический аппарат.

Вносить изменения будем не кардинальные, опираясь на старые компоненты, будем постепенно, лишь слегка их модифицировать. Тем самым, действуя похожим образом как в реальных условиях, сама природа.

В процессе такой эволюции, созданные нами нейроны, научатся взаимодействовать между собой, объединяясь в сети.

Как мы будем это делать

Все сказанное будет подкрепляться теорией. Сначала на простейших принципах линейной функции, создадим наш первый искусственный нейрон. Подтвердим практически его работу – на языке Python выполним задачу по классификации, обучим наш нейрон, в результате чего, он самостоятельно проанализирует данные и классифицирует их. Тем самым максимально автоматизируя процесс классификации. Более того, подавая на вход обученного нейрона новые данные, которые он еще не видел, получим на выходе – верный ответ. Это будет наш первый искусственный интеллект!

Цифровой мир и живая природа очень многообразна. Для выживания в ней, необходима наилучшая приспособляемость к окружающей среде. В живой природе виды эволюционируют, в результате чего приобретают новые навыки и способности для выживания. Так же, когда нашему нейрону потребуется решать задачи, на решение которых, на текущем этапе своей эволюции, он не способен, то для его выживания в цифровом мире, ему тоже будут необходимы новые навыки. Осваивая новые математические принципы, лежащую в основе работы нашего будущего нейрона, мы будем на их основе его немного модифицировать.

Ну и конечно же, подкрепим всё практикой. Разработанные нами алгоритмы, будем применят на языке программирования – Python. Так как, новые математические алгоритмы – модификация предыдущих, то и здесь пойдем по пути постепенного изменения кода. В следствие внесения необходимых изменений в предыдущую программу на Python, и выполнив её, убедимся, что наш нейрон стал еще лучше выполнять предыдущие задачи, или вовсе приобрел способности к выполнению новых. В результате выполнения одной из таких программ, наш обученный нейрон сможет распознавать рукописные цифры! А это уже серьезно!

Все примеры, которые будут реализованы в Python, можно без труда скачать по следующей ссылке:

https://github.com/CaniaCan/neuralmaster

В дальнейшем, мы не раз повторим процесс эволюции к нашему искусственному нейрону. Добавим к нему множество входов и выходов, попутно добавим в его структуру условие – функцию активации. Соответственно узнаем, что такое функции активации, реализуем самые распространённые из них, такие как – единичная функция, сигмоида, RELU, гиперболический тангенс, Softmax.

Следующим этапом нашей эволюции, будет взаимодействие нейронов. Научим их общаться между собой. Или говоря иными словами – объединим в сети. Что в свою очередь, потребует новых навыков и знаний. Словом, теперь мы станем называть нейроны участвующие в её “жизнедеятельности”, нейронной сетью.

На основе таких сетей, на Puthon, напишем программу, способную распознавать рукописные цифры из большой базы данных – 60000 примеров рукописных цифр.

И наконец, мы создадим свёрточную нейронную сеть, и научим её, на той же базе, распознавать рукописные цифры.

ГЛАВА 1

Основа для создания искусственного нейрона

Где используются нейронные сети

Современные вычислительные машины выполняют математические операции с огромной скоростью. Решения различных арифметических и логических операций с числами – суть работы любого компьютера.

Сложение чисел с очень большой скоростью – это огромное преимущество компьютера над мозгом человека. Сложение больших чисел у человека вызывает затруднение, не говоря о скорости их вычисления.

Но есть задачи, с которыми наш мозг справляется куда эффективнее любого компьютера. Если мы взглянем на изображение ниже, то легко можем распознать что на нем изображено:

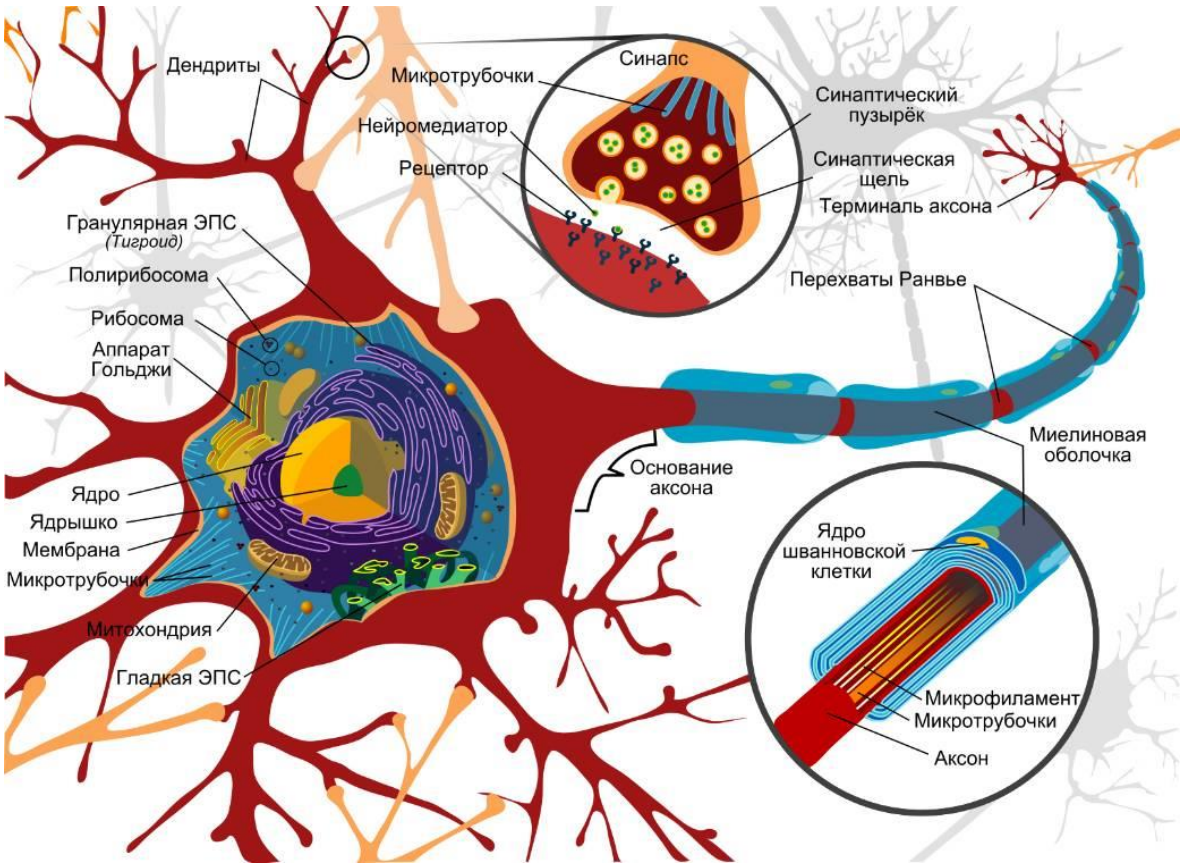
Вы без труда узнаете, что изображено на картинке, так как наш мозг идеальное средство для анализа изображения и его классификации. А вот компьютеру, напротив, очень трудно решать подобные задачи.

Но мы можем использовать вычислительные ресурсы современных компьютеров для моделирования работы мозга человека – искусственной нейронной сети.

Как устроены биологические нейронные сети

Что такое биологический нейрон и нейронные сети? У нас с вами и многих животных есть мозг. Мозг в свою очередь представляет собой сложную биологическую нейронную сеть, которая принимает информацию от органов чувств и обрабатывает её (распознавание слуховой и зрительной информации, распознавание вкуса, тактильных ощущений и т.д.).

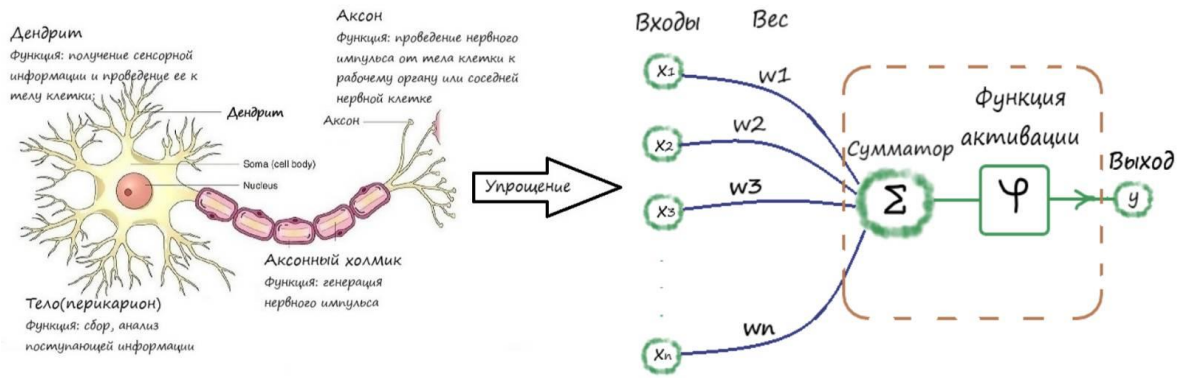
Строение биологического нейрона:



Собственно, эту биологическую модель нейрона мы и будем моделировать. А точнее нам понадобится смоделировать некую структуру, которая принимает на вход сигнал (дендрит), преобразовать этот сигнал по типу – как это происходит в биологическом нейроне, и передать преобразованный сигнал на выход (аксон).

Искусственный нейрон – математическая модель биологического нейрона.

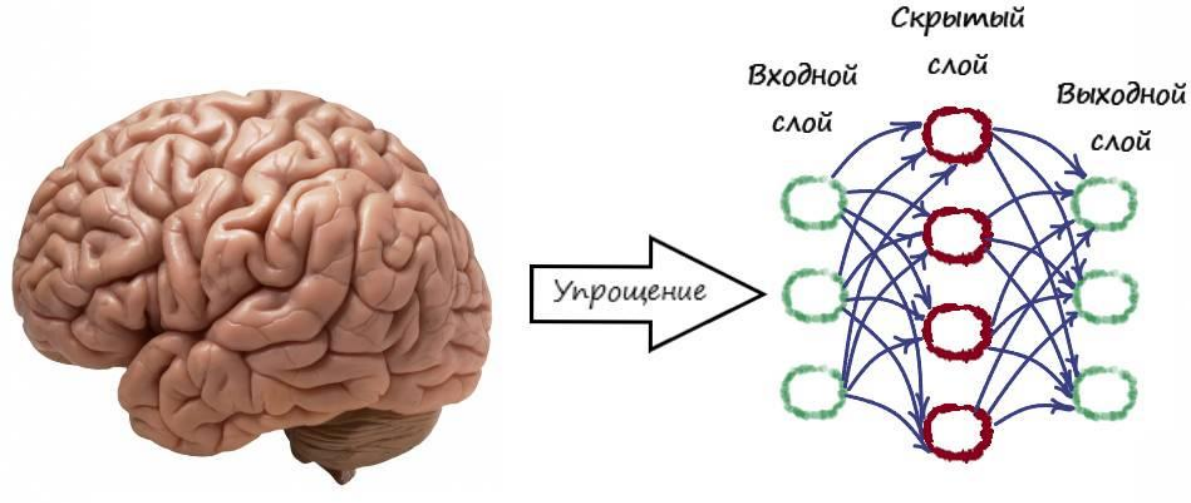
Модель искусственного нейрона (слева – биологический нейрон, справа – искусственный):



Наш мозг, как и любая биологическая нейронная сеть, состоит из множества нейронов.

В человеческом головном мозге насчитывается более 80 миллиардов нейронов, у каждого из который тысячи входов и выходов, и каждый из них соединен с входами других нейронов. И такую модель, в ограниченных объёмах, мы тоже с успехом можем упростить.

Переход к модели искусственных нейронных сетей:

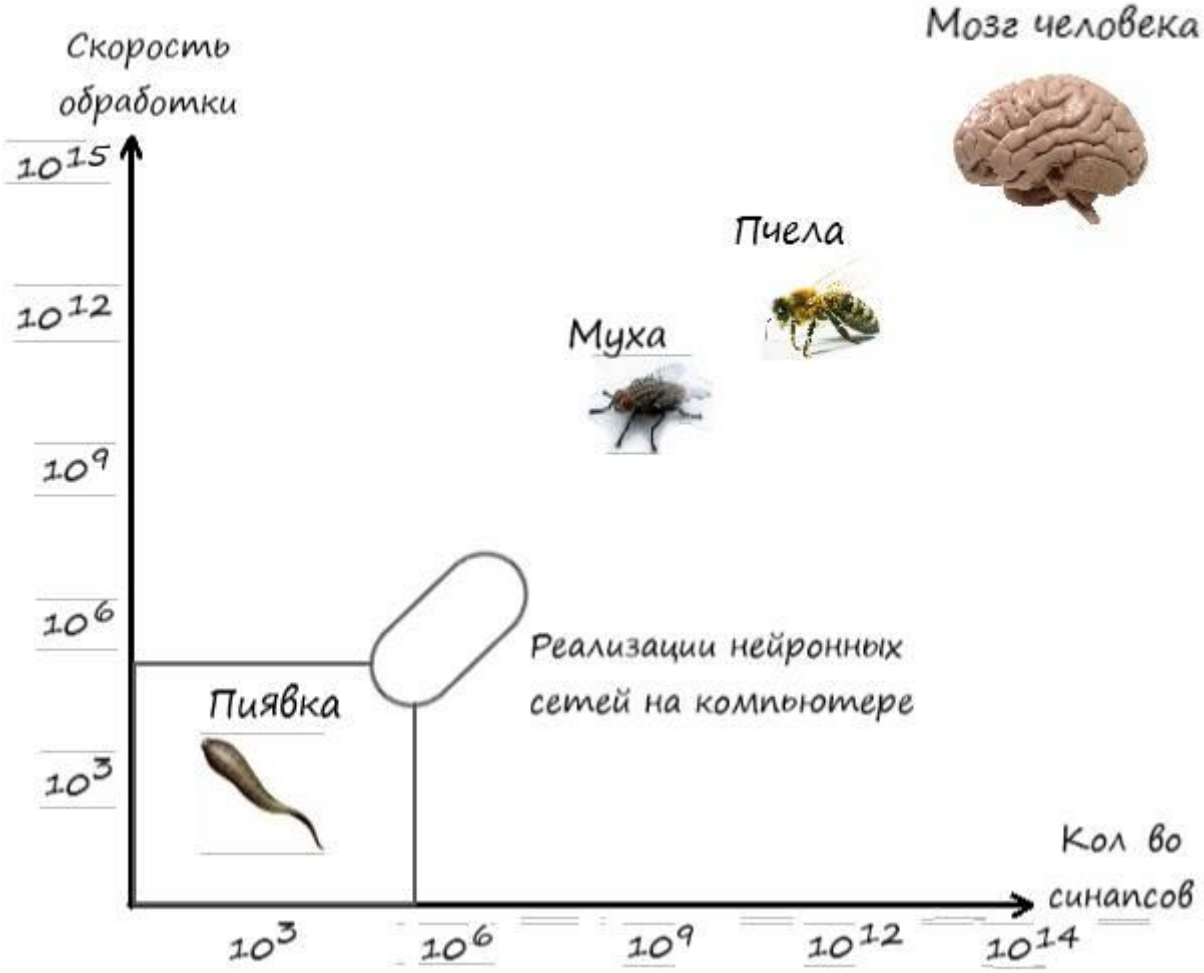


Уровень вычислительной мощности для моделирования ИНС

Мы уже знаем, что в мозге человека более 80 миллиардов нейронов, у каждого из который тысячи входов и каждый из них соединен с выходами других нейронов.

Смоделировать такой объём нейронов и количество их связей, мы на сегодняшний день не сможем. Но, мы можем упростить модель работы мозга, правда в гораздо меньших объёмах. Уровень вычислительной мощности современных компьютеров, при моделировании биологических нейронных сетей, как можно видеть на слайде ниже, немногим выше обычной пиявки.

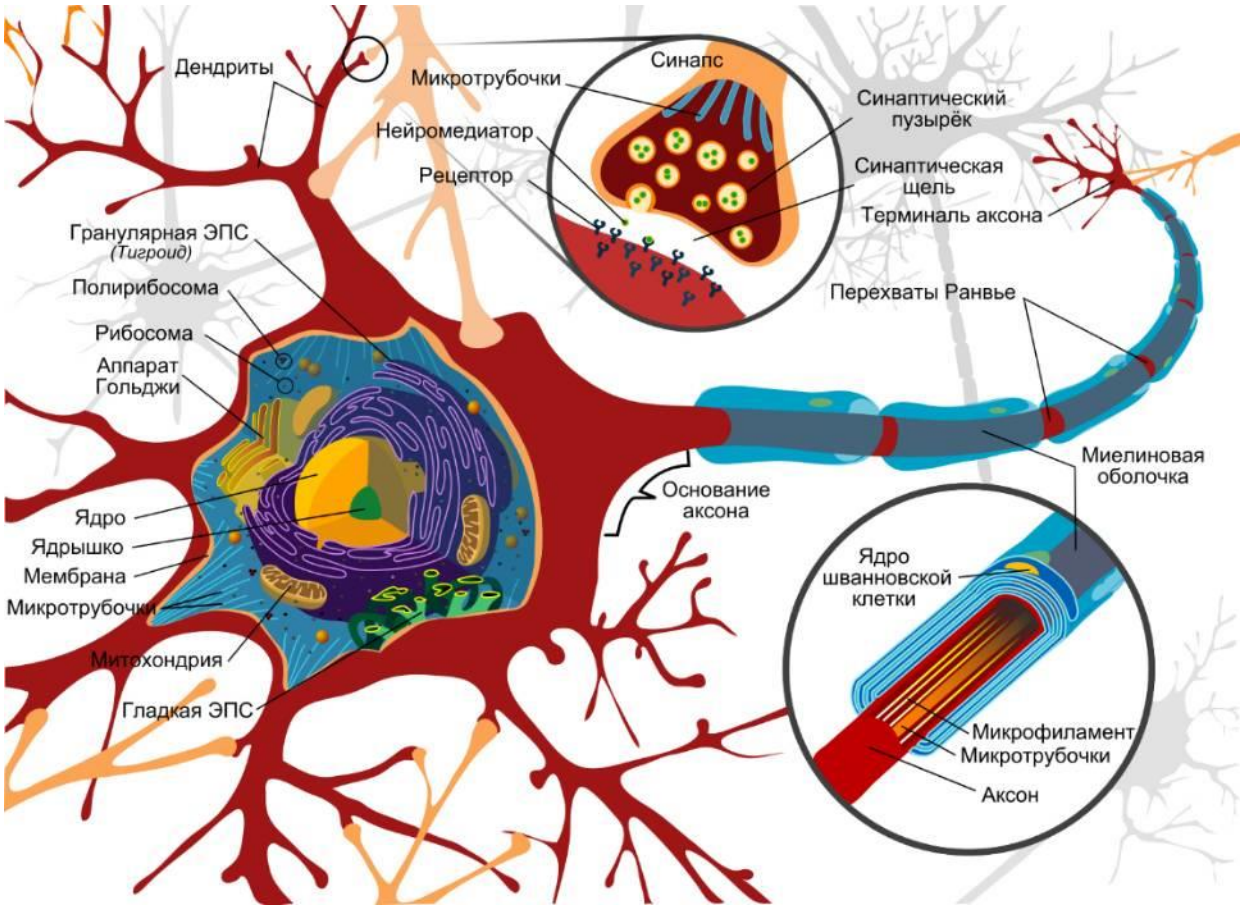
Насколько сильно мы уменьшаем количество нейронов и связей по сравнению с человеческим мозгом:



Как видите, до человека еще достаточно далеко. Но и этого объёма, что будет доступен, будет вполне достаточно для наших задач.

Почему работают нейронные сети

Весь секрет работы нейронных сетей заключается в работе синапсов, которые вы можете видеть на изображении биологического нейрона:



Синапсы – место стыка выхода одного нейрона и входа другого, где происходит усиление и ослабление сигнала. В усилении и ослаблении сигнала и происходит вся суть работы и обучения нейронных сетей. Если при обучении правильно подобрать параметры в синапсах, то входной сигнал, после прохода через нейронную сеть, будет преобразовываться в верный сигнал на выходе.

Все выше сказанное сейчас для вас представляется, лишь теоретической абстракцией и без практики очень трудным к осмыслению, но мы все разберем по полочкам – всю суть работы этого механизма. Действительно, на данном этапе невозможно понять, как работает нейрон, в чем смысл ослабления и усиления сигналов в синапсах, но информация, которую мы получили поможет нам в будущем, когда будем разбираться, что же всё-таки происходит внутри нейрона и нейронных сетях.

Как автоматизировать работу

Наверняка, многим из нас, порой до чёртиков, надоедало повторять одни и те же действия на работе или учёбе. В этот момент кажется, что ничего не может быть хуже каждодневной рутины.

Давайте включим воображение и представим себя офисным работником. Суть нашей работы – классификация данных на два вида. Каждый день, нам приходит список с данными, где может содержаться более 1000 позиций, которые мы самостоятельно должны отделить друг от друга, на основании чего сказать – какой из двух видов стоит за определенной позицией.

Итак, мы пришли на работу и видим на столе очередной список с данными, которые мы должны как можно быстрей классифицировать. А браться за работу, ох как неохота. Эх, если бы работа умела сама себя делать…

А ведь это мысль! Что если создать такую программу, которая многое из наших вакантных обязанностей, брала на себя. Сама с большой точностью, классифицировала

загружаемые в неё данные.

Всё это кажется фантастикой, но всё же реализуемо.

Логичней всего в первую очередь подумать, как это сделать с точки зрения математики. Ведь используя строгую математическую логику, мы поймём, как нам действовать, и добьёмся точных данных на выходе программы.

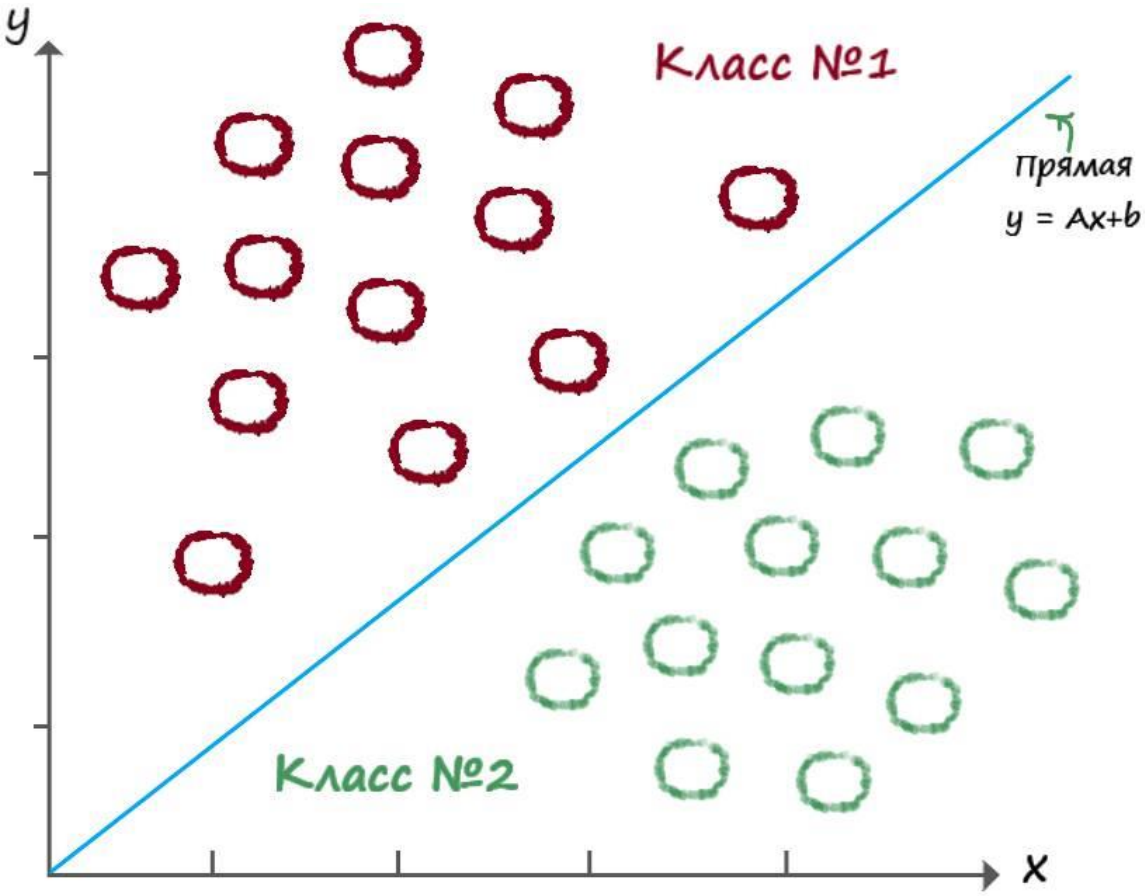
Ну как в любом начинании, нужно начать с самого простого.

Когда то, в младших классах, на уроке математики мы проходили линейную функцию:

$$y = Ax + b$$

Что если сделать так, что на числовых координатах, все данные которые будут находится выше линейной функции, будут принадлежать к одному классу, а ниже к другому. То есть функция прямой будет служить нам как классификатор.

Давайте покажем вышесказанное на слайде:



Отлично! Теперь осталось вспомнить что представляет из себя линейная функция.

Линейная классификация

Вспоминая школьный курс математики, из которого нам должно быть известно, что коэффициент **A** , в уравнении прямой, отвечает за её наклон. Чем больше значение коэффициента **A** , тем больше крутизна наклона линии. А коэффициент **b** – отвечает за точку начала координат по оси Y, через которую проходит прямая.

Раз мы еще толком не знаем, как будем действовать, давайте максимально всё

упрощать. Будем считать, что прямая проходит через начало координат и соответственно параметр прямой **b** , обратим в ноль: **b = 0** . Тогда окончательное выражение нашей разделительной линии, станет еще более простым:

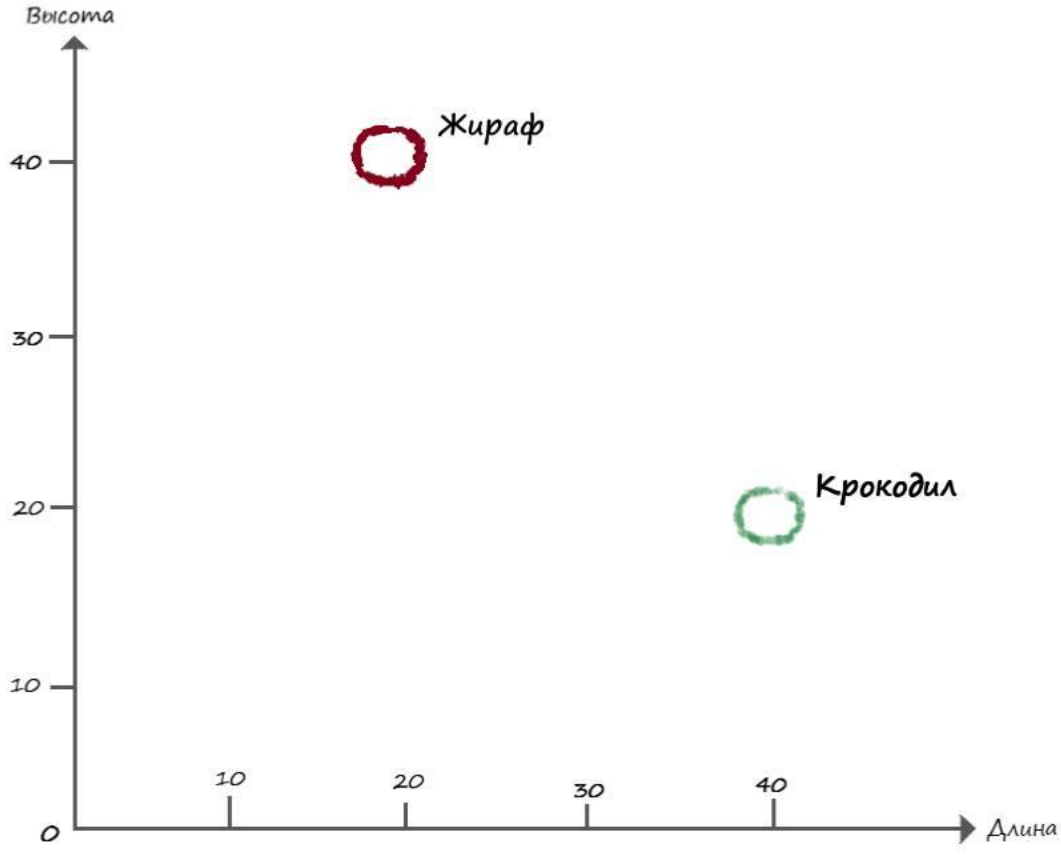
y = Ax

Пусть нашим заданием будет – классифицировать два вида животных, определенной возрастной группы, в два дня от роду, по размеру их тела – высоте и длине.

Для начала, подберем всего две выборки, которые разительно отличаются друг от друга:

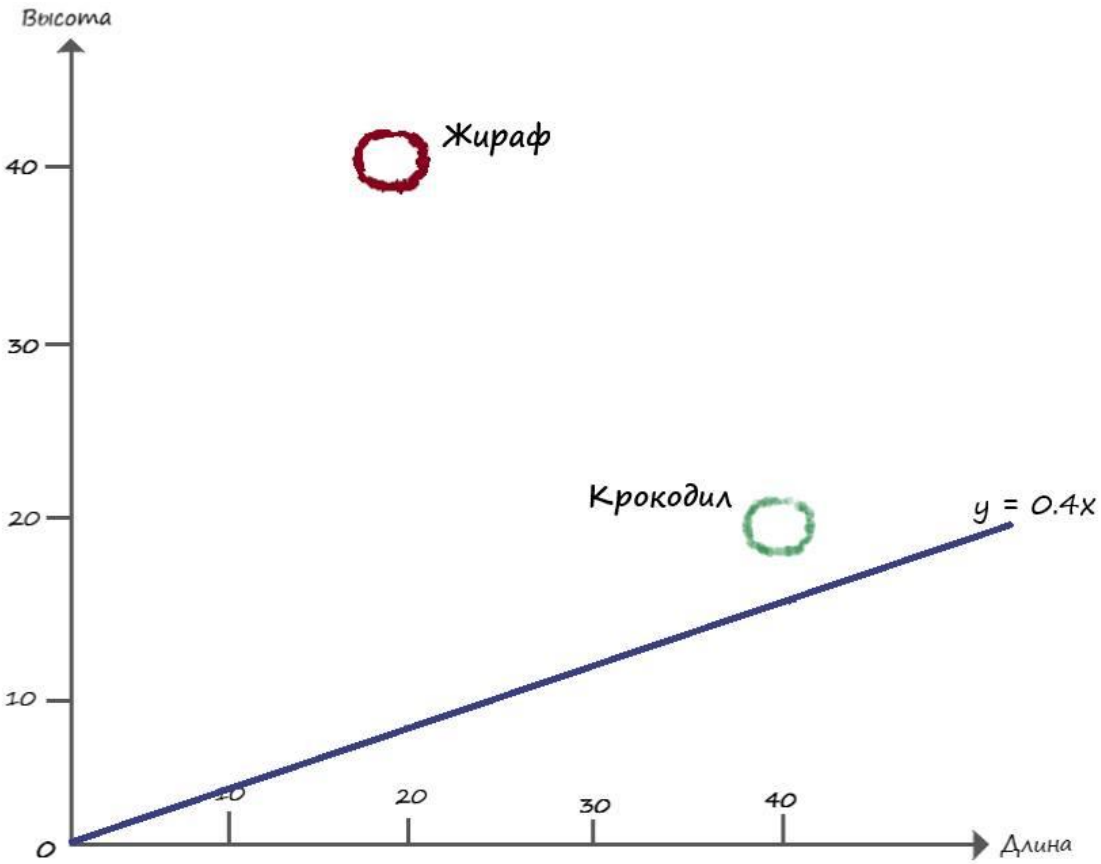
ПримерN°	Высота	Длинна	Вид животного
1	20	40	Крокодил
2	40	20	Жираф

Примем за **x** – значение длины, а за **y** – значения высоты. Визуализируем эти данные на числовой прямой:



Нужно придумать как разделить эти два вида линейной функцией. Попробуем мыслить последовательно.

Для начала, попробуем разделить наши данные случайной разделительной линией. Для этого примем значение коэффициента крутизны любым случайным числом, пусть **A = 0,4** . Тогда наше уравнение разделительной линии примет вид – **y = 0,4x .**



Как следует из графика, линия – **y = 0,4 x** , не отделяет один вид от другого. Для выполнения условия, её необходимо поднять выше. Для этого нам потребуется выработать последовательность команд и математические правила. Говоря иными словами, проработать алгоритм, когда при подаче данных из нашей таблицы (длины и ширины видов животных), в конечном итоге разделительная линия будет четко разделять эти два вида.

Теперь давайте протестируем нашу функцию на первом тренировочном примере, соответствующему виду крокодила, где: высота крокодила – 20, длина – 40. Не важно в чем будем измерять, в какой метрической системе. Самое близкое по условию это сантиметры. Но будем считать, что измеряем в условных единицах. Возьмём пример, где **x=40** (длинна=40), и подставив в него значение нашего коэффициента **A = 0,4** , получим следующий результат:

y = Ax = (0,4) * (40) =16

На выходе получили значение высоты **y = 16** , а верный ответ **y =20 .**

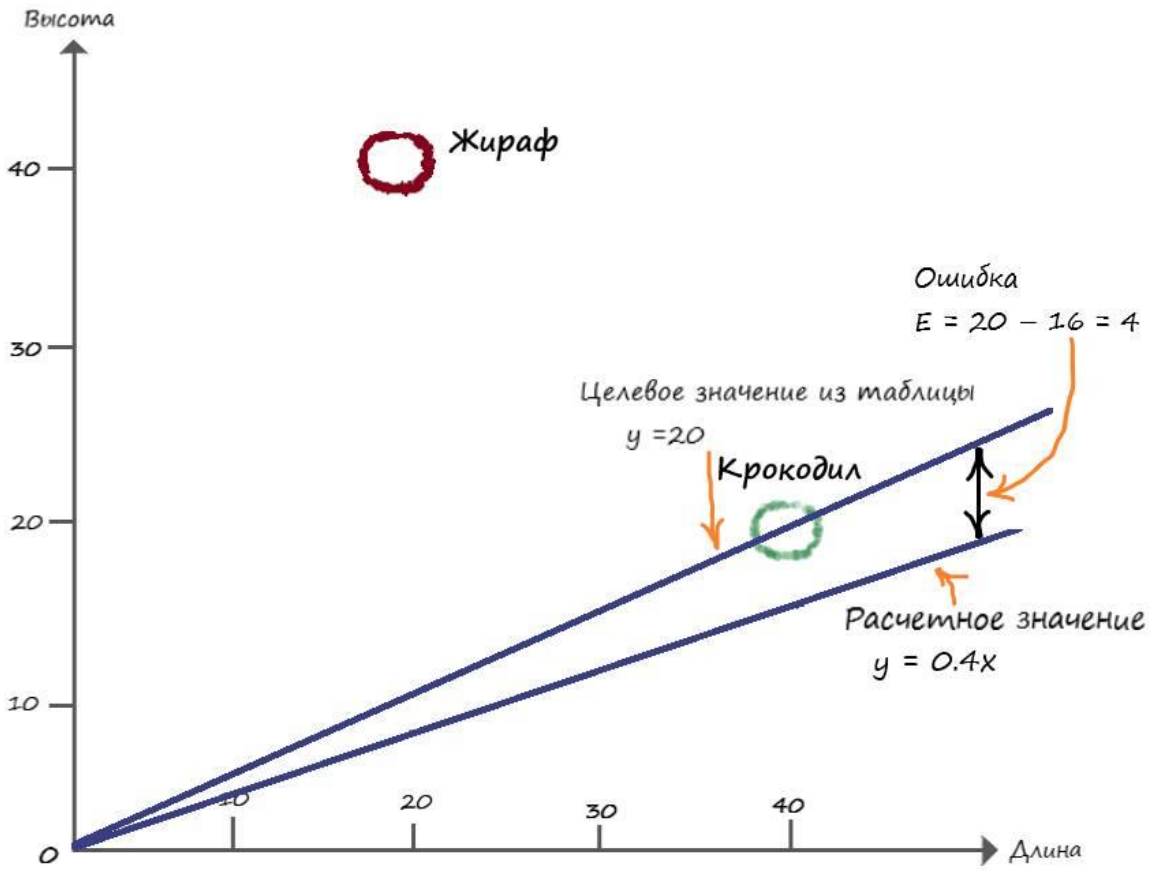
Для того чтоб исправить положение и приподнять нашу линию, введем понятие ошибки **E** , с помощью следующей формулы:

E = целевое значение из таблицы – фактический результат

Следуя этой формуле:

E = 20 – 16 = 4

Теперь давайте приподнимем нашу линию на 4 пункта выше и отобразим это на графике:

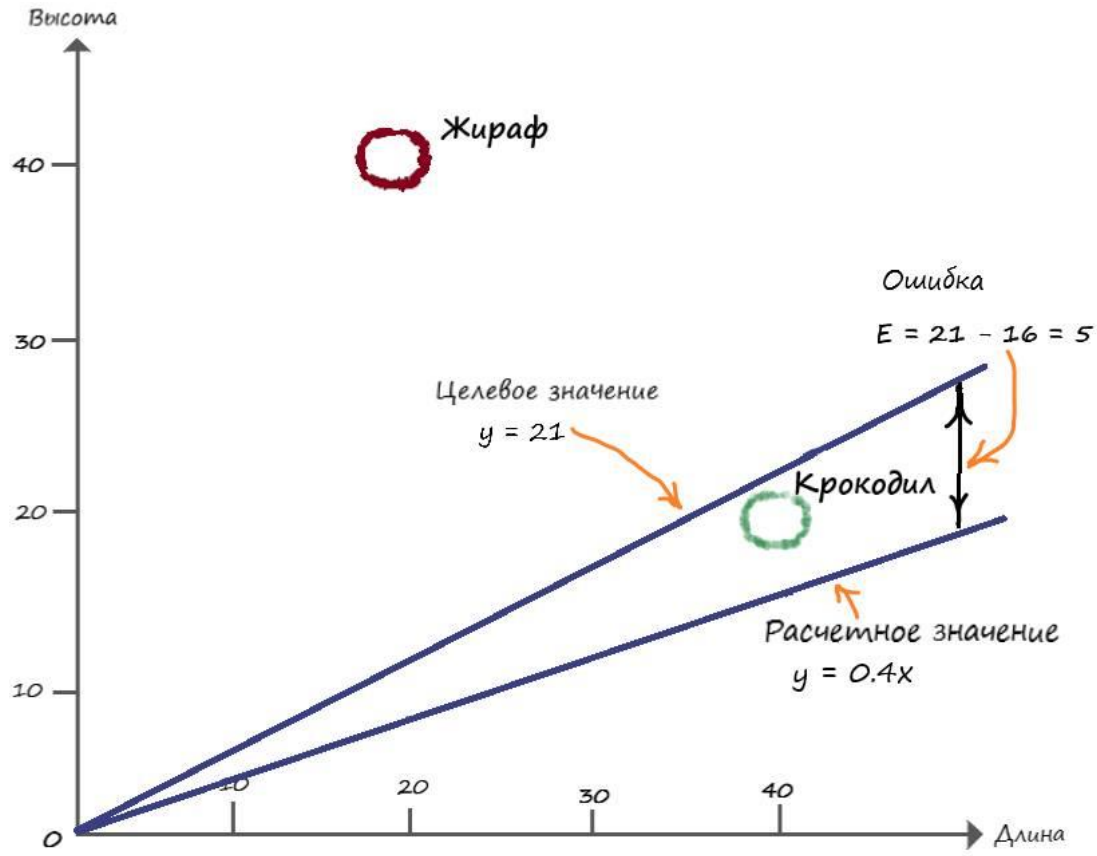


Ну и тут, как мы можем наблюдать, наша линия проходит через точку определяющую вид – крокодил, а нам надо чтобы линия лежала выше.

Решается эта проблема очень легко, давайте примем наши целевые значение чуть больше, положим высоту **y = 21** , вместо **y = 20** . И снова пересчитаем ошибку с новыми параметрами:

E = 21 – 16 = 5

Отобразим новый результат на координатах:



В итоге имеем новую прямую с новым значением коэффициента крутизны. Найдя этот коэффициент, мы как раз и сможем построить нужную нам прямую, на всех значениях оси x (длины).

Для этого нам необходимо через наше значение ошибки **E** , найти искомое изменения коэффициента **A** . Чтоб это сделать, нам нужно знать, как эти две величины связаны между собой, тогда мы бы знали, как изменение одной величины влияет на другую.

Начнем с линейной функции:

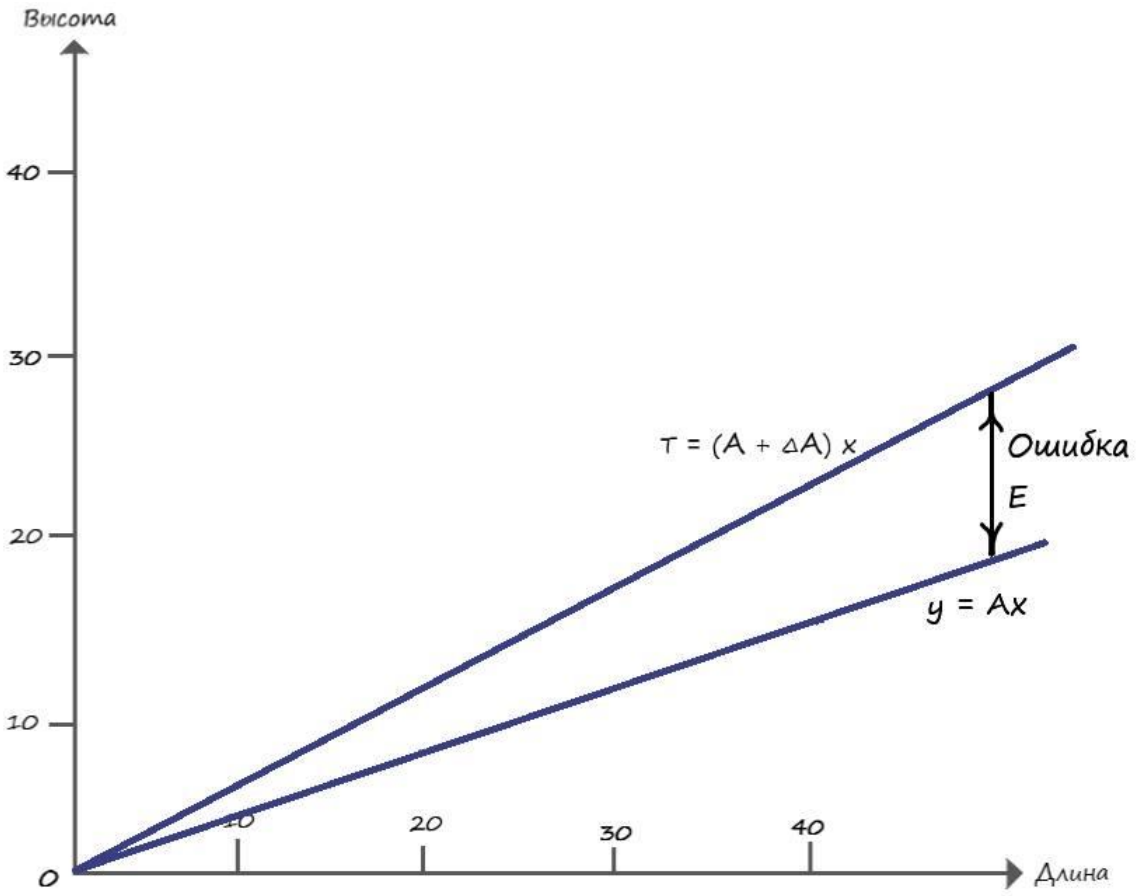
y = Ax

Обозначим переменной **T** – целевое значение (наше значение из таблицы). Если ввести в искомый коэффициент **A** , такую поправку как: **A+ΔA = искомое A .**

Тогда целевое значение можно определить, как:

T = (A + Δ A) x

Отобразим последнее соотношение на графике:



Подставим эти значения в формулу ошибки **E = T – y** :

E = T – y = (A + ΔA) x – Ax = Ax + (ΔA) x – Ax = (ΔA)x
E = (ΔA)x

Теперь зная, как ошибка **E** связана с **ΔA** , нетрудно выяснить что:

ΔA = E / x

Отлично! Теперь мы можем использовать ошибку **E** для изменения наклона классифицирующей линии на величину **ΔA** в нужную сторону.

Давайте сделаем это! При **x = 40** и коэффициенте **A = 0,4** , ошибка **E = 5** , попробуем найти величину **ΔA** :

ΔA = E/x = 5 / 40 = 0,125

Обновим наше начальное значение **A** :

A = A+ ΔA = 0,4 +0,125 = 0,525

Получается новое, улучшенное, значение коэффициента **A = 0,525** . Можно проверить это утверждение, найдя расчетное значение **y** с новыми параметрами:

y = A x = 0,525 * 40 = 21

В точку!

Теперь давайте узнаем на сколько надо изменить коэффициент **A** , чтоб найти верный ответ, для второй выборки из таблицы видов – жираф.

Целевые значения жирафа – высота **y = 40** , длина **x = 20** . Для того чтобы, разделительная линия не проходила через точку с параметрами жирафа, нам необходимо уменьшить целевое значение на единицу – **y = 39** .

Подставляем **x = 20** в линейную функцию, в которой теперь используется обновленное значение **A=0,525** :

y = Ax = 0,525 * 20 = 10,5

Значение – **y = 10,5** , далеко от значения **y = 39** .

Ну и давайте снова предпримем все те действия, что делали для нахождения параметров разделяющей линии в первом примере, только уже для второго значения из нашей таблицы.

E = T – y = 39 – 10,5 = 28,5

Теперь параметр **ΔA** примет следующее значение:

ΔA = E/x = 28,5 / 20 = 1,425

Обновим коэффициент крутизны **A** :

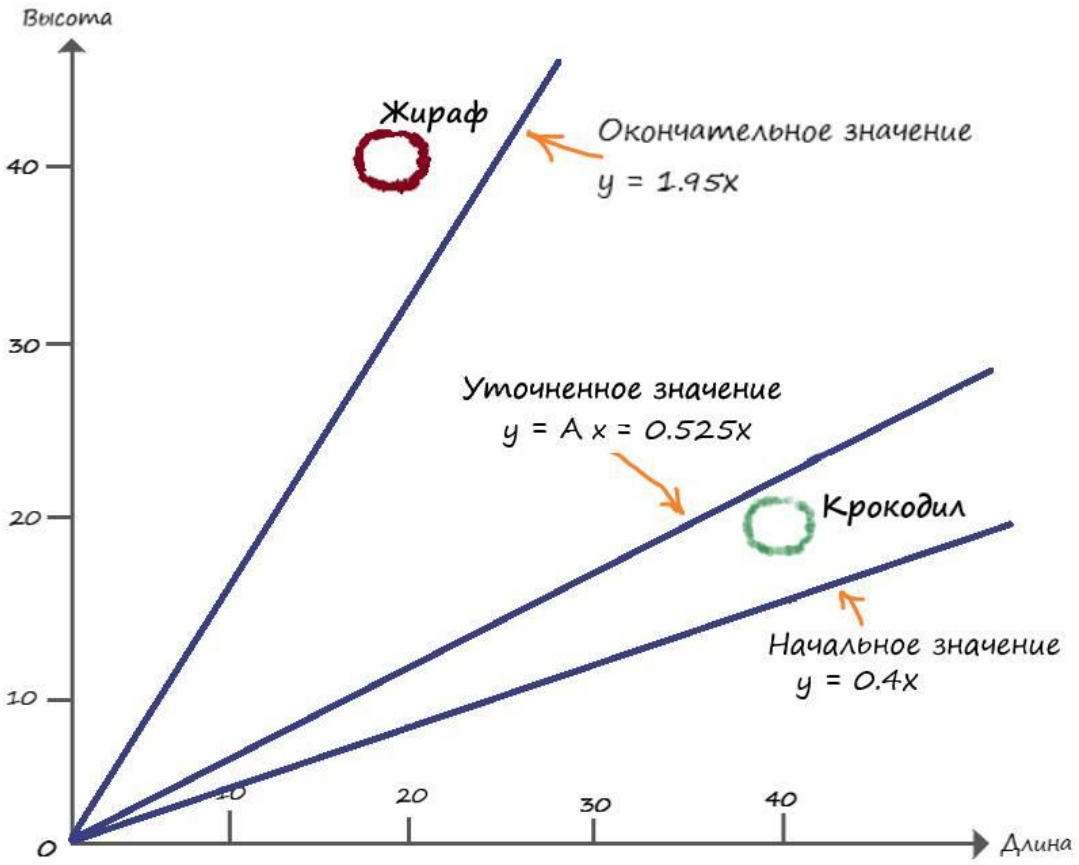
A = A+ ΔA = 0,525 +1,425 = 1,95

Получим обновленный ответ:

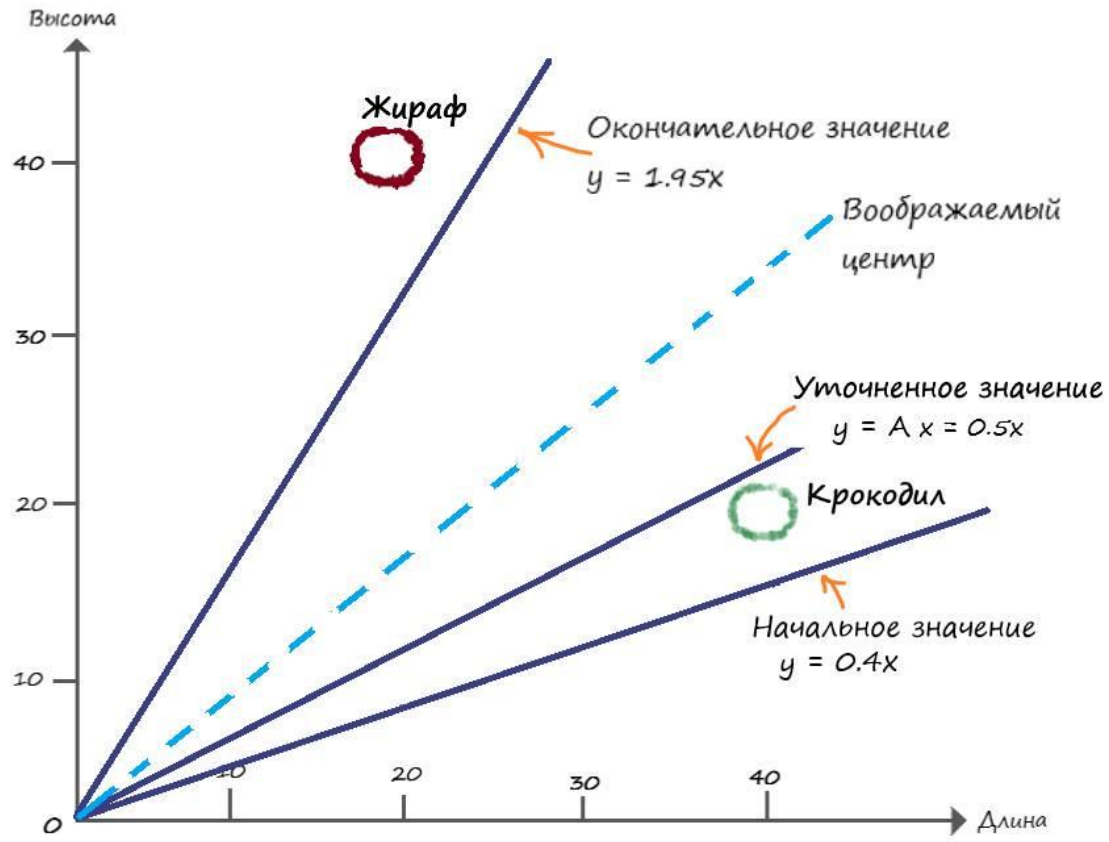
y = A x = 1,95 * 20 = 39

То есть, при **x = 20** , **A = 1,95** и **ΔA = 1,425** – функция возвращает в качестве ответа значение **39** , которое и является желаемым целевым значением.

Представим все наши действия на графике:



Теперь мы наблюдаем, что линия разделила два вида, исходя из табличных значений. Но полученная нами разделяющая линия лежит гораздо выше её воображаемого центра, к которому мы стремимся:



Но и это легко поправимо. Мы добьемся желаемого результата сглаживая обновления, через специальный коэффициент сглаживания – **L** , который часто называют как – **скорость обучения .**

Суть идеи: что каждый раз обновляя **A** , мы будем использовать лишь некоторую долю этого обновления. За счет чего, с каждым тренировочным примером, мы мелкими шагами будем двигаться в нужную нам сторону, и в конечном результате остановимся около воображаемой прямой по центру.

Давайте сделаем такой перерасчет:

ΔA = L * (E / X)

Выберем **L =0,5** в качестве начального приближения. То есть, мы будем использовать поправку вдвое меньшей величины, чем без сглаживания.

Повторим все расчеты, используя начальное значение **A=0,4** . Первый тренировочный пример дает нам **y = Ax = O,4 * 40 = 16** . При **x = 40** и коэффициенте **A = 0,4** , ошибка **E = T – y = 21 – 16 = 5** . Чтобы график прямой, не проходил через точку с нашими координатами, а проходил выше её, то принимаем целевое значение – **T = 21 .**

Рассчитаем поправку: **ΔA = L (E / x) = 0,5*(5 / 40) = 0,0625** . Обновленное значение: **A = A + ΔA = 0,4 + 0,0625= 0,4625 .**

Сглаженное уточнение: **y = Ax = 0,4625 * 40 = 18,5 .**

Теперь перейдем к расчетам следующего тренировочного примера.

Используя обновлённое на первом прогоне значение **A** , для второго тренировочного примера **y = Ax = O,4625 * 20 = 9,25 .**

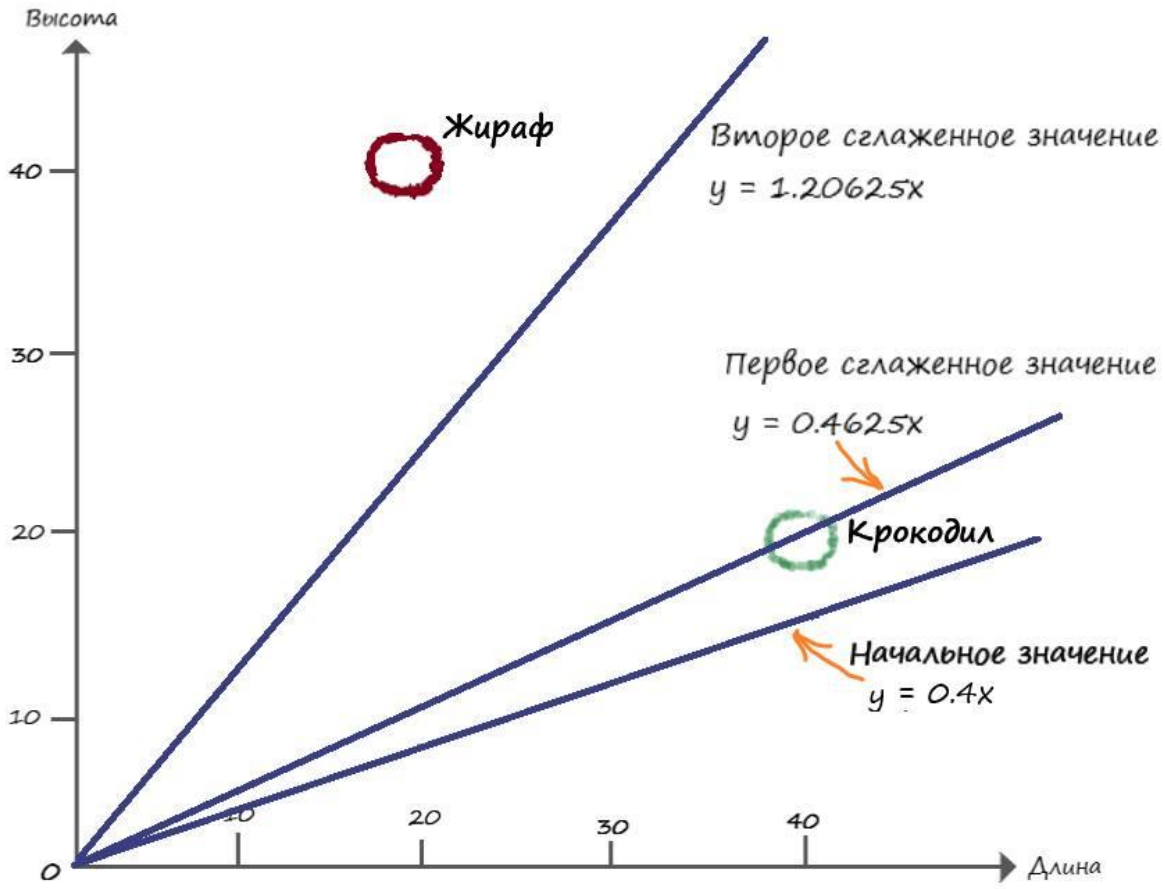
Значение, **y = 9,25** – всё так же далеки от значения **y = 39** , но мы все равно движемся в нужном направлении, но уже с меньшой скоростью.

При **x = 20** и коэффициенте **A = 0, 4625** , ошибка **E = T – y = 39 – 9,25 = 29,75 .** Так как мы хотим, чтобы график прямой, не проходил через точку с нашими координатами, а проходил ниже её, то принимаем целевое значение – **T = 39** . Рассчитаем поправку **ΔA = L (E**

/ x) = 0,5*(29,75 / 20) = 0,74375 . Обновлённое значение **A = A + ΔA = 0,4625+ 0,74375 = 1,20625** .

Сглаженное уточнение **y = = Ax = 1,20625 * 20 = 24,125** .

Теперь еще раз отобразим на координатной диаграмме, начальный, улучшенный и окончательный варианты разделительной линии:



Можно убедиться в том, что сглаживание обновлений приводит к более удовлетворительному расположению разделительной линии.

Если еще уменьшить скорость обучения **L** и повторить расчеты с первым и вторым обучающим примером, то в итоге наша разделительная линия окажется очень близко к воображаемой линии.

Применяя способ уменьшение величины обновлений с помощью коэффициента скорости обучения, ни один из пройденных тренировочных примеров, не будет доминировать в процессе обучения.

ГЛАВА 2

Изучаем Python

В этой главе мы будем создавать собственные нейронные сети. Сначала создадим модель работы искусственного нейрона, а затем научимся моделировать сеть из множества нейронов.

Создаем нейронную сеть на Python

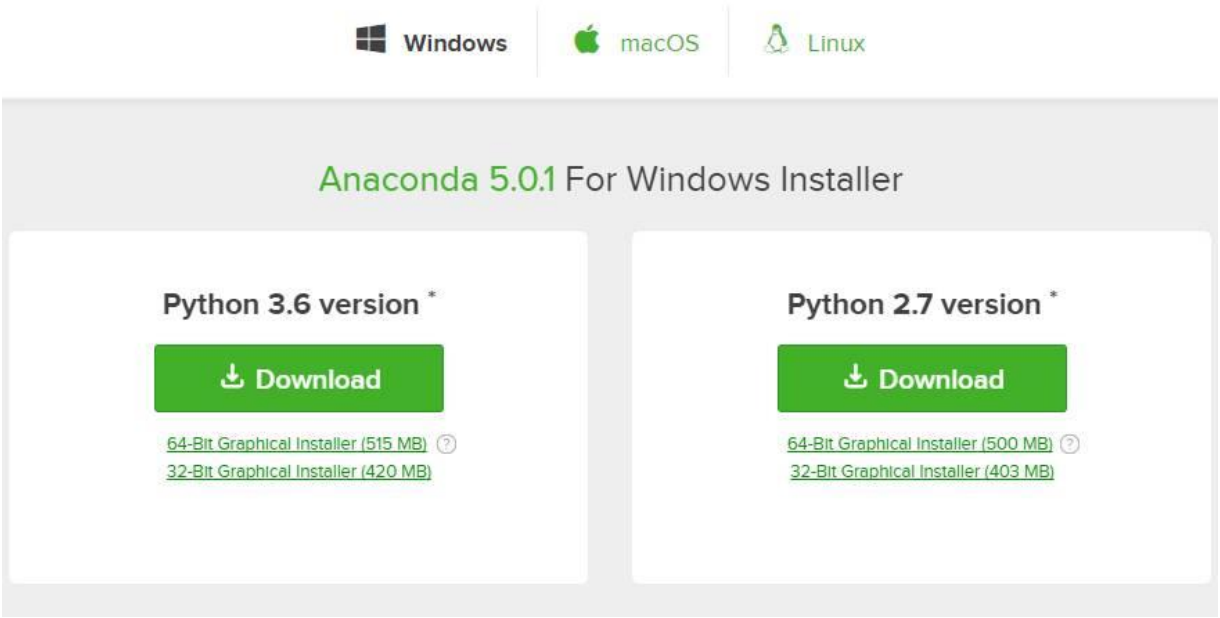
При моделировании нейронных сетей, мы будем использовать язык программирования Python.

Почему Python? Он очень прост в освоении, кроме того, нейронные сети создают и обучают в основном на этом языке. Кроме того, Python очень популярный и распространённый язык программирования.

О Python, можно рассказывать долго и много, но мы будем изучать Python лишь в том объеме, который необходим для достижения нашей цели – изучить работу нейронных сетей.

Установка пакета Anaconda Python

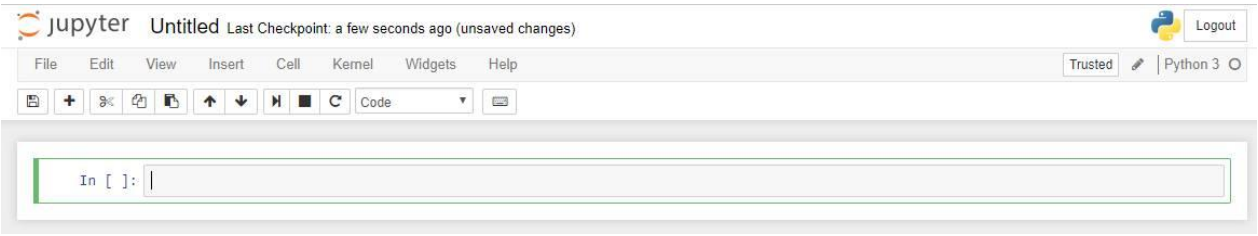
Посетите сайт – http://www.continuum.io/downloads, на котором предлагаются различные варианты установки Anaconda Python. Я использую пакет Anaconda, для операционной системы Windows, вы можете выбрать другие варианты – OS X или Linux. Пакет Anaconda предоставляет удобное средство интерактивной разработки Jupyter Notebook, в котором необычайно удобно писать и проверять программный код. На момент написания книги, доступен пакет Anaconda 5.0.1, и Python 3.6 – который и рекомендую установить.



Если, к тому времени, когда вы посетите сайт, все будет выглядеть иначе, не пугайтесь, сути дела это не поменяет.

Простое введение в Python

После установки пакета Anaconda, запустите интерактивную оболочку Jupyter Notebook, нажмите на кнопку New у правого края окна и выберите в открывшемся меню пункт Python 3, что приведет к открытию пустого блокнота:



Переменные

В переменных всегда что-то хранится (число, объекты, символы, строки). Попробуем создать переменную x со значением 20. И выведем это значение, на экран, при помощи функции – print(). Функция print() – выводит на консоль то, что расположено между её скобками:

С переменными, которые хранят числа, можно выполнять различные простейшие действия: складывать, вычитать, умножать, делить и возводить в степень:

```
x = 5
y = 3
print (x + y) #Сумма
print (x - y) #Разность
print (x * y) #Произведение
print (x / y) #Деление
print (x ** y) #**Возведение в степень
print (x // y) #//показывает в данном примере, сколько троек в пятёрке
print (x % y) #%возвращает остаток от деления
```

```
8
2
15
1.6666666666666667
125
1
2
```

Справа от функции print(), вы можете видеть комментарии. Делаются они очень просто, для этого, перед комментарием, необходимо поставить знак #, и текст после этого знака, в данной строке, Pytnon будет воспринимать, не как программный код, а как обычную текстовую область.

Кроме числовых переменных есть ещё строковые, с которыми мы тоже можем проделать ряд действий:

```
name_question = "Как вас зовут?"
my_name = "\nМоё имя:\nCania Can"

print(name_question, my_name)
```

```
Как вас зовут?
Моё имя:
Cania Can
```

```
num_1 = str (21) #str приводит переменную к строковому типу
num_2 = str (22) #str приводит переменную к строковому типу
res = num_1 + num_2 #В этом случае произойдет объединение строк
print (res, type(res)) #Функция type() возвращает тип переменной
num_1 = int ("21") #int приводит переменную к целому к числу
num_2 = int ("21") #int приводит переменную к целому к числу
res = num_1 + num_2
print (res, type(res))
```

```
2122 <class 'str'>
42 <class 'int'>
```

Функции

Иногда возникает необходимость повторять одни и те же действия, в ходе написания программы, по многу раз. Облегчить наш труд в подобной ситуации, призваны функции.

Давайте представим, что нам очень часто встречается одно и то же действие, а именно сумма двух различных переменных. Написав эту функцию в отдельном модуле, мы в последующем можем обращаться к ней, не переписывав одни и те же действия, по многу раз. Притом функция может возвращать какое-то значение, а может просто выполнить своё действие, например, вывод на консоль информации, при этом ничего не вернув.

Функция – отдельный блок кода, который можно вызывать по её имени из любого места программы:

```
def func_sum (x, a):
    res = x + a
    return res #return - означает что функция возвращает какое то значение
a = func_sum (20, 12)
print (a)
```

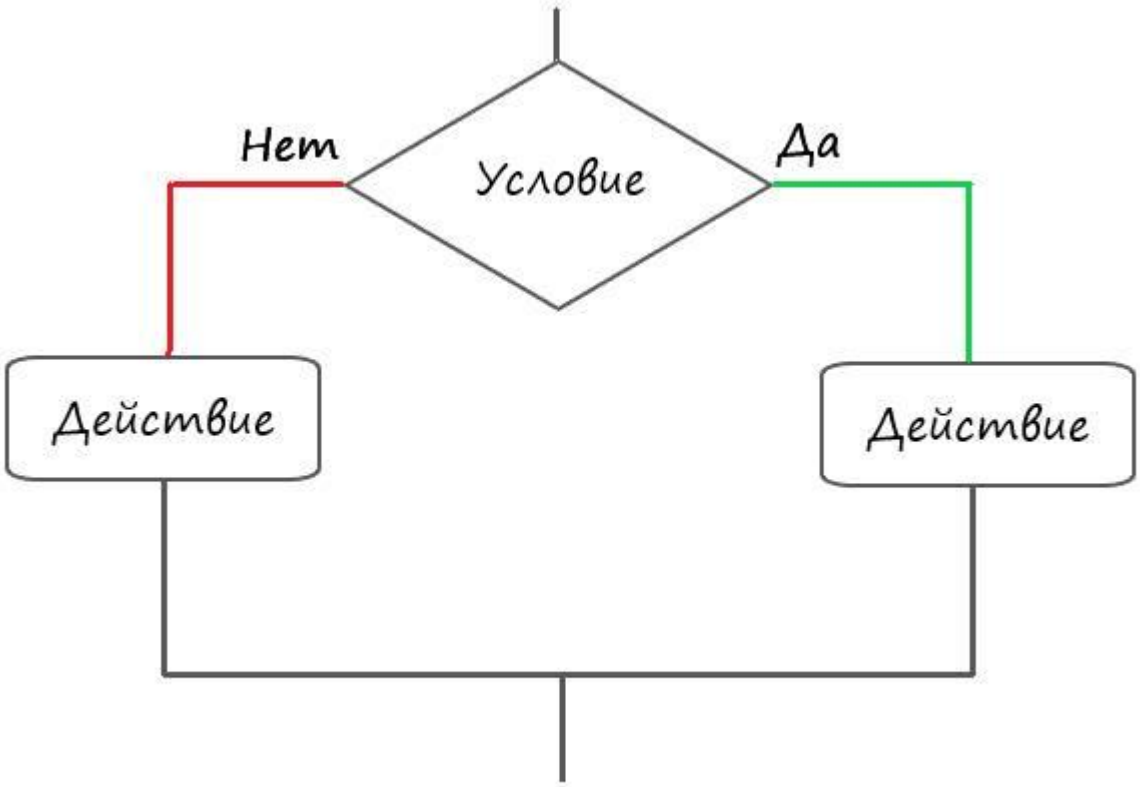
```
32
```

```
def func ():
    x = 34
    y = 45
    z = x + y
    print("Эта функция ничего не возвращает")
    pass #pass - означает что функция ничего не возвращает
print (func ())
```

```
Эта функция ничего не возвращает
None
```

Условные операторы

Условные операторы нужны для того, чтобы выполнить два разных набора действий в зависимости от того, истинно или ложно проверяемое ими утверждение. Иными словами – в зависимости от того, ложно или истинно утверждение, программа, как бы разветвляется, идет по пути, указанным ей этим условием.



Условия

В Python, условия записываются при помощью конструкции if:… else:… if – в переводе с английского – если, else переводится как – иначе.

После ключевого слова if, следует условие, которое им проверяется, если это условие правда, то выполняется тело этого оператора if, если ложно, то тело оператора if, не выполнится.

Давайте рассмотрим это на конкретном примере:

```
x = 15
if x < 10:
    print("Условие (x < 10) - выполнилось!")
print("Точка после условных операторов")
```

```
Точка после условных операторов
```

Здесь, как мы можем наблюдать, условие не выполнилось.

```
x = 15
if x > 10:
    print("Условие (x > 10) - выполнилось!")
print("Точка после условных операторов")
```

```
Условие (x > 10) - выполнилось!
Точка после условных операторов
```

В этот случае, мы наблюдаем, что наше условие выполняется.

```
x = 15
if x < 10:
    print("Условие (x < 10) - выполнилось!")
else:
    print("Условие (x < 10) - не выполнилось!")
print("Точка после условных операторов")
```

```
Условие (x < 10) - не выполнилось!
Точка после условных операторов
```

В этом примере, где задействована вся конструкция if:… else:…, условие if(если) – не выполнено, но если не выполняется условие – if, то тогда сработает условие – else(иначе).

Обратите внимание, в Python все условия принадлежащее оператору, пишутся с определенным отступом!

Массивы

Массив можно представить в виде книжной полки, которая содержат сразу несколько книг(переменных).

Пример массива, содержащего в себе числа и строку:

```
arr = [5, 3, "строка"]
print(arr)
```

```
[5, 3, 'строка']
```

У массива есть такое понятие как индекс, например, по индексу ноль, массива arr, содержится элемент равный числу 5. А по индексу три, находится строка. Количеством индексов, определяется размер массива:

```
arr = [5, 3, "строка"]
print( len(arr) ) #Len - возвращает кол-во элементов массива
```

```
3
```

Обращаясь к индексам элементов, как показано на слайде ниже, мы можем менять

элемент, к адресу которого мы обратились (не забываем, что начало отсчета индексов в массиве, начинается с нуля):

```
# Изменяем элемент с индексом 1, который изначально был равен 3, на значение 10
arr[1] = 10
print(arr)
```

```
[5, 10, 'строка']
```

Для работы с массивами, в наших проектах мы будем использовать пакет numpy.

numpy – очень обширная библиотека, содержащая множество методов по работе с массивами.

Для того чтоб воспользоваться этим инструментом нужно выполнить следующий код:

import numpy

Команда import сообщает Python о необходимости привлечения дополнительных вычислительных ресурсов, для расширения круга уже имеющихся на его вооружении инструментов.

Если мы выполним следующую команду:

import numpy as np

Где, as – префикс, позволяющий сокращать, или изменять имя пакета, указав сокращение np (можно любое другое имя), мы избавляем себя от необходимости писать в программном коде полное имя пакета, т.е. говоря простым языком, заменим имя numpy на сокращенное np.

Давайте создадим с помощью пакета numpy, двухмерный массив (матрицу) с нулевыми элементами:

```
import numpy as np
arr = np.zeros( [2,3] )
print(arr)
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

В коде выше, пакет numpy используется для создания двухмерного массива размерностью 2x3, где 2 – количество строк массива, 3 – количество столбцов в массиве, и во всех ячейках данного массива содержатся нулевые значения.

В массивах с несколькими измерениями, тоже можно изменять элементы, обратившись к их индексам (адресам элементов):

```
arr [0,0] = 2
arr [0,1] = 4
arr [1,0] = 9
arr [1,2] = 14
print(arr)
```

```
[[  2.   4.   0.]
 [  9.   0.  14.]]
```

Срезы

Срезы позволяют обрезать массив, взяв лишь те элементы, которые нам будут нужны. Они работают по следующей схеме: [НАЧАЛО:КОНЕЦ:ШАГ].

Начало – с какого элемента стоит начать (по умолчанию равна 0);

Конец – по какой элемент мы берем элементы (по умолчанию равно длине списка);

Шаг – с каким шагом берем элементы, к примеру, каждый 2 или каждый 3 (по умолчанию каждый 1).

```
#Срезы
arr = [5, 3, 10, 15, 7, 9, 8]
print('arr ', arr)
print('[::2] ', arr [::2]) # Берем каждый 2й элемент
print('[2::2] ', arr [2::2]) # Начиная со 2го элемента, берем каждый 2й элемент
print('[2:5:] ', arr [2:5:]) # Начиная с 2го элемента, берем все элементы по 5й элемент
print('[::] ', arr [::]) # Берем все элементы
```

```
arr  [5, 3, 10, 15, 7, 9, 8]
[::2]  [5, 10, 7, 8]
[2::2]  [10, 7, 8]
[2:5:]  [10, 15, 7]
[::]  [5, 3, 10, 15, 7, 9, 8]
```

А если, например, нам нужен второй элемент с обратной стороны массива, то мы можем обратится к нему следующим образом:

```
print (arr [-2])
```

```
9
```

Циклы

Циклы, необходимы там, где требуется многократные повторения действий. Если, к примеру, мы хотим вывести таблицу квадратов первых четырёх натуральных чисел, то циклы в этом вопросе, будут незаменимыми помощниками.

Когда мы попытаемся вывести квадраты чисел без циклов, то нам придётся выполнять все действия вручную, в нашем случае в 4 строки.

```
print("Квадрат от 1 = " + str(1 ** 2))
print("Квадрат от 2 = " + str(2 ** 2))
print("Квадрат от 3 = " + str(3 ** 2))
print("Квадрат от 4 = " + str(4 ** 2))
```

```
Квадрат от 1 = 1
Квадрат от 2 = 4
Квадрат от 3 = 9
Квадрат от 4 = 16
```

А если нам надо вывести квадраты первых 1000 чисел? Вводить 1000 строк? Нет, для таких случаев и существуют циклы. В Python есть два вида циклов: while и for.

Цикл while повторяет необходимые команды до тех пор, пока остается истинным условие, задаваемое, как и в случае с if, сразу после объявления оператора, как только условие выполнится, цикл прекратит свою работу.

Давайте теперь, с помощью while, выведем таблицу квадратов первых четырёх натуральных чисел:

```
x = 1
while x <= 4:
    print("Квадрат от " + str(x) + " = " + str(x**2))
    x += 1
```

```
Квадрат от 1 = 1
Квадрат от 2 = 4
Квадрат от 3 = 9
Квадрат от 4 = 16
```

Здорово, правда? Всего четырьмя строками кода, мы можем выводить квадраты чисел, до почти любого числа.

Если подробней разобрать работу цикла:

Сначала мы создаем переменную и присваиваем ей число 1. Затем создаем цикл while и проверяем, меньше, или равна четырем наша переменная x. Если меньше, или равна, то будут выполнятся следующие действия:

– вывод на консоль квадрата переменной x;

– в теле оператора, увеличиваем x на единицу, (запись: x+= 1, эквивалентна записи: x = x + 1)

После чего, программа возвращается к условию цикла. Если условие снова истинно, то мы снова выполняем эти два действия. И так до тех пор, пока x не станет больше 4. Тогда условие вернет ложь и цикл больше не будет выполняться.

Цикл for будем использовать, в основном, для того, чтобы перебирать элементы массива, согласно его индексам. Запишем тот же пример, что и с while, с квадратами первых шести натуральных чисел, используя цикл for:

```
for i in [1, 2, 3, 4, 5, 6]:
    print(i ** 2)
```

```
1
4
9
16
25
36
```

Конструкция for i in —создает цикл, организуя счетчик для каждого числа из списка массива, путем назначения текущего значения переменной i. При первом проходе цикла выполняется присваивание i=0, потом i=1, i=2, и так до тех пор, пока мы не дойдем до последнего элемента списка, которому присвоится значение i=6.

Применяя функцию range (), эту операцию можно сделать немногим иначе:

```
for i in range(7):
    print(i ** 2)
```

```
0
1
4
9
16
25
36
```

В данном примере, функция range () – задает последовательность счета натуральных чисел, до конечного значения, указанного в скобках.

Классы и их объекты

В реальной жизни мы чаще оперируем не переменными, а объектами. Стол, стул, человек, кошка, собака, корабль – это все объекты. Наилучший способ знакомства с объектами – это рассмотреть конкретный пример:

```
# класс объектов Cat (кошка)
class Cat:
# Кошки говорят – “Мяу!”

def says (self):
    print (‘Мяу!’)
    pass
pass
```

Запись class Cat – означает что создан класс Cat (кошка), а функция def says(), внутри класса – это метод класса Cat, который выполняет определенные действия связанные с этим классом. В нашем случае созданный нами метод says() выводит на экран – ‘Мяу!’.

Давайте на примере покажем, как создаются объекты класса и работают его методы.

```
classcat = Cat () #создание объекта classCat, класса Cat
classcat.says () #использование метода says (), объекта classCat
```

Методов в классе может содержаться так много, насколько это необходимо, для его

описания. Кошка помимо того, что может говорить: “Мяу!”, обладает и рядом других важных параметров. К ним относятся цвет шерсти, цвет глаз, кличка, и так далее. И все это, можно описать при помощи методов в классе. Давайте опишем выше сказанное в Python:

```
# класс объектов Cat (кошка)
class Cat:
    # Кошки говорят - “Мяу!”
    def says (self):
        print ('Мяу!')
        pass
pass
```

```
classcat = Cat() #создание объекта classcat, класса Cat
classcat.says() #использование метода says (), объекта classcat
```

```
Мяу!
```

Множеству объектов, можно присваивать одинаковый класс и эти объекты в свою очередь, будут обладать одинаковыми методами:

```
# класс объектов Cat (кошка)
class Cat:
    # Кошки говорят - “Мяу!”
    def says (self):
        print ('Мяу!')
        pass
pass

classcat = Cat() #создание объекта classcat, класса Cat
siam = Cat() #создание объекта siam, класса Cat

classcat.says() #использование метода says (), объекта classcat
siam.says() #использование метода says (), объекта siam
```

```
Мяу!
Мяу!
```

Чтобы получить более полное представление о возможностях объектов, давайте добавим в наш класс переменные, которые будут хранить специфические данные этих объектов, а также методы, позволяющие просматривать и изменять эти данные:

```
class Cat:
    # метод для инициализации объекта внутренними данными
    def __init__(self, catname, years):
        self.name = catname
        self.num_years = years

        # получить состояние
    def status(self):
        print('Кличка кошки: ', self.name)
        print('Количество лет кошки: ', self.num_years)
        pass

    # Количество лет кошки
    def number_of_years(self, years):
        self.num_years = years
        pass
```

```
    # Кошка говорит мяу
    def says(self):
        print('Мяу!')
        pass
pass
```

```
Murka = Cat('Murka', 8) #создание объекта Murka, класса Cat
# Узнаем статус объекта Murka
Murka.status()
```

```
Кличка кошки:  Murka
Количество лет кошки:  8
```

```
Murka.says() #использование метода says (), объекта Murka
```

```
Мяу!
```

```
#Изменяем атрибут - количество лет, объетта Murka
Murka.number_of_years(9)
# Узнаем статус с обновленными данными объекта Murka
Murka.status()
```

```
Кличка кошки:  Murka
Количество лет кошки:  9
```

Давайте разбираться что же мы тут написали.

В любом классе можно определить функцию __init__(). Эта функция всегда вызывается, когда мы создаем реальный объект класса, с изначально заданными атрибутами. Атрибут – это переменная, которая относится к классу, в котором она определена. В нашем случае, при создании объекта, мы сразу можем указать его атрибуты – кличку и количество

лет, которые сразу присваиваются этому объекту. Через созданный нами метод status(), мы можем вывести информацию о количестве лет и кличке нашего объекта. Метод number_of_years (self, years), принимает число и изменяет атрибут класса – количество лет. Метод says(), не изменился, он все также говорит голосом нашего объекта – ‘Мяу!’.

ГЛАВА 3

Рождение искусственного нейрона

Моделирование нейрона как линейного классификатора

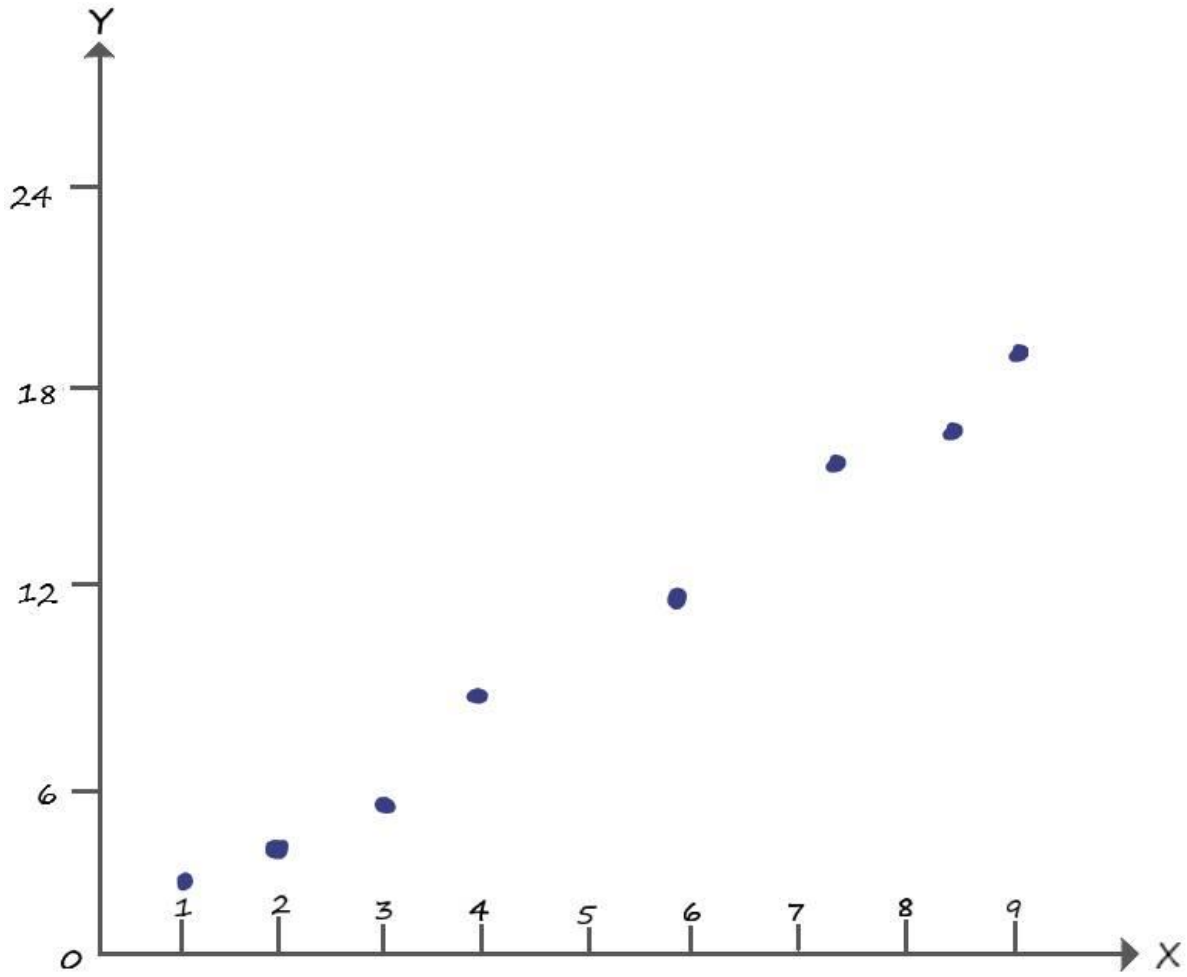
Настало время практически реализовать линейную классификацию. Для этого в Python смоделируем работу искусственного нейрона. Попробуем решить нашу задачу, найдя промежуточные значения, при заданном наборе входных и соответствующим им выходным (целевым) параметрам. Как мы помним – это были высота и длина двух разных видов животных. Это может быть и любой другой условный набор данных, которые можно представить, как параметры размеров одежды, предметов, насекомых, веса, стоимости, градусов и любых других. Отобразим наше задание – список с параметрами двух видов животных:

x (длина)	Y(высота)
1	2,4
2	4,5
3	5,5
3,5	6,4
4	8,5
6	11,7
7,5	16,1
8,5	16,5
9	18,3

В дальнейшем все данные, которые надо анализировать при помощи искусственных нейронов и их сетей, будем называть – **обучающей выборкой** . А процесс изменения коэффициентов, в нашем случае – коэффициент **A** , в зависимости от функции ошибки на выходе, будем называть – **процессом обучения .**

Примем за значение **x** – длины животных, а **Y** – высота. Так как **Y** (игрек большое) – это и есть ответ: **Y = Ax** , то условимся что он и будет целевым значением для нашего нейрона (правильным ответом), а входными данными будут все значения переменной **x .**

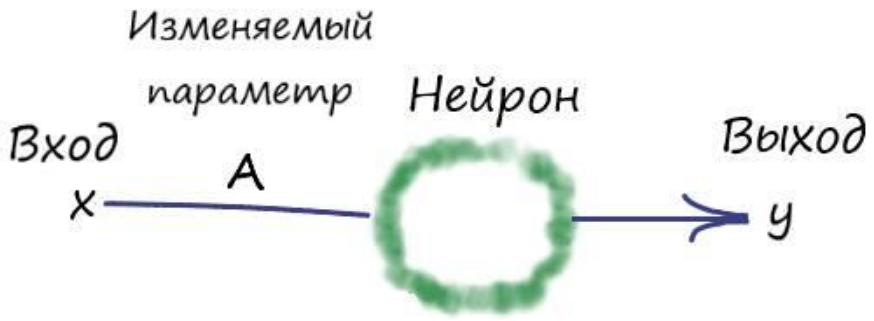
Отобразим для лучшего представления входных данных, график обучающей выборки:



Видно, что наши данные напоминают прямую линию, уравнение которой **Y = = 2* x .** Данные находятся около значений этой функции, но не повторяют их. Задача нашего нейрон суметь с большой точностью провести эту прямую, несмотря на то, что данные по остальным точкам отсутствуют (например, нет данных о **Y** координате с точкой с **x = 5**).

Смоделируем такую структуру, для чего подадим на вход нейрона (дендрит у биологического нейрона), значение **x** , и меняя коэффициент **A** (синапс у биологического нейрона), по правилам, которые мы вывели с линейным классификатором, будем получать выходные значения нейрона **y** (аксон у биологического нейрона). Так же условимся, что Y (большое) – правильный ответ (целевое значение), а **y** (малое) – ответ нейрона (его выход).

Визуализируем структуру нейрона, которую будем моделировать:



Запрограммировав в Python эту структуру, попробуем добиться прямой, которая максимально точно разделит входные параметры.

Программа

Действовать будем так же, как мы действовали, рассчитывая линейный классификатор.

Создадим переменную **A** , являющейся коэффициентом крутизны наклона прямой, и зададим ей любое значение, пусть это будет всё те же **A=0.4** .

```
A = 0.4
```

Запомним начальное значение коэффициента A:

```
A_vis = A
```

Покажем функцию начальной прямой:

```
print('Начальная прямая: ', A, '* X')
```

Укажем значение скорости обучения:

```
lr = 0.001
```

Зададим количество эпох:

```
epochs = 3000
```

Эпоха – значение количества проходов по обучающей выборке. Если в нашей выборке девять наборов, то одна эпоха – это один проход в цикле всех девяти наборов данных.

Зададим наш набор данных, используя массивы. Создадим два массива. В один массив поместим все входные данные – **x** , а в другой целевые значения (ответы) – **Y .**

Создадим массив входных данных x:

```
arr_x = [1, 2, 3, 3.5, 4, 6, 7.5, 8.5, 9]
```

Создадим массив целевых значений (ответы Y):

arr_y = [2.4, 4.5, 5.5, 6.4, 8.5, 11.7, 16.1, 16.5, 18.3]

Задаем в цикле эпох, вложенный цикл – for i in range(len(arr)), который будет последовательно пробегать по входным данным, от начала до конца. А циклом – for e in range(epochs), мы как раз указываем количество таких пробегов (итераций):

```
for e in range(epochs):
for i in range(len(arr)):
```

Функция len(arr) возвращает длину массива, в нашем случае возвращает девять.

Получаем x координату точки из массива входных значений x:

```
x = arr_x[i]
```

А затем действуем как в случае с линейным классификатором:

```
# Получить расчетную y, координату точки
y = A * x
# Получить целевую Y, координату точки
target_Y = arr_y[i]
# Ошибка E = целевое значение – выход нейрона
E = target_Y – y
# Меняем коэффициент при x, в соответствии с правилом A+дельтаA = A
A += lr*(E/x)
```

Напомню, процессом изменения коэффициентов в ходе выполнения цикла программы, называют – **процессом обучения .**

Выведем результат после обучения:

```
print('Готовая прямая: y = ', A, '* X')
```

Полный текст программы:

```
# Инициализируем любым числом коэффициент крутизны наклона прямой
A = 0.4
A_vis = A # Запоминаем начальное значение крутизны наклона
# Вывод данных начальной прямой
print('Начальная прямая: ', A, '* X')

# Скорость обучения
lr = 0.001
# Зададим количество эпох
epochs = 3000

# Создадим массив входных данных x
arr_x = [1, 2, 3, 3.5, 4, 6, 7.5, 8.5, 9]
# Создадим массив целевых значений (ответы Y)
arr_y = [2.4, 4.5, 5.5, 6.4, 8.5, 11.7, 16.1, 16.5, 18.3]

# Прогон по выборке
```

```
for e in range(epochs):
for i in range(len(arr_x)): # len(arr) – функция возвращает длину массива
# Получить x координату точки
x = arr_x[i]

# Получить расчетную y, координату точки
y = A * x

# Получить целевую Y, координату точки
target_Y = arr_y[i]

# Ошибка E = целевое значение – выход нейрона
E = target_Y – y

# Меняем коэффициент при x, в соответствии с правилом A+дельтаA = A
A += lr*(E/x)

# Вывод данных готовой прямой
print('Готовая прямая: y = ', A, '* X')
```

Результатом ее работы будет функция готовой прямой:

```
y = 2.0562708725692196 * X
```

Для большей наглядности, что я специально указал данные в обучающей выборке, так чтобы они лежали около значений функции $y = 2x$. И после обучения нейрона, мы получили ответ очень близкий к этому значению.

Было бы неплохо визуализировать все происходящие на графике прямо в Python.

Визуализация позволяет быстро получить общее представление о том, что мы делаем и чего добились.

Для реализации этих возможностей, нам потребуется расширить возможности Python для работы с графикой. Для этого необходимо импортировать в нашу программу, дополнительный модуль, написанный другими программистами, специально для визуализаций данных и функций.

Ниже приведена инструкция, с помощью которой мы импортируем нужный нам пакет для работы с графикой:

```
import matplotlib.pyplot as plt
```

Кроме того, мы должны дополнительно сообщить Python о том, что визуализировать следует в нашем блокноте, а не в отдельном окне. Это делается с помощью директивы:

```
%matplotlib inline
```

Если не получается загрузить данный пакет в программу, то скорей всего его надо скачать из сети. Делать это удобно через Anaconda Prompt, который устанавливается вместе с пакетом Anaconda.

Для системы Windows, в Anaconda Prompt вводим команду:

```
conda install matplotlib
```

И следуем инструкциям. Для других операционных систем возможно потребуется другая команда.

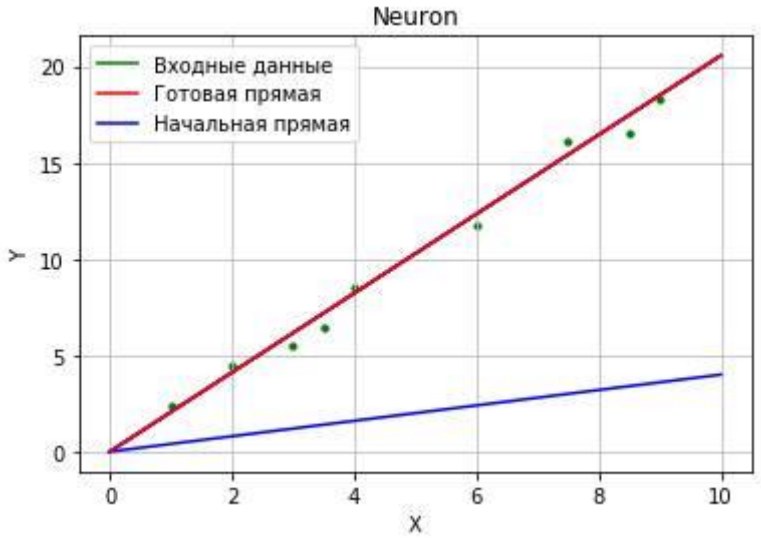
Теперь мы полностью готовы к тому, чтобы представить наши данные и функции в графическом виде.

Выполним код:

```
import matplotlib.pyplot as plt
%matplotlib inline

# Функция для отображения входных данных
def func_data(x_data):
return [arr_y[i] for i in range(len(arr_y))]

# Функция для отображения начальной прямой
def func_begin(x_begin):
return [A_vis*i for i in x_begin]

# Функция для отображения готовой прямой
def func(x):
return [A*i for i in x]

# Значения по X входных данных
x_data = arr_x

# Значения по X начальной прямой (диапазон значений)
x_begin = [i for i in range(0, 11)]

# Значения по X готовой прямой (диапазон значений)
x = [i for i in range(0, 11)]
#x = np.arange(0,11,1)

# Значения по Y входных данных
y_data = func_data(x_data)

# Значения по Y начальной прямой
y_begin = func_begin(x_begin)

# Значения по Y готовой прямой
y = func(x)

# Зададим имена графику и числовым координатам
plt.title("Neuron")
plt.xlabel("X")
plt.ylabel("Y")

# Зададим имена входным данным и прямым
plt.plot(x,y, label='Входные данные', color = 'g')
plt.plot(x,y, label='Готовая прямая', color = 'r')
plt.plot(x,y, label='Начальная прямая', color = 'b')
plt.legend(loc=2) #loc – локация имени, 2 – справа в углу

# представляем точки данных (x,y) кружочками диаметра 10
```

```
plt.scatter(x_data, y_data, color ='g', s=10)
# Начальная прямая
plt.plot(x_begin, y_begin, 'b')
# Готовая прямая
plt.plot(x, y, 'r')
# Сетка на фоне для улучшения восприятия
plt.grid(True, linestyle='-', color='0.75')
# Показать график
plt.show()
```

При выполнении кода, результат визуализации окажется следующим:



Исходники с программами вы можете найти по ссылке: https://github.com/CaniaCan/neuralmaster

Перед тем как описать полученный результат, сперва опишем работу нашего кода пакета matplotlib.

В функциях отображения входных данных – def func_data(x_data), def func_data(x_begin), def func_data(x), возвращаем координаты y, в соответствии со своими значениями по x.

Зададим имена графику – plt.title(), и числовым координатам – plt.xlabel():

```
plt.title("Neuron")
plt.xlabel("X")
plt.ylabel("Y")
```

Зададим имена входным данным и прямым – plt.plot(), в скобках укажем имя и цвет, plt.legend(loc=2) – определяет нахождение данных имен на плоскости:

```
plt.plot(x,y, label='Входные данные', color = 'g')
plt.plot(x,y, label='Готовая прямая', color = 'r')
plt.plot(x,y, label='Начальная прямая', color = 'b')
plt.legend(loc=2) #loc – локация имени, 2 – справа в углу
```

Метод scatter выводит на плоскость точки с заданными координатами:

```
plt.scatter(x_data, y_data, color ='g', s=10)
```

Метод plot выводит на плоскость прямую по заданным точкам:

```
plt.plot(x, y, 'r')
```

Ну и наконец отображаем все что натворили, командой plt.show().

Теперь разберем получившийся график. Синим – отмечена начальная прямая, которая изначально не выполняла никакой классификации. После обучения, значение коэффициента A, стабилизируется возле числа = 2.05. Если провести прямую функции y = Ax = 2.05*x, отмеченной красным на графике, то получим значения близкие к нашим входным данным (на графике – зеленые точки).

А что если, наш обученный нейрон смог бы правильно отвечать на вводимые пользователем данные? Если задать условие, что всё что выше красной линии относится к виду – жирафов, а ниже к виду – крокодилов:

```
x = input("Введите значение ширины X: ")
x = int(x)
T = input("Введите значение высоты Y: ")
T = int(T)
y = A * x

# Условие
if T &gt; y:
print('Это жираф!')
else:
print('Это крокодил!')
```

Функция input – принимает значение, вводимое пользователем. А условие гласит: если целевое значение (вводимое пользователем) больше ответа на выходе нейрона (выше красной линии), то сообщаем что – это жираф, иначе сообщаем что – это крокодил.

После ввода наших значений, получаем ответ:

```
Введите значение ширины X: 4
Введите значение высоты Y: 15
Это жираф!
```

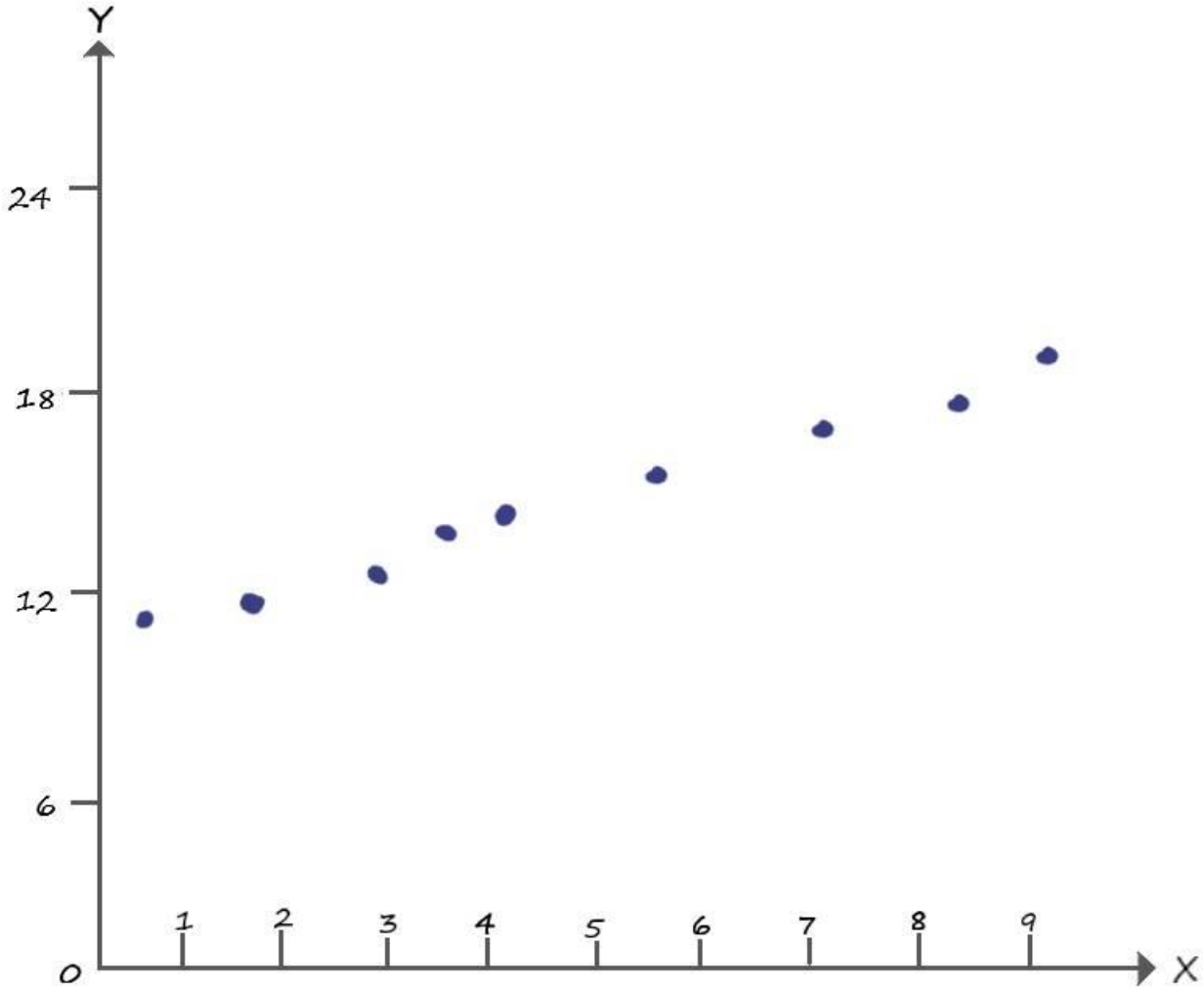
Теперь мы можем поздравить себя! Вся наша работа стала сводиться к тому, чтоб просто подавать на вход нейрона данные, не разбираясь в них самостоятельно. Нейрон сам классифицирует их и даст правильный ответ.

Если бы наши действия на работе сводились к подобным классификациям, то у нас появилась бы куча времени на кофе, очень важных общений в социальных сетях, и даже останется время, чтоб разложить пасьянс. И при всем этом можно выполнять ещё больший объём работы, что конечно же должно вознаграждаться премиальными и повышением зарплаты.

ГЛАВА 4

Добавляем входной параметр

Теперь представим, что нам приходит новое задание. Где, проанализировав самостоятельно данные, мы видим, что их координаты значительно отличаются от прежних. Теперь провести классифицирующую прямую, обладая в своем арсенале лишь коэффициентом крутизны – не выйдет!



Очевидно, что без параметра **b** , которого мы до этого избегали (**b=0**), тут не обойтись.

Вспомним, что параметр **b** , в уравнении прямой **y = Ax + b** , как раз отвечает за точку её пересечения с осью **Y** . На графике выше, такая точка очевидно находится возле координаты – (**x =0** ; **y =11**).

Для того, чтобы выполнить новое задание, придется добавить в наш нейрон, второй вход – отвечающий за параметр **b** .

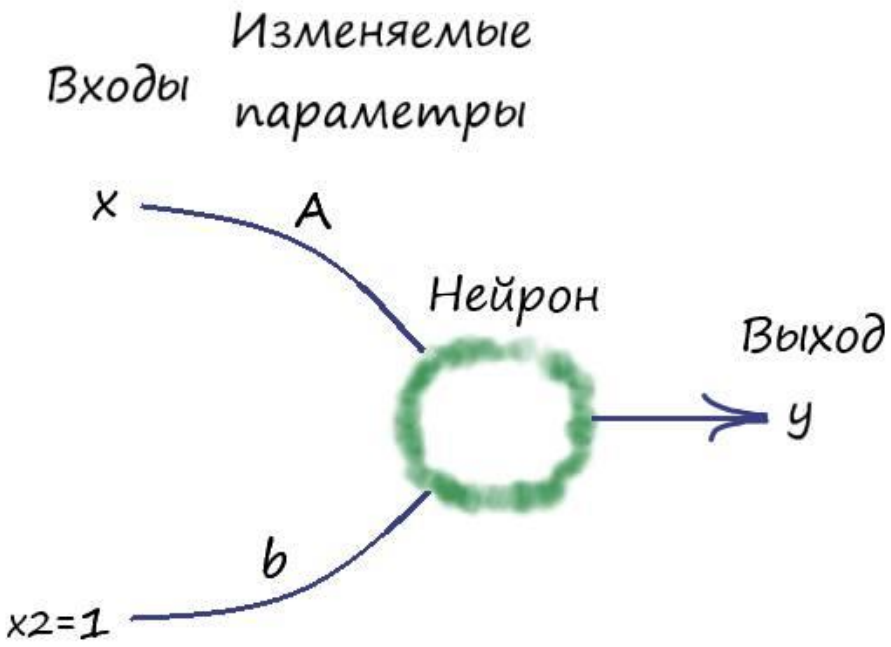
Моделирование нейрона как линейного классификатора со всеми параметрами линейной функции

Определимся с параметром (**b**). Как будет выглядеть второй вход? Какие данные подавать в ходе обучения?

Параметр (**b**) – величина постоянная, поэтому мы добавим его на второй вход нейрона, с постоянным значением входного сигнала, равным единице (**x 2 = 1**). Таким образом, произведение этого входа на значение величины (**b**), всегда будет равно значению самой величины (**b**).

Пришло время для первого эволюционного изменения структуры нашего нейрона!

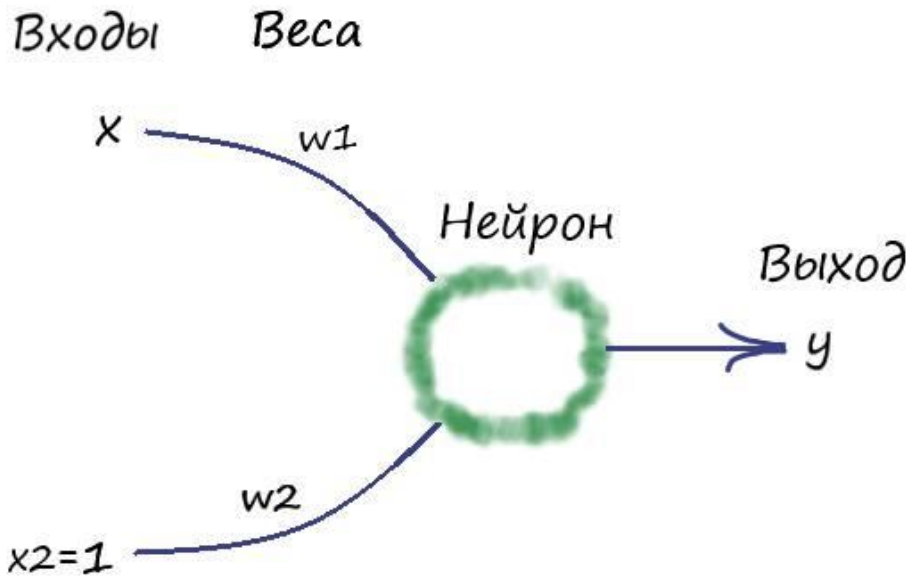
Рассмотрим следующую графическую модель искусственного нейрона:



Где, как говорилось выше, на вход нейрона поступают два входных сигнала **x** (из нашего набора данных) и **x 2 = 1** . После чего, эти значения умножаются со своими изменяемыми параметрами, а далее они суммируются: **A * x + b * x 2** . Значение этой суммы, а по совместительству – значение функции **y = A * x + b * x 2 = A * x + b** , поступает на выход.

Ну и давайте всё представим согласно тем принятым условным обозначениям, которые используются при моделировании искусственных нейронов и нейронных сетей. А именно – коэффициент **A** и параметр **b** , обозначим как **w1** и **w2** соответственно. И теперь будем их называть – **весовыми коэффициентами** .

Ну и конечно же, визуализируем структуру нашего нейрона, с новыми обозначениями:



Переименуем в нашей первой программе коэффициент (**A**) и параметр (**b**), на обозначения весовых коэффициентов, как показано на слайде. Инициализируем их в ней. Дополним небольшую её часть в области с обучением, формулой изменения веса (**w2**), как мы это делали ранее с коэффициентом (**A**).

После чего, область с обучением в программе, будет выглядеть следующим образом:

```
# Прогон по выборке
for e in range(epochs):
for i in range(len(arr)): # len(arr) – функция возвращает длину массива
# Получить x координату точки
x = arr[i]

# Получить расчетную y, координату точки
y = w1 * x + w2

# Получить целевую Y, координату точки
target_Y = arr_y[i]

# Ошибка E = целевое значение – выход нейрона
E = target_Y – y

# Меняем вес при входе x
w1 += lr*(E/x)

# Меняем вес при входе x2 = 1, w2 += lr*(E/x2) = lr*E
w2 += lr*E
```

И забегая вперед, скажу, что тут нас постигнет разочарование – ничего не выйдет…

Дело в том, что вес (**w2**) (бывший параметр (**b**)), вносит искажение в поправку веса (**w1**) (бывшего коэффициента (**A**)) и наоборот. Они действуют независимо друг от друга, что сказывается на увеличении ошибки с каждым проходом цикла программы.

Нужен фактор, который заставит наша веса действовать согласованно, учитывать интересы друг друга, идти на компромиссы, ради нужного результата. И такой фактор у нас уже есть – ошибка.

Если мы придумаем как согласованно со всеми входами уменьшать ошибку с каждым проходом цикла в программе, подгоняя под неё весовые коэффициенты таким образом, что в конечном счете привело к самому минимальному её значению для всех входов. Такое решение, являлось бы общим для всех входов нашего нейрона. То есть, согласованно обновляя веса в сторону уменьшения их общей ошибки, мы будем приближаться к оптимальному результату на выходе.

Поэтому, при числе входов нейрона, больше одного, наши выработанные до этого правила линейной классификации, необходимо дополнить. Нужно использовать ошибку, чтобы математически связать все входы таким образом, при котором они начнут учитывать общие интересы. И как следствие, на выходе получить нужный классификатор.

Итак, мы постепенно подходим к ключевому понятию в обучении нейрона и нейронных сетей – **обучение методом градиентного спуска .**

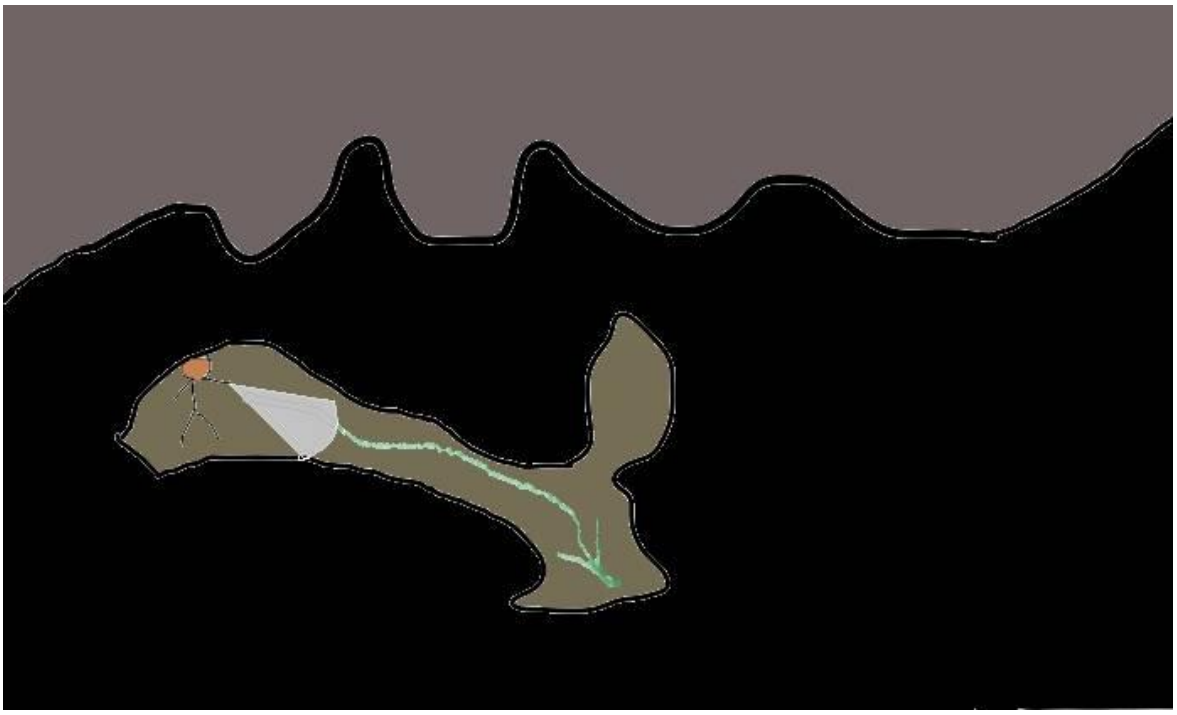
Обновление весовых коэффициентов

Найдем решение, которое, даже будет не идеальным с точки зрения математики, но даст нам правильные результаты, поскольку всё же опирается на математический инструмент.

Для понимания всего процесса, давайте представим себе спуск с холма, со сложным рельефом. Вы спускаетесь по его склону, и вам нужно добраться до его подножья. Кругом кромешная тьма. У вас в руках есть фонарик, света которого едва хватает на пару метров. Все что вы сможете увидеть, в этом случае – по какому участку, в пределах видимости фонаря, проще всего начать спуск и сможете сделать только один небольшой шаг в этом направлении. Действуя подобным образом, вы будете медленно, шаг за шагом, продвигаться вниз.

У такого абстрактного подхода, есть математическая версия, которая называется – **градиентным спуском** . Где подножье холма – минимум ошибки, а шагами в его направлении – обновления весовых коэффициентов.

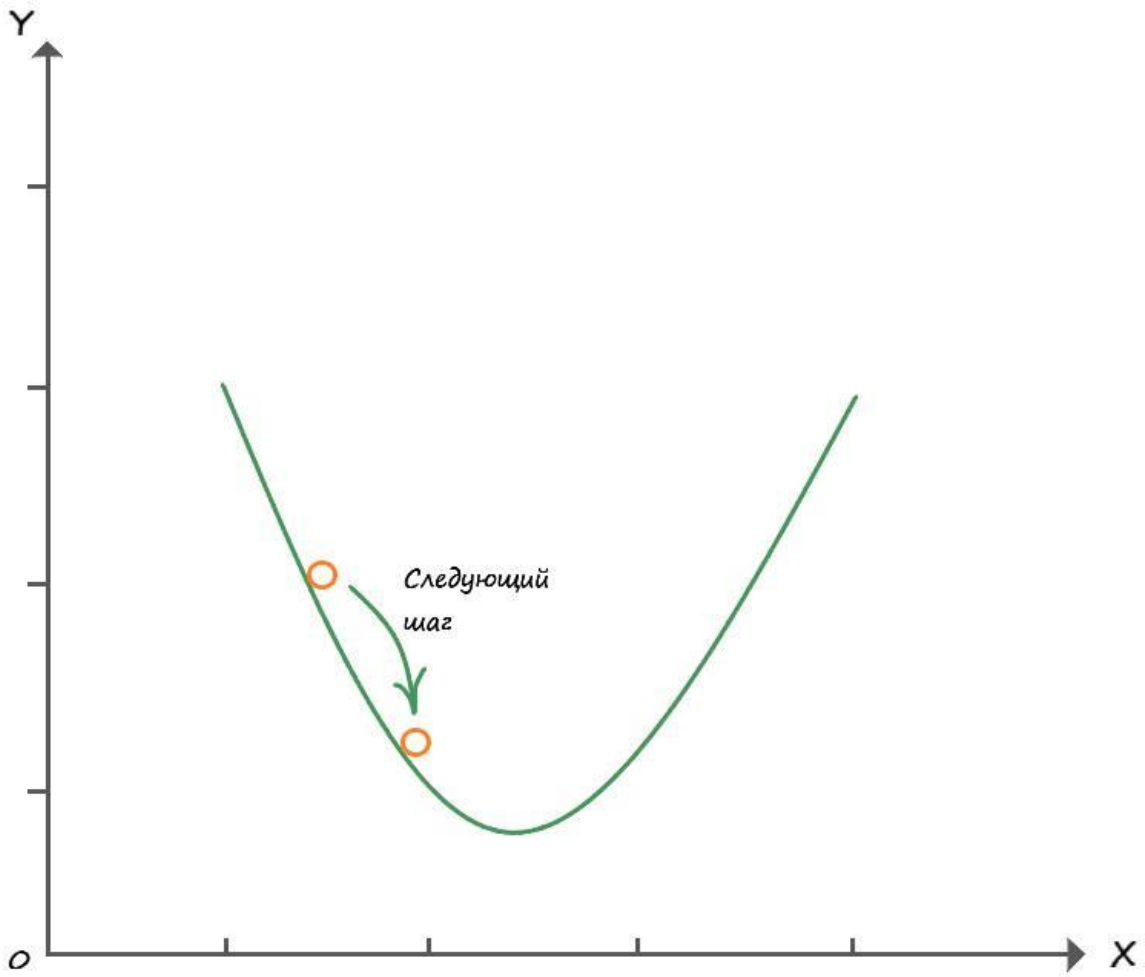
Градиентный спуск – метод нахождения локального минимума или максимума функции с помощью движения вдоль градиента – который, своим направлением указывает направление наибольшего возрастания некоторой величины, значение которой меняется от одной точки пространства к другой, а по величине (модулю) равный скорости роста этой величины в этом направлении.



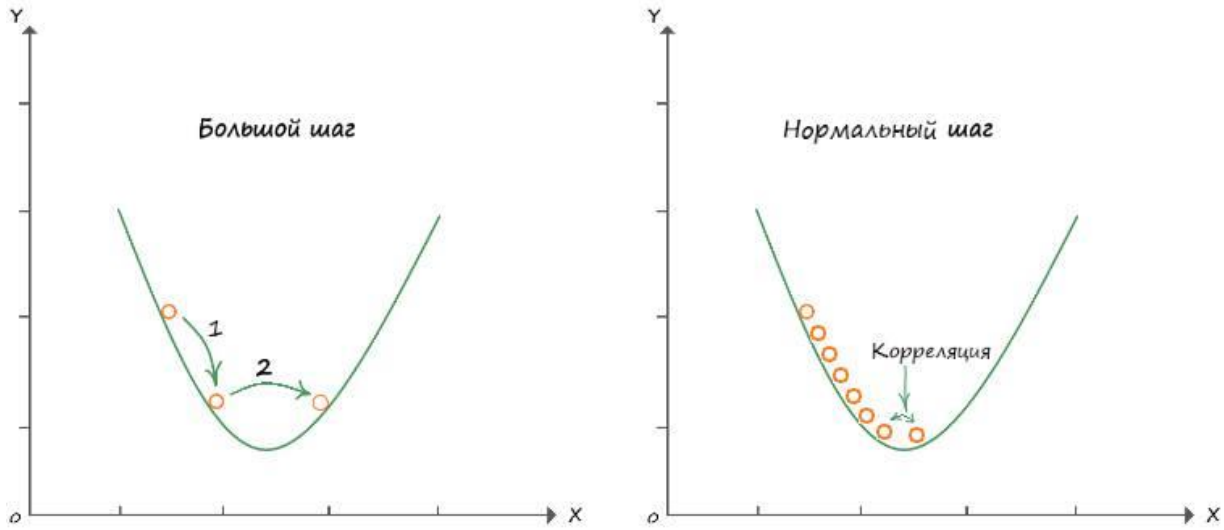
Метод градиентного спуска позволяет находить минимум, даже не располагая знаниями свойств этой функции, достаточными для нахождения минимума другими математическими методами. Если функция очень сложна, где нет простого способа нахождения минимума, мы в этом случае можем применить метод градиентного спуска. Этот метод может не дать нам абсолютно точного ответа. Но все же это лучше, чем вообще не иметь никакого решения. А его суть, как было описано выше – постепенно приближаться к ответу, шаг за шагом, тем самым медленно, но верно, улучшая нашу позицию.

Для наглядности, рассмотрим использование метода градиентного спуска на простейшем примере.

Возьмём график функции, которая своими значениями иллюстрирует склон. Если бы это была функция ошибки, то нам нужно найти такое значение (**x**), которое минимизирует эту функцию:



Значение шага (скорости обучения), как мы говорили ранее, играет тоже не малую роль, при слишком большом значении, мы быстро спускаемся, но можем переступить минимум функции – страдает точность. При очень маленьком значении величины скорости обучения, нахождение минимума потребует гораздо больше времени. Нужно подобрать величину шага такой, чтоб он удовлетворяла нас и по скорости, и по точности. При нахождении минимума, наша точка будет коррелировать, возле значения минимум, в чуть большую и меньшую сторону на величину шага. Это все равно что – когда спустившись вплотную к подножью, мы сделали шаг и оказались чуть выше подножья, повернувшись сделали такой же шаг назад, и поняв, что опять находимся чуть выше, повторяли эти действия до бесконечности. Но при этом, мы все равно находились бы очень близко к подножью, потому как величина шага, в общем объеме, ничтожна, поэтому мы можем говорить – что находимся в самом низу.



Выходной сигнал нейрона представляет собой сложную функцию со многими входными данными, и соответствующие им – весовыми коэффициентами связи. Все они коллективно влияют на выходной сигнал. Как при этом подобрать подходящие значения весов используя метод градиентного спуска? Для начала, давайте правильно выберем функцию ошибки.

Функция выходного сигнала не является функцией ошибки. Но мы знаем, что есть связь между этими функциями, поскольку ошибка – это разность между целевыми тренировочными значениями и фактическими выходными значениями (**E= Y - y**).

Однако и здесь не все так гладко. Давайте взглянем на таблицу с тренировочными данными и выходными значениями для трех выходных узлов вместе с разными функциями ошибок:

Выходное значение нейрона	Целевое значение	Ошибка Целевое - выход	Ошибка $(\text{Целевое} - \text{выход})^2$
0,5	0,6	0,1	0,01
0,7	0,6	-0,1	0,01
1,1	1,1	0	0
Сумма		0	0,02

Функция ошибки, которой мы пользовались ранее (**целевое – выход**), не совсем нам подходит, так как можно видеть, что если мы решим использовать сумму ошибок по всем узлам в качестве общего показателя того, насколько хорошо обучена сеть, то эта сумма равна нулю! Нулевая сумма означает отсутствие ошибки. Отсюда следует, что простая разность значений (**целевое – выход**), не годится для использования в качестве меры величины ошибки.

Во втором варианте, в качестве меры ошибки используется квадрат разности: ((**целевое – выход**)²). Этот вариант предпочтительней первого, поскольку, как видно из таблицы, сумма ошибок на выходе не дает нулевой вариант. Кроме того, такая функция имеет еще ряд преимуществ над первой, делает функцию ошибки непрерывно гладкой, исключая провалы и скачки, тем самым улучшая работу метода градиентного спуска. Еще одно преимущество заключается в том, что при приближении к минимуму градиент уменьшается, что уменьшает корреляцию через точку минимума.

Чтобы воспользоваться методом градиентного спуска, нам нужно применить метод дифференциального исчисления. Не пугайтесь, всё не так сложно, как может показаться.

Дифференциальное исчисление – это просто математически строгий подход к определению величины изменения одних величин при изменении других. Например, мы можем говорить о скорости изменения чего угодно, ускорения или любой другой физической величины, или математической функции.

Не изменяющиеся величина

Если мы представим автомобиль, движущийся с постоянной скоростью в 1,5 км/мин, то отвечая на вопрос, как меняется скорость автомобиля с течением времени, ответ утвердительный никак, ноль, так как его скорость постоянна:

Напомню, дифференциальное исчисление сводится к нахождению изменения одной величины в результате изменения другой. В данном случае нас интересует, как скорость изменяется со временем.

Сказанное, можно записать в следующей математической форме:

$$\frac{ds}{dt} = 0$$

Линейное изменение

А теперь представим тот же автомобиль, с начальной скоростью 1,5 км/мин, но в определенный момент, водитель жмет на газ, и автомобиль начинает набирать скорость (равномерно ускоряться). И по истечении трех минут, от момента, когда мы нажали педаль газа, его скорость станет равной 2,1 км/мин.

Из графика видно, что увеличение скорости автомобиля, происходит с постоянной скоростью изменения (равномерным ускорением), откуда функция зависимости скорости от времени, выглядит как прямая линия.

Изначально, в нулевой момент времени, скорость равна 1,5 км/мин. Далее мы добавляем по 0,2 км в минуту. Таким образом, искомое выражение приобретает следующий вид:

Скорость = 1,5 + (0,2 * время)

S = 1,5 + 0,2 t

В итоговом выражении, вы легко увидите уравнение прямой. Где коэффициент = 0,2 – величина крутизны наклона прямой, а постоянный член = 1,5 – точка через которую проходит линия на оси координат y.

Так будет выглядеть выражение, которое скажет нам о том, что между скоростью движения автомобиля и временем существует зависимость:

$$\frac{ds}{dt} = 0{,}2$$

Каждую минуту, скорость изменяется на значение 0,2.

Не равномерное изменение

Возьмём всё тот же автомобиль, который стоит на месте. Сидя в нем, вы начинаете жать в “пол” педаль газа, удерживая её в этом положении. Скорость движения автомобиля, за счет инерции, будет возрастать не равномерно. Ежеминутное приращение скорости будет с каждой минутой увеличиваться.

Приведем в таблице, значения скорости в каждую минуту:

Время(мин)	Скорость(км/мин)
0	0
1	1
2	4
3	9
4	16
5	25

Эти данные представляют собой выражение:

$$s = t^2$$

Какова скорость изменения скорости автомобиля в каждый момент времени?

Если посмотреть на два предыдущих примера, то в них скорость изменения скорости определялась наклоном графика, коэффициентом крутизны прямой линии **A** . Когда автомобиль двигался с постоянной скоростью, его скорость не изменялась, и скорость изменения скорости равна 0. Когда автомобиль равномерно набирал скорость, скорость его изменения составляла 0,2 км/мин, на протяжении всего времени движения автомобиля в этом режиме.

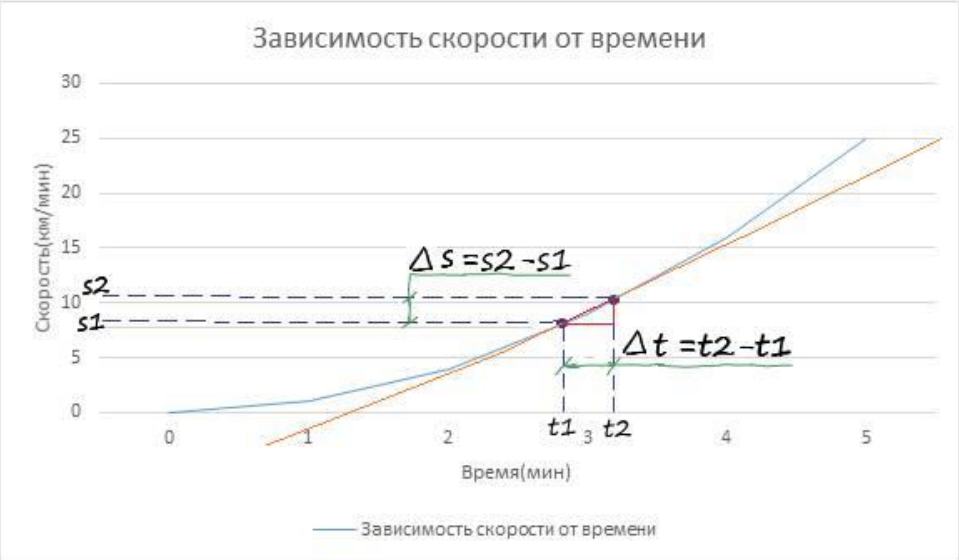
А как тогда поступить в этом случае? Как узнать изменение скорости по кривой?

Применение дифференциального исчисления, понятие производной

После трех минут с момента начала движения (t=3), скорость составит 9 км/мин. Сравним со скоростью в конце пятой минуты. После пяти минут с момента начала движения (t=5), скорость составляет 25 км/мин. Не важно, что скорость 25 км/мин – сопоставима со скоростью пули, ведь это воображаемая машина, и едет она с той скоростью, с какой мы захотим. Если провести касательную линию в этих точках, то окажется, что угол наклона у них совершенно разный:

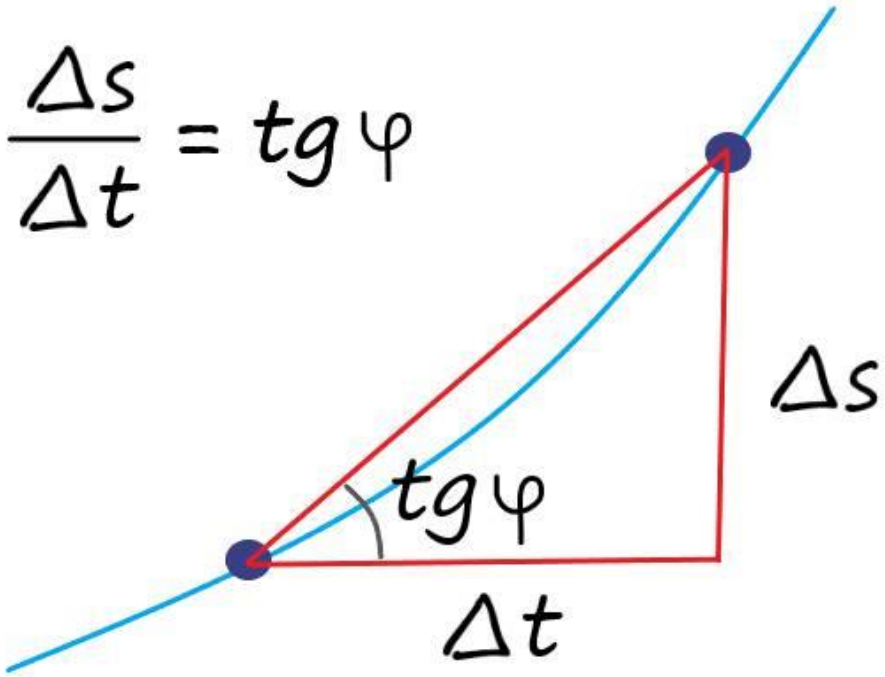
Вы видите, что чем больше скорость в точке касательной, тем её наклон круче. Оба наклона представляют искомую скорость изменения скорости движения. Можно сравнить с вторым примером – линейное изменение.

Но как измерить наклон этих линий? Для этого давайте представим, что наша касательная (t = 3, s = 9), пересекает функцию в двух точках, расстояние между которыми очень мало:



Зная координаты этих точек и проведя проекции по осям, можно вычислить расстояние между этими точками.

Если представить прямоугольный треугольник где гипотенуза – это прямая между двумя точками, а его катеты равны разности проекциям точек по осям (Δt и Δs), то поделив противолежащий катет на прилежащий получим тангенс угла, который и будет являться коэффициентом крутизны. Зная который, как во втором примере, мы легко определим изменение скорости в момент Δt.



Как мы знаем, скорость изменения – это наклон прямой, которую из второго примера мы уже умеем находить. Значит, около точки (t=3), наш коэффициент крутизны будет равен:

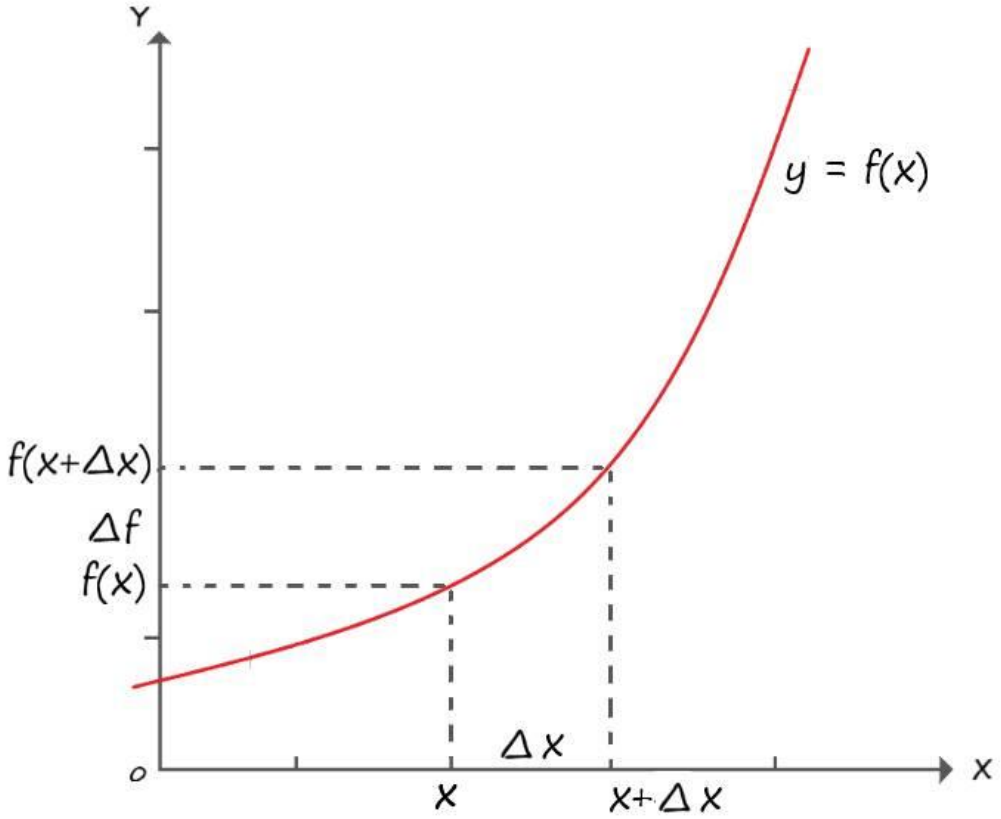
$$\frac{\Delta s}{\Delta t} = \frac{11-7}{3{,}33-2{,}67} = 6{,}06$$

Значит, скорость изменения скорости в момент времени три минуты составляет 6,06 км/мин.

Производная функции

Мы можем говорить о скорости изменения чего угодно – физической величины, экономического показателя и так далее.

Рассмотрим функцию **y = f (x)** . Отметим на оси **X** , некоторое значение аргумента **x** , а на оси **Y** – соответствующее значение функции **y = f (x)** .



Дадим аргументу x, некоторое приращение, обозначенное как **Δx** . Попадаем в точку **x+Δx** . А соответствующие этим значениям аргументов, значение функции обозначим соответственно **f (x)** , **Δ f** и **f (x +Δx)** . Приращение аргумента **Δx** , есть аналог промежутка времени **Δ t** , а соответствующее приращение функции – это аналог пути **Δ s** , пройденного за время **Δ t** .

Если представить, что **Δx** – бесконечно мала, т.е. стремиться к нулю (**Δx->0**), то выражение нахождения изменения скорости можно записать как:

$$\lim_{x\to 0}\frac{\Delta f}{\Delta x}=\lim_{x\to 0}\frac{f(x+\Delta x)-f(x)}{\Delta x}$$

Или исходя из геометрического представления, описанного ранее:

$$\lim_{x->0} \frac{\Delta f}{\Delta x} = \lim_{x->0} tg\varphi = tg\varphi$$

Отсюда вывод, что производная функции **f (x)** в точке **x** – это предел отношения приращения функции к приращению её аргумента, когда приращение аргумента стремиться к нулю.

Нахождение некоторых табличных производных

Решим найденным способом, наш первый пример, когда скорость автомобиля была постоянной, на всем промежутке времени. В этом примере, приращение функции равно нулю (Δ **s = 0**), и соответственно тангенса угла не существует:

$\Delta s = s(t+\Delta t) - s(t) = s(t) - s(t) = 0$

$$\lim_{\Delta x->0} \frac{\Delta s}{\Delta t} = \lim_{\Delta x->0} \frac{0}{\Delta t} = 0$$

Итак, имеем первый результат – производная константы равна нулю. Этот результат мы уже выводили ранее:

$$\frac{ds}{dt} = 0$$

Откуда можно сформулировать правило, что производная константы, равна нулю.

s (t) = c, где c – константа

c′ = 0

Запись **c′** – означает что берется производная по функции.

Во второй примере, когда изменение скорости автомобиля проходило линейно, с постоянным изменением, найти производную функции (**s = 0,2 t + 1,5**), не зная правил дифференцирования сложных функций, мы пока не сможем, поэтому отложим этот пример на потом.

Продолжим с решения третьего примера, когда изменение скорости автомобиля проходило не линейно:

s = t ²

Приращение функции и производная:

s(t) = t²

$\Delta s = s(t+\Delta t) - s(t) = (t+\Delta t)^2 - t^2 = t^2 + 2t\Delta t + \Delta t^2 - t^2 = \Delta t(2t+\Delta t)$

$$\lim_{\Delta t \to 0} \frac{\Delta s}{\Delta t} = \frac{\Delta t(2t+\Delta t)}{\Delta t} = \lim_{\Delta t \to 0} 2t+\Delta t = 2t$$

Вот мы и решили наш третий пример! Нашли формулу точного изменения скорость от времени. Вычислим производную, в всё той же точки t = 3.

s (t) = t ²

s ′(t) = 2*3 = 6

Точный ответ, в пределах небольшой погрешности, почти сошелся с вычисленном до этого приближенным ответом.

Попробуем усложнить пример. Предположим, что скорость движения автомобиля описывается кубической функцией времени:

s (t) = t ³

Приращение и производная:

s(t) = t³

$\Delta s = s(t+\Delta t) - s(t) = t^3 + 3\,t^2\Delta t + 3t\Delta\, t^2 + \Delta\, t^3 - t^3 = \Delta t(3\,t^2 + 3t\Delta t + \Delta t^2)$

$$\lim_{\Delta t \to 0} \frac{\Delta s}{\Delta t} = \lim_{\Delta t \to 0} \frac{\Delta t(3\,t^2 + 3t\Delta t + \Delta t^2)}{\Delta t} =$$

$$= \lim_{\Delta t \to 0} 3\,t^2 + 3t\Delta t + \Delta t^2 = 3\,t^2$$

Из двух последних примеров (с производными функций **s (t) = t ²** и **s (t) = t ³**) следует, что показатель степени числа, становится его произведением, а степень уменьшается на единицу:

s (t) = t^n

$$s'(t) = nt^{n-1}$$

А чему равна производная от аргумента функции? Давайте узнаем…

s (t) = t

Приращение:

$\Delta s = s(t+\Delta t) - s(t) = t + \Delta t - t = \Delta t$

Производная:

$$\lim_{\Delta t \to 0} \frac{\Delta s}{\Delta t} = \lim_{\Delta t \to 0} \frac{\Delta t}{\Delta t} = 1$$

Получается, что производная от переменной:

t ′ = 0

Правила дифференцирования и дифференцирование сложных функций

Дифференцирование суммы

(u + v)′ = u ′ + v ′ , где **u** и **v** – функции.

Пусть **f(x) = u(x) + v(x)** . Тогда:

Δ f = f (x +Δ x) – f (x) = u (x +Δ x) + v (x +Δ x) – u (x) – v (x) = u (x) + Δ u + v (x) + Δ v – u (x) – v (x) = Δ u + Δ v

Тогда имеем:

$$f'(x) = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x} = \lim_{\Delta x \to 0} \frac{\Delta u + \Delta v}{\Delta x} =$$

$$= \lim_{\Delta x \to 0} \frac{\Delta u}{\Delta x} + \frac{\Delta v}{\Delta x}$$

Дроби **Δ u /Δx** и **Δ v /Δx** при **Δx->0** стремятся соответственно к **u ′(x)** и **v ′ (x)** . Сумма этих дробей стремится к сумме **u ′(x) + v ′ (x)** .

f ′(x) = u ′ (x) + v ′ (x)

Дифференцирование произведения

(u * v)′ = u ′ v + v ′ u , где **u** и **v** – функции

Разберем, почему это так. Обозначим **f (x) = u (x) * v (x)** . Тогда:

Δ f = f (x +Δ x) – f (x) = u (x +Δ x) * v (x +Δ x) – u (x) * v (x) = (u (x) + Δ u) * (v (x) + Δ v) – u (x) * v (x) = u (x) v (x) + v (x)Δ u + u (x)Δ v + Δ u Δ v – u (x) v (x) = v (x)Δ u + u (x)Δ v + Δ u Δ v

Далее имеем:

$$f'(x) = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x} = \lim_{\Delta x \to 0} \frac{v(x)\Delta u + u(x)\Delta v + \Delta u \Delta v}{\Delta x} =$$

$$= \lim_{\Delta x \to 0} \frac{\Delta u}{\Delta x} v(x) + \frac{\Delta v}{\Delta x} u(x) + \frac{\Delta u}{\Delta x} \Delta v$$

Первое слагаемое стремиться к **u′(x) v(x)** . Второе слагаемое стремиться к **v′(x)* u(x)** . А третье, в дроби **Δ u /Δ x** , в пределе даст число **u′(x)** , а поскольку множитель **Δv** стремиться к нулю, то и вся эта дробь обратится в ноль. А следовательно, в результате получаем:

f ′(x) = u ′(x) v (x) + v ′(x) u (x)

Из этого правила, легко убедиться, что:

(c * u)′ = c ′ u + c u ′ = c u ′

Поскольку, **c** – константа, поэтому ее производная равна нулю **(c′ = 0)** .

Зная это правило мы без труда, найдем изменение скорости второго примера.

Применим к выражению правило дифференцирование суммы:

s ′ (t) = (0,2t) ′ + (1,5) ′

Теперь по порядку, возьмём выражение – **(0,2t) ′** . Как брать производную произведения константы и переменной мы знаем:

(0,2t) ′ = 0,2

А производная самой константы равна нулю – **(1,5) ′ = 0** .

Следовательно, скорость изменения скорости, второго примера:

s ′ (t) = 0,2

Что совпадает с нашим ответом, полученном ранее во втором примере.

Дифференцирование сложной функции

Допустим, что в некоторой функции, **y** сама является функцией:

f = y²

y = x²+x

Представим дифференцирование этой функции в виде:

$$\frac{df}{dx} = \frac{df}{dy} * \frac{dy}{dx}$$

Нахождение производной в этом случае, осуществляется в два этапа.

$$\frac{df}{dx} = \frac{dy^2}{dy} * \frac{d(x^2 + x)}{dx} =$$

$$= 2y * (2x+1)$$

Мы знаем, как решить производную типа: $\mathbf{dy^2/dy = 2y}$

А также знаем, как решать производную суммы: $\mathbf{x^2 + x = (x^2)' + x' = 2x+1}$

Тогда:

$\mathbf{2(x^2+x) * (2x+1) = (2x^2+2x) * (2x+1) = 4x^3+6x^2+2x}$

Я надеюсь, вам удалось понять, в чем состоит суть дифференциального исчисления.

Используя описанные, методы дифференцирования выражений, вы сможете понять механизм работы метода градиентного спуска.

В качестве небольшого дополнения, приведу список наиболее распространённых табличных производных:

Функция	Производная
x	1
x^n	nx^{n-1}
e^x	e^x
$c(const)$	0
a^x	$a^x \ln a$
$\sqrt[n]{x}$	$\frac{1}{n\sqrt[n]{x^{n-1}}}$
$\sin x$	$\cos x$
$\cos x$	$-\sin x$
$td\ x$	$\frac{1}{\cos^2 x}$
$ctd\ x$	$\frac{1}{-\sin^2 x}$
$\ln x$	$\frac{1}{x}$
$\log_a x$	$\frac{1}{x}\log_a e$
$ld\ x$	$\frac{1}{x} ld\ e$
$\arcsin x$	$\frac{1}{\sqrt{1-x^2}}$
$\arccos x$	$-\frac{1}{\sqrt{1-x^2}}$
$arctd\ x$	$\frac{1}{1+x^2}$
$arcctd\ x$	$-\frac{1}{1+x^2}$

Основные свойства

$$(cu)' = cu'$$

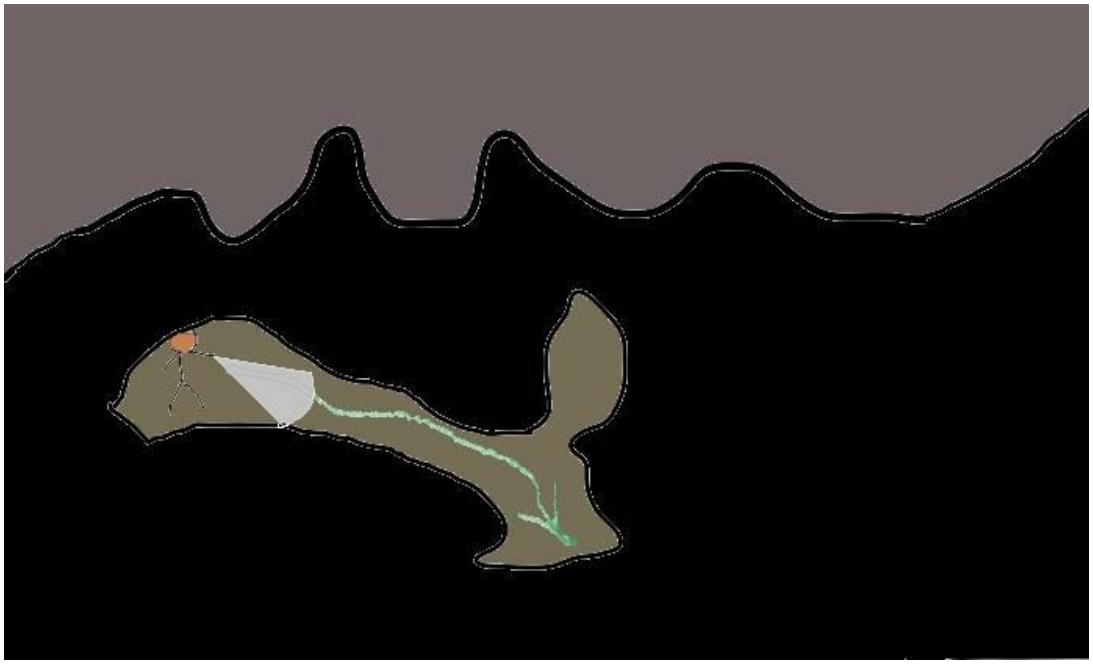
$$(u \pm v)' = u' \pm v'$$

$$(u*v)' = u' * v + u * v'$$

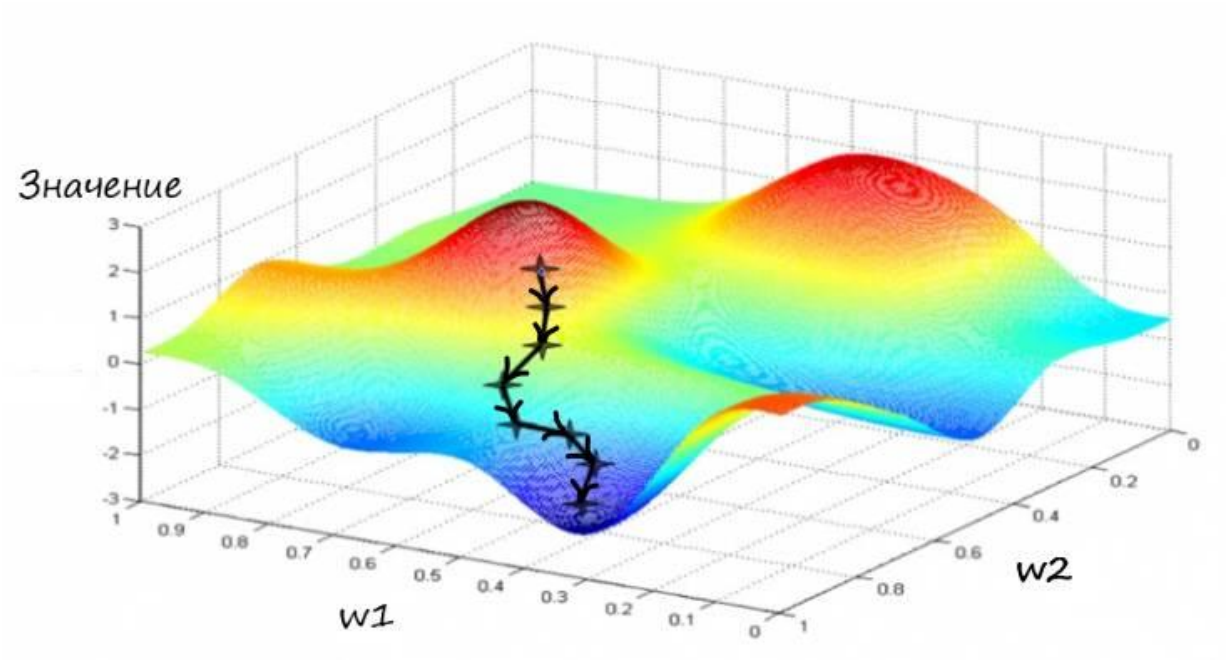
$$\left(\frac{u}{v}\right)' = \frac{u' * v - u * v'}{v^2}$$

Зачем нам дифференцировать функции

Еще раз вспомним как мы спускаемся по склону. Что в кромешной тьме, мы хотим попасть к его подножью, имея в своем арсенале слабенький фонарик.



Опишем эту ситуацию, по аналогии с математическим языком. Для этого проиллюстрируем график метода градиентного спуска, но на этот раз применительно к более сложной функции, зависящей от двух параметров. График такой функции можно представить в трех измерениях, где высота представляет значение функции:



К слову, отобразить визуально такую функцию, с более чем двумя параметрами, как видите, будет довольно проблематично, но идея нахождения минимума методом градиентного спуска останется ровно такой же.

Этот слайд отлично показывает всю суть метода градиентного спуска. Очень хорошо видно, как функция ошибки объединяет весовые коэффициенты, как она заставляет работать их согласованно. Двигаясь в сторону минимума функции ошибки, мы можем видеть координаты весов, которые необходимо изменять в соответствии с координатами точки – которая движется вниз.

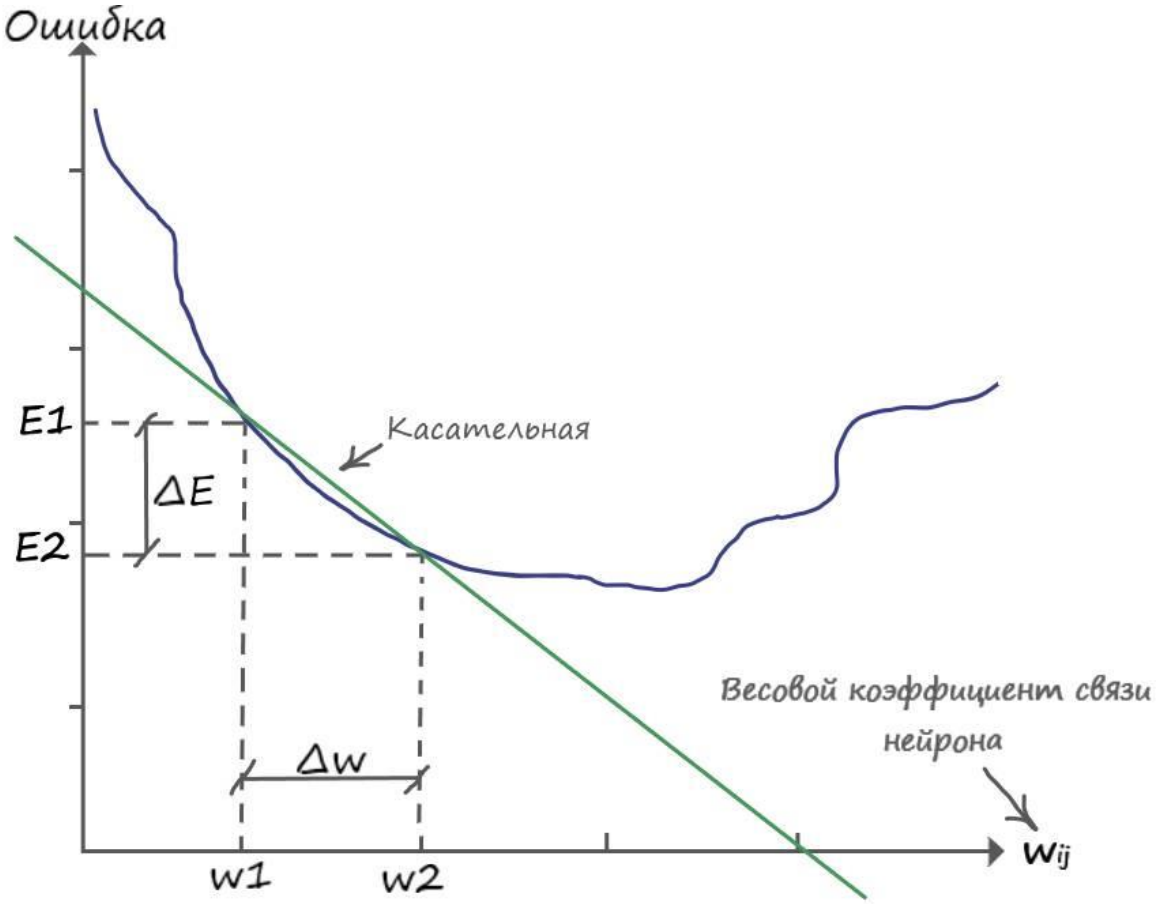
Представим ось значение, как ось ошибка. Очень хорошо видно, что функция ошибки общая для всех значений весов. Соответственно – координаты точки значения ошибки, при определенных значениях весовых коэффициентов, тоже общие.

При нахождении производной функции ошибки (угол наклона спуска в точке), по каждому из весовых коэффициентов, находим новую точку функции ошибки, которая обязательно стремиться двигаться в направлении её уменьшения. Тем самым, находим вектор направления.

А обновляя веса в соответствии со своим входом, на величину угла наклона, находим новые координаты этих коэффициентов. Проекции этих новых координат на ось ошибки (значение низ лежащей точки на графике), приводят в ту самую новую точку функции ошибки.

Как происходит обновление весовых коэффициентов?

Для ответа на этот вопрос, изобразим наш гипотетический рельеф в двумерной плоскости (гипотетический – потому что функция ошибки, зависящая от аргумента весовых коэффициентов, нам не известна). Где значение высоты будет ошибка, а за координаты по горизонтали нахождения точки в данный момент, будет отвечать весовой коэффициент.



Тьму, через которую невозможно разглядеть даже то, что находится под ногами, можно сравнить с тем, что нам не известна функция ошибки. Так как, даже при двух, постоянно изменяющихся, параметрах неизвестных в функции ошибки, провести её точную кривую на координатной плоскости не представляется возможным. Мы можем лишь вычислить её значение в точке, по весовому коэффициенту.

Свет от фонаря, можно сравнить с производной – которая показывает скорость изменения ошибки (где в пределах видимости фонаря, круче склон, чтоб сделать шаг в его направлении). Следуя из основного понятия производной – измерения изменения одной величины, когда изменяется вторая, применительно к нашей ситуации, можно сказать что мы измеряем изменение величины ошибки, когда изменяются величины весовых коэффициентов.

А шаг, в свою очередь, отлично подходит на роль обновления нашего весового коэффициента, в сторону уменьшения ошибки.

Вычислив производную в точке, мы вычислим наклон функции ошибки, который нам нужно знать, чтобы начать градиентный спуск к минимуму:

$$\lim_{w_{ij}->0} \frac{\Delta E}{\Delta w_{ij}} = \lim_{w_{ij}->0} tg\varphi = tg\varphi$$

$$\frac{dE}{dw_{ij}}$$

Ij – определитель веса, в соответствии со своим входом. Если это вход **x1** – то его весовой коэффициент обозначается как – **w11** , а у входа **x2** – обозначается как **-w21** . Чем круче наклон касательной, тем больше скорость изменения ошибки, тем больше шаг.

Запишем в явном виде функцию ошибки, которая представляет собой сумму возведенных в квадрат разностей между целевым и фактическим значениями:

$$\frac{dE}{dw_{ij}} = \frac{d(T-y)^2}{dw_{ij}}$$

Разобьем пример на более простые части, как мы это делали при дифференцировании сложных функций:

$$\frac{dE}{dw_{ij}} = \frac{dE}{dy} * \frac{dy}{dw_{ij}}$$

Продифференцируем обе части поочередно:

$$\frac{dE}{dy} = \frac{d(T-y)^2}{dy} = 2(T-y)*(-1)$$

$$= -2(T-y)$$

Так как выход нейрона – **f(x) = y** , а взвешенная сумма – **y = ∑ I w ij *x i** , где **x i** – известная величина (константа), а весовые коэффициенты **w ij** – переменная, производная по которой, дает как мы знаем единицу, то взвешенную сумму можно разбить на сумму простых множителей:

$$\frac{d(\Sigma_i w_{ij} * x_i)}{d w_{ij}} =$$

$$= x_1 \frac{d(w_{11})}{d w_{11}} + x_2 \frac{d(w_{21})}{d w_{21}} \dots x_i \frac{d(w_{ij})}{d w_{ij}}$$

Откуда нетрудно найти:

$$\frac{d(w_{ij} * x_i)}{d w_{ij}} = x_i$$

Значит, для того чтобы обновить весовой коэффициент по своей связи:

$$\frac{dE}{dw_{ij}} = \frac{dE}{dy} * \frac{dy}{dw_{ij}} = -2(T - y) * x_i$$

Прежде чем записать окончательный ответ, избавимся от множителя 2 в начале выражения. Мы спокойно можем это сделать, поскольку нас интересует только направление градиента функции ошибки. Не столь важно, какой множитель будет стоять в начале этого выражения, 1, 2 или любой другой (лишь немного потеряем в масштабировании, направление останется прежним). Поэтому для простоты избавимся от неё, и запишем окончательный вид производной ошибки:

$$\frac{dE}{dw_{ij}} = \frac{dE}{dy} * \frac{dy}{dw_{ij}} = -(T - y) * x_i$$

Всё получилось! Это и есть то выражение, которое мы искали. Это ключ к тренировке эволюционировавшего нейрона.

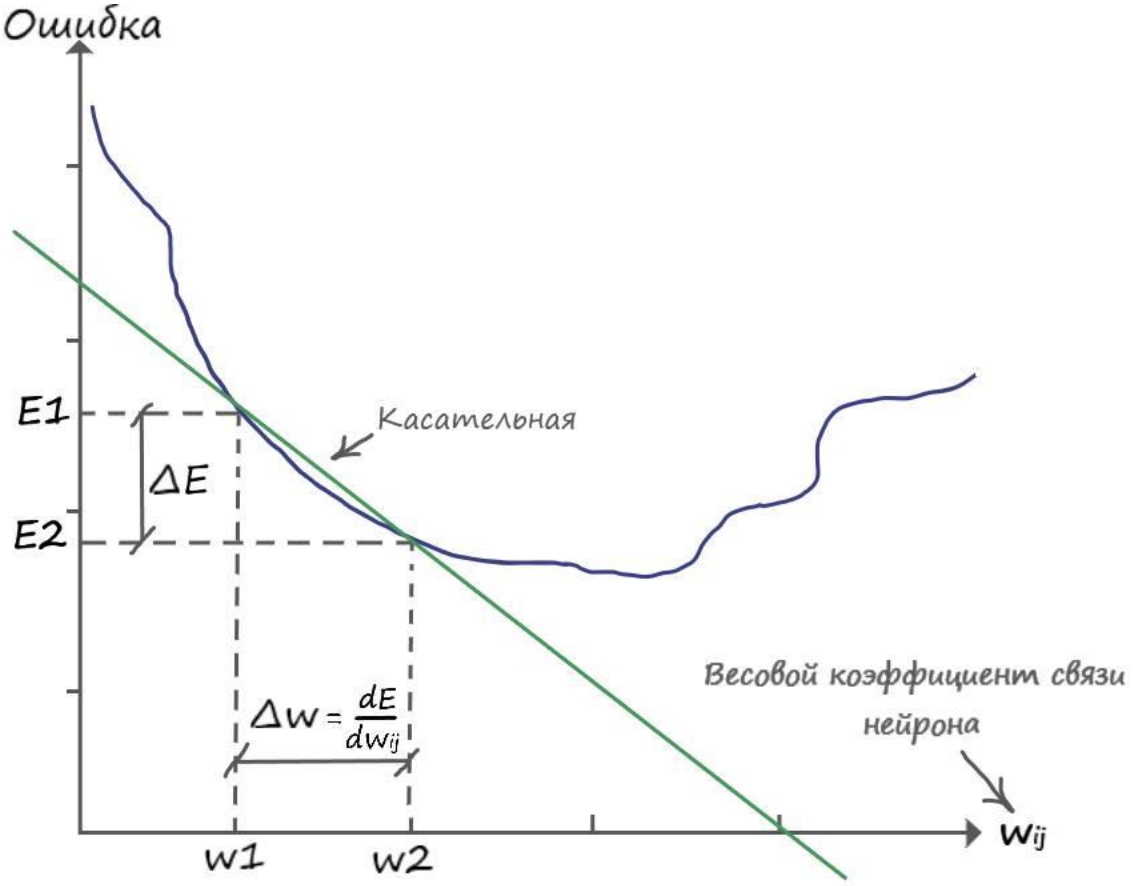
Как мы обновляем весовые коэффициенты

Найдя производную ошибки, вычислив тем самым наклон функции ошибки (подсветив фонариком, подходящий участок для спуска), нам необходимо обновить наш вес в сторону уменьшения ошибки (сделать шаг в сторону подсвеченного фонарем участка). Затем повторяем те же действия, но уже с новыми (обновлёнными) значениями.

Для понимания как мы будем обновлять наши коэффициенты (делать шаги в нужном направлении), прибегнем к помощи так уже нам хорошо знакомой – иллюстрации. Напомню, величина шага зависит от крутизны наклона прямой (tgφ). А значит величина, на которую мы обновляем наши веса, в соответствии со своим входом, и будет величиной производной по функции ошибки:

$$\Delta wij = \frac{dE}{dw_{ij}}$$

Вот теперь иллюстрируем:



Из графика видно, что для того чтобы обновить вес в большую сторону, до значения (**w2**), нужно к старому значению (**w1**) прибавить дельту (Δ**w**), откуда: (**w2 = =w1+Δw**). Приравняв (Δ**w**) к производной ошибки (величину которой уже знаем), мы спускаемся на эту величину в сторону уменьшения ошибки.

Так же замечаем, что (**E 2 – E 1 = -Δ E**) и (**w 2 – w 1 = Δ w**), откуда делаем вывод:

Δ w = -Δ E /Δ w

Ничего не напоминает? Это почти то же, что и дельта линейного классификатора (**ΔA = E/x**), подтверждение того что наша эволюция прошла с поэтапным улучшением математического моделирования. Таким же образом, как и с обновлением коэффициента (**A = A+ΔA**), линейного классификатора, обновляем весовые коэффициенты:

новый w ij = старый w ij -(– Δ E /Δ w)

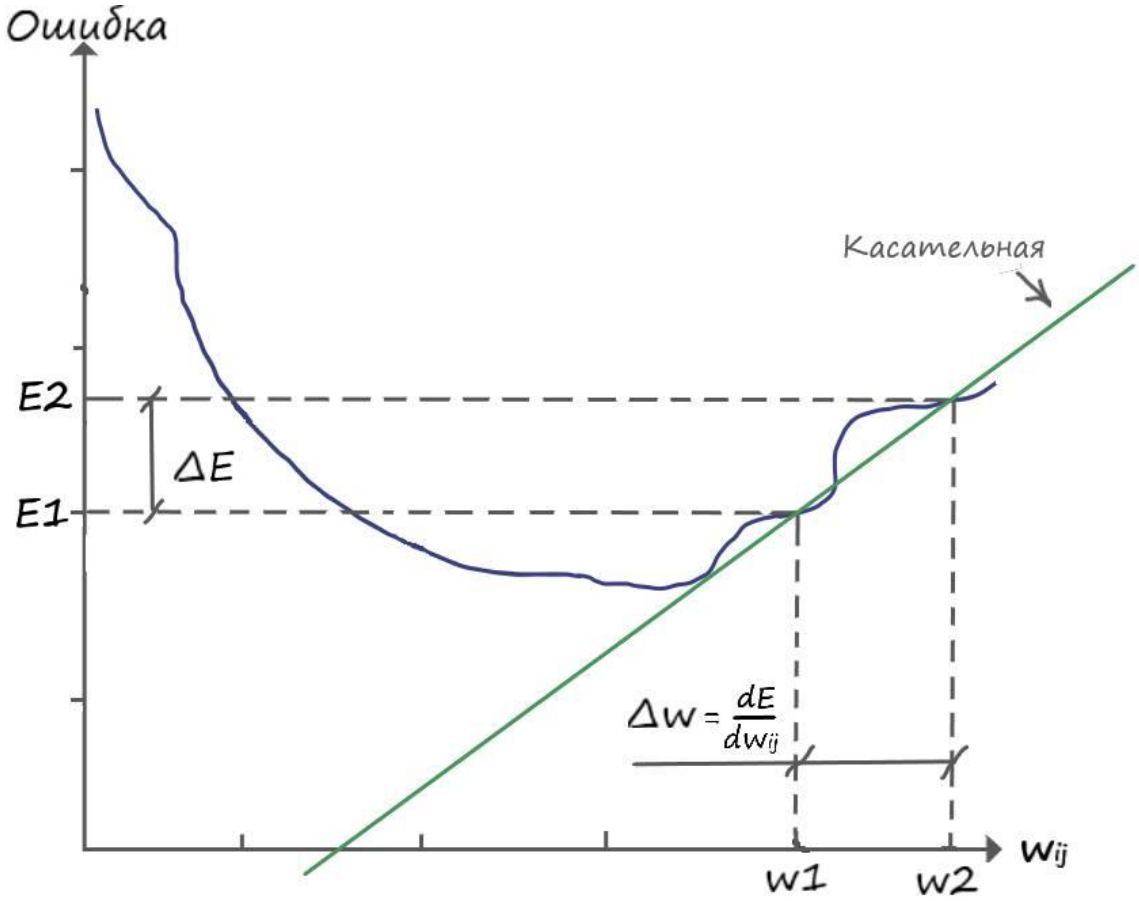
Знак минус, для того чтобы обновить вес в большую сторону, для уменьшения ошибки. На примере графика – от **w1** до **w2** .

В общем виде выражение записывается как:

новый w ij = старый w ij – dE /dw ij

Еще одно подтверждение, постепенного, на основе старого аппарата, хода эволюции, в сторону улучшения классификации искусственного нейрона.

Теперь, зайдем с другой стороны функции ошибки:



Снова замечаем, что (**E 2 – E 1 = Δ E**) и (**w 2 – w 1 = Δ w**), откуда делаем вывод:

Δ w = Δ E /Δ w

В этом случае, для обновления весового коэффициента, в сторону снижения функции ошибки, а значит до значения находящееся левее (**w1**), необходимо от значения (**w1**) вычесть дельту (**Δw**):

новый w ij = старый w ij - Δ E /Δ w

Получается, что независимо от того, какого знака производная ошибки от весового коэффициента по входу, вычитая из старого значения – значение этой производной, мы движемся в сторону уменьшения функции ошибки. Откуда можно сделать вывод, что последнее выражение, общее для всех возможных случаев обновления градиента.

Запишем еще раз, обновление весовых коэффициентов в общем виде:

новый w ij = старый w ij – dE /dw ij

Но мы забыли еще об одной важной особенности… Сглаживания! Без сглаживания величины дельты обновления, наши шаги будут слишком большие. Мы подобно кенгуру, будем прыгать на большие расстояния и можем перескочить минимум ошибки! Используем прошлый опыт, чтоб устранить этот недочёт.

Вспоминаем старое выражение при нахождении сглаженного значения дельты линейного классификатора: **ΔA = L*(E/x)** . Где (**L**) – скорость обучения, необходимая для того, чтобы мы делали спуск, постепенно, небольшими шашками.

Ну и наконец, давайте запишем окончательный вариант выражения при обновлении весовых коэффициентов:

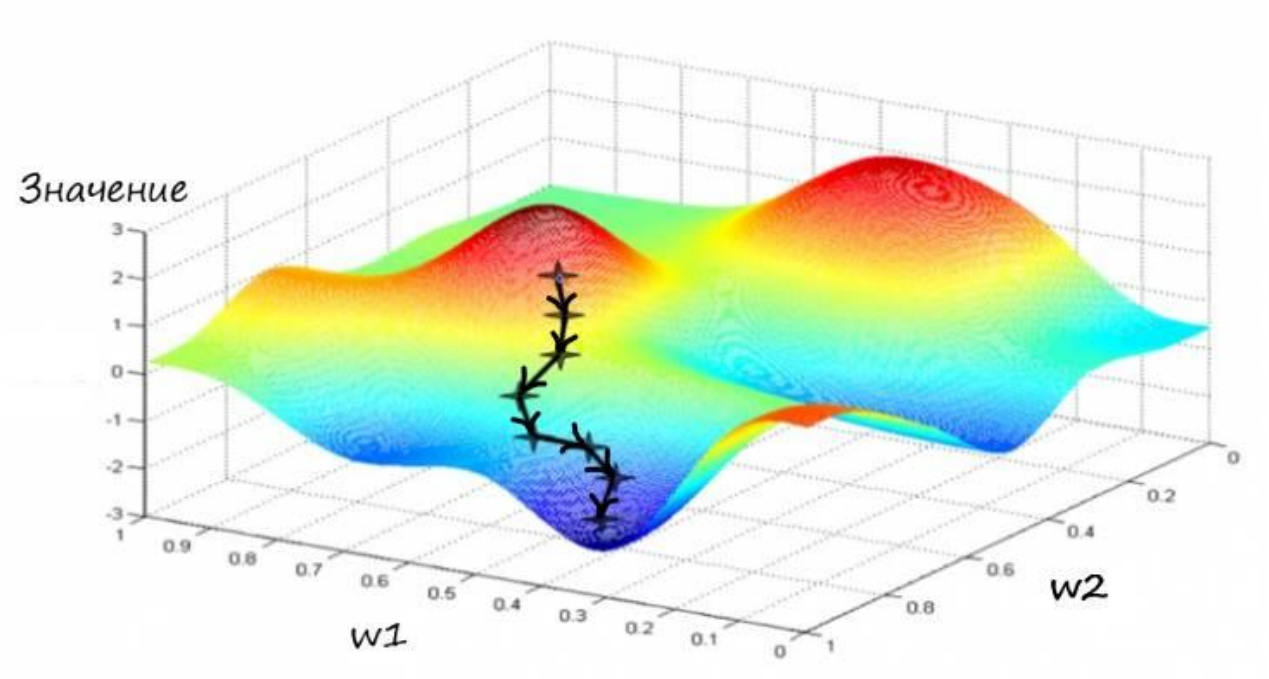
новый w ij = старый w ij – L *(dE /dw ij)

Еще раз можем убедиться, в постепенном улучшении свойств, в ходе эволюции искусственного нейрона. Много из того что реализовывали ранее остается, лишь небольшая часть подверглась эволюционному улучшению.

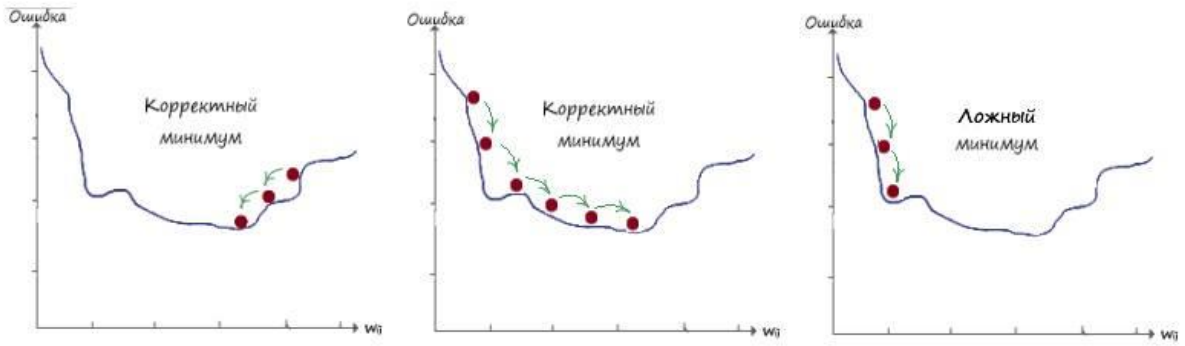
Ложный минимум

Если еще раз взглянуть на трехмерную поверхность, можно увидеть, что метод

градиентного спуска может привести в другую долину, которая расположена правее, где минимум значения будет меньше относительно той долины, куда попали мы сейчас, т.е. эта долина не является самой глубокой.



На следующей иллюстрации показано несколько вариантов градиентного спуска, один из которых приводит к ложному минимуму.



Поздравляю! Мы прошли самую основу в теории нейронных сетей – **метод градиентного спуска** . Освоив этот материал, в дальнейшем, изучение теории искусственных нейронных сетей, не будет представлять для вас значимого труда.

Как работает эволюционировавший нейрон

Ну вот и настало время проверить практически, все наши умозаключения, касающиеся работы нашего искусственного нейрона, после первой эволюции. Для этого прибегнем к помощи Python, но сначала покажем наш список с данными, с которого мы это всё затеяли:

x (длина)	Y(высота)
1	4,3
2	7
3	8
3,5	10,1
4	11,3
6	14,2
7,5	18,5
8,5	19,3
9	21,4

Если по координатам построить точки на плоскости, то мы заметим, что их значения лежат возле значений графика функции – **y = 2 x + 2,5** .

Программа

```
import random
# Инициализируем любым числом крутизны наклона прямой w1 = A
w1 = 0.4
w1_vis = w1 # Запоминаем начальное значение крутизны наклона
# Инициализируем параметр w2 = b – отвечающий за точку прохождения прямой через ос Y
w2 = random.uniform(-4, 4)
w2_vis = w2 # Запоминаем начальное значение параметра
# Вывод данных начальной прямой
print('Начальная прямая: ', w1, '* X + ', w2)

# Скорость обучения
lr = 0.001
# Зададим количество эпох
epochs = 3000
# Создадим массив (выборку входных данных) входных данных x1
arr_x1 = [1, 2, 3, 3.5, 4, 6, 7.5, 8.5, 9]

# Значение входных данных второго входа всегда равно 1
x2 = 1
# Создадим массив значений (целевых значений)
arr_y = [4.3, 7, 8.0, 10.1, 11.3, 14.2, 18.5, 19.3, 21.4]
# Прогон по выборке
for e in range(epochs):
for i in range(len(arr_x1)): # len(arr) – функция возвращает длину массива
# Получить x координату точки
x1 = arr_x1[i]

# Получить расчетную y, координату точки
y = w1 * x1 + w2

# Получить целевую Y, координату точки
target_Y = arr_y[i]

# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y – y)

# Меняем вес при x, в соответствии с правилом обновления веса
w1 -= lr * E * x1
```

```
# Меняем вес при x2 = 1
#w2 -= rate * E * x2 # Т.к. x2 = 1, то этот множитель можно не писать
w2 -= lr * E

# Вывод данных готовой прямой
print('Готовая прямая: ', w1, '* X + ', w2)
```

Данный код, как и все другие, вы можете скачать по ссылке: https://github.com/CaniaCan/neural master

Опишем код программы:

В самом начале программы импортируем модуль для работы со случайными числами:

```
import random
```

При помощи которого, случайным числом, создаем весовой коэффициент параметра (w2 = b) – отвечающий за точку прохождения прямой через ос Y:

```
w2 = random.uniform(-4,  4)
```

Метод модуля random – uniform(from, to), генерирует случайное вещественное число от from до to включительно.

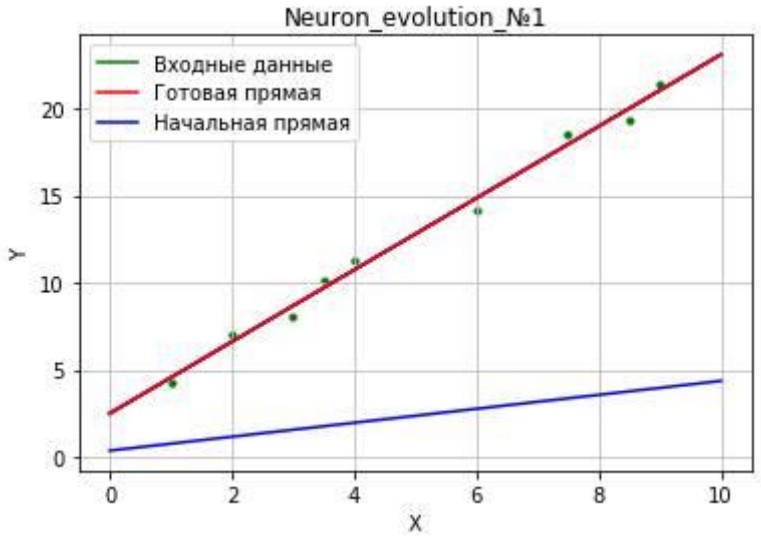
В нашей программе, как видно, не так много изменений, по сравнению с той что мы написали до этого. Мы добавили второй вход (x2 = 1), со своим весовым коэффициентом (w2). Коэффициент (A) – переименовали в весовой коэффициент (w1), параметр (b) – в весовой коэффициент (w2). Ну и конечно же, реализовали новую улучшенную функцию ошибки, и обновление весовых коэффициентов по методу градиентного спуска.

В результате чего, наш эволюционировавший нейрон, теперь гораздо лучше справляется с задачей классификации. Теперь он может классифицировать данные по двум входам, тем самым получая линейный классификатор с пересечением прямой по всей оси Y, а не только строго в точке нуля.

Давайте взглянем на результат чтобы убедиться в этом:

Начальная прямая: 0.4 * X + 0.3652477754014445

Готовая прямая: 2.058410130422831 * X + 2.5013583972057263



Вы видите! Как наш искусственный нейрон прекрасно справляется с задачей. Даже еле различимые на глаз данные, он легко смог линейно разделить.

Теперь зададим условие, как это делали ранее. Если данные расположены выше классифицирующий линии, то это вид жирафа, а все что ниже – крокодилы. Будем делать это

подавая на входы, значения, которые нейрон до этого не видел и посмотрим, сможет ли обученный нейрон, самостоятельно определить к какому виду они принадлежат.

```
x1 = input("Введите значение ширины X: ")
x1 = int(x1)
T = input("Введите значение высоты Y: ")
T = int(T)
y = w1 * x1 + w2

# Условие
if T &gt; y:
print('Это жираф!')
else:
print('Это крокодил!')
```

После ввода наших значений, следует условие, которое проверяет, какого вида эти данные, жирафы или крокодилы, и возвращает ответ на поставленный вопрос.

Введите значение ширины X: 4
Введите значение высоты Y: 15
Это жираф!

Резюмируя проделанную работу:

Получив задание, классифицировать два вида животных, по параметрам, определяющим размеры их тела, с некоторой выборкой данных (значений и ответов), мы смогли запрограммировать искусственный нейрон, основываясь на элементарных знаниях математики, а именно линейной функции, проходящей через начало координат (**y = Ax**). Определив, что, данные лежащие выше прямой относились бы к одному классу, а все точки данных лежащих ниже – к другим. Тем самым мы лишили бы себя утомительной работы по самостоятельному анализу полученных данных, для классификации их на два вида. Говоря иными словами, мы доверили этот процесс искусственному нейрону, который мы создали на основе знания линейного классификатора. Теперь нейрон самостоятельно классифицирует все данные поступившие на его единственный вход. Более того, после процесса обучения, с обученным коэффициентом (**A**), мы легко можем задать условие, которое по вводимым пользователем значениям, определяло, к какому виду они принадлежат.

Мы полностью автоматизировали процесс классификации! Избавили себя от рутины сейчас и в последующем. И это только на самой простейшей форме “искусственной жизни” нейрона, с одним входом и выходом!

Но биологическая, как и цифровая, природа, не столь однообразна. До этого мы рассматривали “тепличные данные” – (**y = Ax**). Данные – которые мы могли классифицировать, имея лишь один вход. Во многих случаях классификации обойтись одним коэффициентом (**A**), линейной функции, невозможно, приходится использовать весь спектр возможности линейной функции. Для использования этих дополнительных возможностей, необходимо эволюционировать искусственный нейрон, добавив к нему еще один вход.

Добавив на второй вход параметр (**b**), отвечающий за точку прохождения прямой через ось **Y** , в качестве обучаемого коэффициента, мы получаем весь арсенал возможностей линейной функции (**y = Ax + b**) при классификации.

Так как у параметра (**b**), в линейной функции (**y = Ax 1+ b**), нет произведения на значение переменной, то на второй вход, в качестве данных, всегда поступает единица (**x 2 = 1**). Откуда на выходе получаем взвешенную сумму: **y = Ax 1+ bx 2** . При **x2 = 1** , на выходе получаем **y = Ax 1+ b** . И наконец, назвав коэффициенты, при входных данных – весовыми коэффициентами, изменили их обозначение – **w 1 = A** , а **w 2 = b** , в итоге: **y = w 1 x 1+ w 2 .**

Но обучая наш нейрон, как в первом случае, на выходе мы не получим нужных ответов. Оказалось, всё дело в том, что второй вход, участвует в процессе обучения независимо от

первого, и наоборот. Каждый тянет одеяло на себя. Оба входа, как бы мешают друг другу подстроить свои веса. Вследствие чего, при вычислении ошибки, получали непредсказуемый результат для подстройки обоих весовых коэффициентов. И было бы здорово, если бы с каждым последующим обучающим примером, мы смогли уменьшать функцию ошибки.

Для решения этой проблемы, нам пришлось ознакомится с методом градиентного спуска. В ходе рассмотрения этого метода, мы ознакомились с производными, узнали о правилах дифференцирования. В следствии чего, научились обновлять весовые коэффициенты, в сторону уменьшения ошибки по каждому из входов.

Суть метода – обновление весовых коэффициентов на своих входах, в зависимости от функции ошибки, таким образом, чтобы плавно двигаться в сторону её уменьшения. Другими словами, найти на каждом из входов, такое значение веса, чтоб ошибка на выходе, для всех этих весовых коэффициентов, была минимальной и как следствие удовлетворяла их всех.

Получив необходимые выражения, убедились, что изменений в математике функционирования искусственного нейрона, не так уж и много. Подобно биологической эволюции, наша тоже произошла постепенно. Ранее приобретённые навыки для классификации, лишь немногим усовершенствовались, а новые в свою очередь, выходят исходя из старых.

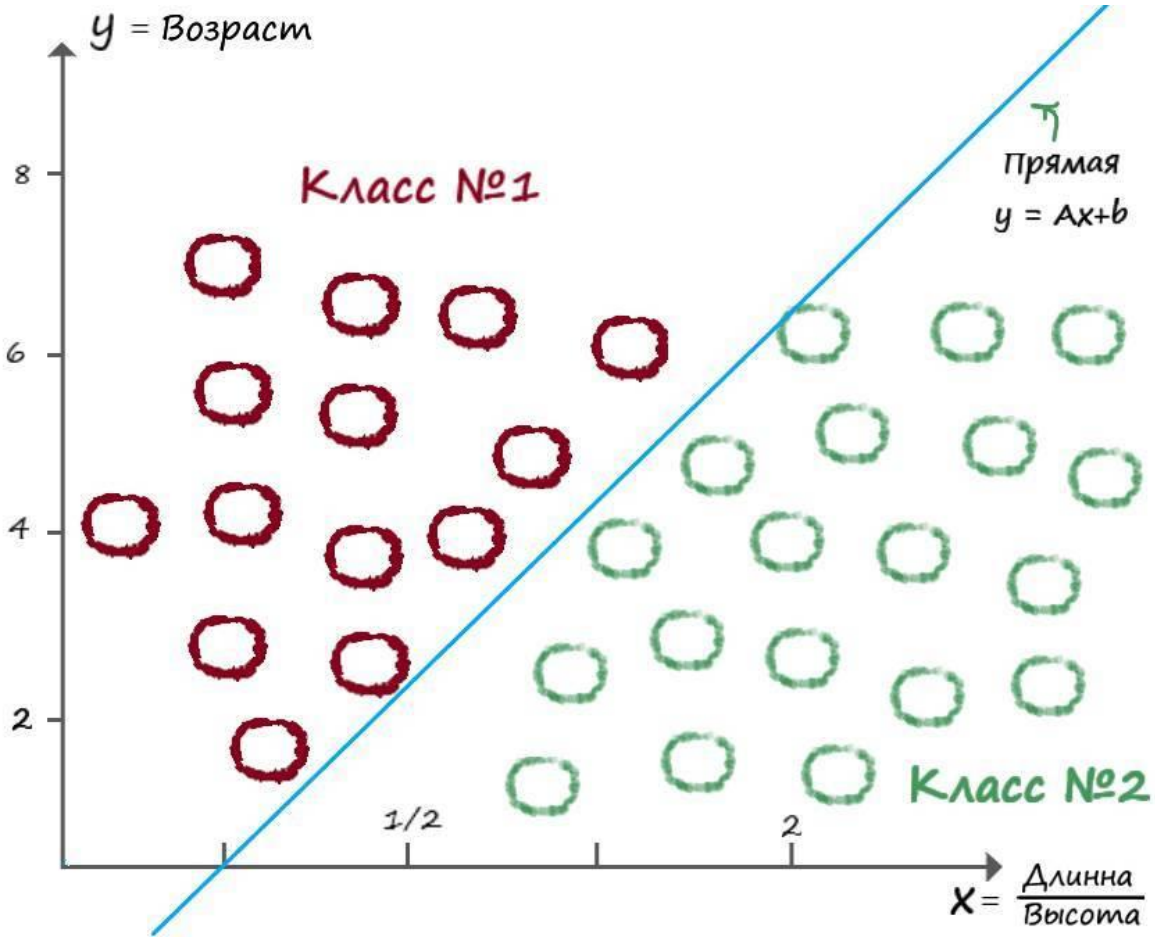
ГЛАВА 5

Больше входных данных

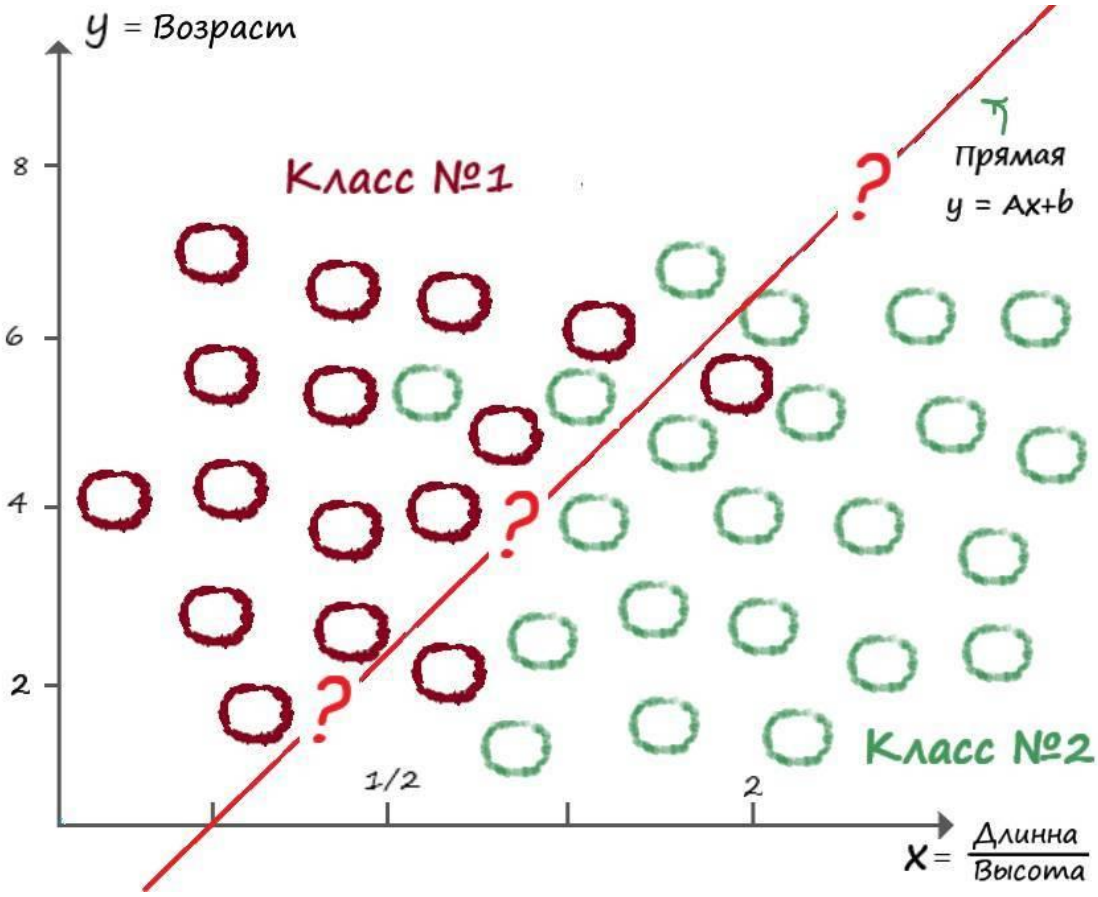
А что будет если добавить на вход искусственного нейрона, еще больше данных? Для начала, хотя бы еще один…

Проблемы линейной классификации

Допустим поступило новое задание, не совсем похожее на предыдущее. Теперь от нас хотят классифицировать виды животных, но уже с дополнительным параметром – возраст. Тестовая выборка дается уже по трем параметрам – ширина, высота, возраст. Первое что приходит в голову – объединить два параметра в одно. Если принять соотношение длины к высоте за один параметр, то мы можем смело действовать, как раньше:



Но проанализировав всё задание самостоятельно, мы пришли к такому выводу:



Как видим – данные пересекаются. И действительно, природу, как и всё что нас окружает, далеко не всегда можно классифицировать прямой. Даже один и тот же вид животных, может обитать в разных климатических зонах и условиях, что может сильно сказываться на параметрах его тела.

Что же делать? Ну для начала не будем паниковать и попробуем найти решение, пойдя по простому пути.

Логические функции

Рассмотрим, что будет на выходе нашего нейрона, добавив к нему еще один вход. Для этого, будем подавать на его вход данные логических функций.

Логическая функция принимает на вход два аргумента. Их значения, целевые значения, тоже известны. Логические функции могут принимать только дискретные аргументы (0 или 1).

Рассмотрим логическую функцию (И). Такая функция равна нулю для любого набора входных аргументов, кроме набора (x1 = 1, x2 = 1):

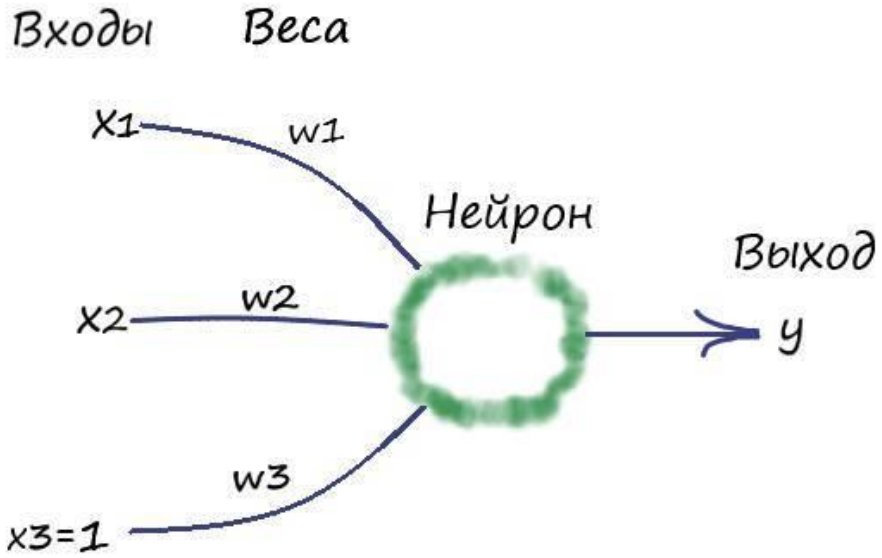
x1	x2	Y – логическое И
0	0	0
1	0	0
0	1	0
1	1	1

Функцию логического (И), для упрощения, еще называют – логическом произведением. В самом деле:

x1 * x2 = 0 * 0 = 0

x1 * x2 = 1 * 0 = 0
x1 * x2 = 0 * 1 = 0
x1 * x2 = 1 * 1 = 1

Раз мы решили добавить еще один вход на наш нейрон, то как будет выглядеть функция выхода? Ну первое что приходит в голову, раз мы в первом случае суммировали, по аналогии с линейной функцией, два произведения входных данных и весовых коэффициентов (**y = w1x1 + w2**), то почему бы не попробовать действовать подобным образом. Тогда представим линейный классификатор функцией – **y = w1x1 + w2x2 + w3**. Ну и конечно же, эволюционируем наш нейрон, добавив еще одну "ногу" на вход:



Если присмотреться, наш нейрон уже и в правду напоминает какой то, простейший живой организм.

Так как у нас всего четыре обучающие выборки, то давайте самостоятельно, без написания программы, проанализируем, что будет происходить на выходе и какие должны быть значение весовых коэффициентов:

x1w1 + x2w2 + w3 = 0 * 0,5 + 0 * 0,5 + 0 = 0
x1w1 + x2w2 + w3 = 1 * 0,5 + 0 * 0,5 + 0 = 0,5
x1w1 + x2w2 + w3 = 0 * 0,5 + 1 * 0,5 + 0 = 0,5
x1w1 + x2w2 + w3 = 1 * 0,5 + 1 * 0,5 + 0 = 1

Значение весовых коэффициентов не обязательно получились бы такими, как здесь, я их взял относительно решения уравнения при **x1 = 1** и **x2 =2**. Но в каких бы из четырех решений логической функции (И), мы не подстраивали наши веса, общего решения нам не найти.

Ну и эта проблема, решается без труда. Помните, как в конце двух написанных нами программ на Python, мы в конце задавали условие – если координата размеров тела животного превышает значение нашего классификатора, то он относится к одному виду, иначе – к другому.

Так вот если задать условие в нашем примере, которое гласило бы – что на выходе получаем единицу, только в том случае – если значение на выходе больше или равно заданному порогу (любое заданное нами число).

Пусть заданный порог равен 3. Тогда, если еще раз самостоятельно проанализировать логическую функцию. И решить её относительно уравнения при **x1 = 1** и **x2 =2**, с учетом условия порога:

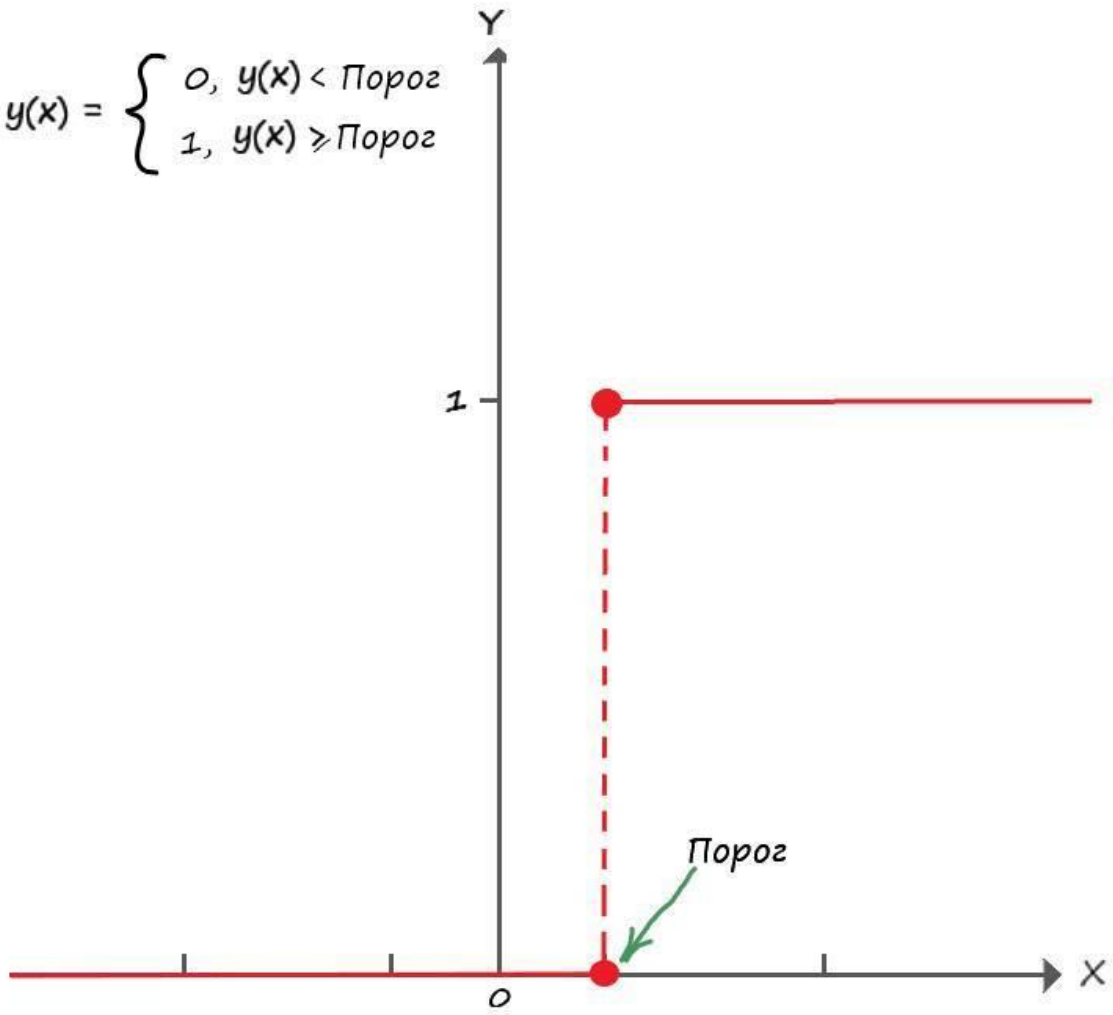
x1w1 + x2w2 + w3 = 0 * 1,5 + 0 * 1,5 + 0 = 0
x1w1 + x2w2 + w3 = 1 * 1,5 + 0 * 1,5 + 0 = 0
x1w1 + x2w2 + w3 = 0 * 1,5 + 1 * 1,5 + 0 = 0
x1w1 + x2w2 + w3 = 1 * 1,5 + 1 * 1,5 + 0 = 1

В последнем выражении, как мы говорили, единица на выходе, только при условии: **x1w1 + x2w2 + w3 ≥ 3**. Опять же, коэффициенты могут быть несколько иными, больше чем (**x1=1,5 и x2=1,5**), или к примеру: **x1w1 + x2w2 + w3 = 1 * 1,4+ + 1 * 1,4 + 0,3 = 1**, смысл решения не меняется, а главное оно есть!

Теперь, задав условие сразу при обучении нейрона, мы как бы расширили диапазон его возможностей. Так, с помощью условия при обучении, мы можем получать верные ответы на выходе искусственного нейрона не только с помощью классифицирующей прямой.

Функция единичного скачка

Попробуем проиллюстрировать наше условие на координатной плоскости:



Наше условие имеет вполне видимые очертания на координатах. Можно сказать – что наше условие является некой функцией. И у такой функции есть название – **функция единичного скачка**. А сами функции участвующие в процессе обучения называют – **функциями активации**. Всего существует более десятка функций активации. В дальнейшем мы разберем самые популярные из них.

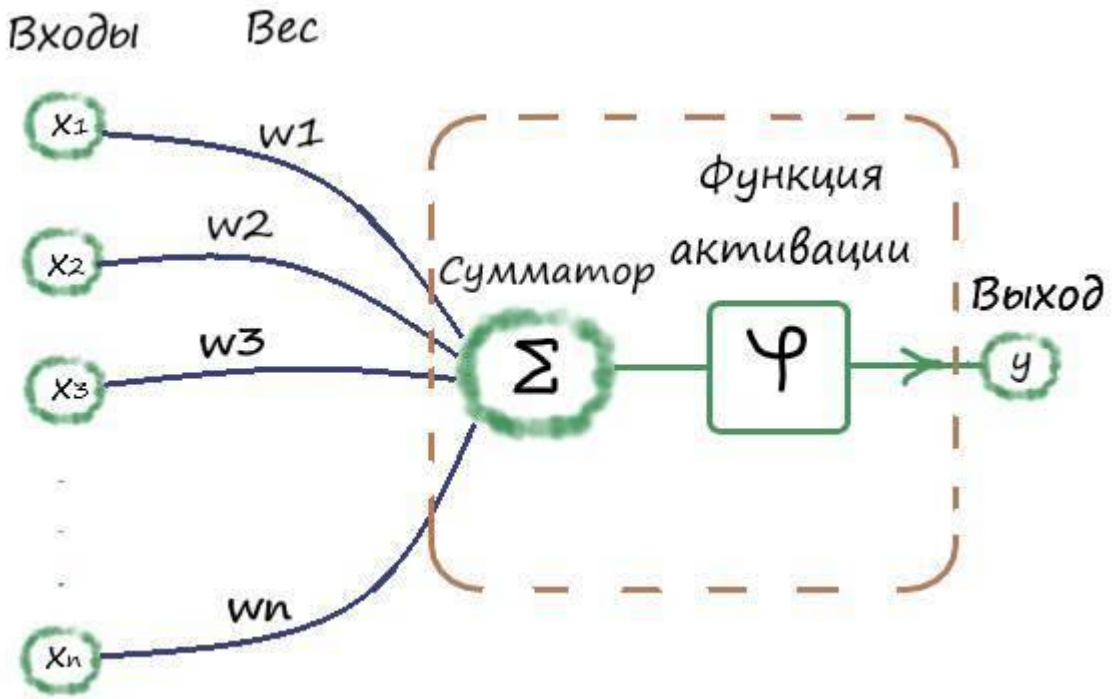
Запишем функцию активации единичного скачка на математическом языке:

$$y(x) = \begin{cases} 0, & y(x) < Порог \\ 1, & y(x) \geqslant Порог \end{cases}$$

Искусственный нейрон

Опять настало время для эволюции нашего искусственного нейрона. Теперь нам нужно внести в его структуру, дополнительный вход и функцию активации. И чтоб не терять время зря, давайте сразу условимся – входных значений теперь у него может быть любое множество и это будет его окончательный вид, до того, как, он не приобретет способность к объединению с другими нейронами, создавая тем самым нейронную сеть.

Окончательная модель внутренней структуры искусственного нейрона:



Искусственный нейрон – математическая модель биологического нейрона.

С функцией активации мы уже ознакомились, а вот про сумматор впервые слышим. Собственно, здесь ничего нового. Назначение сумматора – суммировать все входные произведения как мы условились до этого, при рассмотрении логической функции (И): **x1w1 + x2w2 + w3**. Дополнительно о роли сумматора и функции активации мы поговорим ещё чуть ниже.

Опишем еще раз принцип работы искусственного нейрона, и работу нашего нейрона в частности.

Искусственный нейрон, как и биологический, через дендриты, имеет множество входных сигналов (**x1, x2, x3…xn**). Сигнал проходящий по связи от входа к сумматору, умножается со своим весовым коэффициентом (**x1*w1, x2*w2, x3*w3… xn*wn**). Весовые коэффициенты играют роль синапсов, именно они имеют ключевую роль в обучении, изменяя в большую или меньшую сторону их значения, мы тем самым усиливаем или ослабляем связанный с соответствующим весом входной сигнал. Например, возьмём вход (**x1**), положим сигнал на этот вход–(**x1= = 0,9**), а соответствующий этому входу весовой коэффициент – (**w1 = 0,2**). Тогда их произведение даст значение на вход сумматора – **0,18**. Нетрудно понять, что чем больше весовой коэффициент, тем большее количество входного сигнала поступит на сумматор и наоборот, а нулевое значение веса – обнулит входной сигнал на входе сумматора. После чего, все произведения по входным сигналам и соответствующим им весовым коэффициентам, передаются в сумматор. Уже исходя из его названия можно понять, в чем заключается суть его работы. Он просто суммирует все поступившие на его вход сигналы:

$$x1w1+x2w2+\cdots+xnwn=\sum_{i=1}^{n} xiwi$$

Результатом работы сумматора является число, называемое – **взвешенной суммой.**

Взвешенная сумма – это сумма входных сигналов, умноженных на соответствующие им весовые коэффициенты.

Роль сумматора очевидна – он агрегирует все входные сигналы в одно число, взвешенную сумму, которая характеризует поступивший на искусственный нейрон сигнал в целом, выступая как степень его общего возбуждения.

Но, просто подавая взвешенную сумму на выход – мы получим лишь линейный классификатор, который значительно ограничен по своему функционалу, так как ранее мы выяснили, что далеко не всё можно линейно классифицировать. Нейрон должен не просто принять взвешенную сумму, но и должным образом её обработать, сформировав тем самым нужный выходной сигнал. Для обработки взвешенной суммы используется – функция активации.

Роль функции активации – преобразовать взвешенную сумму в определенное число, которое и будет является выходом нейрона.

$$y = f\left(\sum_{i=1}^{n} xiwi\right)$$

Надеюсь, теперь вы получили полное представление о внутренней структуре искусственного нейрона.

Практикум по функции логического (И)

Ну что же, снова подкрепим теорию – практикой. Напишем в Python, программу, для обучения искусственного нейрона решать логическую функцию (И). Для этого добавим в нашу программу третий вход, а в качестве обучающей выборки будет выступать функция логического (И). Так же, внесем в цикл обучения условие преодоления порога – функцию активации единичного скачка.

Текст программы:

```
import random
import numpy as np


# Инициализируем любым числом крутизны наклона прямой w1 = A
w1 = random.uniform(-4, 4)
w1_vis = w1 # Запоминаем начальное веса w1


# Инициализируем параметр w2 = b – отвечающий за точку прохождения прямой через ось Y
w2 = random.uniform(-4, 4)
w2_vis = w2 # Запоминаем начальное значение веса w2


# Инициализируем свободный параметр w3 = b
```

```
w3 = random.uniform(-4,  4)
w3_vis = w3 # Запоминаем начальное значение веса w3



# Вывод данных начальных значений весовых коэффициентов
print('Начальные  весовые коэффициенты:',  '\nw1 = ', w1, '\nw2 = ', w2, '\nw3 = ', w3)



# Скорость обучения
lr = 0.001
# Зададим количество эпох
epochs = 5000
# Зададим порог единичной функции активации
bias = 3



# Создадим массив данных функции логического (И)
log_and  = np.array([[0,  0, 0],
[1, 0, 0],
[0, 1, 0],
[1, 1, 1]])



# Прогон по выборке
for e in range(epochs):
for i in range(log_and.shape[0]): # shape – возвращает размерность массива, [0] – индекс числа строк в массиве
# Получить x1 координату точки
x1 = log_and[i,  0] # i – строка, 0 -столбец



# Получить x2 координату точки
x2 = log_and[i,  1] # i – строка, 1 -столбец



# Взвешенная  сумма
y = (w1 * x1) + (w2 * x2) + w3



if y >= bias:
# Когда превышено пороговое значение, выход должен быть – y = 1
y = 1
# Получить целевую Y, координату точки
target_Y  = log_and[i,  2] # i – строка, 2 -столбец



# Ошибка E = -(целевое значение – выход нейрона)
```

```
E = – (target_Y  – y)


# Меняем вес при x1
w1 -= lr * E * x1


# Меняем вес при x2
w2 -= lr * E * x2


# Меняем вес при x3 = 1
w3 -= lr * E
else:
# Когда не превышено пороговое значение, выход должен быть – y = 0
y = 0
# Получить целевую Y, координату точки
target_Y  = log_and[i,  2] # i – строка, 2 -столбец


# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y  – y)


# Меняем вес при x1
w1 -= lr * E * x1


# Меняем вес при x2
w2 -= lr * E * x2


# Меняем вес при x3 = 1
w3 -= lr * E


# Вывод данных готовой прямой
print('\nОбученные  весовые коэффициенты:',  '\nw1 = ', w1, '\nw2 = ', w2, '\nw3 = ', w3)


print('\nПроверка  логической функции  (И):')
print('( 0, 0,', int((0*w1  + 0*w2 + w3)>=3),  ')')
print('( 1, 0,', int((1*w1  + 0*w2+  w3)>=3),  ')')
print('( 0, 1,', int((0*w1  + 1*w2+  w3)>=3),  ')')
print('( 1, 1,', int((1*w1  + 1*w2+  w3)>=3),  ')')
```

Результат работы программы:

Начальные весовые коэффициенты:
w1 = 0.9491869921653935
w2 = 1.28402471824886
w3 = -0.08281146388163485

Обученные весовые коэффициенты:
w1 = 1.23318699217
w2 = 1.56802471825
w3 = 0.201188536118

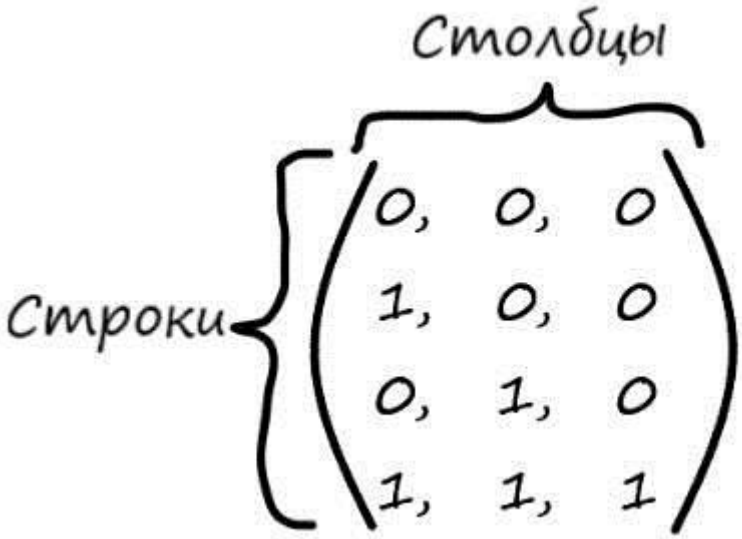
Проверка логической функции (И):
(0, 0, 0)
(1, 0, 0)
(0, 1, 0)
(1, 1, 1)
По традиции, разберем код программы.
Как и ранее, инициализируем начальные весовые коэффициенты случайным числом в диапазоне от -4 до 4 и запоминаем их для их последующего вывода:

```
w1 = random.uniform(-4,  4)
w1_vis = w1 # Запоминаем начальное веса w1
w2 = random.uniform(-4,  4)
w2_vis = w2 # Запоминаем начальное значение веса w2
w3 = random.uniform(-4,  4)
w3_vis = w3 # Запоминаем начальное значение веса w3
```

После определение параметров скорости обучения и количества эпох, создаем двумерный массив (обучающую выборку) логической функции (И):

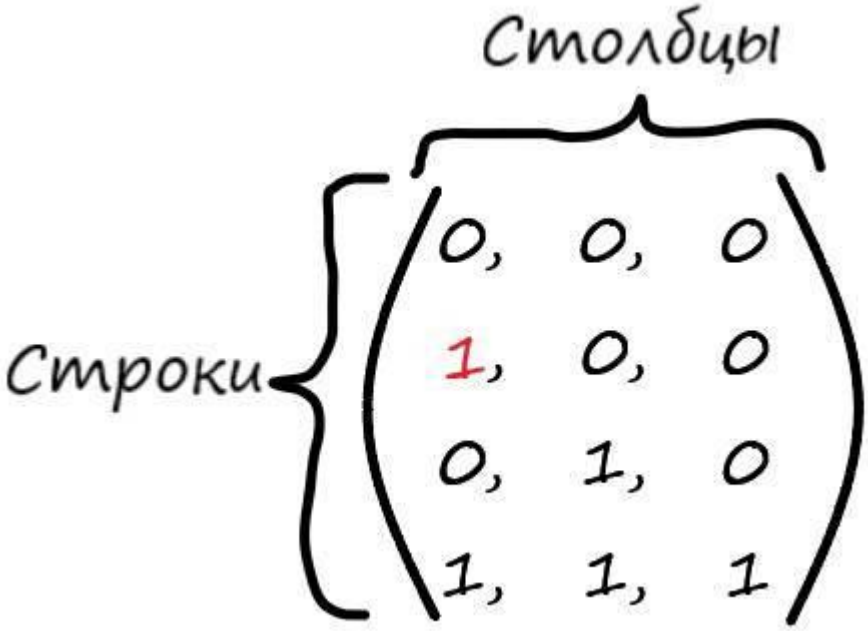
```
# Создадим массив данных функции логического (И)
log_and = np.array([[0,  0, 0],
[1, 0, 0],
[0, 1, 0],
[1, 1, 1]])
```

Индексация двумерного массива не представляет из себя ничего



сложного:

Например, если мы хотим обратится к элементу 0 столбца, 1 строки:



То для этого нам потребуется следующая запись в Python:

```
array[1, 0]
```

Первое число в квадратных скобках – индекс строки массива, второе – индекс столбца.
Можно визуализировать индексацию двухмерного массива, следующим образом:

В программе первые 2 столбца массива логической функции, представляют из себя входные данные, а 3 столбец – целевые значения.

Так будет выглядеть цикл обучения:

```
# Прогон по выборке
for e in range(epochs):
for i in range(log_and.shape[0]): # shape – возвращает размерность массива, [0] – индекс числа строк в массиве
```

Здесь, log_and.shape[0]: метод shape – возвращает количество строк и столбцов, их значения содержаться по 0 и 1 адресу функции. Так как в квадратных скобках мы обращаемся только к 0 элементу этой функции, в котором содержится число строк в массиве, то соответственно получаем вложенный в эпохи цикл из 4 транзакций.

Далее получаем координаты входных данных по каждой строке обучающей выборки (логической функции (И)):

```
# Получить x1 координату точки
x1 = log_and[i, 0] # i – строка, 0 -столбец
```

```
# Получить x2 координату точки
```

```
x2 = log_and[i, 1] # i – строка, 1 –столбец
```

Рассчитываем взвешенную сумму:

```
# Взвешенная сумма
y = (w1 * x1) + (w2 * x2) + w3
```

Ну и наконец, создаем условие функции активации единичного скачка:

```
if y >= bias:
# Когда превышено пороговое значение, выход должен быть – y = 1
y = 1
# Получить целевую Y, координату точки
target_Y = log_and[i, 2] # i – строка, 2 – столбец

# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y – y)

# Меняем вес при x1
w1 -= lr * E * x1

# Меняем вес при x2
w2 -= lr * E * x2

# Меняем вес при x3 = 1
w3 -= lr * E
else:
# Когда не превышено пороговое значение, выход должен быть – y = 0
y = 0
# Получить целевую Y, координату точки
target_Y = log_and[i, 2] # i – строка, 2 – столбец

# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y – y)
```

```
# Меняем вес при x1
w1 -= lr * E * x1


# Меняем вес при x2
w2 -= lr * E * x2


# Меняем вес при x3 = 1
w3 -= lr * E
```

Здесь все то же очень просто. Если взвешенная сумма равна, или превышает порог, то она должна принимать значение 1, которое гласит – что ответ верный:

```
# Когда превышено пороговое значение, выход должен быть – y = 1
y = 1
```

Из нашей обучающей выборки получаем ответ (целевое значение):

```
# Получить целевую Y, координату точки
target_Y = log_and[i, 2] # i – строка, 2 –столбец
```

Еще раз напомню, что здесь мы обращаемся к i строке 2 столбца массива логической функции, то есть к целевым значениям.

Далее, рассчитываем ошибку и меняем весовые коэффициенты, всё так как мы делали до этого:

```
# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y – y)


# Меняем вес при x1
w1 -= lr * E * x1


# Меняем вес при x2
```

```
w2 -= lr * E * x2


# Меняем вес при x3 = 1
w3 -= lr * E
```

В условии – иначе взвешенная сумма меньше порога – выход нейрона равен 0:

```
# Когда не превышено пороговое значение, выход должен быть – y = 0
y = 0
```

Далее так же, рассчитываем ошибку и меняем весовые коэффициенты. После чего выводим полученные, после обучения, весовые коэффициенты и делаем проверку логической функции с уже обученными коэффициентами:

```
# Вывод данных готовой прямой
print('\nОбученные весовые коэффициенты:', '\nw1 = ', w1, '\nw2 = ', w2, '\nw3 = ', w3)


print('\nПроверка логической функции (И):')
print('( 0, 0,', int((0*w1 + 0*w2 + w3)>=3), ')')
print('( 1, 0,', int((1*w1 + 0*w2+ w3)>=3), ')')
print('( 0, 1,', int((0*w1 + 1*w2+ w3)>=3), ')')
print('( 1, 1,', int((1*w1 + 1*w2+ w3)>=3), ')')
```

Результат работы программы:

```
Начальные весовые коэффициенты:
w1 = 0.9491869921653935
w2 = 1.28402471824886
w3 = -0.08281146388163485


Обученные весовые коэффициенты:
w1 = 1.23318699217
w2 = 1.56802471825
w3 = 0.201188536118
Проверка логической функции (И):
( 0, 0, 0 )
( 1, 0, 0 )
```

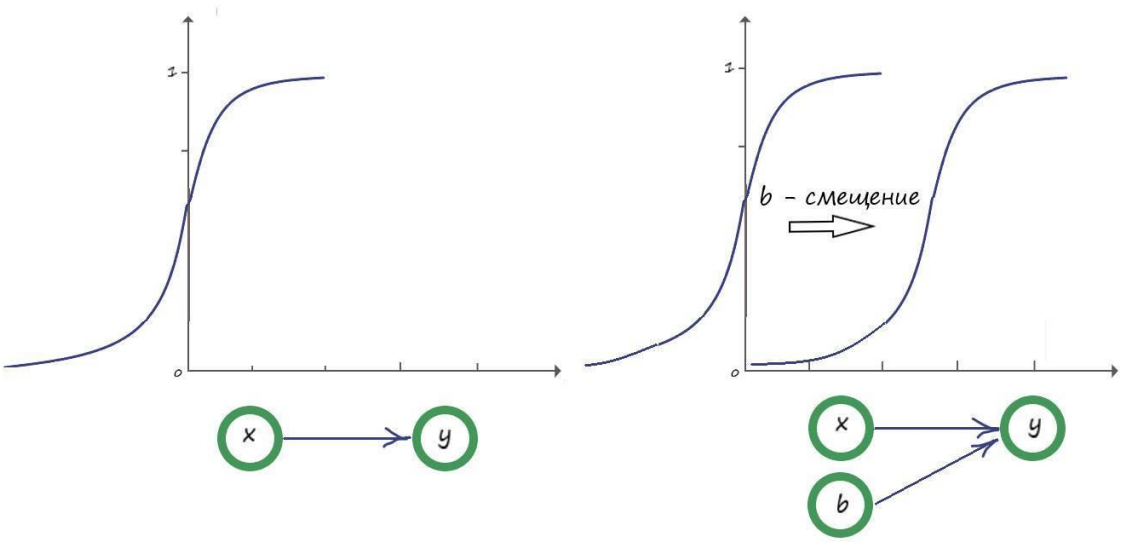
(0, 1, 0)
(1, 1, 1)

Данный код, вы можете найти по следующей ссылке:
https://github.com/CaniaCan/neuralmaster

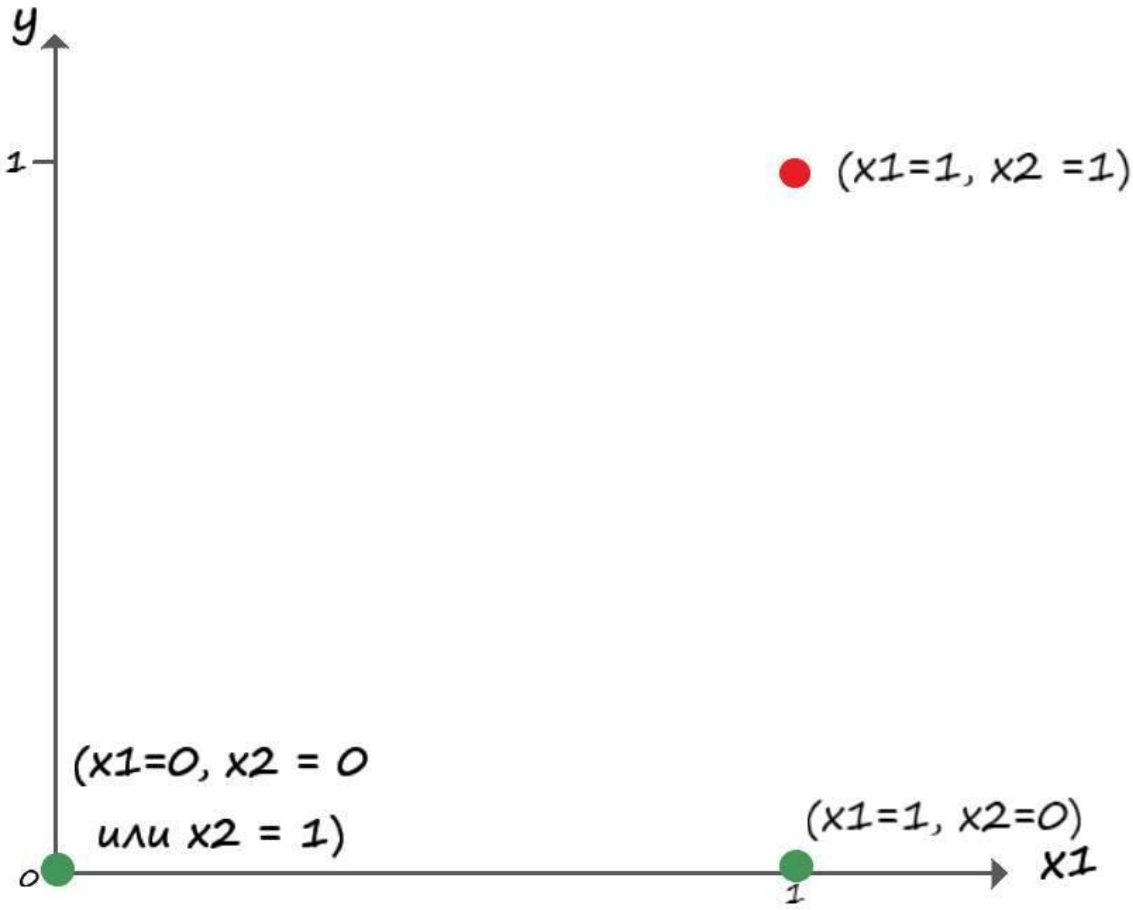
Как видим, мы смогли обучить нейрон решать задачу с дополнительным входным параметром и без использования линейного классификатора. Так же легко заметить, что в данной задаче параметр **b** линейной функции, здесь уже не нужен. В самом деле, решить логическую функцию (И) можно и без него, например, если коэффициенты будут следующими – (**w1 = 1,7 w2 = 1,4**), то с учетом пороговой функции:

x1w1 + x2w2 + w3 = 0 * 1,7 + 0 * 1,4 + 0 = 0
x1w1 + x2w2 + w3 = 1 * 1,7 + 0 * 1,4 + 0 = 0
x1w1 + x2w2 + w3 = 0 * 1,7 + 1 * 1,4 + 0 = 0
x1w1 + x2w2 + w3 = 1 * 1,7 + 1 * 1,4 + 0 = 1

Хотя в целом, он далеко не бесполезен. Его основное свойство сдвигать значения активационной функции, относительно оси координат, в право или лево. Его так и называют – нейрон смещения. Это его свойство часто используют в нейронных сетях:

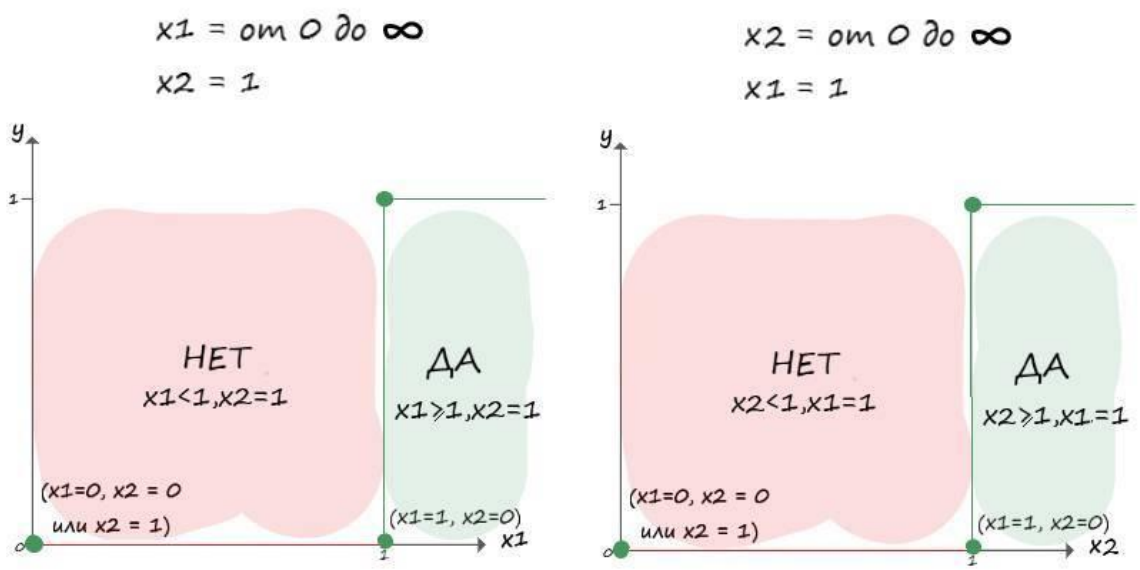


Если визуализировать, по отдельному входу, например – **x1**, все входные параметры и соответствующие ответы на выходе, получим следующую картину:



Из графика видно, что при одинаковых значениях: **x1 = 1**, ответ на выходе содержится не в единственном экземпляре. А благодаря функции активации, ответ будет зависеть от остальных входных параметров.

С полученными значениями весов, нейрон интерпретирует все входные значения, что ниже 1, за нулевое, так как на его выходе получаем ноль:



Сам график очень напоминает пороговую функцию, с порогом 1. Иначе быть не могло, так как пороговая функция оперирует только 0 или 1.

Функция логического (ИЛИ)

Похожим образом разберем еще одну логическую функцию (ИЛИ).

Логическое (ИЛИ), имеет следующую таблицу истинности:

x1	x2	Y – логическое ИЛИ
0	0	0
1	0	1
0	1	1
1	1	1

Функцию логического (ИЛИ), можно назвать – логическом сложением. В самом деле:

x1 + x2 = 0 + 0 = 0
x1 + x2 = 1 + 0 = 1
x1 + x2 = 0 + 1 = 1
x1 + x2 = 1 + 1 = 1

Так как значения логических функций – бинарные, то логическое сложение: 1+1 = 1.

Внесем изменения в нашу последнюю программу, в части массива данных и проверки обученного нейрона. А также уберем параметр **b**, посмотрим какими будут коэффициенты без него:

```
import random
import numpy as np
# Инициализируем любым числом крутизны наклона прямой w1 = A
w1 = random.uniform(-4, 4)
w1_vis = w1 # Запоминаем начальное веса w1



# Инициализируем параметр w2 = b – отвечающий за точку прохождения прямой через ос Y
w2 = random.uniform(-4, 4)
w2_vis = w2 # Запоминаем начальное значение веса w2



# Вывод данных начальных значений весовых коэффициентов
print('Начальные весовые коэффициенты:', '\nw1 = ', w1, '\nw2 = ', w2)



# Скорость обучения
lr = 0.001
# Зададим количество эпох
epochs = 5000
# Зададим порог единичной функции активации
bias = 3



# Создадим массив данных функции логического (И)
log_or = np.array([[0, 0, 0],
[1, 0, 1],
[0, 1, 1],
[1, 1, 1]])
# Прогон по выборке
for e in range(epochs):
for i in range(log_or.shape[0]): # shape – возвращает размерность массива, [0] – индекс числа строк в массиве
# Получить x1 координату точки
x1 = log_or[i, 0] # i – строка, 0 -столбец



# Получить x2 координату точки
x2 = log_or[i, 1] # i – строка, 1 -столбец



# Взвешенная сумма
y = (w1 * x1) + (w2 * x2)



if y >= bias:
# Когда превышено пороговое значение, выход должен быть – y = 1
y = 1
```

```
# Получить целевую Y, координату точки
target_Y = log_or[i, 2] # i – строка, 2 -столбец



# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y – y)



# Меняем вес при x1
w1 -= lr * E * x1



# Меняем вес при x2
w2 -= lr * E * x2



else:
# Когда не превышено пороговое значение, выход должен быть – y = 0
y = 0
# Получить целевую Y, координату точки
target_Y = log_or[i, 2] # i – строка, 2 -столбец



# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y – y)



# Меняем вес при x1
w1 -= lr * E * x1



# Меняем вес при x2
w2 -= lr * E * x2



# Вывод данных готовой прямой
print('\nОбученные весовые коэффициенты:', '\nw1 = ', w1, '\nw2 = ', w2)
print('\nПроверка логической функции (И):')
print('( 0, 0,', int((0*w1 + 0*w2)>=3), ')')
print('( 1, 0,', int((1*w1 + 0*w2)>=3), ')')
print('( 0, 1,', int((0*w1 + 1*w2)>=3), ')')
print('( 1, 1,', int((1*w1 + 1*w2)>=3), ')')
```

Результат работы программы:

Начальные весовые коэффициенты:
w1 = 3.611365134992452
w2 = 3.8646266194092878

Обученные весовые коэффициенты:
w1 = 3.61136513499
w2 = 3.86462661941

Проверка логической функции (И):
(0, 0, 0)
(1, 0, 1)
(0, 1, 1)
(1, 1, 1)

Бинго! И снова удача!

Если более детально взглянуть на процесс обучения по одному из входов, то получим следующую картину:

1)

Входной параметр активен (

x=1

), нейрон выдал правильный ответ (превышен порог), но ошибся (

y

=1, но

Y

= 0

):

Ошибка: **E = – (Y – y) = -(0 – 1) = 1**;

Обновление веса: **w = w – (E * x1) = w – 1**

2)

Входной параметр не активен (

x=0

), нейрон выдал правильный ответ (превышен порог), но ошибся (

y

=1, но

Y

= 0

):

Ошибка: **E = – (Y – y) = -(0 – 1) = 1**;

Обновление веса: **w = w – (E * x1) = w – 0 = w**

Сразу делаем вывод, что при неактивном входном параметре, весовой коэффициент не меняется при любых целевых значениях и выходе нейрона. Поэтому, в дальнейшем не будем рассматривать варианты с неактивным входом.

3)

Входной параметр активен (

x=1

), нейрон выдал правильный ответ (превышен порог), и не ошибся (

y

=1, но
Y
= 1
):
Ошибка: **E = – (Y – y) = -(1 – 1) = 0**;
Обновление веса: **w = w – (E * x1) = w – 0 = w**
4)
Входной параметр активен (
x=1
), нейрон выдал не правильный ответ (не превышен порог), и не ошибся (
y
=0 но
Y
= 0
):
Ошибка: **E = – (Y – y) = -(0 – 0) = 0**;
Обновление веса: **w = w – (E * x1) = w – 0 = w**
5)
Входной параметр активен (
x=1
), нейрон выдал не правильный ответ (не превышен порог), и ошибся (
y
=0 но
Y
= 1
):
Ошибка: **E = – (Y – y) = -(1 – 0) = -1**;
Обновление веса: **w = w – (E * x1) = w + 1 = w + 1**

Если внимательно присмотреться в эти данные, то можно сформулировать следующее правило:

Если наш нейрон дал правильный ответ (ошибка равна 0), то ничего не предпринимаем.

Если нейрон выдал правильный ответ, но ошибся, то мы должны его наказать за это – уменьшить веса тех связей, через которые прошел сигнал. А иными словами, те веса, которые связанны с возбудившимися входами, уменьшаются.

Если нейрон выдал не правильный ответ и ошибся, то мы должны увеличить все веса, через которые прошел сигнал. Таким образом мы как бы говорим сети, что такие связи, а значит и связанные с ними входы – правильные.

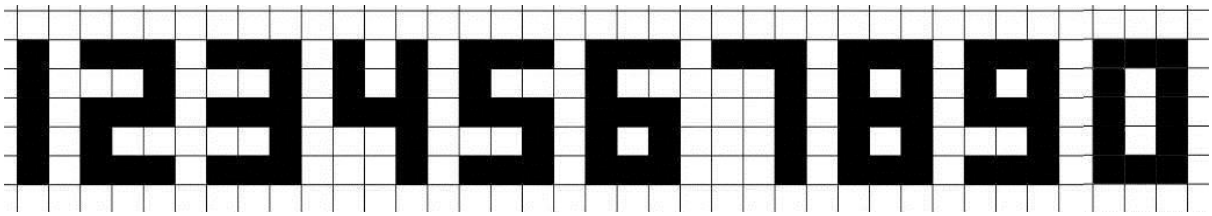
У такого правила есть свое имя – **дельта правило.**

Дельта правило – метод обучения нейрона по принципу градиентного спуска по поверхности ошибки.

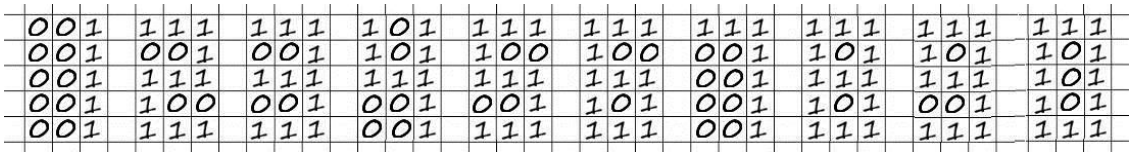
Ну что же, если обратится к тому, что мы до сих пор прошли в этой главе, то смело можно утверждать, что теперь, после очередного этапа эволюции, наш нейрон научился принимать несколько входных параметров, а также решать задачи, которые невозможно решить линейной классификацией, путем внесения в его структуру функции активации.

Распознавание цифры

Любое цифровое изображение, как всем известно, состоит из пикселей. Очень удобно каждый пиксель представлять определенным числом. Так как мы пока оперируем дискретными значениями 0 и 1, то черный цвет будем представлять, как 1, а белый как 0. То есть наши цифры пока будут в черно – белом формате. Цифры будут состоять из черных пикселей разрешением 3x5:



Как и говорилось ранее, за белый пиксель отвечает 0, а черный – 1. Поэтому наши десять цифр от 0 до 9 в бинарном формате будут выглядеть следующим образом:



Для записи каждой цифры у нас используется пять строк и три столбца в каждой. Теперь уберем все переносы строк, чтобы получить для каждой цифры от 0 до 9 одну длинную строку длиной в 15 символов:

1 – 001001001001001

2 – 111001111001111

.

.

.

9 – 111101111001111

0 – 111101101101111

Цифры в таком строковом формате уже можно использовать для обучения нейрона.

С входными данными определились. А как быть с целевыми значениями? Давайте условимся, что правильный ответ будет находится по 0 адресу массива цифры. Добавим в начало массива элемент целевого значения:

1 – 1001001001001001

2 – 2111001111001111

.

.

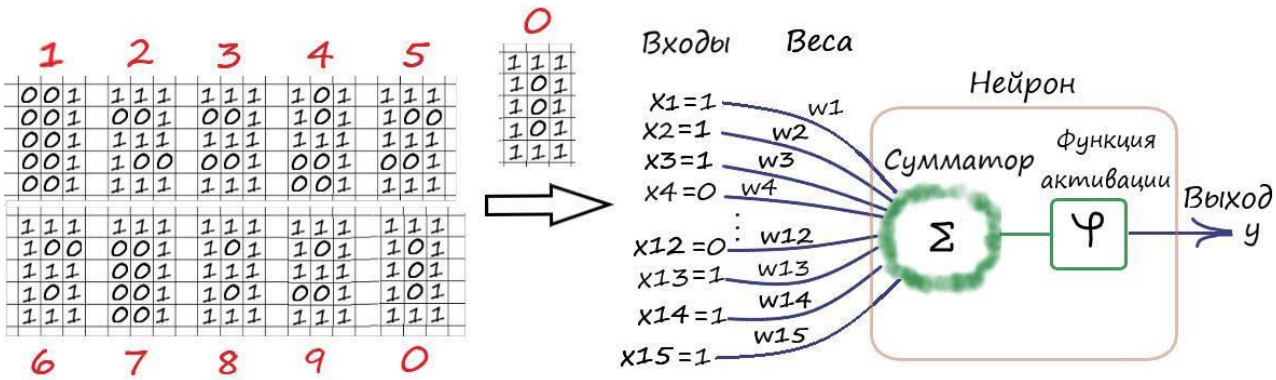
.

9 – 9111101111001111

0 – 0111101101101111

Вот собственно мы и имеем обучающую выборку, характеризующей определенное число, с входными данными и целевыми значениями.

А вот так визуально можно представить входные данные, например, характеризующие число 0, поступающие на вход искусственного нейрона:



Пусть нашей новой задачей будет распознавание нужной для нас цифры. Например, пусть это будет цифра 0 (можно любую другую).

Как мы уже говорили: обучающей выборкой будут все цифры от 0 до 9 в строковом формате, в виде одномерного массива, где 0 элемент – целевое значение. Когда нейрон обучится безошибочно распознавать нужную нам цифру (0), тогда, после обучения, мы проверим его "эрудицию", на тестовой выборке. Её создадим похожим образом, как и обучающую, но она будет выглядеть немного иначе. Таких данных, как в тестовой выборке, нейрон еще не видел до этого, на вход будут подаваться уже искаженные изображения нуля.

Программа

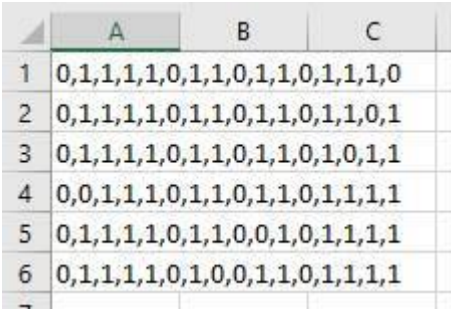
Теперь давайте запишем все цифры от 0 до 9, в отдельный файл с расширением .csv. Это достаточно популярный формат, создается в обычном офисном пакете типа "Excel", и кроме того, "Python" умеет с ним работает.

Вот так будет выглядеть наша обучающая выборка:

	A	B	C
1	0,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1		
2	1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1		
3	2,1,1,1,0,0,1,1,1,1,1,0,0,1,1,1		
4	3,1,1,1,0,0,1,1,1,1,0,0,1,1,1,1		
5	4,1,0,1,1,0,1,1,1,1,0,0,1,0,0,1		
6	5,1,1,1,1,0,0,1,1,1,0,0,1,1,1,1		
7	6,1,1,1,1,0,0,1,1,1,1,0,1,1,1,1		
8	7,1,1,1,0,0,1,0,0,1,0,0,1,0,0,1		
9	8,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1		
10	9,1,1,1,1,0,1,1,1,1,0,0,1,1,1,1		
11			
12			

Как видим, тут все достаточно просто. В каждой строке пишем, через запятую, значение пикселей определенной цифры, 0 или 1. Не забываем, что в начале списка следует ответ – какая эта цифра.

Теперь давайте создадим отдельный файл с тестовой выборкой. Запишем в неё шесть видов искаженной цифры – ноль:



	A	B	C
1	0,1,1,1,1,0,1,1,0,1,1,0,1,1,1,0		
2	0,1,1,1,1,0,1,1,0,1,1,0,1,1,0,1		
3	0,1,1,1,1,0,1,1,0,1,1,0,1,0,1,1		
4	0,0,1,1,1,0,1,1,0,1,1,0,1,1,1,1		
5	0,1,1,1,1,0,1,1,0,0,1,0,1,1,1,1		
6	0,1,1,1,1,0,1,0,0,1,1,0,1,1,1,1		

Для начала, загрузим в программу тренировочные данные из файла:

```
# Загрузить и подготовить тренировочные данные из формата CSV в список training_data = open("dataset/Data_train.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data.close() # закрываем файл csv
```

Функция open() – открывает файл и возвращает представляющий его объект. Аргументами данной функции являются: путь где находится наш файл (dataset/Data_train.csv), и режим открытия файла (в нашем случае ‘r’ – режим чтения).

Таким же образом подгружаем данные для теста:

```
# Загрузить и подготовить тестовые данные из формата CSV в список
test_data = open("dataset/Data_test.csv", 'r') # 'r' – открываем файл для чтения
test_data_list = test_data.readlines()# Загрузить и подготовить тестовые данные из формата CSV в список
test_data.close() # закрываем файл csv
```

Так как, начальные значения весовых коэффициентов могут принимать любые значения, то для простоты, обозначим их на старте как нулевые. Так как у нас 15 входов, то нам потребуется 15 связей. Запишем их в виде одномерного массива с 15 нулевыми элементами:

```
# Инициализация весов нейрона
weights = np.zeros(15)
```

Метод zeros(), модуля numpy, создает нулевые элементы массива, в количестве указанным в скобках (аргумент).

Задаем остальные параметры:

```
# Скорость обучения
lr = 1
```

```
# Зададим количество эпох
epochs = 1000
```

```
# Зададим порог единичной функции активации
bias = 3
```

Так как мы оперируем только 0 и 1, то и скорость обучения примем равной единице.

Цикл обучения:

```
# Прогон по обучающей выборке
for e in range(epochs):
for i in training_data_list:
# Получить входные данные числа
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])



# Получить целевое значение Y, (ответ – какое это число)
target_Y = int(all_values[0]) # перевод символов в int, 0 элемент – ответ
```

Переводим целевой результат в бинарный вид. Так как мы ищем только значение ноль, значит только он будет верным = 1.

```
# остальные ответы, будут неверными, соответственно они обращаются в ноль.
if target_Y == 0:
target_Y = 1
else:
target_Y = 0



# Взвешенная сумма
y = np.sum(weights * inputs_x)



if y >= bias:
# Когда равно или превышено пороговое значение, выход должен быть – y = 1
y = 1



# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y – y)



# Меняем веса по каждому из входов (дельта правило)
weights -= lr * E * inputs_x



else:
# Когда не превышено пороговое значение, выход должен быть – y = 0
y = 0



# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y – y)
```

```
# Меняем веса по каждому из входов (дельта правило)
weights  -= lr * E * inputs_x
```

Здесь, в начале цикла получаем данные для приёма на вход нейрона:

```
# Получить входные данные числа
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
```

Метод split() – разбивает строки на список, который по умолчанию делает разбиение по пробелам. Указав в его аргументе – (','), мы разбиваем строки на список по запятым. Затем, начиная с первого элемента списка, записываем данные в переменную inputs_x.

Таким же образом получаем целевое значение, которое хранится в нулевом адресе массива числа:

```
# Получить целевое значение Y, (ответ – какое это число)
target_Y = int(all_values[0])  # перевод символов в int, 0 элемент – ответ
```

Так как мы ищем только число ноль и оперируем бинарными значениями 0 и 1, то верным ответом на выходе (y = 1) будет цифра, у которой нулевой элемент равен нулю (первая строка в списке файла тренировочных значений – число ноль):

0 – 0111101101101111

Что легко осуществить с помощью простого условия:

```
# Переводим целевой результат в бинарный вид. Так как мы ищем только значение ноль, значит только он будет верным = 1.
# остальные ответы, будут неверными, соответственно они обращаются в ноль.
if target_Y  == 0:
target_Y  = 1
else:
target_Y  = 0
```

Далее, действуем уже известным нам способом, находим взвешенную сумму и обновляем весовые коэффициенты с учетом функции активации:

```
# Взвешенная сумма
y = np.sum(weights  * inputs_x)
```

```
if y >= bias:
# Когда равно или превышено пороговое значение, выход должен быть – y = 1
y = 1
```

```
# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y  – y)



# Меняем веса по каждому из входов (дельта правило)
weights  -= lr * E * inputs_x



else:
# Когда не превышено пороговое значение, выход должен быть – y = 0
y = 0
# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y  – y)



# Меняем веса по каждому из входов (дельта правило)
weights  -= lr * E * inputs_x
```

А чтоб убедиться в том, что наш нейрон обрел “интеллект” и умеет отличать цифру ноль от остальных, выведем результаты на консоль:

```
# Вывод обученых весов
print('Весовые коэффициенты:\n',weights)



for i in training_data_list:
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
print(i[0],  ' это 0? ', np.sum(weights  * inputs_x)>=bias)



# Проход по тестовой выборке
t = 0 # Счетчик номера нуля тестовой выборки
for i in test_data_list:
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
t += 1
print('Узнал  0 – ',t, '?', np.sum(weights  * inputs_x)>=bias)
```

Полный текст программы:

```
import numpy as np
```

```
    # Загрузить и подготовить тренировочные данные из формата CSV в список
    training_data  = open("dataset/Data_train.csv",  'r') # 'r' – открываем файл для чтения
    training_data_list  =  training_data.readlines()  #  readlines()  –  читает  все  строки  в  файле  в переменную training_data_list
    training_data.close()  # закрываем файл csv



    # Загрузить и подготовить тестовые данные из формата CSV в список
    test_data = open("dataset/Data_test.csv",  'r') # 'r' – открываем файл для чтения
    test_data_list = test_data.readlines()# Загрузить и подготовить тестовые данные из формата CSV в список
    test_data.close() # закрываем файл csv



    # Инициализация  весов нейрона
    weights  = np.zeros(15)



    # Скорость обучения
    lr = 1



    # Зададим количество эпох
    epochs = 1000



    # Зададим порог единичной функции активации
    bias = 3



    # Прогон по обучающей выборке
    for e in range(epochs):
    for i in training_data_list:
    # Получить входные данные числа
    all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    inputs_x = np.asfarray(all_values[1:])



    # Получить целевое значение Y, (ответ – какое это число)
    target_Y  = int(all_values[0])  # перевод символов в int,  0 элемент  – ответ



    # Переводим целевой результат в бинарный вид. Так как мы ищем только значение ноль, значит только он будет верным = 1.
    # остальные ответы, будут неверными, соответственно они обращаются в ноль.
```

```
if target_Y == 0:
target_Y = 1
else:
target_Y = 0



# Взвешенная сумма
y = np.sum(weights * inputs_x)



if y >= bias:
# Когда равно или превышено пороговое значение, выход должен быть – y = 1
y = 1



# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y – y)



# Меняем веса по каждому из входов (дельта правило)
weights -= lr * E * inputs_x



else:
# Когда не превышено пороговое значение, выход должен быть – y = 0
y = 0



# Ошибка E = -(целевое значение – выход нейрона)
E = – (target_Y – y)



# Меняем веса по каждому из входов (дельта правило)
weights -= lr * E * inputs_x



# Вывод обученых весов
print('Весовые коэффициенты:\n',weights)



# Еще раз пройдем по обучающей выборке
for i in training_data_list:
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
print(i[0], ' это 0? ', np.sum(weights * inputs_x)>=bias)
```

```
# Проход по тестовой выборке
t = 0 # Счетчик номера нуля тестовой выборки
for i in test_data_list:
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
t += 1
print('Узнал  0 – ',t, '?', np.sum(weights  * inputs_x)>=bias)
```

Результат работы программы:

Весовые коэффициенты:
[0. 0. -2. 3. 0. 2. 0. -7. -2. 3. 0. 1. 0. 0. -2.]

0 это 0? True
1 это 0? False
2 это 0? False
3 это 0? False
4 это 0? False
5 это 0? False
6 это 0? False
7 это 0? False
8 это 0? False
9 это 0? False

Узнал 0 – 1 ? True
Узнал 0 – 2 ? True
Узнал 0 – 3 ? True
Узнал 0 – 4 ? True
Узнал 0 – 5 ? True
Узнал 0 – 6 ? True

Этот и другие исходники можно найти по ссылке:

https://github.com/CaniaCan/neuralmaster

Отлично, обученный нейрон распознал всю тестовую выборку! Хотя данные из этой выборки он до этого не встречал. Всего один единственный искусственный нейрон, способен решать такую сложную для компьютера задачу, как распознавание цифры.

В дополнение, как видно, наш нейрон прекрасно справился с поставленной задачей без использования параметра линейной функции b. При его наличии, он так же принимал бы активное участие в обучении, остальные веса так же обновлялись с его учетом, и в результате получили бы тоже верные ответы.

Преимущества и недостатки между функцией единичного скачка и линейной функцией

Да, введя в цикл обучения функцию активации (единичного скачка), мы значительно расширили функционал нашего нейрона. Теперь он способен по мимо классификации, еще и распознавать числа. Но избавившись от линейной классификации, заменив её на функцию единичного скачка, мы теперь можем оперировать лишь бинарными значениями – 0 и 1. Такое ограничение вызвано не следствием ухода от линейной классификации, а выбором функции активации. Так как сама функция единичного скачка способна работать лишь с такими типами данных. Очевидно, для того чтобы оперировать всем разнообразием числовых данных – необходима другая активационная функция.

Логистическая функция (Сигмоидальная функция или сигмоида)

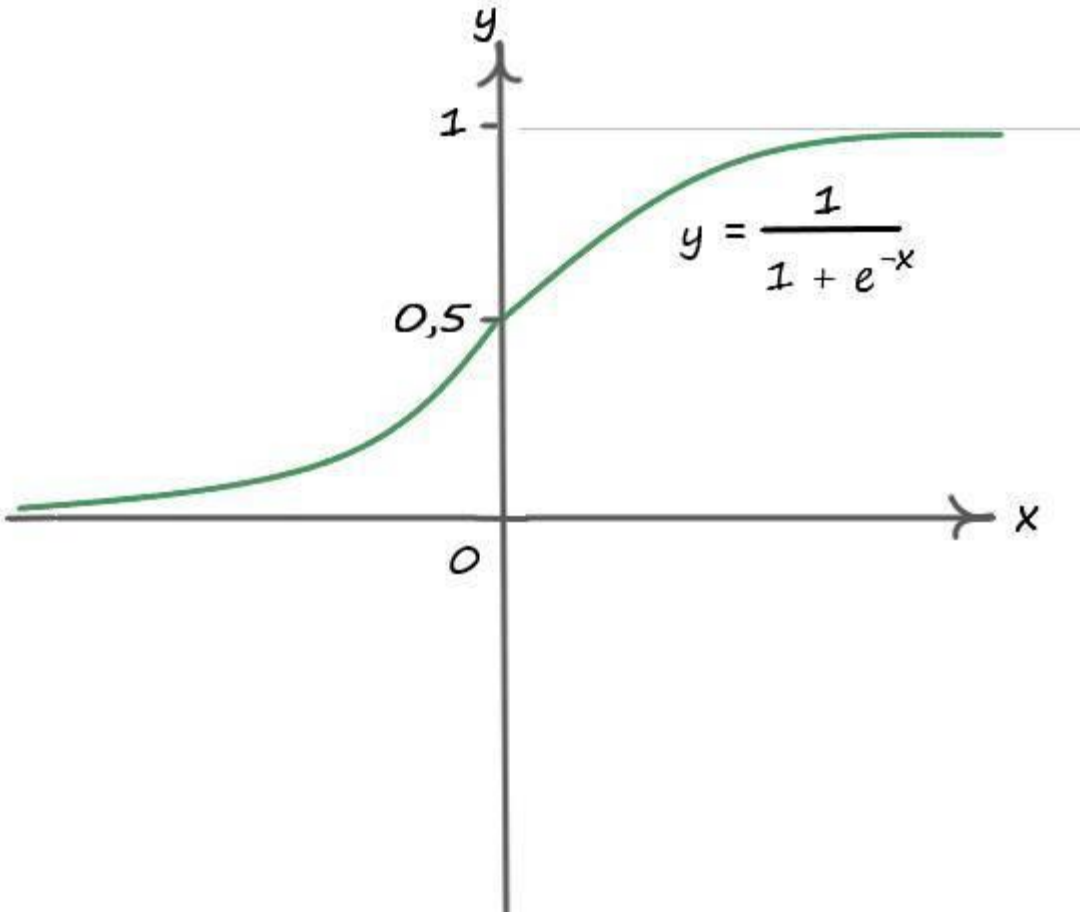
Сигмоида – это гладкая монотонная нелинейная функция, имеющая форму буквы "S", которая применяется для «сглаживания» значений некоторой величины.

Формула сигмоиды:

$$y = \frac{1}{1 + e^{-x}}$$

Напомню, буквой e в математике принято обозначать константу, равную 2,7182…

Функция сигмоиды стремиться к нулю при бесконечно малом значении аргумента x, и стремиться к единице, при бесконечно большом значении аргумента. Но она никогда не равна нулю, и не равна единице. При нулевом значении аргумента, функция равна значению 0,5. Что легко проверить, e в степени 0 равна единице, следовательно, значение сигмоиды: 1/1+1 = 0,5:



Нетрудно заметить, что при больших или очень малых значениях сигнала, кривая сигмоидальной функции заметно спрямляется. Это чревато некоторыми проблемами. При нахождении градиента и последующим обновлением весов, величина градиента и соответственно обновления будет очень незначительной. Использование небольших значений градиента, значительно ограничивает способность нейрона к обучению. Иными словами, использование значений входных данных, значительно превышающих диапазон значений функции активации, заметно снижает или делают невозможным обучение. Так, например, если при использовании сигмоидальной функции активации, в обучающей выборке содержаться большие значения (например, от -1000 до 1000), то нейрон навряд ли сможет правильно пройти своё обучение. Если мы имеем такие входные параметры, то решить проблему обучения нейрона, можно **стандартизацией данных.**

Стандартизация данных не обязательный материал в этой книге, так как в дальнейшем она нам не понадобится. Но если вы серьезно решите заняться наукой нейронных сетей, её необходимо знать.

Стандартизация данных – процесс создания стандартных тестовых шкал или стандартизованных шкал. Для этого осуществляется Z – преобразование первичных данных по следующей формуле:

$$Z_i = \frac{x_i - X_{cp}}{bco}$$

где **xi** – величина входного параметра в определенной строке,
Zi – стандартизованная величина для **xi,**
Xcp – среднее арифметическое значение столбца входных параметров
bco – стандартное отклонение входных параметров
С первыми тремя неизвестными мы знакомы, а со стандартным отклонением нужно ещё познакомится.

Что измеряет стандартное отклонение?

Допустим у нас имеется некий набор входных данных **x:**

100	900
2000	850
0,5	750
1500	830

Средние значения по столбцам **Xcp**:

900,125	832,5

Стандартное отклонение, нам как раз и скажет, как отклоняются значения в выборке от среднего. С точки зрения стандартного отклонения, среднее значение будет эталонным и равняться нулю. Стандартное отклонение рассчитывается следующим образом:

$$bco = \frac{\sqrt{\sum_{i=1}^{n} (xi - Xcp)^2}}{\sqrt{n-1}}$$

Где **n** – объём выборки. В нашем случае **n = 4.**
Теперь нетрудно узнать значения стандартных отклонений bco:

1003,18	62,38

Видим, что в первом столбце каждое значение в выборке в среднем откланяется на **1003,18** от среднего значения, а во втором на **62,38**. Это говорит о том, что разброс значений во втором столбце гораздо меньше чем в первом. Можно проверить результаты в программе “Excel”, воспользовавшись функцией – СТАНДОТКЛОН.

Ну и воспользовавшись формулой стандартизации данных, вычислим стандартизированные данные по столбцам **Zi =(xi – Xcp)/bco**:

Zi =100 – 900,125/1003,18 = -0,80	Zi =900 – 832,5/62,38 = 1,08
Zi =2000 – 900,125/1003,18 = 1,10	Zi =850– 832,5/62,38 = 0,28
Zi =0,5 – 900,125/1003,18 = -0,90	Zi =750– 832,5/62,38 = -1,32
Zi =150 – 900,125/1003,18 = 0,60	Zi =830– 832,5/62,38 = -0,04

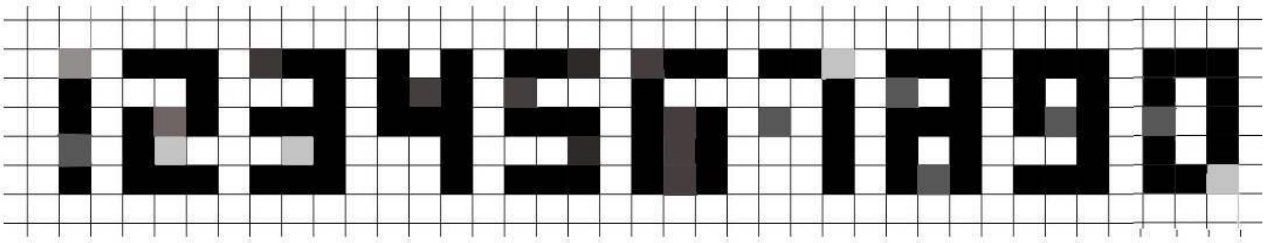
А вот эти данные располагаются близко к области значений сигмоидальной функции, хоть и некоторые её значения выходят за диапазон сигмоиды. Но всё же, этого достаточно для нормального обучения нейрона. В таких областях кривая функции не так распрямлена, как при больших значениях.

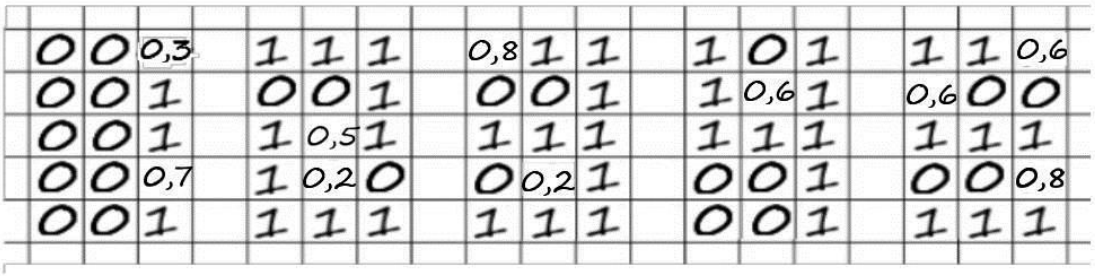
Теперь можно с уверенностью говорить, что мы можем применять сигмоидальную функцию на всем диапазоне входных данных!

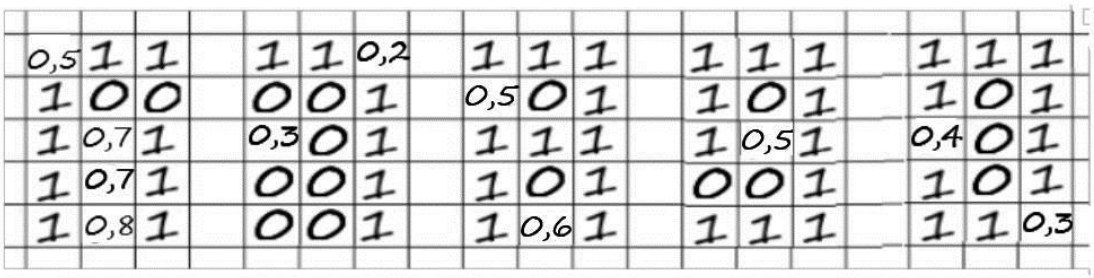
Распознаем цифру при помощи логистической функции активации

Распознавать цифру 0 при помощи единичной функции мы уже умеем. Теперь давайте попробуем сделать то же самое, но с помощью сигмоидальной функции.

Для чего нам такой диапазон значений? Ну, даже если обратится к нашему набору данных с цифрами, то они не так часто встречаются в строго черно белом формате. Они могут быть в цветном формате, или иметь градации серого. Так же на картинках с цифрами, могут располагаться различные шумы, например, “шальной” пиксель, который будет её искажать. Давайте проиллюстрируем сказанное:







Надеюсь все хорошо помнят – как вычислять градиент?

$$\frac{dE}{dw_{ij}} = \frac{dE}{dy} * \frac{dy}{dw_{ij}}$$

$$\frac{dE}{dy} = \frac{d(T-y)^2}{dy} = 2(T-y)*(-1)$$
$$= -2(T-y)$$

Так как выход нейрона – **f(x) = y**, а **y** это логистическая функция:

$$y = \frac{1}{1+e^{-x}}$$

то выражение **dy/dw**, возможно решить продифференцировав как сложную функцию.
С единичной функцией было все проще, так как это все же условие, а не функция.
Давайте пробовать дифференцировать сигмоиду:

$$y = f(Si), \; Si = \sum_{ij=1}^{n} wij*xi$$

Здесь n – число входов нейрона, **xi** – сигнал на **i**-ом входе, а **f(S)** – функция активации. Тогда получим:

$$\frac{dy}{dw_{ij}} = \frac{f'(Si)}{dw_{ij}} = f'(Si)Si' = f'(Si)xi$$

Соберем наше уравнение избавившись от двойки:

$$\frac{dE}{dw_{ij}} = \frac{dE}{dy} * \frac{dy}{dw_{ij}} = -(T - y) * f'(S_i) * x_i$$

Как видим, добавился один элемент (**f′(Si)**) – производная функции активации. В нашем случае – логистической функции (сигмоиды). К счастью, её производная довольна проста:

$$\frac{d}{dx}\,\text{сигмоида} = \text{сигмоида}(x) * (1 - \text{сигмоида}(x))$$

Но всё же, для общего развития давайте разберем, как мы это получили.
Воспользовавшись основным свойством производных от частного:

$$\left(\frac{u}{v}\right)' = \frac{u' * v - u * v'}{v^2}$$

Получим:

$$\left(\frac{1}{1+e^{-x}}\right)' = \frac{1' * (1+e^{-x}) + 1*(1+e^{-x})'}{(1+e^{-x})^2} = \frac{0+(-e^{-x})}{(1+e^{-x})^2} = \frac{-e^{-x}}{(1+e^{-x}}$$

Так как:

$$(1+e^{-x})^2 = (1+e^{-x})(1+e^{-x})$$

$$e^{-x} = 1+e^{-x} - 1$$

То получим окончательный вид формулы производной сигмоиды:

$$\frac{-e^{-x}}{(1+e^{-x})^2} = \frac{1}{(1+e^{-x})}\left(1 - \frac{1}{(1+e^{-x})}\right) =$$

$$= \text{сигмоида}(x) * (1 - \text{сигмоида}(x))$$

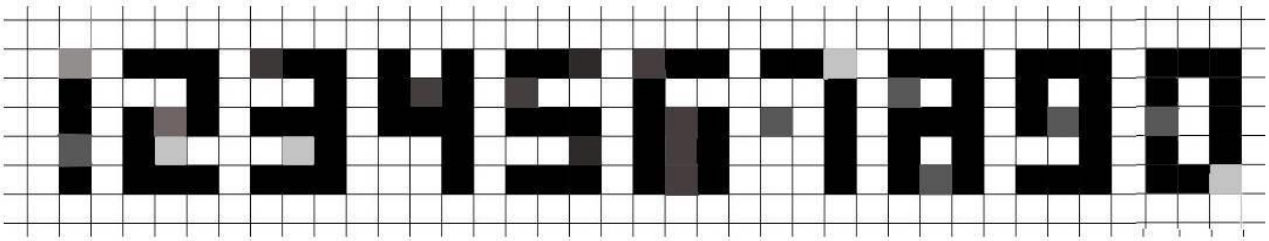
Что и требовалось доказать.

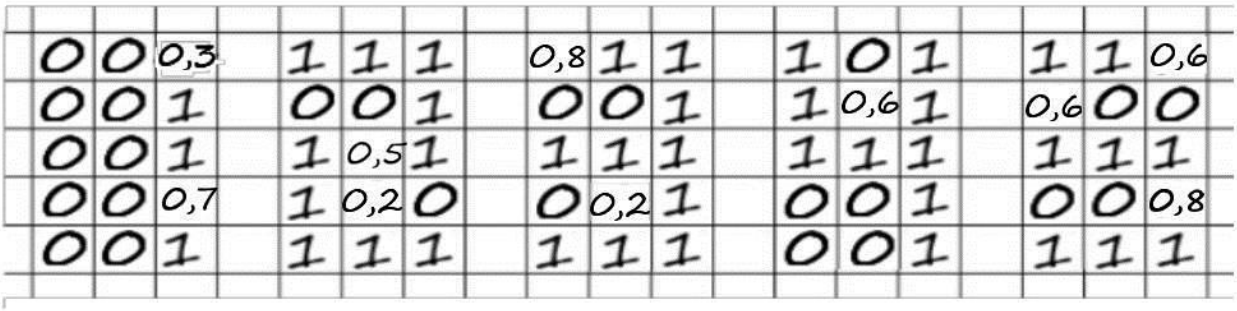
Практика распознавания цифры с применением сигмоиды

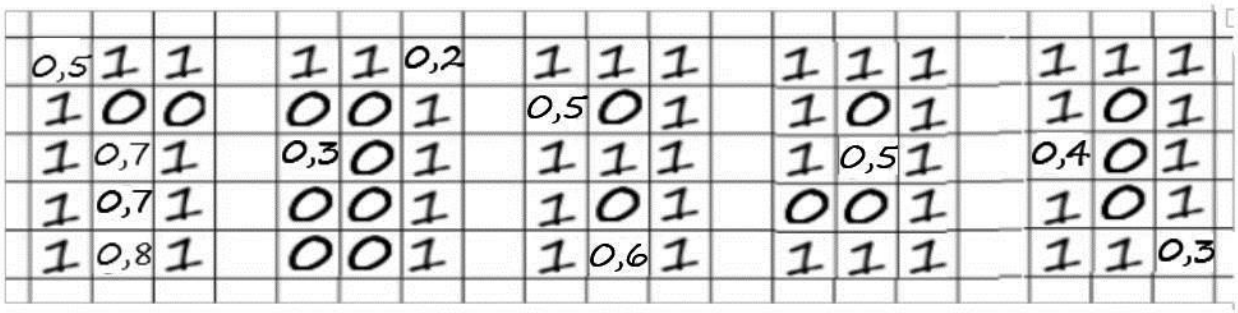
И по традиции закрепим результат практической работой. Здесь мы будем распознавать нужную нам цифру, а именно всё тот же ноль, с помощью сигмоидальной функции. Как говорилось ранее, при использовании данной функции, входные значения могут принимать значения от 0 до 1.

Если данные значительно выходят за эти пределы, то можно применить стандартизацию входных данных. В дальнейшем, подавая на входы нейрона стандартизированные данные, нейрон сможет прекрасно обучаться.

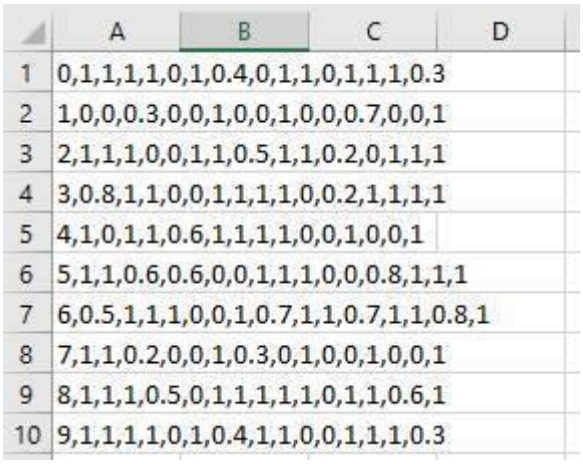
На вход будем подавать цифры в оттенках серого. Градация оттенков будет лежать в пределах от 0 до 1, выходить за эти пределы, следовательно, стандартизировать данные не будем:







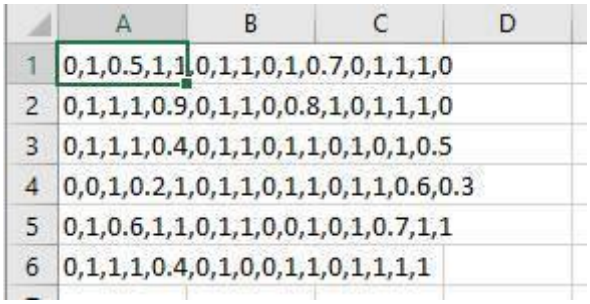
Входные данные так же помещаем в файл с расширением .csv:



	A	B	C	D
1	0,1,1,1,1,0,1,0.4,0,1,1,0,1,1,1,0.3			
2	1,0,0,0.3,0,0,1,0,0,1,0,0,0,0.7,0,0,1			
3	2,1,1,1,0,0,1,1,0.5,1,1,0.2,0,1,1,1			
4	3,0.8,1,1,0,0,1,1,1,1,0,0.2,1,1,1,1			
5	4,1,0,1,1,0.6,1,1,1,1,0,0,1,0,0,1			
6	5,1,1,0.6,0.6,0,0,1,1,1,0,0,0.8,1,1,1			
7	6,0.5,1,1,1,0,0,1,0.7,1,1,0.7,1,1,0.8,1			
8	7,1,1,0.2,0,0,1,0.3,0,1,0,0,1,0,0,1			
9	8,1,1,1,0.5,0,1,1,1,1,1,0,1,1,0.6,1			
10	9,1,1,1,1,0,1,0.4,1,1,0,0,1,1,1,0.3			

Напомню: нулевой элемент – целевой результат (ответ).

Данные для теста так же помещаем в отдельный файл:



	A	B	C	D
1	0,1,0.5,1,1,0,1,1,0,1,0.7,0,1,1,1,0			
2	0,1,1,1,0.9,0,1,1,0,0.8,1,0,1,1,1,0			
3	0,1,1,1,0.4,0,1,1,0,1,1,0,1,0,1,0.5			
4	0,0,1,0.2,1,0,1,1,0,1,1,0,1,1,0.6,0.3			
5	0,1,0.6,1,1,0,1,1,0,0,1,0,1,0.7,1,1			
6	0,1,1,1,0.4,0,1,0,0,1,1,0,1,1,1,1			

В программе так же подключаем все необходимые модули и значения из файлов .csv:

```
import numpy as np

# Загрузить и подготовить тренировочные данные из формата CSV в список
training_data  = open("dataset/Data_train.csv",  'r') # 'r' – открываем файл для чтения
training_data_list = training_data.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data.close()  # закрываем файл csv
```

```
# Загрузить и подготовить тестовые данные из формата CSV в список
test_data = open("dataset/Data_test.csv", 'r') # 'r' – открываем файл для чтения
test_data_list = test_data.readlines()# Загрузить и подготовить тестовые данные из формата CSV в список
test_data.close() # закрываем файл csv
```

Инициализируем параметры:

```
# Инициализация весов нейрона
weights = np.zeros(15)

# Скорость обучения
lr = 0.1

# Зададим количество эпох
epochs = 50000
```

Создаем цикл обучения:

```
# Прогон по обучающей выборке
for e in range(epochs):
for i in training_data_list:
# Получить входные данные числа
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])

# Получить целевое значение Y, (ответ – какое это число)
target_Y = int(all_values[0]) # перевод символов в int, 0 элемент – ответ

# Так как мы ищем только значение ноль, значит только он будет верным ответом = 1.
# остальные ответы, будут неверными, соответственно они обращаются в ноль.
if target_Y == 0:
target_Y = 1
else:
target_Y = 0
```

```
# Взвешенная сумма
x = np.sum(weights  * inputs_x)
# Функция активации
y = 1/(1+np.exp(-x))
```

```
# Ошибка E = -(целевое значение – выход нейрона)
E = -(target_Y  – y)
```

```
# Меняем веса по каждому из входов
weights  -= lr * E * y * (1.0 – y) * inputs_x
```

Здесь всё почти так же, как мы писали с функцией единичного скачка. Изменения коснулись условия – его нет, и значение на выходе проходит через активационную функцию – сигмоиду. Так же есть изменение в области обновления весовых коэффициентов. Сюда добавили производную функции активации (сигмоиды).

Далее следует код вывода результатов на консоль:

```
# Вывод обученных весов
print('Весовые коэффициенты:\n',weights)
```

```
# Еще раз пройдем по обучающей выборке
for i in training_data_list:
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
x = np.sum(weights  * inputs_x)
print(i[0],  'Вероятность что 0: ', 1/(1+np.exp(-x)))
```

```
# Проход по тестовой выборке
t = 0 # Счетчик номера нуля тестовой выборки
for i in test_data_list:
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
t += 1
x = np.sum(weights  * inputs_x)
```

```
print('Вероятность что узнал 0 -',t, '?', 1/(1+np.exp(-x)))
```

Полный тест программы:

```
import numpy as np


# Загрузить и подготовить тренировочные данные из формата CSV в список
training_data  = open("dataset/Data_train.csv",  'r') # 'r' – открываем файл для чтения
training_data_list  =  training_data.readlines()  #  readlines()  –  читает  все  строки  в  файле  в переменную training_data_list
training_data.close()  # закрываем файл csv


# Загрузить и подготовить тестовые данные из формата CSV в список
test_data = open("dataset/Data_test.csv",  'r') # 'r' – открываем файл для чтения
test_data_list = test_data.readlines()# Загрузить и подготовить тестовые данные из формата CSV в список
test_data.close() # закрываем файл csv


# Инициализация  весов нейрона
weights  = np.zeros(15)


# Скорость обучения
lr = 0.1


# Прогон по обучающей выборке
for e in range(epochs):
for i in training_data_list:
# Получить входные данные числа
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])


# Получить целевое значение Y, (ответ – какое это число)
target_Y = int(all_values[0])  # перевод символов в int, 0 элемент – ответ


# Так как мы ищем только значение ноль, значит только он будет верным ответом = 1.
# остальные ответы, будут неверными, соответственно они обращаются в ноль.
if target_Y == 0:
target_Y = 1
else:
target_Y = 0
```

```
# Взвешенная сумма
x = np.sum(weights  * inputs_x)
# Функция активации
y = 1/(1+np.exp(-x))



# Ошибка E = -(целевое значение – выход нейрона)
E = -(target_Y  – y)



# Меняем веса по каждому из входов
weights  -= lr * E * y * (1.0 – y) * inputs_x



# Вывод обученых весов
print('Весовые коэффициенты:\n',weights)



# Еще раз пройдем по обучающей выборке
for i in training_data_list:
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x  = np.asfarray(all_values[1:])
x = np.sum(weights  * inputs_x)
print(i[0],  'Вероятность что 0: ', 1/(1+np.exp(-x)))



# Проход по тестовой выборке
t = 0 # Счетчик номера нуля тестовой выборки
for i in test_data_list:
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x  = np.asfarray(all_values[1:])
t += 1
x = np.sum(weights  * inputs_x)



print('Вероятность что узнал 0 -',t, '?', 1/(1+np.exp(-x)))
```

Результат работы программы:

```
Весовые коэффициенты:
[ 0.44879589  -0.18430999  -0.02023563  2.0697523  -0.07193334  0.17307025
–2.1606589  -5.35195696  -1.45815401  3.73096992  -1.31490847  0.26365056
0.52117981  0.94419683  -4.20017402]
```

0 Вероятность что 0: 0.987440019679
1 Вероятность что 0: 0.00493344023472
2 Вероятность что 0: 0.00579884946173
3 Вероятность что 0: 1.14519734014e-05
4 Вероятность что 0: 3.43432571255e-05
5 Вероятность что 0: 4.53774043971e-05
6 Вероятность что 0: 0.00591089049169
7 Вероятность что 0: 0.00365001935213
8 Вероятность что 0: 0.00130998489482
9 Вероятность что 0: 0.00885106765671

Вероятность что узнал 0 – 1 ? 0.964467249281
Вероятность что узнал 0 – 2 ? 0.98802530455
Вероятность что узнал 0 – 3 ? 0.614232538622
Вероятность что узнал 0 – 4 ? 0.905331401561
Вероятность что узнал 0 – 5 ? 0.81811184231
Вероятность что узнал 0 – 6 ? 0.74017734823

Найти исходники, вы можете по следующей ссылке:

https://github.com/CaniaCan/neuralmaster

Видим, что после прогона по обучающей выборке, наш нейрон распознает число ноль без всякого труда (0,98744). Не удивительно, ведь он обучался на этой выборке. А вот по результатам прогона по тестовой выборке, значений которых нейрон еще не видел, вероятность заметно варьируется. Например, лучше всего он распознал второе тестовое обозначение нуля, его вероятность равна 0,988%. А хуже всех он распознал третье обозначение нуля – 0,614%. Но все же по многим значениям, результаты заметно выше 0,5. И можно говорить, что наш нейрон не ошибся в распознавании цифры ноль, ни на одном из примеров.

Вот теперь преимущества функций активации, в частности сигмоиды, над линейной функцией очевидны. Мы можем принимать на вход неограниченное число параметров, из широкого диапазона числовых значений (с учетом статистической нормализации данных).

Распознать число с использованием одной лишь линейной функцией классификации, не представлялось бы возможным.

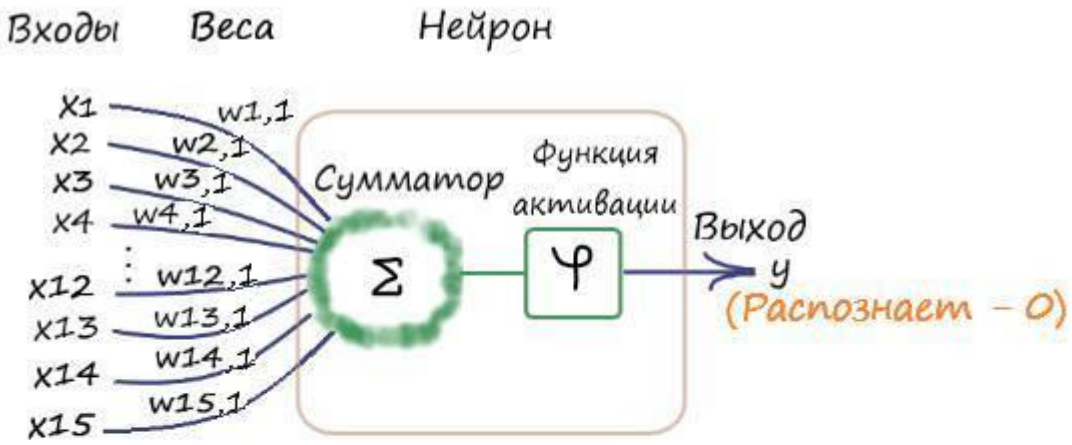
ГЛАВА 6

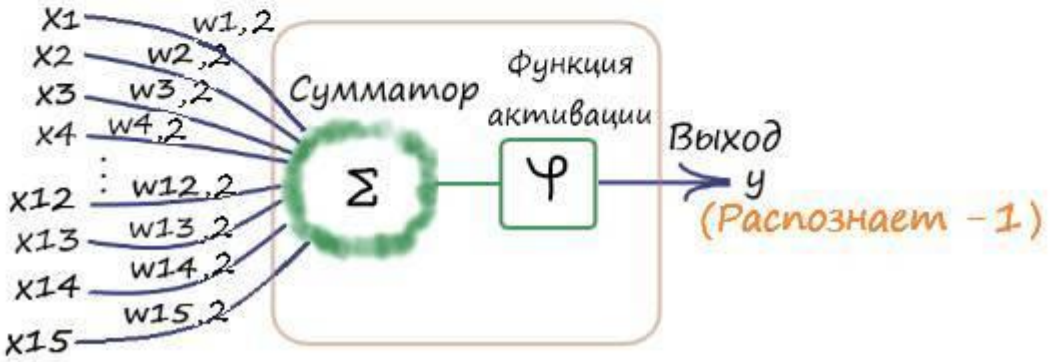
Распознаем больше данных

В этой главе рассмотрим сети из входных данных и нейронов. С помощью таких сетей, мы сможем распознавать ещё больше данных на выходе.

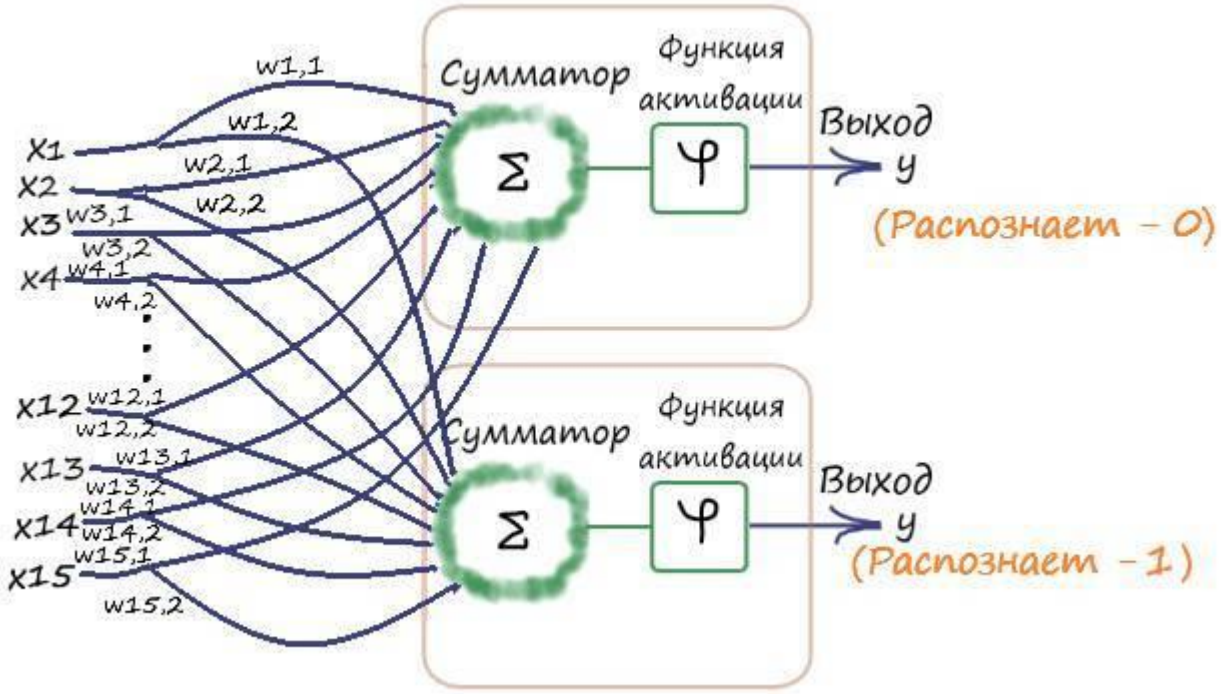
Как распознать несколько данных

Всё что мы делали до этого, при помощи линейной классификации и функций активации, мы могли лишь отделить, или распознать один конкретный тип данных от других. А что, если нам понадобится распознать не только цифру 0, а еще и к примеру цифру 1. Логичней всего представить второй такой же нейрон, как и с цифрой ноль, только вместо нуля он будет распознавать цифру 1:

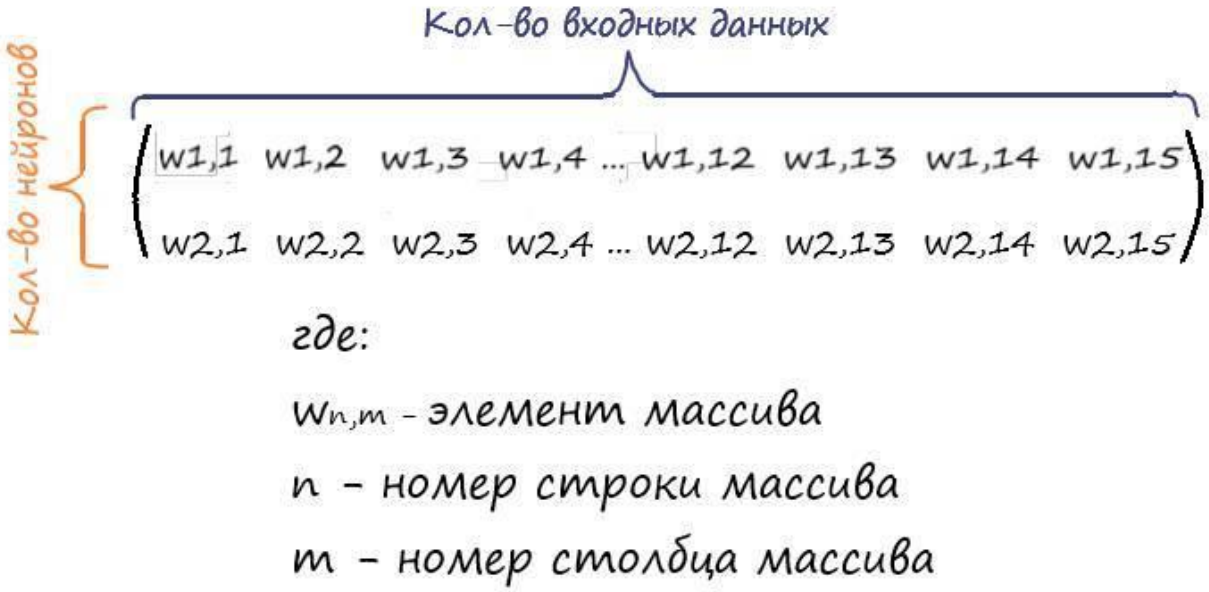




Ну а так как они одновременно получают одни и те же данные, то логично будет их объединить:



Теперь входы у нейронов общие, а весовые коэффициенты разные. У каждого коэффициента есть свой идентификационный номер, который говорит, к какому входу он принадлежит. Такой набор коэффициентов удобно представлять в виде двухмерного массива, где номера коэффициентов будут адресами элементов:



Действия с матрицами

В дальнейшем нам придётся выполнять действия с матрицами, а именно – перемножать. У матриц свои правила перемножения.

Вот пример умножения одной матрицы на другую:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} * \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1*5)+(2*7) & (1*6)+(2*8) \\ (3*5)+(4*7) & (4*6)+(4*8) \end{pmatrix} =$$

$$= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Возможно, вы уже догадались по каким правилам производится умножение. Если нет, то посмотрите на иллюстрацию, как получился первый элемент результирующей матрицы:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} * \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1*5)+(2*7) & (1*6)+(2*8) \\ (3*5)+(4*7) & (4*6)+(4*8) \end{pmatrix} =$$

$$= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Видим, что строки в первой матрицы, умножаются на соответствующие столбцы второй.
В общем виде, правило умножения матриц будет выглядеть как:

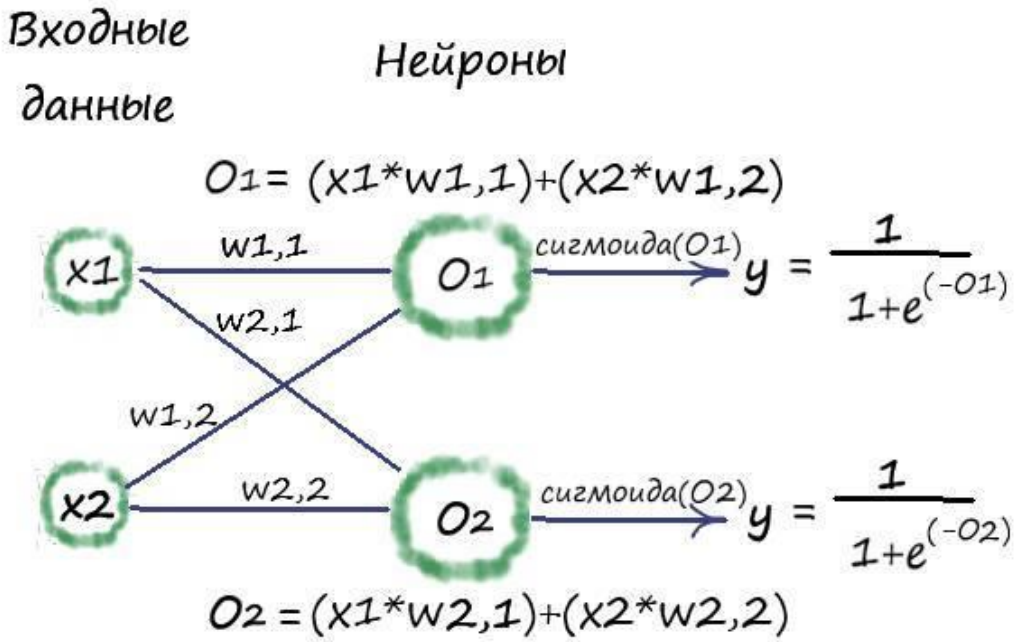
$$\begin{pmatrix} a & b\ldots \\ c & d\ldots \end{pmatrix} * \begin{pmatrix} e & f \\ g & h \\ \ldots & \ldots \end{pmatrix} = \begin{pmatrix} (a*e)+(b*g)+\ldots & (a*f)+(b*h)+\ldots \\ (c*e)+(b*d)+\ldots & (c*f)+(d*h)+\ldots \end{pmatrix}$$

Если число столбцов в первой матрице, не равно числу строк во второй – **то такие матрицы невозможно перемножить**. Это правило мы должны обязательно учитывать в будущем:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} e & f \end{pmatrix} =$$ невозможно (число строк и столбцов между матрицами не совпадает

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} e \\ f \end{pmatrix} = \begin{pmatrix} (a*e)+(b*f) \\ (c*e)+(d*f) \end{pmatrix}$$

Как действия с матрицами могут нам помочь? Представим, для простоты, что имеется два входа и два нейрона. Каждый вход имеет одну связь с каждым нейроном, через которую проходит сигнал. Такой вид связи, называется – **полносвязным:**



$$\begin{pmatrix} W1,1 & W1,2 \\ W2,1 & W2,2 \end{pmatrix} * \begin{pmatrix} X1 \\ X2 \end{pmatrix} = \begin{pmatrix} (X1*W1,1)+(X2*W1,2) \\ (X1*W2,1)+(X2*W2,2) \end{pmatrix} = \begin{pmatrix} O1 \\ O2 \end{pmatrix}$$

Это выражение можно записать ещё в более компактной форме:

O= W * I;

Где **W** – матрица весов, **I** – матрица входных данных, а **O** – результирующая матрица взвешенных сумм нейронов.

Так же замечаем, что немного изменилась нумерация индексов весов. Теперь, индекс n (номер строки) – показывает с каким по счету нейроном установлена данная связь, а индекс **m** (номер столбца) – показывает индекс номера входных данных.

Не нужно заботится о том, сколько узлов приходит на нейроны. Увеличение входных данных, приводит к увеличению размера матрицы по столбцам. А увеличение количества нейронов, ведет к увеличению размера матрицы по строкам.

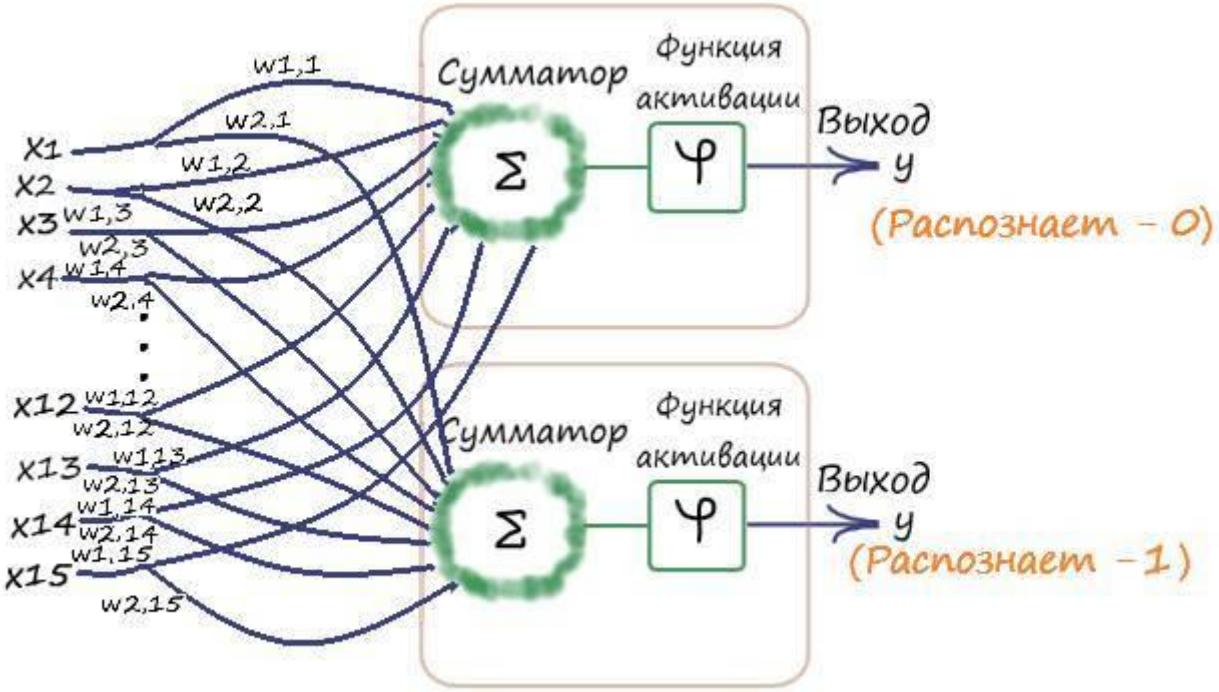
Вот так, легко и не принужденно, с помощью умножений матриц, мы можем вычислять взвешенную сумму в нейронах!

Практика и ещё раз практика

Напишем программу, которая будет распознавать несколько данных на выходе, а именно не только цифру 0, из нашего набора входных данных, но ещё и цифру 1. Для этого дополним данные в тестовой выборке, заполним её кроме шести тестовых значений нуля, шестью тестовыми значениями единицы:

	A	B	C	D
1	0,1,0.5,1,1,0,1,1,0,1,0.7,0,1,1,1,0			
2	0,1,1,1,0.9,0,1,1,0,0.8,1,0,1,1,1,0			
3	0,1,1,1,0.4,0,1,1,0,1,1,0,1,0,1,0.5			
4	0,0,1,0.2,1,0,1,1,0,1,1,0,1,1,0.6,0.3			
5	0,1,0.6,1,1,0,1,1,0,0,1,0,1,0.7,1,1			
6	0,1,1,1,0.4,0,1,0,0,1,1,0,1,1,1,1			
7	1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1			
8	1,0,0,0.7,0,0,1,0,0,1,0,0,0.5,0,0,1			
9	1,0.2,0,1,0,0,1,0,0,1,0.1,0,1,0,0,0.8			
10	1,0,0,1,0,0,0.5,0,0,1,0,0,0.9,0,0,1			
11	1,0,0,0.6,0,0.1,1,0,0,1,0,0,1,0,0,1			
12	1,0.1,0,1,0,0,0.8,0,0,0.9,0,0,0.7,0,0,1			

Структура такай сети уже приводилась, но давайте еще раз об этом вспомним, с учетом изменения нумерации весов:



Создавать в программе массивы весов, удобно в конструкторе класса. Для этого создадим класс – neuron_Net и инициализируем параметры в его конструкторе (__init__):

```
# Определение класса нейронной сети
class neuron_Net:


# Инициализация весов нейронной сети
def __init__(self, input_num, neuron_num, learningrate): #констр.(кол-во входов, кол-во нейронов)
# МАТРИЦА ВЕСОВ
# Задаем матрицу весов как случайное от -0,5 до 0,5
self.weights = (np.random.rand(neuron_num, input_num) -0.5)
# Задаем параметр скорости обучения
self.lr = learningrate


pass
```

Здесь функцией np.random.rand(neuron_num, input_num) -0.5), задаем значение весов от -0,5 до 0,5. Затем, эти значения присваиваем в переменную.

После чего, здесь же (в классе), создаём метод (функцию класса) обучения сети:

```
# Метод обучения нейронной сети
def train(self, inputs_list, targets_list): # принимает (вх. список данных, ответы)
# Преобразовать список входов в вертикальный массив. .T – транспонирование
inputs_x = np.array(inputs_list, ndmin=2).T # матрица числа
targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов: какое это число



# ВЫЧИСЛЕНИЕ СИГНАЛОВ
# Вычислить сигналы в нейронах. Взвешенная сумма.
x = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs
# Вычислить сигналы, выходящие из нейрона. Функция активации – сигмоида(x)
y = 1/(1+np.exp(-x))



# ВЫЧИСЛЕНИЕ ОШИБКИ
# Ошибка E = -(цель – фактическое значение)
E = -(targets_Y – y)



# ОБНОВЛЕНИЕ ВЕСОВ
# Меняем веса по каждой связи
self.weights -= self.lr * np.dot((E * y * (1.0 – y)), np.transpose(inputs_x))



pass
```

Функции np.array(inputs_list, ndmin=2).T и np.array(targets_list, ndmin=2).T, переворачивают входной вектор и вектор выходных значений соответственно. Что бы понять зачем это нужно, необходимо ещё немного дополнить знания по работе над матрицами. Нам нужно ознакомится с таким действием, как транспонирование матрицы. Чтобы наглядно понять, о чём идет речь, лучше всего проиллюстрировать данный процесс:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^T = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

$$(a \quad b \quad c)^T = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix}^T = (a \quad b \quad c)$$

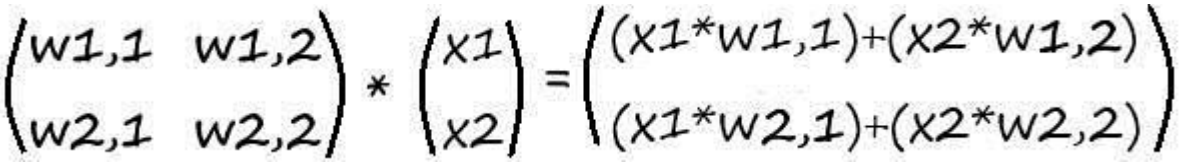
Ну и теперь все становится предельно ясно. Знак “T” справа над матрицей – означает что данная матрица транспонируется. А сам процесс транспонирования – это ни что иное, как замена в массиве строк на столбцы и наоборот.

После транспонирования наши входные и выходные данные, будут представляться в удобном для восприятия виде, а точнее – в вертикальных массивах, с которыми в дальнейшем мы уже сможем работать:

$$(x1 \quad x2 \quad x3 \ldots x13 \quad x14 \quad x15) = \begin{pmatrix} x1 \\ x2 \\ x3 \\ \vdots \\ x13 \\ x14 \\ x15 \end{pmatrix}$$

$$(y1 \quad y2) = \begin{pmatrix} y1 \\ y2 \end{pmatrix}$$

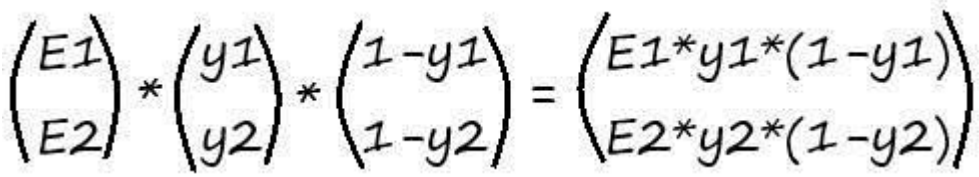
Ну а применив к входным данным процесс умножения матрицы весов, который нам уже известен, мы сможем получать значения взвешенной суммы:



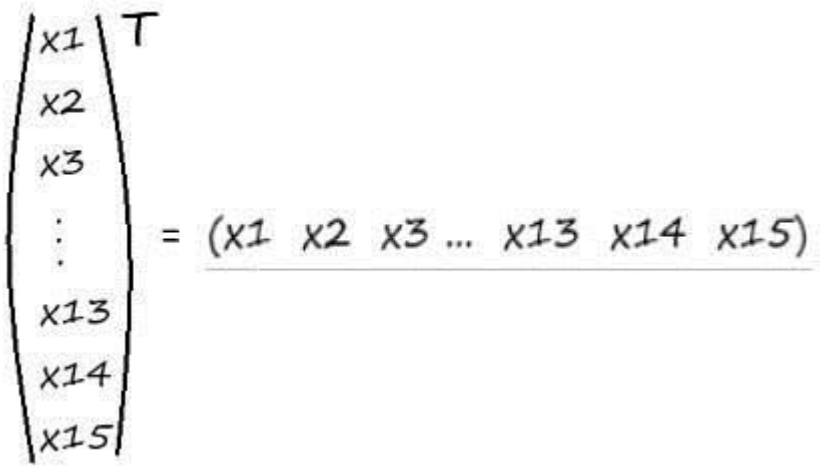
Этим и занимается выражение – np.dot(self.weights, inputs_x). Метод .dot, библиотеки numpy, перемножает, по правилам, матрицы, которые содержаться в его аргументах.

После чего, так же, как и в предыдущей программе, находим выходные значения, применив к взвешенной сумме функцию активации – y=1/1+np.exp(-x). Следом вычисляем ошибку на своих выходах – E = -(targets_Y – y), и затем обновляем весовые коэффициенты – self.weights -= self.lr * np.dot((E*y*(1.0 – y)), np.transpose(inputs_x)).

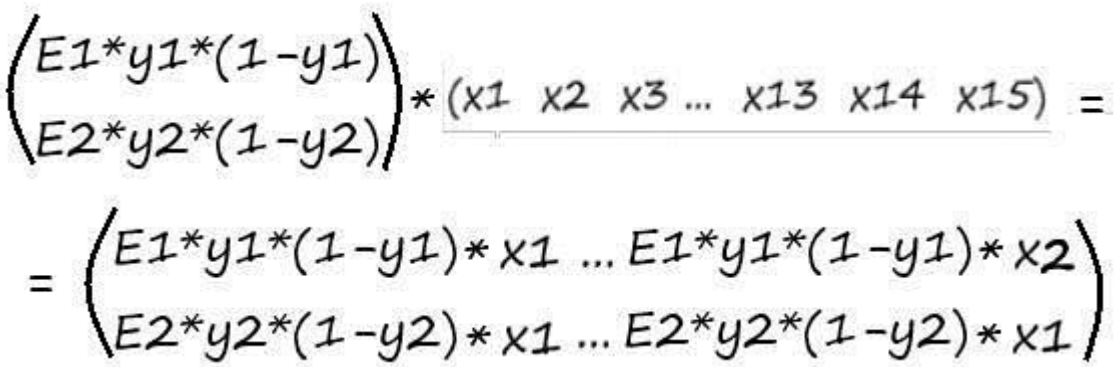
Если подробнее рассмотреть последнее выражение – np.dot((E*y*(1.0 – y)), np.transpose(inputs_x)), то можно заметить что произведение матрицы ошибки E, матриц y и (1– y), выполняются поэлементно:



А уже результат этой матрицы, перемножается с матрицей входных данных. А мы знаем, что по правилу перемножения матриц, число столбцов в первой матрицы, должно быть таким же, как и число строк во второй матрице. Для этого нам необходимо развернут (транспонировать) матрицу входов:



Эти преобразования осуществляет метод – .transpose(). А выражение np.transpose(inputs_x), транспонирует входную матрицу. Ну а всё выражение целиком – np.dot((E*y*(1.0 – y)), np.transpose(inputs_x)), перемножает между собой аргументы (матрицы):



После чего результат перемножается на скорость обучения и обновляются весовые коэффициенты – self.weights -= self.lr * np.dot((E * y * (1.0 – y)), np.transpose(inputs_x)).

Здесь же, в классе, создаем метод для прохода тестовых значений через сеть, который возвращает результаты на выходе:

```
# Метод прогона тестовых значений
def query(self, inputs_list): # Принимает свой набор тестовых данных
# Преобразовать список входов в вертикальный 2D массив.
inputs_x = np.array(inputs_list, ndmin=2).T


# Вычислить сигналы в нейронах. Взвешенная сумма.
x = np.dot(self.weights, inputs_x)
# Вычислить сигналы, выходящие из нейрона. Сигмоида(x)
```

```
y = 1/(1+np.exp(-x))


return y
```

Далее задаем параметры нашей сети:

```
# Количество входных данных, нейронов
data_input = 15
data_neuron = 2


# Скорость обучения
learningrate = 0.1


# Создать экземпляр нейронной сети
n = neuron_Net(data_input, data_neuron, learningrate)
```

И обучаем её:

```
# Зададим количество эпох
epochs = 40000
# Прогон по обучающей выборке
for e in range(epochs):
for i in training_data_list:
# Получить входные данные числа
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])


# Получить целевое значение Y, (ответ – какое это число)
targets_Y = int(all_values[0]) # перевод символов в int, 0 элемент – ответ


# создать целевые выходные значения (все 0.01, кроме нужной метки, которая равна 0.99)
targets_Y = np.zeros(data_neuron) + 0.01


# Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
```

```
if int(all_values[0]) <= 1: # цель <= 1 потому как распознаём только 2 числа, 0 и 1.
targets_Y[int(all_values[0])] = 0.99


n.train(inputs_x, targets_Y) # наш метод train – обучение нейронной сети
```

В этом отрывке, многое осталось от прошлого кода, только теперь выходные данные представляют из себя массив из двух элементов. Сначала создаем нулевой массив выходных данных – targets_Y = np.zeros(data_neuron) + 0.01, а затем помещаем значение 0.99 в индекс, который означает какое это число. Например, если это число выходит за пределы, которые мы распознаем, например число 3, то условие if int(all_values[0]) <= 1, даст целевые результаты – targets_Y =([0.01,0.01]).T, то есть оба ответа неверные. Если это одно из чисел, которое мы распознаем, к примеру 1, то условие if int(all_values[0]) <= 1, будет выполнятся и целевым результатом будет – targets_Y =([0.01,0.99]).T, то есть ответ что это 1. И ответ будет располагаться на своем индексе. Собственно, номер индекса в котором будет располагаться значение – 0.99, нам и будет говорить какое это число.

Минимальные ответы – 0.01 и максимальные – 0.99, если прменяется сигмоида – не должны быть изначально в 0 или 1, так как это может привести к потере данных, в прошлом нам везло и сеть удачно распознавала данные со значениями 0 и 1 в целевых данных.

После всех необходимых преобразований с входными данными и целевыми значениями, приступаем к обучению сети, путем обращения к методу – .train, нашего класса:

```
n.train(inputs_x, targets_Y)
```

Далее, по традиции следует вывод полученных результатов после обучения:

```
# Вывод обученных весов
print('Весовые коэффициенты:\n', n.weights)


# Еще раз пройдем по обучающей выборке
for i in training_data_list:
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
# Прогон по сети
outputs = n.query(inputs_x)
print(i[0], 'Вероятность:\n', outputs)


# Если вероятность больше 0,5 и номер выхода совпадает с ответом, то считаем что сеть,
#на своем определенном выходе, узнала цифру.
for i in training_data_list:
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
```

```
# Прогон по сети
outputs = n.query(inputs_x)
# индекс самого высокого значения соответствует метке
label = np.argmax(outputs)
if outputs[label]>0.5 and int(all_values[0]) == label:
print(i[0], 'Узнал?: ', 'Да!')
else:
print(i[0], 'Узнал?: ', 'Нет!')
```

```
# Проход по тестовой выборке
t = 0 # Счетчик номера нуля тестовой выборки
t1 = 0 # Счетчик номера единицы тестовой выборки
for i in test_data_list:
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
t += 1
# Прогон по сети
outputs = n.query(inputs_x)
# индекс самого высокого значения соответствует метке
label = np.argmax(outputs)
if t <= 6:
print('Вероятность что узнал 0 -',t, '?', outputs[label])
else:
t1 += 1
print('Вероятность что узнал 1 -',t1, '?', outputs[label])
```

```
t = 0 # Счетчик номера нуля тестовой выборки
t1 = 0 # Счетчик номера единицы тестовой выборки
# Если вероятность больше 0,5 и номер выхода совпадает с ответом, то считаем что сеть,
#на своем определенном выходе, узнала цифру.
for i in test_data_list:
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
# Прогон по сети
outputs = n.query(inputs_x)
# индекс самого высокого значения соответствует метке
label = np.argmax(outputs)
t += 1
if outputs[label]>0.5 and int(all_values[0]) == label:
print(i[0], 'Узнал?:',t, 'Да!')
else:
t1 += 1
print(i[0], 'Узнал?:',t1, 'Нет!')
```

При тестовых проходах, по обучающей и тестовой выборок, сначала выводим вероятности распознавания цифры (ответы сети), а затем, через условие: если эти ответы больше значения 0.5, и индекс максимального числа из матрицы результатов ответа сети, равен индексу ответов из матрицы целевых значений, то сеть распознала цифру, в противном случае не распознала.

Полный текст программы:

```
import numpy as np


# Загрузить и подготовить тренировочные данные из формата CSV в список
training_data = open("dataset/Data_train.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data.close() # закрываем файл csv


# Загрузить и подготовить тестовые данные из формата CSV в список
test_data = open("dataset/Data_test.csv", 'r') # 'r' – открываем файл для чтения
test_data_list = test_data.readlines()# Загрузить и подготовить тестовые данные из формата CSV в список
test_data.close() # закрываем файл csv


# Определение класса нейронной сети
class neuron_Net:


# Инициализация весов нейронной сети
def __init__(self, input_num, neuron_num, learningrate): #констр.(кол-во входов, кол-во нейронов)
# МАТРИЦА ВЕСОВ
# Задаем матрицу весов как случайное от -0,5 до 0,5
self.weights = (np.random.rand(neuron_num, input_num) -0.5)
# Задаем параметр скорости обучения
self.lr = learningrate


pass


# Метод обучения нейронной сети
def train(self, inputs_list, targets_list): # принимает (вх. список данных, ответы)
# Преобразовать список входов в вертикальный массив. .T – транспонирование
inputs_x = np.array(inputs_list, ndmin=2).T # матрица числа
targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов: какое это число


# ВЫЧИСЛЕНИЕ СИГНАЛОВ
```

```
# Вычислить сигналы в нейронах. Взвешенная сумма.
x = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs
# Вычислить сигналы, выходящие из нейрона. Функция активации – сигмоида(x)
y = 1/(1+np.exp(-x))



# ВЫЧИСЛЕНИЕ ОШИБКИ
# Ошибка E = -(цель – фактическое значение)
E = -(targets_Y – y)



# ОБНОВЛЕНИЕ ВЕСОВ
# Меняем веса по каждой связи
self.weights -= self.lr * np.dot((E * y * (1.0 – y)), np.transpose(inputs_x))



pass



# Метод прогона тестовых значений
def query(self, inputs_list): # Принимает свой набор тестовых данных
# Преобразовать список входов в вертикальный 2D массив.
inputs_x = np.array(inputs_list, ndmin=2).T



# Вычислить сигналы в нейронах. Взвешенная сумма.
x = np.dot(self.weights, inputs_x)
# Вычислить сигналы, выходящие из нейрона. Сигмоида(x)
y = 1/(1+np.exp(-x))



return y



# ЗАДАЁМ ПАРАМЕТРЫ СЕТИ
# Количество входных данных, нейронов
data_input = 15
data_neuron = 2



# Скорость обучения
learningrate = 0.1



# Создать экземпляр нейронной сети
n = neuron_Net(data_input, data_neuron, learningrate)
```

```
# ОБУЧЕНИЕ
# Зададим количество эпох
epochs = 40000
# Прогон по обучающей выборке
for e in range(epochs):
for i in training_data_list:
# Получить входные данные числа
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])



# Получить целевое значение Y, (ответ – какое это число)
targets_Y = int(all_values[0])  # перевод символов в int, 0 элемент – ответ



# создать целевые выходные значения (все 0.01, кроме нужной метки, которая равна 0.99)
targets_Y = np.zeros(data_neuron)  + 0.01



# Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
if int(all_values[0])  <= 1: # цель <= 1 потому как распознаём только 2 числа, 0 и 1.
targets_Y[int(all_values[0])]  = 0.99



n.train(inputs_x,  targets_Y) # наш метод train – обучение нейронной сети



# Вывод обученных весов
print('Весовые коэффициенты:\n',  n.weights)



# Еще раз пройдем по обучающей выборке
for i in training_data_list:
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
# Прогон по сети
outputs = n.query(inputs_x)
print(i[0],  'Вероятность:\n', outputs)



# Если вероятность больше 0,5 и номер выхода совпадает с ответом, то считаем что сеть,
#на своем определенном выходе, узнала цифру.
for i in training_data_list:
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
```

```
# Прогон по сети
outputs = n.query(inputs_x)
# индекс самого высокого значения соответствует метке
label = np.argmax(outputs)
if outputs[label]>0.5 and int(all_values[0]) == label:
print(i[0], 'Узнал?: ', 'Да!')
else:
print(i[0], 'Узнал?: ', 'Нет!')



# Проход по тестовой выборке
t = 0 # Счетчик номера нуля тестовой выборки
t1 = 0 # Счетчик номера единицы тестовой выборки
for i in test_data_list:
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
t += 1
# Прогон по сети
outputs = n.query(inputs_x)
# индекс самого высокого значения соответствует метке
label = np.argmax(outputs)
if t <= 6:
print('Вероятность что узнал 0 -',t, '?', outputs[label])
else:
t1 += 1
print('Вероятность что узнал 1 -',t1, '?', outputs[label])



t = 0 # Счетчик номера нуля тестовой выборки
t1 = 0 # Счетчик номера единицы тестовой выборки
# Если вероятность больше 0,5 и номер выхода совпадает с ответом, то считаем что сеть,
#на своем определенном выходе, узнала цифру.
for i in test_data_list:
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
# Прогон по сети
outputs = n.query(inputs_x)
# индекс самого высокого значения соответствует метке
label = np.argmax(outputs)
t += 1
if outputs[label]>0.5 and int(all_values[0]) == label:
print(i[0], 'Узнал?:',t, 'Да!')
else:
t1 += 1
print(i[0], 'Узнал?:',t1, 'Нет!')
```

Результат работы программы:

Весовые коэффициенты:

```
[[ 0.55376239  -0.2659114  -0.55801795  1.63794241  -0.39180754  0.18253672
–1.86935497  -4.51957151  -1.10400713  3.68316632  -1.31690796  0.48928634
0.63951451  0.6593403  -3.74255197]
[-4.18510101  -3.35072555  -0.15142484  -0.89959198  -0.54914752  1.80364436
–2.10131379  -0.79061126  0.94477939  0.27198919  -0.42900452  0.13232677
–0.54182952  0.16586589  1.24938759]]
```

```
0 Вероятность:
[[ 0.98282717]
[ 0.0018778  ]]
1 Вероятность:
[[ 0.01110791]
[ 0.98279182]]
2 Вероятность:
[[ 0.01280932]
[ 0.00169931]]
3 Вероятность:
[[ 4.97172575e-05]
[ 2.29715820e-03]]
4 Вероятность:
[[ 0.00010462]
[ 0.0130299  ]]
5 Вероятность:
[[ 0.00018235]
[ 0.00010789]]
6 Вероятность:
[[ 0.01247943]
[ 0.00070262]]
7 Вероятность:
[[ 0.01036096]
[ 0.01685072]]
8 Вероятность:
[[ 0.00498377]
[ 0.00085023]]
9 Вероятность:
[[ 0.01543393]
[ 0.00064968]]
```

```
0 Узнал?: Да!
1 Узнал?: Да!
2 Узнал?: Нет!
3 Узнал?: Нет!
4 Узнал?: Нет!
5 Узнал?: Нет!
6 Узнал?: Нет!
7 Узнал?: Нет!
8 Узнал?: Нет!
9 Узнал?: Нет!
```

Вероятность что узнал 0 – 1 ? [0.95590294]
Вероятность что узнал 0 – 2 ? [0.98378171]
Вероятность что узнал 0 – 3 ? [0.63522571]
Вероятность что узнал 0 – 4 ? [0.92786908]
Вероятность что узнал 0 – 5 ? [0.78988343]
Вероятность что узнал 0 – 6 ? [0.76715129]
Вероятность что узнал 1 – 1 ? [0.98163393]
Вероятность что узнал 1 – 2 ? [0.98125632]
Вероятность что узнал 1 – 3 ? [0.94877843]
Вероятность что узнал 1 – 4 ? [0.95536855]
Вероятность что узнал 1 – 5 ? [0.9817356]
Вероятность что узнал 1 – 6 ? [0.95543842]

0 Узнал?: 1 Да!
0 Узнал?: 2 Да!
0 Узнал?: 3 Да!
0 Узнал?: 4 Да!
0 Узнал?: 5 Да!
0 Узнал?: 6 Да!
1 Узнал?: 7 Да!
1 Узнал?: 8 Да!
1 Узнал?: 9 Да!
1 Узнал?: 10 Да!
1 Узнал?: 11 Да!
1 Узнал?: 12 Да!

Код программы можно найти по следующей ссылке:

https://github.com/CaniaCan/neuralmaster

В обучающей выборке вероятность нахождения числа превышает 0.98. Большинство вероятностей нахождения числа из тестовой выборки, превышает 0.95, а в среднем это значение около 0,91. А в итоге, в обучающей и тестовой выборке, сеть распознала всё без исключения, 100% результат. Что весьма неплохо!

Как вы понимаете, добавив еще восемь нейронов и объединив их связями с входными данными, где каждый нейрон будет отвечать за свою отдельную цифру, можно распознавать весь набор чисел (от 0 до 9), а добавив ещё нейронов, отвечающих за символы, можно и помимо чисел распознавать и символы, как печатные, так и рукописные.

Входными значениями, могут служить не только значения цифр или символов, они могут быть любыми другими. Например, хотя бы всё те же параметры животных, входными параметрами которых будут являться их вес, пол, возраст, любые другие параметры их тела и так далее. А целевыми результатами, будет ответ какое животное соответствует этим всем параметрам. Входных параметров и целевых результатов, может быть огромное множество.

Резюмируя всё выше сказанное, можно с уверенностью утверждать, что мы проделали огромный путь, в ходе которого наш нейрон эволюционно многому научился. От начала своего появления, с возможностью принимать только один входной параметр и классифицировать только два вида входных данных, до того что он приобрел способности принимать на вход

неограниченное число данных, и объединившись с ними в сети, стало возможным получать неограниченное число ответов на выходе.

ГЛАВА 7

Нейронные сети

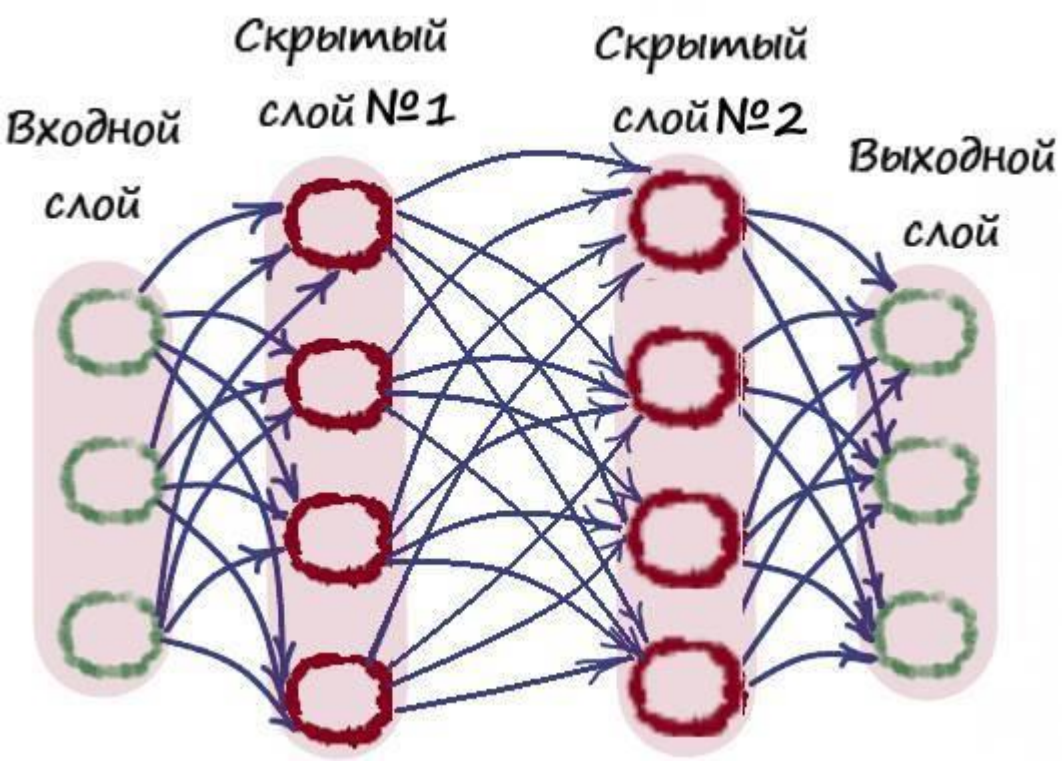
В этой главе рассмотрим ситуации, когда с выходов одних нейронов (аксонов), сигналы поступают на вход других (дендриты).

Виды искусственных нейронных сетей

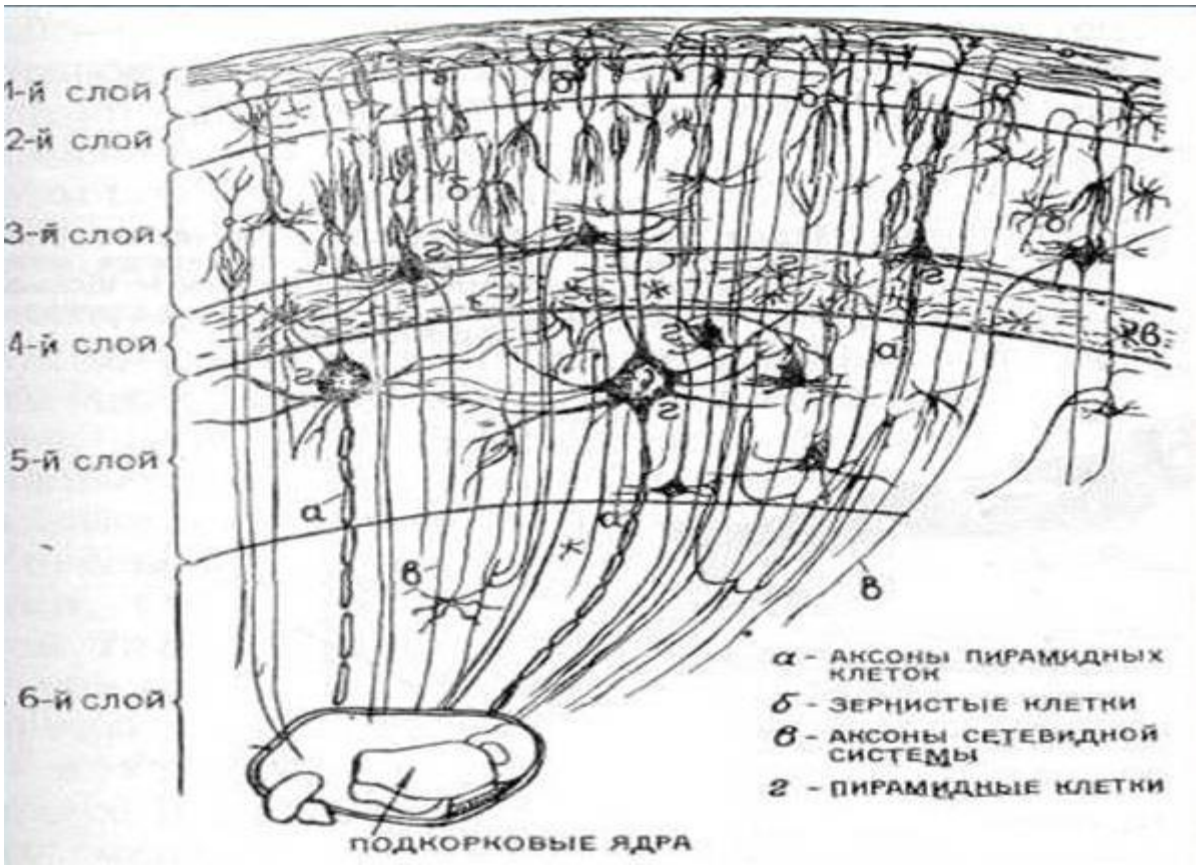
Мы разобрались со структурой искусственного нейрона.

Искусственные нейронные сети, состоят из множественных соединений нейронов между собой.

До этого мы рассмотрели вариант, когда входные данные сразу подавались на входы нейронов. В большинстве нейронных сетей, входные данные называют – **входным слоем.** Входной слой выполняет задачу – распределения входных сигналов между нейронами, никаких вычислений он не производит. Группа нейронов, на которые поступает сигнал от входного слоя, называется – **скрытым слоем**. Скрытых слоев может быть несколько. Ну а группу нейронов, с которых мы считываем данные (ответы сети), называют – **выходным слоем**. Иллюстрация выше сказанного, наглядно всё покажет:



Подобная структура нейронных сетей копирует многослойную структуру биологического головного мозга.



Неслучайно скрытый слой получил такое название. Разработать алгоритмы обучения нейронов срытого слоя, удалось в последнюю очередь. До этого, как и мы с вами, умели обучать нейроны лишь с одним слоем.

Многослойные нейронные сети, имеющие хотя бы один скрытый слой, обладают гораздо большими возможностями, чем однослойные.

Преимущества многослойных сетей

Ограничения при использовании лишь одного слоя, можно показать на решение задачи, путем обучения однослойной сети, логического “исключающее или”, или других логический функций, которые мы рассматривали, и как её еще называют – “XOR”. Логическая функция – “исключающее или”, имеет следующую таблицу истинности:

x1	x2	Y – Исключающее ИЛИ
0	0	0
1	0	1
0	1	1
1	1	0

Если взять нашу однослойную сеть, которую мы программировали последней, заменить в файлах входные и тестовые данные, на функцию “исключающее или” и поменять параметры по количеству входных данных и нейронов, то в итоге получим сеть, которая будет пытаться обучиться решить задачу данной функции.

Меняем входные данные в обучающей выборке, на значения из таблицы истинности логического “исключающее или”:

	A
1	0,0,0
2	1,1,0
3	1,0,1
4	0,1,1

Менять данные в тестовой выборке, в данном примере нет необходимости, так как значения идентичны с обучаемой. Тестировать будем на обучающих примерах.

Чтобы не менять программу, оставим за нулевым элементом право на целевое значение логического “исключающее или”.

Изменим параметры количества данных и нейронов:

```
# Количество входных данных, нейронов
data_input = 2
data_neuron = 2
```

Сделаем вывод выходных данных:

```
# Прогоним входные данные из обучающей выборки через обученную сеть
for i in training_data_list:
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
# Прогон по сети
outputs = n.query(inputs_x)
print(all_values[1], 'XOR', all_values[2], outputs)
```

И всё, наша программа готова.

Но все же стоит привести полный текст программы:

```
import numpy as np
# Загрузить и подготовить тренировочные данные из формата CSV в список
training_data = open("dataset/Data_train.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data.close() # закрываем файл csv
```

```
    # Загрузить и подготовить тестовые данные из формата CSV в список
    test_data = open("dataset/Data_test.csv", 'r') # 'r' – открываем файл для чтения
    test_data_list = test_data.readlines()# Загрузить и подготовить тестовые данные из формата CSV в список
    test_data.close() # закрываем файл csv
    # Определение класса нейронной сети
    class neuron_Net:
    # Инициализация весов нейронной сети
    def __init__(self, input_num, neuron_num, learningrate): #констр.(кол-во входов, кол-во нейронов, скорость обучения)
    # МАТРИЦА ВЕСОВ
    # Задаем матрицу весов как случайное от -0,5 до 0,5
    self.weights = (np.random.rand(neuron_num, input_num) -0.5)
    # Задаем параметр скорости обучения
    self.lr = learningrate



    pass



    # Метод обучения нейронной сети
    def train(self, inputs_list, targets_list): # принимает (вх. список данных, ответы)
    # Преобразовать список входов в вертикальный массив. .T – транспонирование
    inputs_x = np.array(inputs_list, ndmin=2).T # матрица числа
    targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов: какое это число



    # ВЫЧИСЛЕНИЕ СИГНАЛОВ
    # Вычислить сигналы в нейронах. Взвешенная сумма.
    x = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs
    # Вычислить сигналы, выходящие из нейрона. Функция активации – сигмоида(x)
    y = 1/(1+np.exp(-x))



    # ВЫЧИСЛЕНИЕ ОШИБКИ
    # Ошибка E = -(цель – фактическое значение)
    E = -(targets_Y – y)



    # ОБНОВЛЕНИЕ ВЕСОВ
    # Меняем веса по каждой связи
    self.weights -= self.lr * np.dot((E * y * (1.0 – y)), np.transpose(inputs_x))



    pass
```

```
# Метод прогона тестовых значений
def query(self, inputs_list): # Принимает свой набор тестовых данных
# Преобразовать список входов в вертикальный 2D массив.
inputs_x = np.array(inputs_list, ndmin=2).T



# Вычислить сигналы в нейронах. Взвешенная сумма.
x = np.dot(self.weights, inputs_x)
# Вычислить сигналы, выходящие из нейрона. Сигмоида(x)
y = 1/(1+np.exp(-x))



return y
# ЗАДАЁМ ПАРАМЕТРЫ СЕТИ
# Количество входных данных, нейронов
data_input = 2
data_neuron = 2



# Скорость обучения
learningrate = 0.1



# Создать экземпляр нейронной сети
n = neuron_Net(data_input, data_neuron, learningrate)
# ОБУЧЕНИЕ
# Зададим количество эпох
epochs = 10000
# Прогон по обучающей выборке
for e in range(epochs):
for i in training_data_list:
# Получить входные данные числа
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])



# Получить целевое значение Y, (ответ – какое это число)
targets_Y = int(all_values[0]) # перевод символов в int, 0 элемент – ответ



# создать целевые выходные значения (все 0.01, кроме нужной метки, которая равна 0.99)
targets_Y = np.zeros(data_neuron) + 0.01



# Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
targets_Y[int(all_values[0])] = 0.99
```

```
n.train(inputs_x, targets_Y) # наш метод train – обучение нейронной сети
# Вывод обученных весов
print('Весовые коэффициенты:\n', n.weights)
# Прогоним входные данные из обучающей выборки через обученную сеть
for i in training_data_list:
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
# Прогон по сети
outputs = n.query(inputs_x)
print(all_values[1], 'XOR', all_values[2], outputs)
```

Результат работы программы:

```
Весовые коэффициенты:
[[ 0.0082007  0.0082007]
[-0.0082007  -0.0082007]]


0 XOR 0
[[ 0.5]
[ 0.5]]
1 XOR 0
[[ 0.50205016]
[ 0.49794984]]
0 XOR 1
[[ 0.50205016]
[ 0.49794984]]
1 XOR 1
[[ 0.50410026]
[ 0.49589974]]
```

Как видим, нашей сети не удалось решить задачу подобного рода. Значения на выходе сети, примерно совпадают – 0,5. Что бы мы не предпринимали с этими значениями (складывали, искали максимальное и т.д.), решить данную задачу мы так и не сможем. Вероятность нужного числа на выходе, всегда будет 50% на 50%, что конечно же никого не может удовлетворить.

Попробовав решить подобным образом и другие логические функции “И” и “ИЛИ”, результат был бы тоже отрицательный. Имея лишь один слой, логистическая функция, в отличии от пороговой функции активации нейронов, не сможет верно подобрать веса. И действительно, на примере логического “И”:

x1w1 + x2w2 + w3 = 0 * 0,5 + 0 * 0,5 = 0
x1w1 + x2w2 + w3 = 1 * 0,5 + 0 * 0,5 = 0,5
x1w1 + x2w2 + w3 = 0 * 0,5 + 1 * 0,5 = 0,5
x1w1 + x2w2 + w3 = 1 * 0,5 + 1 * 0,5 = 1

Так же, как и с линейной функцией, в отличие от функции активации единичного скачка, логистическая функция бессильна при решении данной задачи.

Но всё же, функция “исключающее или”, выбрана не зря. Решая её даже с помощью активационной функции единичного скачка, нас так же постигнет неудача. Все дело в том, что, исходя из двух выражений логической функции: 0+0=0 и 1+1=0, какие бы мы не выбирали значения весов и порога, ответы на выходе из нейронов не будут совпадать с целевыми значениями.

Но и у этой проблемы есть своё решение. Но для этого, стоит немного рассказать о булевой алгебре.

Как вы уже знаете, операцию логического “ИЛИ”, часто называют логическим сложением. Правила здесь такие же, как и в обычном сложении, за исключением 1+1=1, а не 2. А операцию логического “И”, называю логическим произведением, здесь всё в целом производится аналогично обычному умножению. Есть еще одна распространённая логическая функция – функция логического “НЕ”. Из её названия следует, что её предназначение инвертировать значения. Если это 0, то пройдя логическое “НЕ”, 0 станет 1.

$\overline{x}$ – логическое "НЕ"

Если $x = 1$, то $\overline{x} = 0$

Если $x = 0$, то $\overline{x} = 1$

Так как же математически решить функцию “исключающее или”? Просто! Если мы применим следующие выражение:

$$Y = (\overline{x1} * x2) + (x1 * \overline{x2})$$

И применив его к таблице истинности функции “исключающее или”, получим следующие результаты:

1) $x1=0, x2=0$:

$(\overline{x1} * x2) + (x1 * \overline{x2}) = (1 * 0) + (0 * 1) = 0$

2) $x1=1, x2=0$:

$(\overline{x1} * x2) + (x1 * \overline{x2}) = (0 * 0) + (1 * 1) = 1$

3) $x1=0, x2=1$:

$(\overline{x1} * x2) + (x1 * \overline{x2}) = (1 * 1) + (0 * 0) = 1$

4) $x1=1, x2=1$:

$(\overline{x1} * x2) + (x1 * \overline{x2}) = (0 * 1) + (1 * 0) = 0$

Отлично! Математически разобрались. Теперь как это передать в нашу сеть?
Если мы, в нашей сети, условно передадим одному из нейронов решать задачу:

$$(\overline{x1} * x2)$$

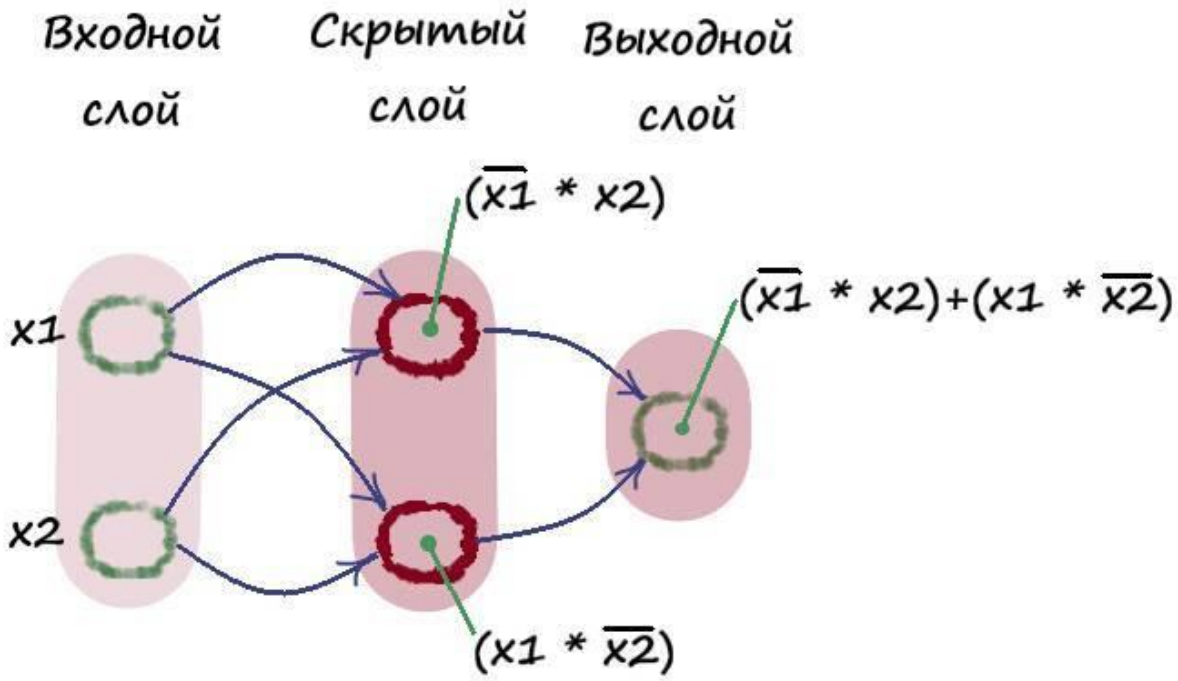
, а другому:

$$(x1 * \overline{x2})$$

и на выходе просуммировать значения, то должно все получиться.

Теперь нужно как то, заставить нейроны самостоятельно выполнять эти действия. И нужно это сделать так, чтобы это происходило автоматически, в процессе обучения, чтобы нейроны сами решили, без нашей подсказки, кому какие действия производить.

Но все же, без подсказки не обойтись, иначе нейроны так ничего и не поймут. Поэтому, логичнее всего если в роли подсказчика выступал бы ещё один нейрон, стоящий следом. Он и будет вычислять ошибку на выходе, после чего сообщать о ней двум предыдущим нейронам. Так же, роль по суммированию этих двух выражений, автоматически отводится выходному нейрону. Так как эта сумма будет представлять, ни что иное как взвешенную сумму:



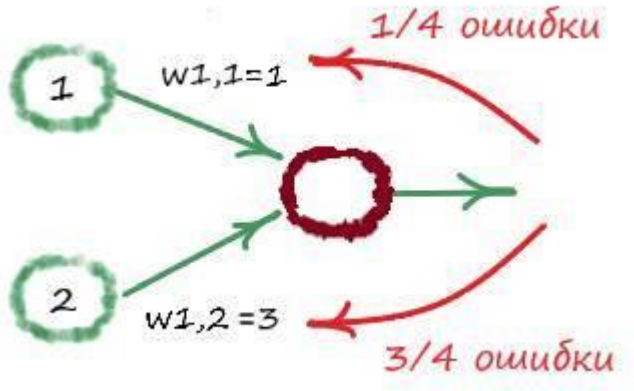
Тем самым передавая назад значение ошибки, выходной нейрон будет говорить позади стоящим, на какую величину они ошибаются. В результате этих действий, нейроны как бы сами разберутся, кто какое из двух выражений будет выполнять.

Такой вид сети, а именно состоящая из трёх слоёв, самая распространённая в искусственных нейронных сетях. Именно в скрытом слое происходит основное вычисление, в этом его предназначение.

Распространение ошибки назад

Как находить ошибку на выходе мы прекрасно усвоили: **E = -(Y – y)**. Но как находить ошибку в скрытых слоях? Как распространять ошибку назад относительно выхода?

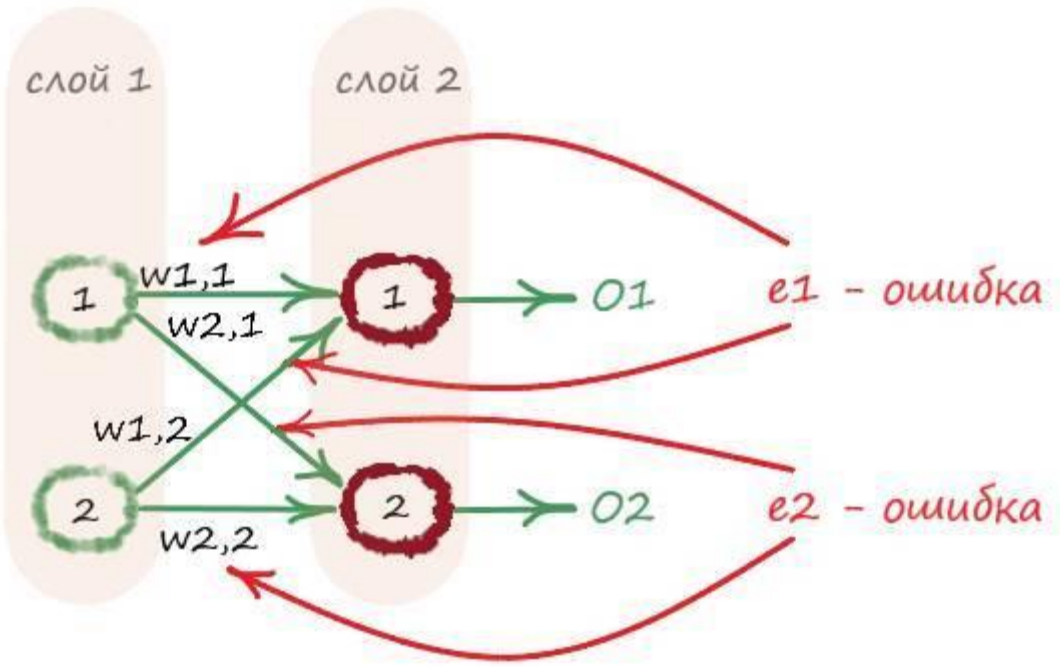
Использовать лишь одну величину ошибки на выходе для всех связей, затея глупая. Так как величина ошибки, определяется вкладами всех связей, а не только одной. Значит нам нужно каким-то образом, распределить эту ошибку между всеми узлами, пропорционально их вкладам в неё. То есть, большая часть ошибки приписывается тем связям, которые имеют больший вес, потому что они оказывают большее влияние на величину ошибки:



В данном случае, сигнал на выходном нейроне, формируется за счет двух позади стоящих нейронов. Их весовые коэффициенты равны 1 и 3, соответственно. Если ошибку на выходе, разделить пропорционально между двух позади стоящих нейронов, то для того чтобы обновить меньший вес (**w1,1=1**) следует использовать **1/4** части величины ошибки, а для обновления большего веса (**w1,2=3**) понадобится **3/4** её части. Если представить, что сеть имеет тысяча или более нейронов, связанных подобным образом с выходным нейроном, мы бы распределяли выходную ошибку между всеми этими связями, пропорционально их вкладам, которые определяются размерами весов соответствующих связей.

Теперь мы будем использовать весовые коэффициенты не только для расчетов распространения сигналов по нейронной сети, от входа к выходу, но еще и для вычисления распространения ошибки от выходного слоя к входному. У такого метода распространения ошибки есть созвучное название – **метод обратного распространения ошибки**.

Теперь рассмотрим ситуацию распространения ошибки при большем количестве нейронов на выходе:



Как мы можем видеть, для обновления весов, в процессе обучения, необходима информация о всех ошибках на выходных узлах. Но мы можем всё так же спокойно, распространить ошибку назад, как делали до этого, пропорционально весовым коэффициентам соответствующих связей. И то обстоятельство, что на выходе имеется более одного нейрона, ничего не меняет.

Таким образом, ошибка **e1** распределяется пропорционально связям **w1,1** и **w1,2**. Точно так же ошибка **e2** распределяется пропорционально связям **w2,1** и **w2,2**.

Запишем в математической форме, как распределяются величины ошибки по связям. Например, ошибка **e1** будет поправлять веса **w1,1** и **w1,2**. При её распределении между этими связями, доля ошибки для w1,1 будет составлять:

$$\frac{w1,1}{w1,1 + w1,2}$$

Доля ошибки **e1** для обновления веса **w1,2**, будет распределяться аналогичным образом:

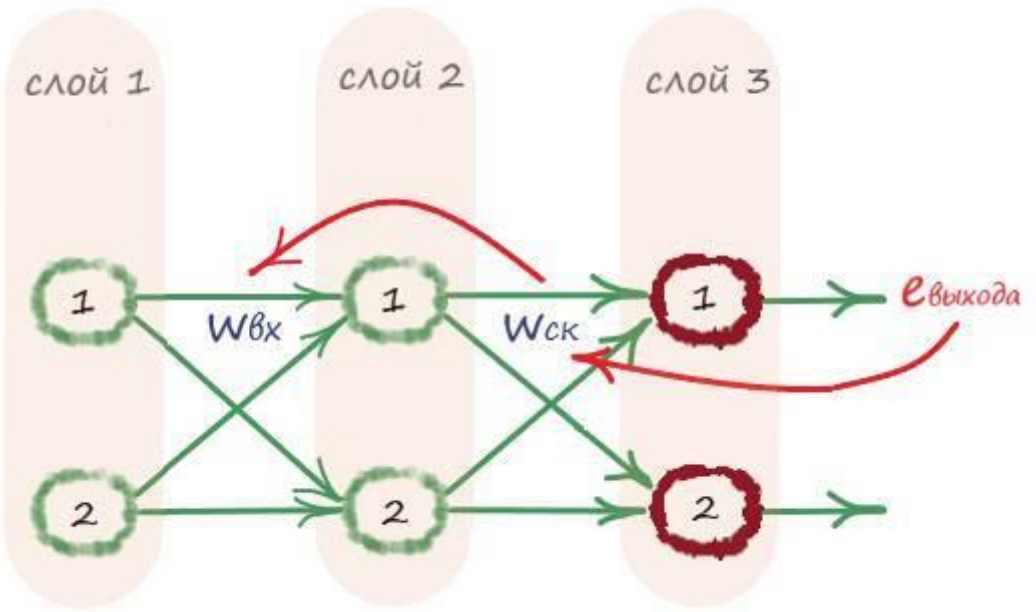
$$\frac{w1,2}{w1,1 + w1,2}$$

Эти выражения нам ещё раз говорят, что узлы, которые внесли больший вклад в ошибку, получают больший сигнал о ней, и наоборот, узлы, сделавшие меньший вклад, получают меньший сигнал.

Например, если **w1,1 = 1** и **w1,2 = 3**, то доля ошибки **e1** для **w1,1**, будет составлять: **1/1+3 = 1/4**, тогда как для другого веса **w1,2: 3/1+3 = 3/4**.

Распространение ошибки между слоями

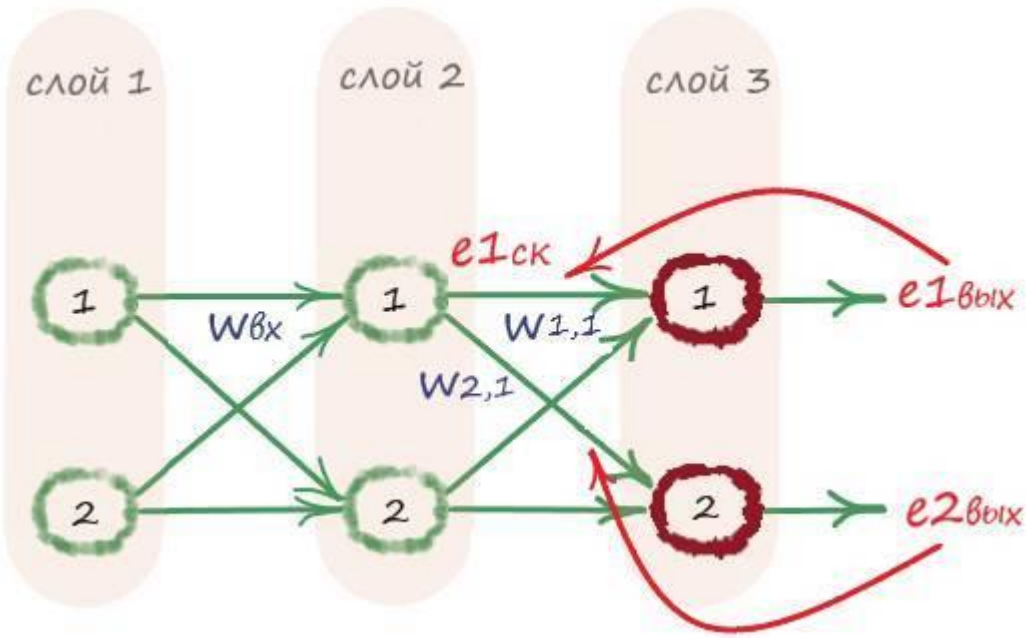
При наличии большего количества слоёв мы просто повторили описанные выше действия для каждого слоя, продвигаясь от выхода к входу:



Здесь **wск** – вес связи скрытого слоя, **wвх** – вес связи входного слоя. Еще раз можно увидеть почему этот процесс называется обратным распространением ошибки.

При распространении входных сигналов в прямом направлении, в нейронах скрытого слоя после взвешенной суммы всех сигналов и функции активации, образуется один выходной сигнал, который в последствии может умножаться с весами других связей. А как определить ошибку в нейронах скрытого слоя, при обратном её распространении?

Как определить ошибку на выходе мы знаем – разность целевых и выходных значений. А вот у скрытого слоя, нет целевых или желаемых выходных значений. Но если мы просуммируем все значения ошибок, по всем исходящим связям нейрона, то получим общее её значение. Этот процесс очень похож на прямое распространение сигнала, где в качестве сигнала используется ошибка:



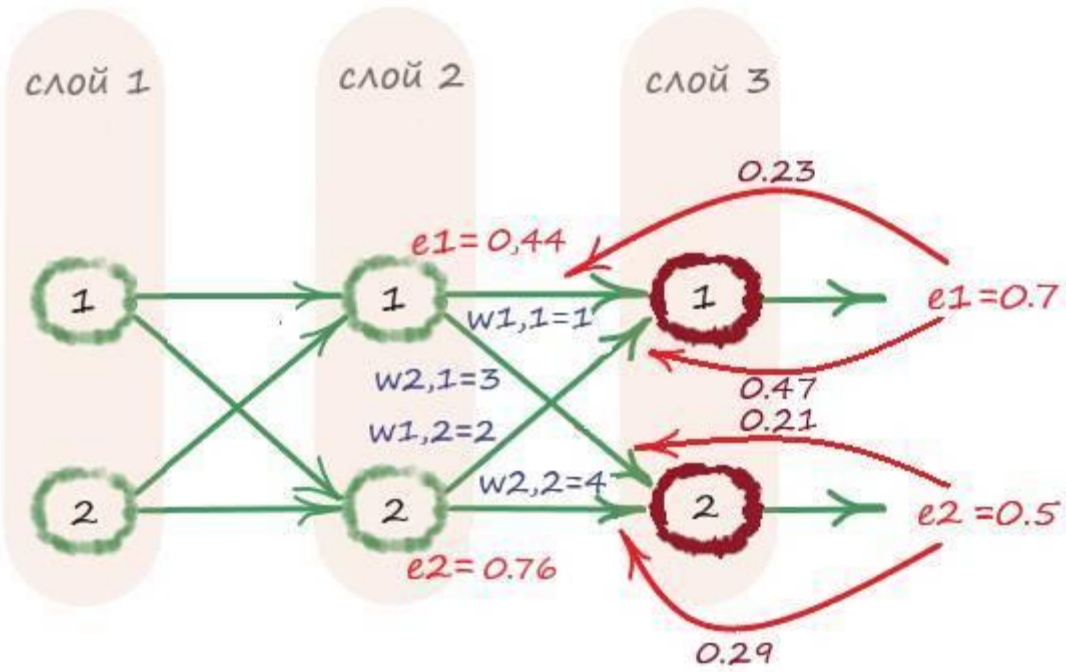
На иллюстрации видно, что имеется некоторая доля ошибки **e1вых**, приписываемая связи с весом **w1,1**, и есть доля ошибки **e2вых**, приписываемая связи **w2,1**.

Вышесказанное можно записать как:

e1ск = сумма ошибок связей w1,1 и w2,1

$$= e1_{вых} * \frac{w1,1}{w1,1 + w1,2} + e2_{вых} * \frac{w2,1}{w2,1 + w2,2}$$

Чтобы понять, как эта теория действует на практике, приведем иллюстрацию, обратного распространения ошибки в трехслойной сети, на примере конкретных значений:

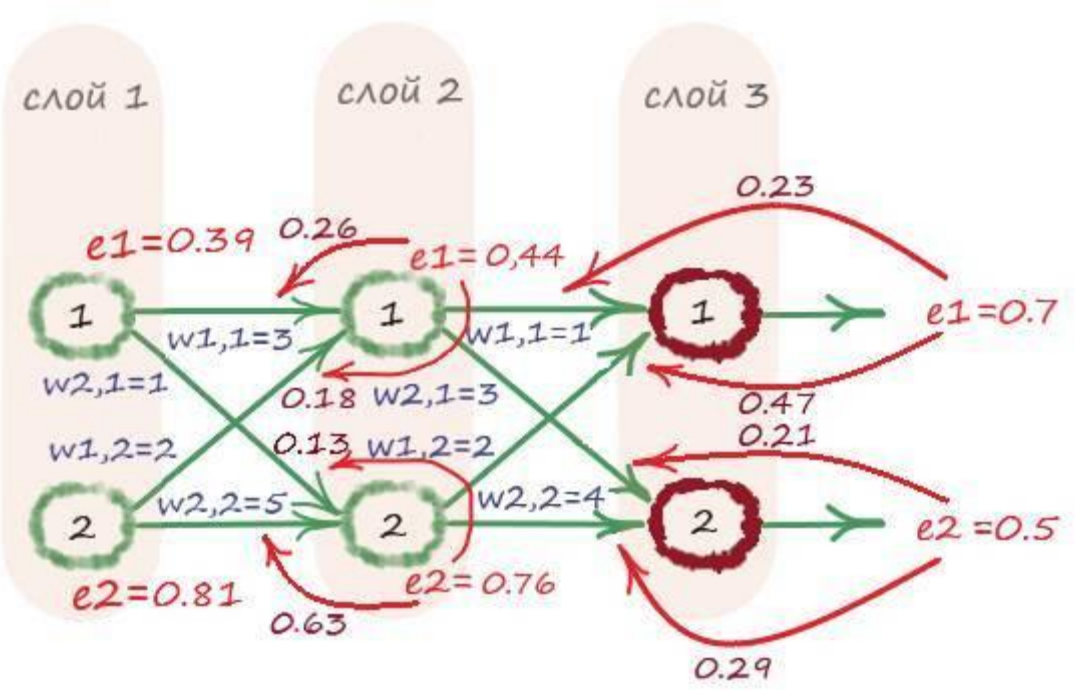


Проследим за обратным распространением одной из ошибок. Возьмём первый нейрон выходного слоя. Мы видим доли ошибки **e1** между его связями с весами **w1,1=1** и **w1,2=2**, равные 0,23 и 0,47 соответственно. Как это получать, мы уже знаем:

e1 * (w1,1/w1,1+ w1,2) = 0.7 * (1/1+2) = 0.23
e1 * (w1,2/w1,1+ w1,2) = 0.7 * (2/1+2) = 0.47

А объединённая ошибка в первом нейроне скрытого слоя, представляет собой сумму распределённых ошибок, в данном случае 0,23+0,21 = 0,44.

Распространение ошибки ближе к входному слою, производится аналогичным способом:



Применив данный способ, мы будем знать ошибку по всем нейронам и слоям, что даст возможность обновить веса всех связей, в нужной пропорции.

Метод обратного распространения ошибки в матричной форме

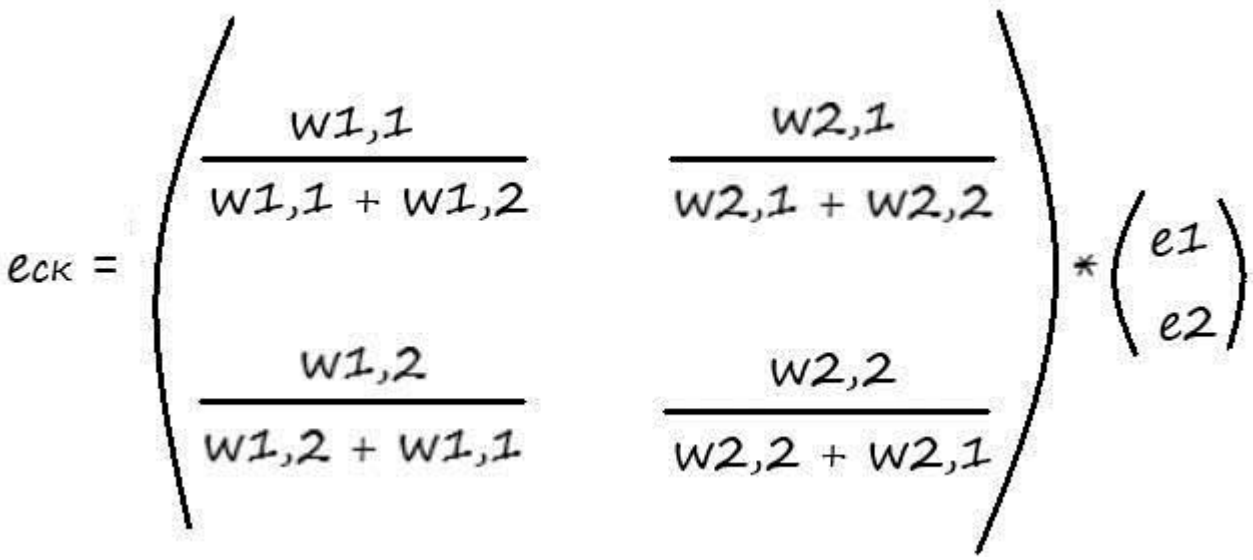
Мы можем значительно упростить расчеты, связанные с обратным распространением ошибки. Так же, как и в прямом распространении сигнала, нам помогут матрицы. Попробуем описать этот процесс.

Ошибку на выходе можно представить, как:

$$Евых = \begin{pmatrix} e1 \\ e2 \end{pmatrix}$$

Далее нам нужно построить матрицу ошибок скрытого слоя. Ошибка первого нейрона скрытого слоя, формируется от суммы двух сигналов. Этими сигналами являются: **e1 * (w1,1/w1,1+ w1,2)** и **e2 * (w2,1/w2,1+ w2,2)**. А ошибка второго нейрона скрытого слоя формируется от суммы **e1 * (w1,2/w1,2+ w1,1)**и **e2 * (w2,2/w2,2+ w2,1).**

Теперь можно эти действия записать как умножение матриц:

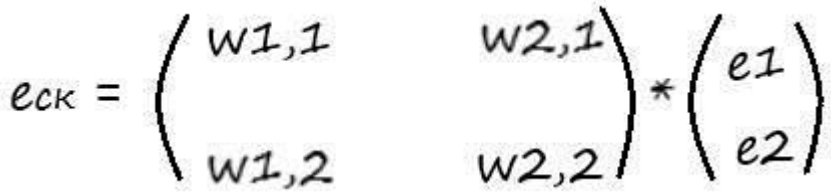


Мы можем еще больше упростить данный процесс, для ещё более эффективного вычисления.

Если нам удаться переписать это выражение, в том виде, как мы это делали с прямым распространением сигнала, только в виде сигналов будут выступать ошибки и идти будем в обратном направлении, мы получим большие преимущества.

Как это сделать? Если посмотреть на выражение, которое приведено выше, можно заметить, что наибольшую роль в величине распространения ошибки играет произведение выходных ошибок на связанные с ним веса **e*wij**. Чем больше будет вес, тем большая величина доли ошибки будет передаваться в скрытый слой. Поэтому, если пренебречь знаменателем, в элементах матрицы весов, то в целом так важная нам пропорциональность распространения ошибки от значений связей, будет сохранена, потеряем только лишь её масштабирование, что не так страшно. Таким образом, на примере первого элемента матрицы весов, выражение **e1 * (w1,1/w1,1+ w1,2)**, упроститься до **e1 * w1,1.**

В следствии чего, получим выражение:



Собственно, получилась матрица весов, которую мы строили ранее, но теперь она перевернута. Правый верхний элемент, стал левым нижним и наоборот. Если к этой матрице применить транспонирование, то все станет на свои места.

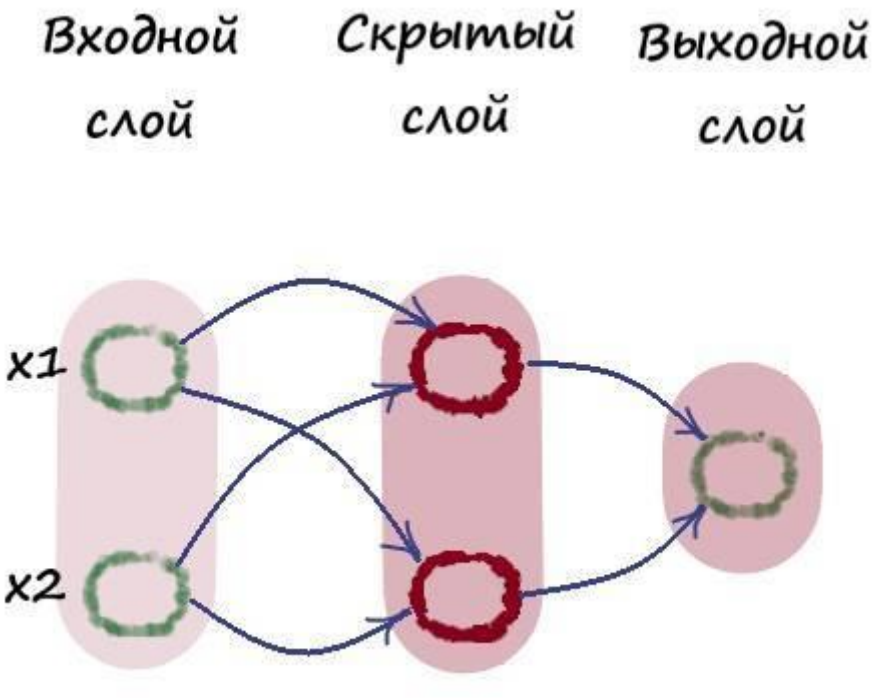
Вот теперь мы своего достигли. Теперь мы можем применять матричный подход к обратному распространению ошибки:

$$e_{ск} = w^{T}_{вых} * e_{вых}$$

Как оказалось, такая упрощенная модель, работает ничуть не хуже, той более сложной, где если бы мы оставили знаменатели в матрице весов. Главная деталь, а именно то, что учитываются весовые коэффициенты связей, даёт справедливо нужными порциями, распределить ответственность за ошибки.

Практика по решению логических функций с использованием логистических функций

Ну что же, пришло время убедиться в правильности принятых выше решений. А самое убедительное доказательство – это практика. Давайте для начала решим задачу “исключающее или”, реализовав в программе следующую структуру:



Подготовим тренировочные данные, они же будут и тестовыми, где первый элемент в строке будет являться целевым значением:

	A
1	0,0,0
2	1,1,0
3	1,0,1
4	0,1,1

Импортируем модуль по работе с массивами:

```
import numpy as np
```

Загружаем в переменную training_data_list наши тренировочные данные:

```
# Загрузить и подготовить тренировочные данные из формата CSV в список
training_data = open("dataset/Data_train.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data.close() # закрываем файл csv
```

Инициализируем параметры сети:

```
# Определение класса нейронной сети
class neuron_Net:

# Инициализация параметров нейронной сети
def __init__(self, input_num, neuron_num, output_num, learningrate):
# МАТРИЦА ВЕСОВ
# Задаем матрицу весов как случайное
self.weights = (np.random.rand(neuron_num, input_num) +0.0)
self.weights_out = (np.random.rand(output_num, neuron_num) +0.0)

# Задаем параметр скорости обучения
self.lr = learningrate

pass
```

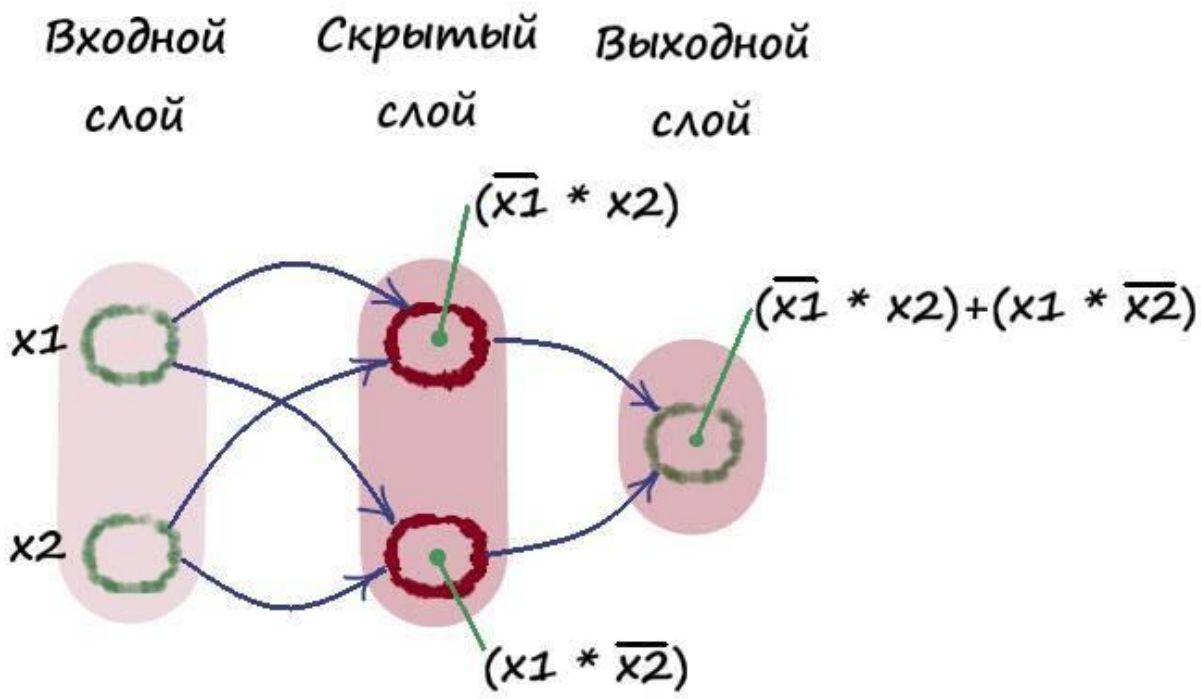
Создаем метод обучения сети:

```
# Метод обучения нейронной сети
def train(self, inputs_list, targets_list): # принимает (вх. список данных, ответы)
# Преобразовать список входов в вертикальный массив. .T – транспонирование
inputs_x = np.array(inputs_list, ndmin=2).T # матрица числа
targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов
```

```
# ВЫЧИСЛЕНИЕ СИГНАЛОВ
# Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
x1 = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs
# Вычислить сигналы, выходящие из нейронов скрытого слоя. Функция активации – сигмоида(x)
y1 = 1/(1+np.exp(-x1))
# Вычислить сигналы в нейронах выходного слоя. Взвешенная сумма.
x2 = np.dot(self.weights_out, y1) # dot – умножение матриц X = W*I = weights * inputs


# ВЫЧИСЛЕНИЕ ОШИБКИ
# Ошибка выходного слоя: E = -(цель – фактическое значение)
E = -(targets_Y – x2)
# Ошибка скрытого слоя
E_hidden = np.dot(self.weights_out.T, E)


# ОБНОВЛЕНИЕ ВЕСОВ
# Меняем веса связей, исходящих из скрытого слоя
self.weights_out -= self.lr * np.dot((E * x2), np.transpose(y1))
# Меняем веса связей, исходящих из входного слоя
self.weights -= self.lr * np.dot((E_hidden * y1 * (1.0 – y1)), np.transpose(inputs_x))


pass
```

Здесь в выражении: E_hidden = np.dot(self.weights_out.T, E) мы собственно и распространили ошибку от выходного слоя к скрытому!

Заметьте, что на выходном слое, мы используем линейную функцию, активационной функции здесь нет. Это сделано для того, о чём говорилось ранее, чтобы использовать только взвешенную сумму нейронов скрытого слоя:



Затем создаем метод прогона значений для тестирования сети, в прямом направлении:

```
# Метод прогона тестовых значений
def query(self, inputs_list): # Принимает свой набор тестовых данных
# Преобразовать список входов в вертикальный 2D массив.
inputs_x = np.array(inputs_list, ndmin=2).T


# Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
x1 = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs
# Вычислить сигналы, выходящие из нейронов скрытого слоя. Функция активации – сигмоида(x)
y1 = 1/(1+np.exp(-x1))
# Вычислить сигналы в нейронах выходного слоя. Взвешенная сумма.
x2 = np.dot(self.weights_out, y1) # dot – умножение матриц X = W*I = weights * inputs


return x2
```

Теперь зададим параметры нашей сети:

```
# ЗАДАЁМ ПАРАМЕТРЫ СЕТИ
# Количество входных данных, слоев, нейронов
data_input = 2
```

```
data_neuron = 2
data_output = 1


# Скорость обучения
learningrate  = 0.2


# Создать экземпляр нейронной сети
n = neuron_Net(data_input,  data_neuron,  data_output, learningrate)
```

Здесь ничего нового, два параметра на входе, два нейрона скрытого слоя, один нейрон в выходном слое.

Подготавливаем входные данные и целевые результаты для обучения сети. После чего передаем их в соответствующий метод:

```
# ОБУЧЕНИЕ
# Зададим количество эпох
epochs = 70000
# Прогон по обучающей выборке
for e in range(epochs):
for i in training_data_list:


# Получить входные данные числа
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])


# Получить целевое значение Y, (ответ – какое это число)
targets_Y = int(all_values[0])  # перевод символов в int,  0 элемент – ответ
#targets_Y = np.asfarray(all_values[0],int)


n.train(inputs_x,  targets_Y) # наш метод train – обучение нейронной сети
```

Выводим результаты:

```
# Вывод обученных весов
print('Весовые коэффициенты:\n',  n.weights)


# Прогоним входные данные из обучающей выборки через обученную сеть
for i in training_data_list:
```

all_values = i.split(',') # split(',') – разделить строку на символы где запятая "," символ разделения
#all_values = np.asfarray(all_values,int) # Перевод списка в int
inputs_x = np.asfarray(all_values[1:])
# Прогон по сети
outputs = n.query(inputs_x)
print(int(all_values[1]), 'XOR', int(all_values[2]), '=' , float(outputs), '\n')

Приведу еще раз полный текст программы:

```
import numpy as np
# Загрузить и подготовить тренировочные данные из формата CSV в список
training_data = open("dataset/Data_train.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data.close() # закрываем файл csv


# Определение класса нейронной сети
class neuron_Net:


# Инициализация параметров нейронной сети
def __init__(self, input_num, neuron_num, output_num, learningrate):
# МАТРИЦА ВЕСОВ
# Задаем матрицу весов как случайное
self.weights = (np.random.rand(neuron_num, input_num) +0.0)
self.weights_out = (np.random.rand(output_num, neuron_num) +0.0)


# Задаем параметр скорости обучения
self.lr = learningrate


pass


# Метод обучения нейронной сети
def train(self, inputs_list, targets_list): # принимает (вх. список данных, ответы)
# Преобразовать список входов в вертикальный массив. .T – транспонирование
inputs_x = np.array(inputs_list, ndmin=2).T # матрица числа
targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов


# ВЫЧИСЛЕНИЕ СИГНАЛОВ
```

```
    # Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
    x1 = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs
    # Вычислить сигналы, выходящие из нейронов скрытого слоя. Функция активации – сигмоида(x)
    y1 = 1/(1+np.exp(-x1))
    # Вычислить сигналы в нейронах выходного слоя. Взвешенная сумма.
    x2 = np.dot(self.weights_out, y1) # dot – умножение матриц X = W*I = weights * inputs



    # ВЫЧИСЛЕНИЕ ОШИБКИ
    # Ошибка выходного слоя: E = -(цель – фактическое значение)
    E = -(targets_Y – x2)
    # Ошибка скрытого слоя
    E_hidden = np.dot(self.weights_out.T, E)



    # ОБНОВЛЕНИЕ ВЕСОВ
    # Меняем веса связей, исходящих из скрытого слоя
    self.weights_out -= self.lr * np.dot((E * x2), np.transpose(y1))
    # Меняем веса связей, исходящих из входного слоя
    self.weights -= self.lr * np.dot((E_hidden * y1 * (1.0 – y1)), np.transpose(inputs_x))



    pass



    # Метод прогона тестовых значений
    def query(self, inputs_list): # Принимает свой набор тестовых данных
    # Преобразовать список входов в вертикальный 2D массив.
    inputs_x = np.array(inputs_list, ndmin=2).T



    # Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
    x1 = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs
    # Вычислить сигналы, выходящие из нейронов скрытого слоя. Функция активации – сигмоида(x)
    y1 = 1/(1+np.exp(-x1))
    # Вычислить сигналы в нейронах выходного слоя. Взвешенная сумма.
    x2 = np.dot(self.weights_out, y1) # dot – умножение матриц X = W*I = weights * inputs



    return x2



    # ЗАДАЁМ ПАРАМЕТРЫ СЕТИ
    # Количество входных данных, слоев, нейронов
    data_input = 2
    data_neuron = 2
```

```
data_output = 1


# Скорость обучения
learningrate  = 0.2


# Создать экземпляр нейронной сети
n = neuron_Net(data_input,  data_neuron,  data_output, learningrate)


# ОБУЧЕНИЕ
# Зададим количество эпох
epochs = 70000
# Прогон по обучающей выборке
for e in range(epochs):
for i in training_data_list:


# Получить входные данные числа
all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])


# Получить целевое значение Y, (ответ – какое это число)
targets_Y = int(all_values[0])  # перевод символов в int, 0 элемент – ответ
#targets_Y = np.asfarray(all_values[0],int)


n.train(inputs_x,  targets_Y) # наш метод train – обучение нейронной сети


# Вывод обученных весов
print('Весовые коэффициенты:\n',  n.weights)


# Прогоним входные данные из обучающей выборки через обученную сеть
for i in training_data_list:
all_values  =  i.split(',')  #  split(',')  –  разделить  строку  на  символы  где  запятая  ","  символ
разделения
#all_values  = np.asfarray(all_values,int)  # Перевод списка в int
inputs_x = np.asfarray(all_values[1:])
# Прогон по сети
outputs = n.query(inputs_x)
print(int(all_values[1]),  'XOR', int(all_values[2]),  '=' , float(outputs),  '\n')
```

Результаты работы программы:

Весовые коэффициенты:
[[7.81432203 7.82825685]
[2.89844863 2.90151462]]

0 XOR 0 = 0.02727560400468576

1 XOR 0 = 0.993182895138375

0 XOR 1 = 0.9905392833501487

1 XOR 1 = 0.10921815839185456

Как видим, более или менее мы справились с поставленной задачей. Если увеличить число нейронов скрытого слоя в два раза, доведя тем самым их значение до четырех, то мы резко увеличим точность нашей сети:

Весовые коэффициенты:
[[-2.33442417 -1.10601149]
[3.85660315 4.00658125]
[6.00766732 0.47282647]
[0.38238545 6.08253924]]

0 XOR 0 = 0.0058097515306574365

1 XOR 0 = 0.9999538649241559

0 XOR 1 = 0.9999292912654436

1 XOR 1 = 0.008841038108958976

Посмотрим, что будет, если в выходном слое будет два нейрона, каждый будет отвечать за свое число, один за ноль, а второй за единицу. Для этого потребуется внести не так много изменений в нашу программу. Главным образом они коснуться области определения количество нейронов и подготовки данных на вход и целевых значений:

```
# ЗАДАЁМ ПАРАМЕТРЫ СЕТИ
# Количество входных данных, нейронов
data_input = 2
data_neuron = 4
data_output = 2


# Скорость обучения
learningrate = 0.2


# Создать экземпляр нейронной сети
n = neuron_Net(data_input, data_neuron, data_output, learningrate)


# ОБУЧЕНИЕ
# Зададим количество эпох
epochs = 80000
# Прогон по обучающей выборке
for e in range(epochs):
for i in training_data_list:
# Получить входные данные числа
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])


# Получить целевое значение Y, (ответ – какое это число)
targets_Y = int(all_values[0]) # перевод символов в int, 0 элемент – ответ


# создать целевые выходные значения (все 0.01, кроме нужной метки, которая равна 0.99)
targets_Y = np.zeros(data_output) + 0.01


# Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
targets_Y[int(all_values[0])] = 0.99
```

n.train(inputs_x, targets_Y) # наш метод train – обучение нейронной сети

В остальном все то же самое, как в предыдущей программе. Поэтому приводить весь текст программы целиком, не имеет смысла.

Результат работы программы:

Весовые коэффициенты:
[[12.95040394 -6.49232526]
[-9.08125352 -8.94676728]
[-13.09373108 6.56168093]
[7.32763501 -14.51411564]]

0 XOR 0 = 0
[[0.97987662]
[0.02012364]]
1 XOR 0 = 1
[[0.0181055]
[0.98189458]]
0 XOR 1 = 1
[[0.01725169]
[0.98274808]]
1 XOR 1 = 0
[[0.985501]
[0.0144984]]

Ссылка на материалы программ:

https://github.com/CaniaCan/neuralmaster

Ещё раз убеждаемся в том, что это работает.

А для еще большей убедительности, решим, с помощью обратного распространения ошибки, задачу логического “ИЛИ”:

```
import numpy as np
# Загрузить и подготовить тренировочные данные из формата CSV в список
training_data = open("dataset/Data_or.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data.close() # закрываем файл csv
# Определение класса нейронной сети
class neuron_Net:
```

```
    # Инициализация весов нейронной сети
    def __init__(self, input_num, neuron_num, output_num, learningrate): #констр.(кол-во входов, кол-во нейронов)
    # МАТРИЦА ВЕСОВ
    # Задаем матрицу весов как случайное от -0,5 до 0,5
    self.weights = (np.random.rand(neuron_num, input_num) +0.0)
    self.weights_out = (np.random.rand(output_num, neuron_num) +0.0)



    # Задаем параметр скорости обучения
    self.lr = learningrate



    pass



    # Метод обучения нейронной сети
    def train(self, inputs_list, targets_list): # принимает (вх. список данных, ответы)
    # Преобразовать список входов в вертикальный массив. .T – транспонирование
    inputs_x = np.array(inputs_list, ndmin=2).T # матрица числа
    targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов: какое это число



    # ВЫЧИСЛЕНИЕ СИГНАЛОВ
    # Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
    x1 = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs
    # Вычислить сигналы, выходящие из нейронов скрытого слоя. Функция активации – сигмоида(x)
    y1 = 1/(1+np.exp(-x1))
    # Вычислить сигналы в нейронах выходного слоя. Взвешенная сумма.
    x2 = np.dot(self.weights_out, y1)



    # ВЫЧИСЛЕНИЕ ОШИБКИ
    # Ошибка E = -(цель – фактическое значение)
    E = -(targets_Y – x2)
    # Ошибка скрытого слоя
    E_hidden = np.dot(self.weights_out.T, E)



    # ОБНОВЛЕНИЕ ВЕСОВ
    # Меняем веса исходящих из скрытого слоя
    self.weights_out -= self.lr * np.dot((E * x2), np.transpose(y1))
    # Меняем веса исходящих из входного слоя
    self.weights -= self.lr * np.dot((E_hidden * y1 * (1.0 – y1)), np.transpose(inputs_x))
```

```
    pass



    # Метод прогона тестовых значений
    def query(self, inputs_list):  # Принимает свой набор тестовых данных
    # Преобразовать список входов в вертикальный 2D массив.
    inputs_x = np.array(inputs_list, ndmin=2).T



    # Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
    x1 = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs
    # Вычислить сигналы, выходящие из нейронов скрытого слоя. Функция активации – сигмоида(x)
    y1 = 1/(1+np.exp(-x1))
    # Вычислить сигналы в нейронах выходного слоя. Взвешенная сумма.
    x2 = np.dot(self.weights_out, y1)



    return x2
    # ЗАДАЁМ ПАРАМЕТРЫ СЕТИ
    # Количество входных данных, нейронов
    data_input = 2
    data_neuron = 2
    data_output = 1



    # Скорость обучения
    learningrate = 0.2



    # Создать экземпляр нейронной сети
    n = neuron_Net(data_input, data_neuron, data_output, learningrate)
    # ОБУЧЕНИЕ
    # Зададим количество эпох
    epochs = 70000
    # Прогон по обучающей выборке
    for e in range(epochs):
    for i in training_data_list:
    # Получить входные данные числа
    all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    inputs_x = np.asfarray(all_values[1:])



    # Получить целевое значение Y, (ответ – какое это число)
    targets_Y = int(all_values[0]) # перевод символов в int, 0 элемент – ответ
```

```
# Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
targets_Y = np.asfarray(all_values[0],int)


n.train(inputs_x, targets_Y) # наш метод train – обучение нейронной сети
# Вывод обученных весов
print('Весовые коэффициенты:\n', n.weights)
# Прогоним входные данные из обучающей выборки через обученную сеть
for i in training_data_list:
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
inputs_x = np.asfarray(all_values[1:])
# Прогон по сети
outputs = n.query(inputs_x)
print(int(all_values[1]), 'OR', int(all_values[2]), '=', float(outputs), '\n')
```

Результаты работы программы:

```
Весовые коэффициенты:
[[ 2.95719849  3.05322881]
[ 0.1741739  0.19161757]]


0 OR 0 = 0.0015221568573919875
1 OR 0 = 0.9999989042032791
0 OR 1 = 0.9999985457530554
1 OR 1 = 1.000000835153846
```

Окончательно убеждаемся в правильно выбранном варианте решения задач типа “исключающего или”, путем добавления ещё одного слоя в сеть.

Работа скрытого слоя

Мы поняли, что однослойная сеть не способна решать некоторый круг задач. Для их решения нам понадобилось ввести ещё один слой, наделив наши нейроны эволюционной способностью, связывать свои выходы со входами других нейронов. Теперь искусственные нейроны, пусть и в упрощенной форме, по своему функционалу соответствуют своим биологическим прототипам.

Мы поняли, что скрытые слои дают вариантность ответов на выходе. Внеся в них нелинейные функции, в нашем примере – сигмоиду, мы добились нелинейности ответов на выходе.

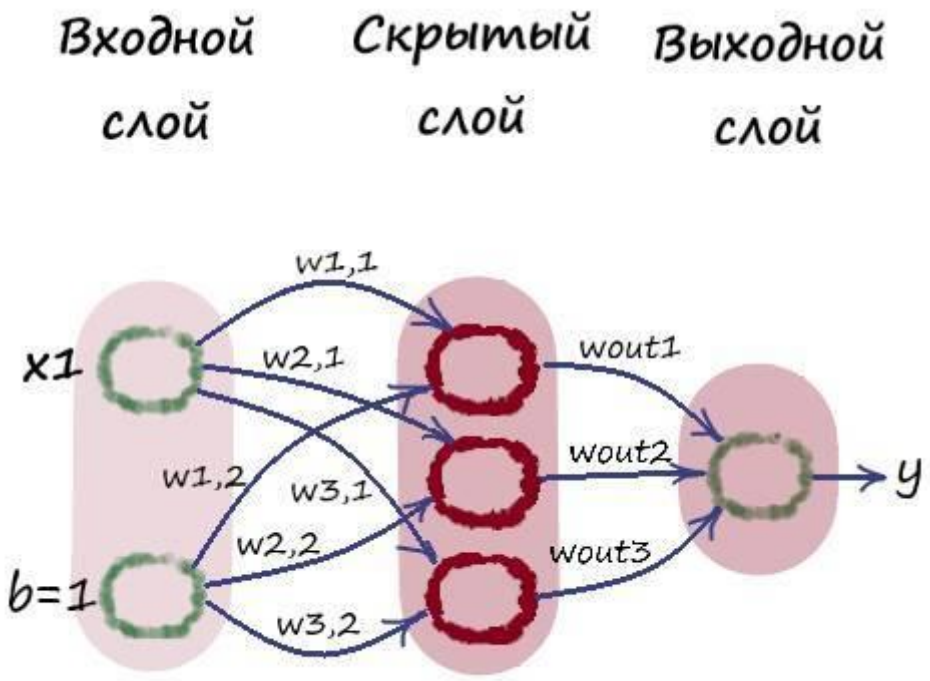
Но что это значит? Как наглядно понять какие процессы происходят в скрытых слоях и почему они приносят в сеть такие возможности? На что влияет количество нейронов в этих слоях?

Самым наглядным примером для ответов на эти вопросы, будет решения задачи аппроксимации математических функций с помощью нейронных сетей. Иными словами, получать на выходе из сети такие значения, которые очень близко соответствовали значению непрерывной математической функции.

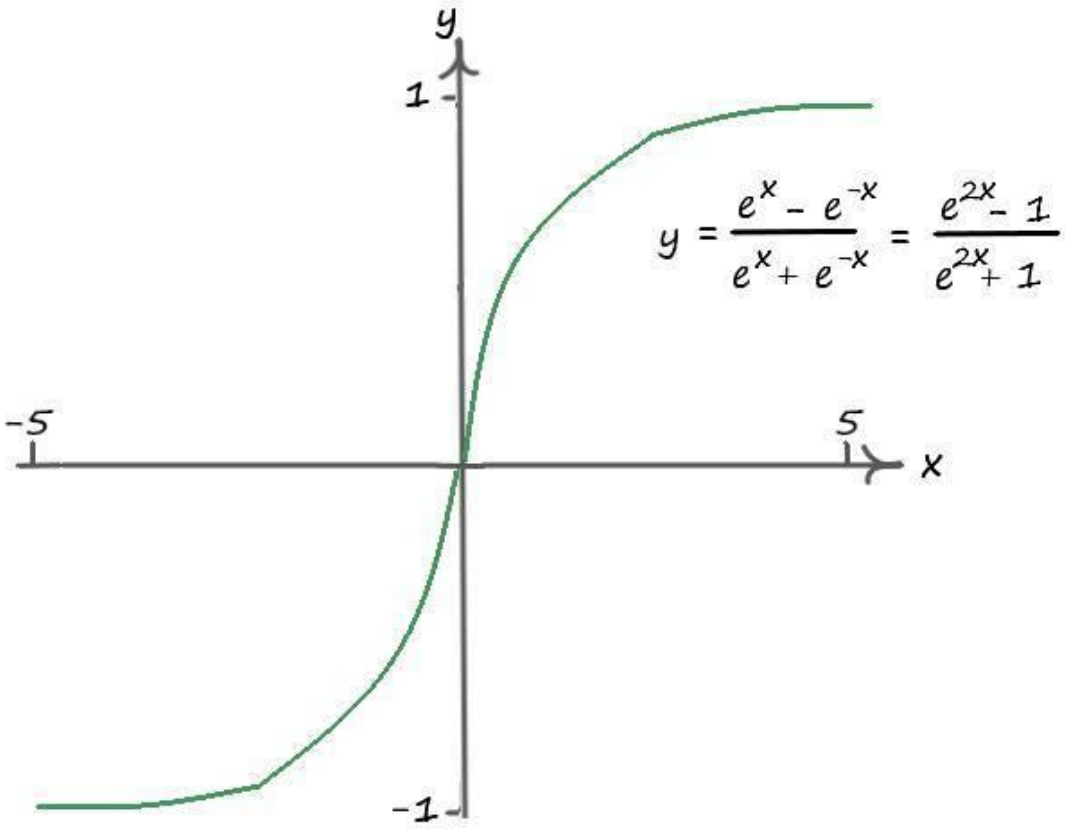
Согласно теореме аппроксимации – нейронная сеть с одним скрытым слоем может аппроксимировать любую непрерывную функцию многих переменных с любой точностью.

Реализуем сеть, которая будет аппроксимировать функцию синуса. Как известно синус – функция непрерывная.

Сеть будет состоять, из двух нейронов на входе, один из которых будет отвечать за смещение функции активации в скрытом слое (параметр b), трех нейронов скрытого слоя и одного нейрона на выходе. Функция активации скрытого слоя – гиперболический тангенс, а выходной слой не имеет функций активаций:



Гиперболический тангенс во многом напоминает логистическую функцию, но диапазон значений у него немного расширен, а именно от 1 до -1:



В данном примере эта функция приведена для ознакомления с ней. Вполне можно было, почти с одинаковым успехом, воспользоваться обычной логистической функцией.

Производная гиперболического тангенса – не очень сложна. Она есть в табличных производных:

$$td'x = \frac{1}{\cos^2 x}$$

Так как:

$$\frac{1}{\cos^2 x} = \frac{\sin^2 x + \cos^2 x}{\cos^2 x} = td^2 x + 1$$

Следовательно, величина обновления:

$$\frac{dE}{dw} = \frac{dE}{dOk} * \frac{dOk}{dwij} = E * (1 - td^2x) * Ok^T$$

С теорией всё, приступим к практике. Для начала загрузим известные нам модули для работы с массивами и графиками функций:

```
import numpy as np
import math
import matplotlib.pyplot as plt
%matplotlib inline
```

Загрузим модули подсчета времени и для работы с выводом данных прогресса обучения на консоль:

```
from time import time, sleep #Для замера времени выполнения обучения
from tqdm import tqdm #Для вывода прогресса обучения
from tqdm import tqdm_notebook #Для вывода прогресса обучения. Только для python notebook
```

Если данные модули не подключаются, их необходимо установить через Anaconda Promt, используя команду conda install tqdm.

Создаем данные синусоиды и выводим их на координаты:

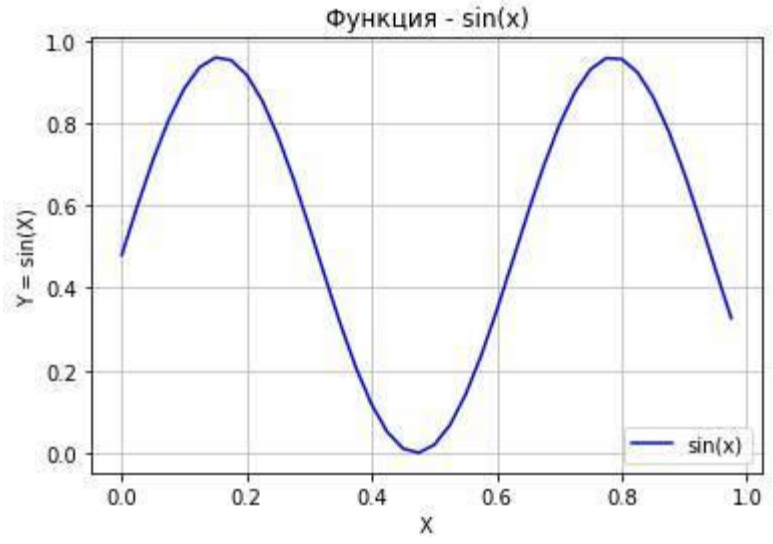
```
                #СОЗДАЕМ ИСХОДНЫЕ ДАННЫЕ
# Зададим имена графику и числовым координатам
plt.title("Функция – sin(x)")
plt.xlabel("X")
plt.ylabel("Y = sin(X)")
```

```
X_sin = []
Y_sin = []
x = 0
while x < 1:
Y_sin += [ (0.48*(math.sin(x*10))+0.48) ] # Создание целевых данных
X_sin += [x] # Создание входных данных
fx = open ('dataset/Data_sin_x.csv', 'w') # Создаем или открываем для записи. w записываем в фаел
fy = open ('dataset/Data_sin_y.csv', 'w') # Создаем или открываем для записи. w записываем в фаел
fx.write (str(X_sin)) # Читаем входные данные в массив
fy.write (str(Y_sin)) # Читаем целевые данные в массив
```

```
x += 0.025 # Шаг данных


fx.close()
fy.close()
#Создаем массивы данных для вывода
X_sin2 = np.zeros(len(X_sin))
Y_sin2 = np.zeros(len(Y_sin))
X_sin2 = np.asfarray(X_sin)
Y_sin2 = np.asfarray(Y_sin)


#Вывод исходной синусоиды
plt.plot(X_sin, Y_sin, color = 'blue', linestyle = 'solid',
label = 'sin(x)')
# локация имени функции
plt.legend(loc=4) #loc – локация имени, 4 – справа внизу
# Сетка на фоне для улучшения восприятия
plt.grid(True, linestyle='-', color='0.75')
# Показать график
plt.show()
```

Как вы уже поняли в папке с данными создаются два файла с входными и целевыми значениями соответственно. Ограничим, для наглядности, значением 0,48, амплитуду и частоту синусоиды.

Результат данного сета:



Теперь загрузим эти данные в переменные:

```
# ЗАГРУЖАЕМ ДАННЫЕ И ЗАПИСЫВАЕМ В МАССИВЫ
```

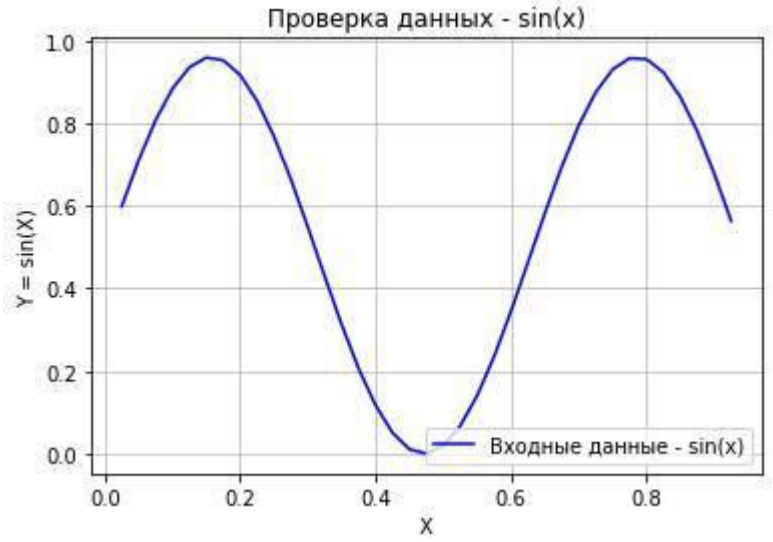
```
# Загрузить и подготовить тренировочные данные из формата CSV в список
training_data = open("dataset/Data_sin_x.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data.close() # закрываем файл csv



# Загрузить и подготовить целевые данные из формата CSV в список
target_data = open("dataset/Data_sin_y.csv", 'r') # 'r' – открываем файл для чтения
target_data_list = target_data.readlines() # readlines() – читает все строки в файле в переменную training_data_list
target_data.close() # закрываем файл csv



for i in training_data_list:
# Получить входные данные числа
all_values = i.split(',') # Разбиваем на символы
ty = len(all_values)-2 # Переменная размера данных. -2 чтоб избежать оштбок
inputs_ = np.asfarray(all_values[1:ty]) # Массив входных данных



for i in target_data_list:
# Получить целевые данные числа
all_values_t = i.split(',') # Разбиваем на символы
targets_ = np.asfarray(all_values_t[1:ty]) # Массив целевых данных
```

На всякий случай проверим загруженные данные:

```
        # ПРОВЕРЯЕМ ВХОДНЫЕ ДАННЫЕ
# Значения по X входных данных
x_data = inputs_
print(len(x_data)) # Размер входных данных
# Значения по Y входных данных
y_data = targets_
print(len(y_data)) # Размер целевых данных



# Зададим имена графику и числовым координатам
plt.title("Проверка данных – sin(x)")
plt.xlabel("X")
plt.ylabel("Y = sin(X)")



# Начальная прямая
plt.plot(x_data, y_data, 'b', label = 'Входные данные – sin(x)')
plt.legend(loc=4) #loc – локация имени, 4 – справа внизу
```

```
# Сетка на фоне для улучшения восприятия
plt.grid(True, linestyle='-', color='0.75')
# Показать график
plt.show()
```

Результат работы сета:



Создаем класс и методы:

```
# Определение класса нейронной сети
class neuron_Net:



# Инициализация весов нейронной сети
def __init__(self, input_num, neuron_num, output_num, learningrate): #констр.(кол-во входов, кол-во нейронов)
# МАТРИЦА ВЕСОВ
# Задаем матрицу весов как случайное от -0,5 до 0,5
self.weights = np.random.normal(+0.0, pow(input_num, -0.5), (neuron_num, input_num))
self.weights_out = np.random.normal(+0.0, pow(neuron_num, -0.5), (output_num, neuron_num))
self.weights_out_bias = 0.01



# Задаем параметр скорости обучения
self.lr = learningrate



pass



# Метод обучения нейронной сети
def train(self, inputs_list, targets_list): # принимает (вх. список данных, ответы)
```

```
    # Преобразовать список входов в вертикальный массив. .T – транспонирование
    inputs_x = np.array(inputs_list, ndmin=2).T # матрица числа
    targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов: какое это число



    # ВЫЧИСЛЕНИЕ СИГНАЛОВ
    # Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
    x1 = np.dot(self.weights, inputs_x)# dot – умножение матриц X = W*I = weights * inputs
    # Вычислить сигналы, выходящие из нейрона
    #y1 = 1/(1+np.exp(-x1)) #Сигмоида
    #y1 = np.maximum(x1, 0) #RELU
    y1 = (np.exp(2*x1)-1)/(np.exp(2*x1)+1) #Тангенс
    # Вычислить сигналы в нейронах выходного слоя. Взвешенная сумма.
    x2 = np.dot(self.weights_out, y1)



    # ВЫЧИСЛЕНИЕ ОШИБКИ
    # Ошибка E = -(цель – фактическое значение)
    E = -(targets_Y – x2)
    # Скрытая ошибка слоя-это output_errors, разделенные на весы, рекомбинированные на скрытых узлах
    E_hidden = np.dot(self.weights_out.T, E)



    # ОБНОВЛЕНИЕ ВЕСОВ
    # Меняем веса по каждой связи
    self.weights_out -= self.lr * np.dot((E * x2), np.transpose(y1))
    # Меняем веса по каждой связи
    #self.weights -= self.lr * np.dot((E_hidden * y1 * (1.0 – y1)), np.transpose(inputs_x)) #Сигмоида
    #self.weights -= self.lr * np.dot((E_hidden * (y1 > 0)), np.transpose(inputs_x)) #RELU
    self.weights -= self.lr * np.dot((E_hidden * (1.0 – np.power(y1, 2))), np.transpose(inputs_x)) #Тангенс
    pass



    # Метод прогона тестовых значений
    def query(self, inputs_list):
    # Преобразовать список входов в вертикальный 2D массив.
    inputs_x = np.array(inputs_list, ndmin=2).T



    #Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
    x1 = np.dot(self.weights, inputs_x)# dot – умножение матриц X = W*I = weights * inputs
    # Вычислить сигналы, выходящие из нейрона
    #y1 = 1/(1+np.exp(-x1)) #Сигмоида
    #y1 = np.maximum(x1, 0) #RELU
    y1 = (np.exp(2*x1)-1)/(np.exp(2*x1)+1) #Тангенс
    # Вычислить сигналы в нейронах выходного слоя. Взвешенная сумма.
    x2 = np.dot(self.weights_out, y1)
    return x2
```

```
# Метод возвращает сигнал – wouti * tanh(wi * x + bi)
def querynum2(self, inputs_list, numnet): # Принимает входные данные и номер комбинации
# Преобразовать список входов в вертикальный 2D массив.
inputs_x = np.array(inputs_list, ndmin=2).T



#Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
x1 = np.dot(self.weights, inputs_x)
# Вычислить сигналы, выходящие из нейрона
#y1 = 1/(1+np.exp(-x1)) #Сигмоида
#y1 = np.maximum(x1, 0) #RELU
y1 = (np.exp(2*x1)-1)/(np.exp(2*x1)+1) #Тангенс
y12 = np.dot(self.weights_out[0,numnet], y1[numnet])
return y12
```

Метод querynum2(), будет возвращать одно их линейных выражений гиперболического тангенса из взвешенной суммы.

Например, выходное значение представляет собой сумму линейных выражений гиперболических тангенсов f(x) = wout0 * tanh(w0 * x + b0) +… + wout2* tanh(w2 * x + b2). Передавая в аргументах номер одной из линейной комбинации, вернем его значение, если это будет 0, то вернем комбинацию wout0 * tanh(w0 * x + b0).

Задаем параметры и обучаем нашу сеть:

```
        # ЗАДАЁМ ПАРАМЕТРЫ СЕТИ
# Количество входных данных, нейронов
data_input = 2
data_neuron = 3
data_output = 1



# Скорость обучения
learningrate = 0.01



# Создать экземпляр нейронной сети
n = neuron_Net(data_input, data_neuron, data_output, learningrate)



# ОБУЧЕНИЕ
# Зададим количество эпох
epochs = 50000
start = time()
# Прогон по обучающей выборке
#for e in range(epochs):
for e in tqdm(range(epochs)):
```

```
for i in range(len(x_data)):



# Получить входные данные числа
inputs_x = x_data[i]
# Добавляем второй вход bias = 1
inputs_x = np.append(inputs_x, 1)
targets_Y = y_data[i] # перевод символов в int, 0 элемент – ответ
#round(x, 1) #Округление числа



n.train(inputs_x, targets_Y) # наш метод train – обучение нейронной сети



time_out = time() – start
print("Время выполнения: ", time_out, " сек" )
```

Выводим данные:

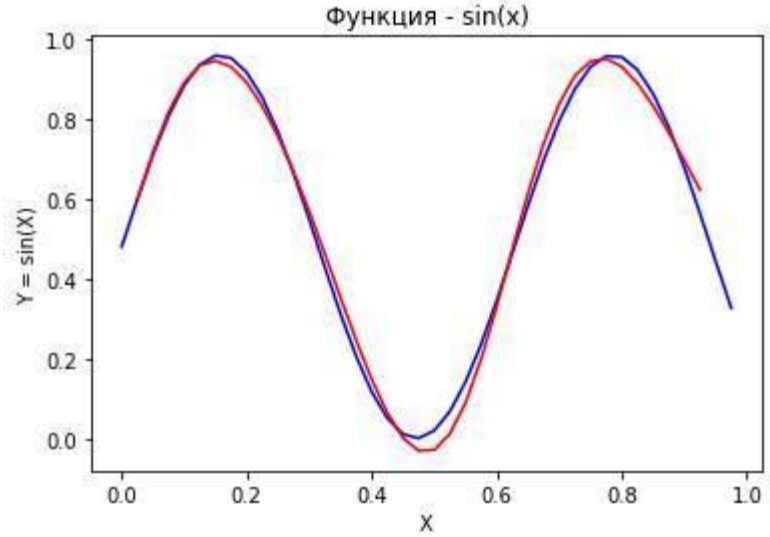
```
# Вывод обученных весов
print('Весовые коэффициенты:\n', n.weights)
print('Весовые коэффициенты от скрытого слоя:\n', n.weights_out)



# Создание значений на выходе сети
outputs_ = np.array([])
for i in range(len(x_data)):
inputs_x = x_data[i]
inputs_x = np.append(inputs_x, 1) # Еще раз создаем массив входных данных
# Прогон по сети
outputs_ = np.append(outputs_, n.query(inputs_x)) # Еще раз создаем массив выходных данных
обученной сети
#outputs = n.query(inputs_x)



# Создание значений на выходе нейрона срытого слоя
outputs_num0 = np.array([])
outputs_num1 = np.array([])
outputs_num2 = np.array([])
#outputs_num3 = np.array([])
#outputs_num4 = np.array([])
for i in range(len(x_data)):
# Получить входные данные числа
inputs_num = x_data[i]
inputs_num = np.append(inputs_num, 1)
outputs_num0 = np.append(outputs_num0, n.querynum2(inputs_num, 0))
```

```
outputs_num1 = np.append(outputs_num1, n.querynum2(inputs_num, 1))
outputs_num2 = np.append(outputs_num2, n.querynum2(inputs_num, 2))
#outputs_num3 = np.append(outputs_num3, n.querynum2(inputs_num, 3))
#outputs_num4 = np.append(outputs_num4, n.querynum2(inputs_num, 4))



# Еще раз выводем синусоиду
X_sin = []
Y_sin = []
x = 0.0
while x < 1:
Y_sin += [ (0.48*(math.sin(x*10))+0.48) ]
X_sin += [x]
x += 0.025



X_sin2 = np.zeros(len(X_sin))
Y_sin2 = np.zeros(len(Y_sin))
X_sin2 = np.asfarray(X_sin)
Y_sin2 = np.asfarray(Y_sin)



plt.plot(X_sin, Y_sin, color = 'b', linestyle = 'solid',
label = 'Входные данные – sin(x)')



# Зададим имена графику и числовым координатам
plt.title("Функция – sin(x)")
plt.xlabel("X")
plt.ylabel("Y = sin(X)")



# Обученная сеть
plt.plot(x_data, outputs_, color = 'red', label = 'Обученная сеть – sin(x)')
```

Можно не использовать tqdm для вывода на консоль статус бара. Для этого пользуйтесь обычным циклом – for e in range(epochs).

Мы каждый раз создаем массивы данных чтобы было удобно пользоваться своими сетами, можно было этого не делать и использовать уже полученные значения.

Результатом работы нашей сети будут данные красного цвета:



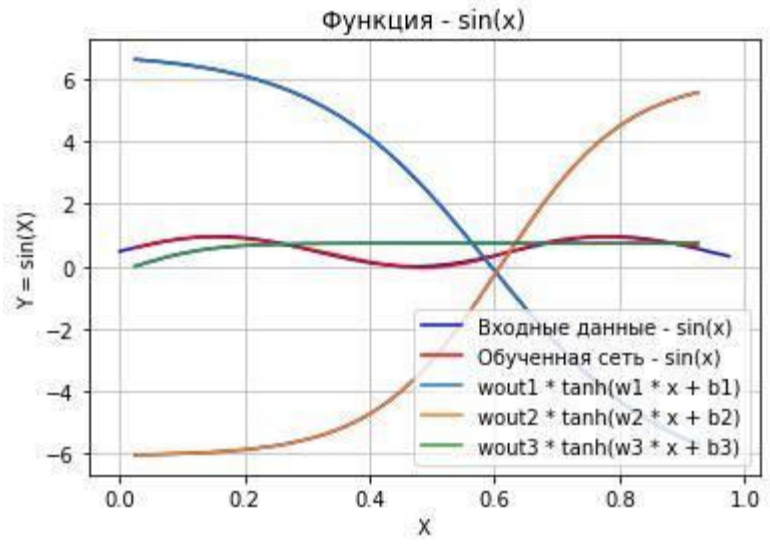
Мы видим, что значения на выходе сети почти полностью повторяют заданную нами синусоиду. Результаты можно ещё улучшить, добавляя количество нейронов в скрытом слое и экспериментируя со скоростью обучения, но для нас и такого результата будет достаточно.

Далее выводим на плоскость еще и результаты линейных комбинаций гиперболических тангенсов на выходе сети:

```
# Выводим целевую синусоиду
X_sin = []
Y_sin = []
x = 0.0
while  x < 1:
Y_sin  += [ (0.48*(math.sin(x*10))+0.48)  ]
X_sin  += [x]
x += 0.025



X_sin2  = np.zeros(len(X_sin))
Y_sin2  = np.zeros(len(Y_sin))
X_sin2  = np.asfarray(X_sin)
Y_sin2  = np.asfarray(Y_sin)



plt.plot(X_sin,  Y_sin, color = 'b', linestyle  = 'solid',
label  = 'Входные данные – sin(x)')
# Зададим имена графику и числовым координатам
plt.title("Функция  – sin(x)")
plt.xlabel("X")
plt.ylabel("Y  = sin(X)")



# Обученная сеть
plt.plot(x_data,  outputs_, color = 'red', label  = 'Обученная сеть – sin(x)')
```

```
# Выход сети по отдельным связям – wouti * tanh(wi * x + bi)
plt.plot(x_data, outputs_num0, color = 'b', linestyle = 'solid')
plt.plot(x_data,outputs_num0, label='wout1 * tanh(w1 * x + b1)') # label=имя функции


plt.plot(x_data, outputs_num1, color = 'b', linestyle = 'solid')
plt.plot(x_data,outputs_num1, label='wout2 * tanh(w2 * x + b2)')


plt.plot(x_data, outputs_num2, color = 'b', linestyle = 'solid')
plt.plot(x_data,outputs_num2, label='wout3 * tanh(w3 * x + b3)')


#plt.plot(x_data, outputs_num3, color = 'b', linestyle = 'solid')
#plt.plot(x_data,outputs_num3, color = 'g',label='wout4 * tanh(w4 * x + b4)')


#plt.plot(x_data, outputs_num4, color = 'b', linestyle = 'solid')
#plt.plot(x_data,outputs_num4, color = 'c', label='wout5 * tanh(w5 * x + b5)')
plt.legend(loc=4) #loc – локация имени, 4 – справа внизу


# Сетка на фоне для улучшения восприятия
plt.grid(True, linestyle='-', color='0.75')
# Показать график
plt.show()
```

Результат работы сета:



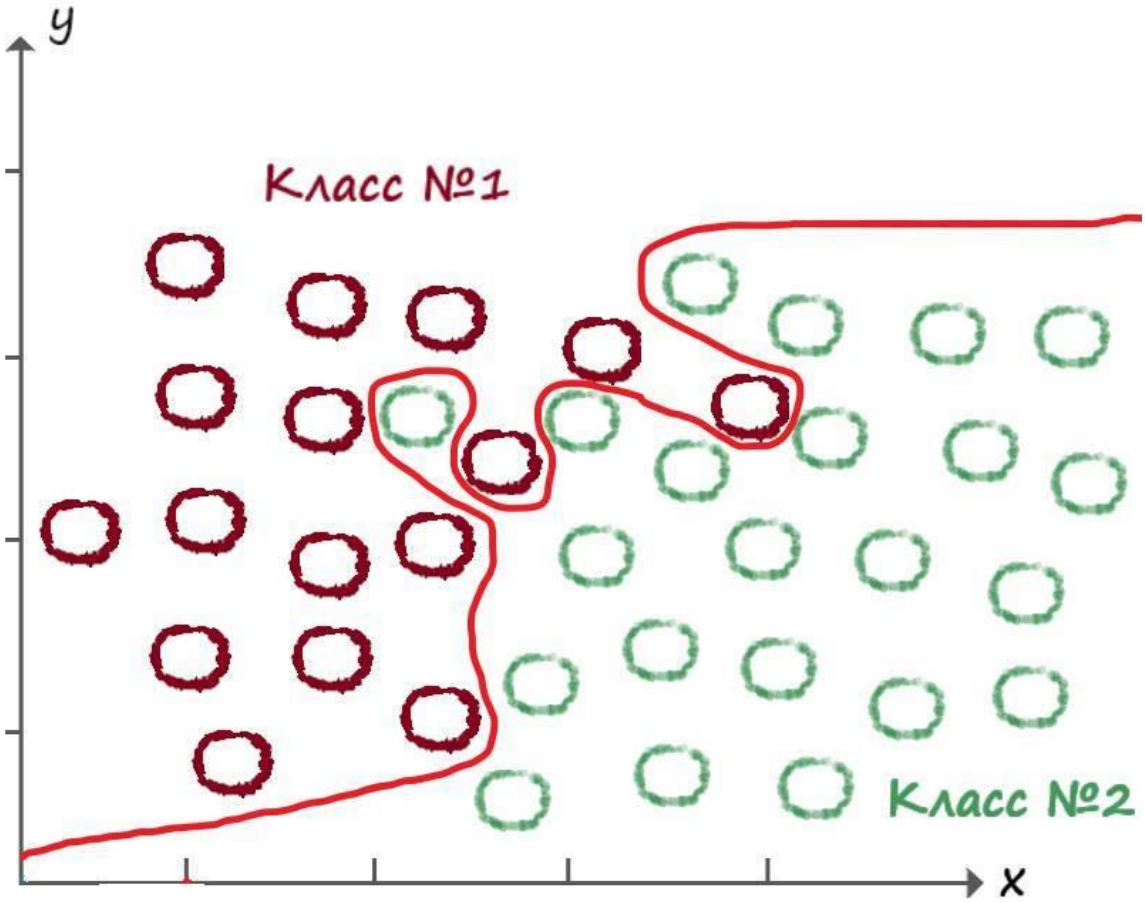
На данной диаграмме отчетливо видно, как работает скрытый слой. Пусть на вход поступило значение 0,2. Взвешенная сумма будет представлять собой сумму всех трех значений с выхода нейронов скрытого слоя:

f(x) = wout0 * tanh(w0 * x + b0) + wout1* tanh(w1 * x + b1) + wout2* tanh(w2 * x + b2) = 6,0 – 5,8 + 0,7 = 0,9

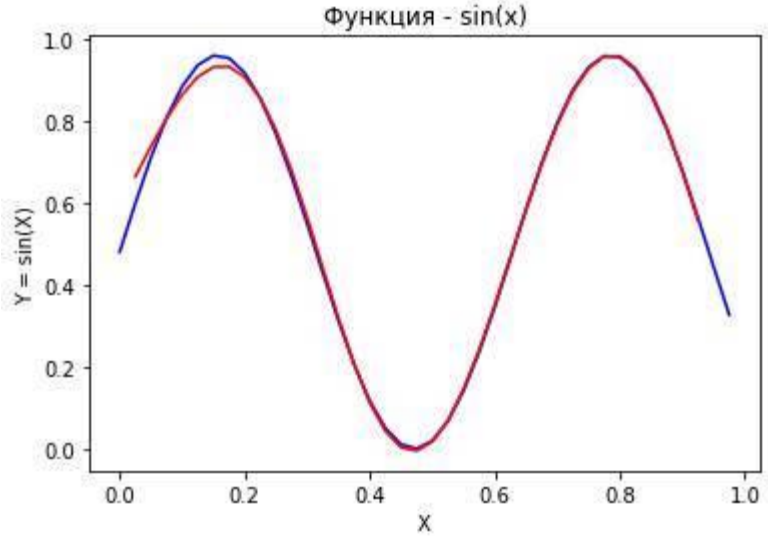
Проекции на ось ординат даны с приближениями.

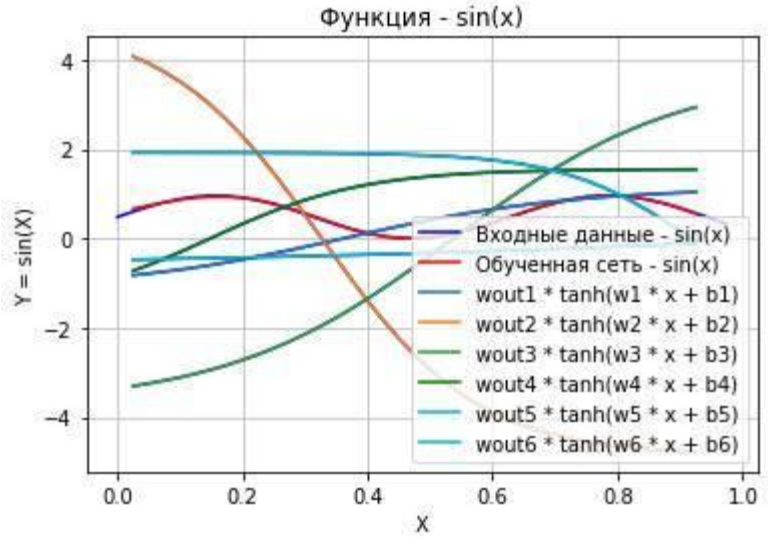
Так же отчетливо видна работа параметра **b**, который задает смещение функций относительно ординаты (по горизонтали).

Говоря иными словами, скрытый слой с нелинейной функцией активации, вносит нелинейную вариантность ответов на выходе. Если проиллюстрировать это на примере обычной классификации на два вида, то график мог выглядеть примерно таким образом:



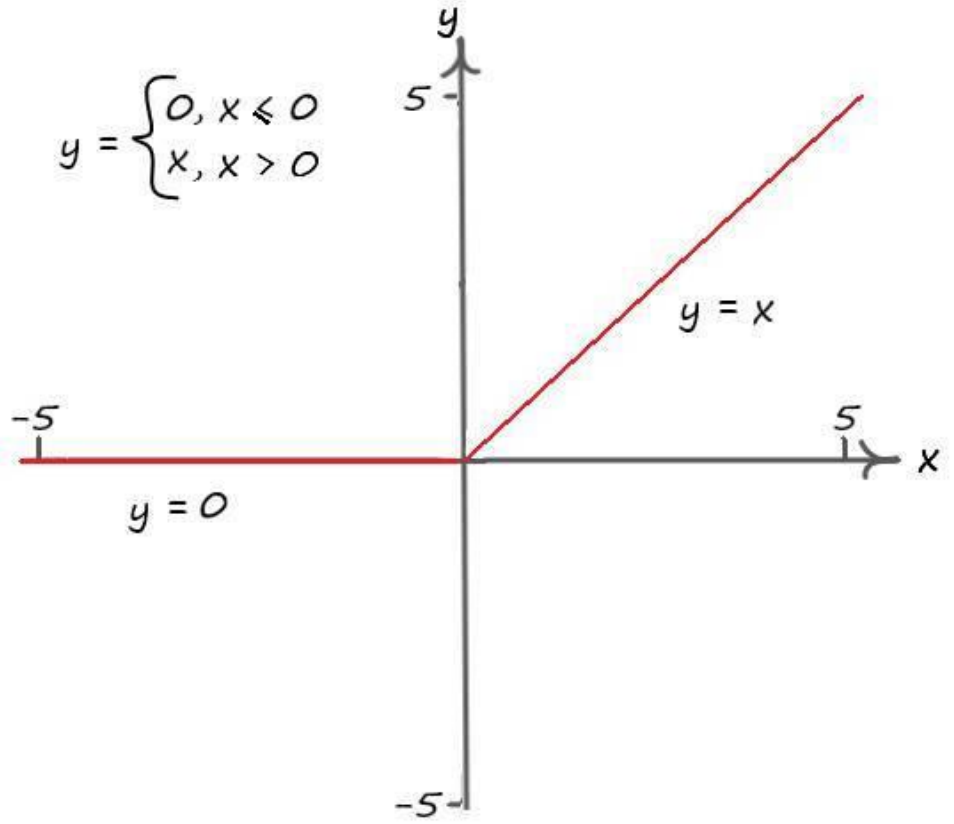
А как скажется увеличение нейронов скрытого слоя, на выходных значениях сети. Ну тут интуитивно понятно, что значения будут ещё более сглаженными. Убедимся в этом, увеличив вдвое количество нейронов в скрытом слое:





Функция стала более сглаженной, но этого не очень заметно на данной диаграмме.

Для того чтобы более наглядней рассмотреть, как влияет количество нейронов скрытого слоя, на выходные значения, нужно применить в нем другую функцию активации – RELU “выпрямитель”:



Ну а производная такой функции совсем элементарная. Производная линейной функции, проходящей через начало координат – это константа (**a**) помноженная на значение (**x**). Следовательно, получается:

y = ax, y′ = ax′ = a, так как **a = x** и **y = x, при x>0**, то **y′ = y > 0**

Реализуем сеть с семью нейронами скрытого слоя и с функцией активации RELU:

```
import numpy as np
import math
import matplotlib.pyplot as plt
%matplotlib inline
```

```
from time import time, sleep #Для замера времени выполнения обучения
from tqdm import tqdm #Для вывода прогресса обучения
from tqdm import tqdm_notebook #Для вывода прогресса обучения. Только для python notebook
```

```
#СОЗДАЕМ ИСХОДНЫЕ ДАННЫЕ
# Зададим имена графику и числовым координатам
plt.title("Функция – sin(x)")
plt.xlabel("X")
plt.ylabel("Y = sin(X)")
```

```
X_sin = []
Y_sin = []
x = 0
while x < 1:
Y_sin += [ (0.48*(math.sin(x*10))+0.48) ] # Создание целевых данных
X_sin += [x] # Создание входных данных
fx = open ('dataset/Data_sin_x.csv', 'w') # Создаем или открываем для записи. w записываем в фаел
fy = open ('dataset/Data_sin_y.csv', 'w') # Создаем или открываем для записи. w записываем в фаел
fx.write (str(X_sin)) # Читаем входные данные в массив
fy.write (str(Y_sin)) # Читаем целевые данные в массив
x += 0.025 # Шаг данных



fx.close()
fy.close()
#Создаем массивы данных для вывода
X_sin2 = np.zeros(len(X_sin))
Y_sin2 = np.zeros(len(Y_sin))
X_sin2 = np.asfarray(X_sin)
Y_sin2 = np.asfarray(Y_sin)



#Вывод исходной синусоиды
plt.plot(X_sin, Y_sin, color = 'blue', linestyle = 'solid',
label = 'sin(x)')
# локация имени функции
plt.legend(loc=4) #loc – локация имени, 4 – справа внизу
# Сетка на фоне для улучшения восприятия
plt.grid(True, linestyle='-', color='0.75')
# Показать график
plt.show()



# ЗАГРУЖАЕМ ДАННЫЕ И ЗАПИСЫВАЕМ В МАССИВЫ
# Загрузить и подготовить тренировочные данные из формата CSV в список
training_data = open("dataset/Data_sin_x.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data.close() # закрываем файл csv



# Загрузить и подготовить целевые данные из формата CSV в список
target_data = open("dataset/Data_sin_y.csv", 'r') # 'r' – открываем файл для чтения
target_data_list = target_data.readlines() # readlines() – читает все строки в файле в переменную training_data_list
target_data.close() # закрываем файл csv
```

```
for i in training_data_list:
# Получить входные данные числа
all_values = i.split(',') # Разбиваем на символы
ty = len(all_values)-2 # Переменная размера данных. -2 чтоб избежать оштбок
inputs_ = np.asfarray(all_values[1:ty]) # Массив входных данных



for i in target_data_list:
# Получить целевые данные числа
all_values_t = i.split(',') # Разбиваем на символы
targets_ = np.asfarray(all_values_t[1:ty]) # Массив целевых данных



# ПРОВЕРЯЕМ ВХОДНЫЕ ДАННЫЕ
# Значения по X входных данных
x_data = inputs_
print(len(x_data)) # Размер входных данных
# Значения по Y входных данных
y_data = targets_
print(len(y_data)) # Размер целевых данных



# Зададим имена графику и числовым координатам
plt.title("Проверка данных – sin(x)")
plt.xlabel("X")
plt.ylabel("Y = sin(X)")



# Начальная прямая
plt.plot(x_data, y_data, 'b', label = 'Входные данные – sin(x)')
plt.legend(loc=4) #loc – локация имени, 4 – справа внизу



# Сетка на фоне для улучшения восприятия
plt.grid(True, linestyle='-', color='0.75')
# Показать график
plt.show()



# Определение класса нейронной сети
class neuron_Net:



# Инициализация весов нейронной сети
def __init__(self, input_num, neuron_num, output_num, learningrate): #констр.(кол-во входов,
кол-во нейронов)
# МАТРИЦА ВЕСОВ
# Задаем матрицу весов как случайное от -0,5 до 0,5
```

```
    self.weights  = np.random.normal(+0.0,  pow(input_num,  -0.5), (neuron_num,  input_num))
    self.weights_out  = np.random.normal(+0.0,  pow(neuron_num,  -0.5), (output_num, neuron_num))



    # Задаем параметр скорости обучения
    self.lr  = learningrate



    pass



    # Метод обучения нейронной сети
    def train(self,  inputs_list,  targets_list):  # принимает (вх. список данных, ответы)
    # Преобразовать список входов в вертикальный массив. .T – транспонирование
    inputs_x = np.array(inputs_list,  ndmin=2).T  # матрица числа
    targets_Y = np.array(targets_list,  ndmin=2).T  # матрица ответов: какое это число



    # ВЫЧИСЛЕНИЕ  СИГНАЛОВ
    # Вычислить сигналы в нейронах скрытого слоя. Взвешенная  сумма.
    x1 = np.dot(self.weights,  inputs_x)# dot – умножение матриц X = W*I = weights  * inputs
    # Вычислить сигналы, выходящие из нейрона
    #y1 = 1/(1+np.exp(-x1))  #Сигмоида
    y1 = np.maximum(x1,  0) #RELU
    #y1 = (np.exp(2*x1)-1)/(np.exp(2*x1)+1)  #Тангенс
    # Вычислить сигналы в нейронах выходного слоя. Взвешенная  сумма.
    x2 = np.dot(self.weights_out,  y1)



    # ВЫЧИСЛЕНИЕ  ОШИБКИ
    # Ошибка E = -(цель – фактическое значение)
    E = -(targets_Y  – x2)
    # Скрытая ошибка слоя-это output_errors, разделенные на весы, рекомбинированные на скрытых узлах
    E_hidden = np.dot(self.weights_out.T,  E)



    # ОБНОВЛЕНИЕ  ВЕСОВ
    # Меняем веса по каждой связи
    self.weights_out  -= self.lr  * np.dot((E * x2), np.transpose(y1))
    # Меняем веса по каждой связи
    #self.weights  -= self.lr  * np.dot((E_hidden * y1 * (1.0 – y1)), np.transpose(inputs_x)) #Сигмоида
    self.weights  -= self.lr  * np.dot((E_hidden  * (y1 > 0)), np.transpose(inputs_x)) #RELU
    #self.weights  -= self.lr  *  np.dot((E_hidden  *  (1.0  –  np.power(y1,  2))),  np.transpose(inputs_x)) #Тангенс
    pass
```

```
# Метод прогона тестовых значений
def query(self, inputs_list): # Принимает свой набор тестовых данных
# Преобразовать список входов в вертикальный 2D массив.
inputs_x = np.array(inputs_list, ndmin=2).T


#Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
x1 = np.dot(self.weights, inputs_x)# dot – умножение матриц X = W*I = weights * inputs
# Вычислить сигналы, выходящие из нейрона
#y1 = 1/(1+np.exp(-x1)) #Сигмоида
y1 = np.maximum(x1, 0) #RELU
#y1 = (np.exp(2*x1)-1)/(np.exp(2*x1)+1) #Тангенс
# Вычислить сигналы в нейронах выходного слоя. Взвешенная сумма.
x2 = np.dot(self.weights_out, y1)
return x2


# Метод возвращает сигнал – wouti * tanh(wi * x + bi)
def querynum2(self, inputs_list, numnet): # Принимает входные данные и номер комбинации
# Преобразовать список входов в вертикальный 2D массив.
inputs_x = np.array(inputs_list, ndmin=2).T


#Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
x1 = np.dot(self.weights, inputs_x)
# Вычислить сигналы, выходящие из нейрона
#y1 = 1/(1+np.exp(-x1)) #Сигмоида
y1 = np.maximum(x1, 0) #RELU
#y1 = (np.exp(2*x1)-1)/(np.exp(2*x1)+1) #Тангенс
y12 = np.dot(self.weights_out[0,numnet], y1[numnet])
return y12


# ЗАДАЁМ ПАРАМЕТРЫ СЕТИ
# Количество входных данных, нейронов
data_input = 2
data_neuron = 7
data_output = 1


# Скорость обучения
learningrate = 0.01


# Создать экземпляр нейронной сети
n = neuron_Net(data_input, data_neuron, data_output, learningra
```

```
    # ОБУЧЕНИЕ
    # Зададим количество эпох
    epochs = 60000
    start = time()
    # Прогон по обучающей выборке
    #for e in range(epochs):
    for e in tqdm(range(epochs)):
    for i in range(len(x_data)):



    # Получить входные данные числа
    inputs_x = x_data[i]
    # Добавляем второй вход bias = 1
    inputs_x = np.append(inputs_x, 1)
    targets_Y = y_data[i] # перевод символов в int, 0 элемент – ответ
    #round(x, 1) #Округление числа



    n.train(inputs_x, targets_Y) # наш метод train – обучение нейронной сети



    time_out = time() – start
    print("Время выполнения: ", time_out, " сек" )



    # Вывод обученных весов
    print('Весовые коэффициенты:\n', n.weights)
    print('Весовые коэффициенты от скрытого слоя:\n', n.weights_out)



    # Создание значений на выходе сети
    outputs_ = np.array([])
    for i in range(len(x_data)):
    inputs_x = x_data[i]
    inputs_x = np.append(inputs_x, 1) # Еще раз создаем массив входных данных
    # Прогон по сети
    outputs_ = np.append(outputs_, n.query(inputs_x)) # Еще раз создаем массив выходных данных
обученной сети
    #outputs = n.query(inputs_x)



    # Создание значений на выходе нейрона срытого слоя
    outputs_num0 = np.array([])
    outputs_num1 = np.array([])
    outputs_num2 = np.array([])
    outputs_num3 = np.array([])
    outputs_num4 = np.array([])
    outputs_num5 = np.array([])
    outputs_num6 = np.array([])
```

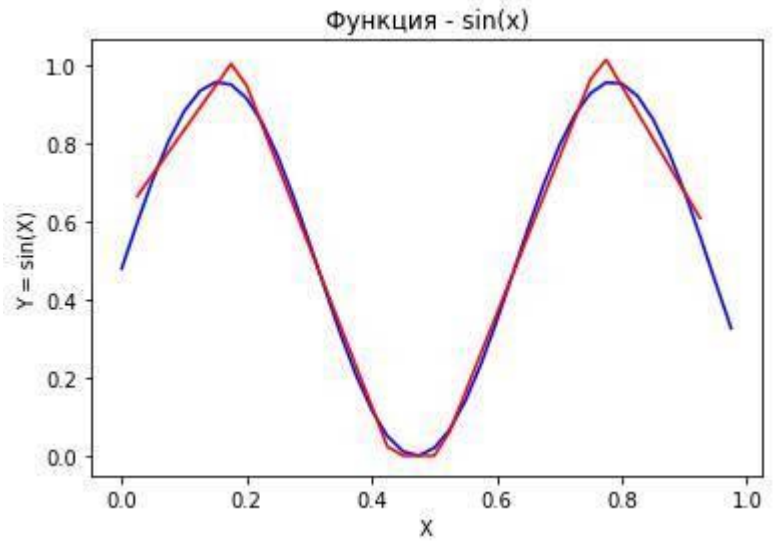
```
for i in range(len(x_data)):
# Получить входные данные числа
inputs_num = x_data[i]
inputs_num = np.append(inputs_num, 1)
outputs_num0 = np.append(outputs_num0, n.querynum2(inputs_num, 0))
outputs_num1 = np.append(outputs_num1, n.querynum2(inputs_num, 1))
outputs_num2 = np.append(outputs_num2, n.querynum2(inputs_num, 2))
outputs_num3 = np.append(outputs_num3, n.querynum2(inputs_num, 3))
outputs_num4 = np.append(outputs_num4, n.querynum2(inputs_num, 4))
outputs_num5 = np.append(outputs_num5, n.querynum2(inputs_num, 5))
outputs_num6 = np.append(outputs_num6, n.querynum2(inputs_num, 6))



# Еще раз выводем синусоиду
X_sin = []
Y_sin = []
x = 0.0
while x < 1:
Y_sin += [ (0.48*(math.sin(x*10))+0.48) ]
X_sin += [x]
x += 0.025



X_sin2 = np.zeros(len(X_sin))
Y_sin2 = np.zeros(len(Y_sin))
X_sin2 = np.asfarray(X_sin)
Y_sin2 = np.asfarray(Y_sin)



plt.plot(X_sin, Y_sin, color = 'b', linestyle = 'solid',
label = 'Входные данные – sin(x)')



# Зададим имена графику и числовым координатам
plt.title("Функция – sin(x)")
plt.xlabel("X")
plt.ylabel("Y = sin(X)")



# Обученная сеть
plt.plot(x_data, outputs_, color = 'red', label = 'Обученная сеть – sin(x)')



# Выводим целевую синусоиду
X_sin = []
Y_sin = []
x = 0.0
while x < 1:
Y_sin += [ (0.48*(math.sin(x*10))+0.48) ]
```

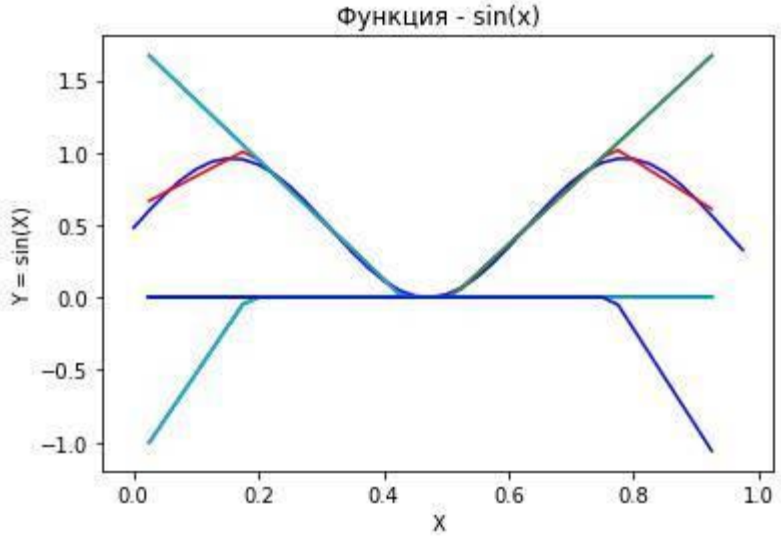
```
X_sin += [x]
x += 0.025



X_sin2 = np.zeros(len(X_sin))
Y_sin2 = np.zeros(len(Y_sin))
X_sin2 = np.asfarray(X_sin)
Y_sin2 = np.asfarray(Y_sin)



plt.plot(X_sin, Y_sin, color = 'b', linestyle = 'solid',
label = 'Входные данные – sin(x)')
# Зададим имена графику и числовым координатам
plt.title("Функция – sin(x)")
plt.xlabel("X")
plt.ylabel("Y = sin(X)")



# Обученная сеть
plt.plot(x_data, outputs_, color = 'red', label = 'Обученная сеть – sin(x)')



# Выход сети по отдельным связям – wouti * tanh(wi * x + bi)
plt.plot(x_data, outputs_num0, color = 'b', linestyle = 'solid')
plt.plot(x_data,outputs_num0, label='wout1 * tanh(w1 * x + b1)') # label=имя функции



plt.plot(x_data, outputs_num1, color = 'b', linestyle = 'solid')
plt.plot(x_data,outputs_num1, label='wout2 * tanh(w2 * x + b2)')



plt.plot(x_data, outputs_num2, color = 'b', linestyle = 'solid')
plt.plot(x_data,outputs_num2, label='wout3 * tanh(w3 * x + b3)')



plt.plot(x_data, outputs_num3, color = 'b', linestyle = 'solid')
plt.plot(x_data,outputs_num3, color = 'g',label='wout4 * tanh(w4 * x + b4)')



plt.plot(x_data, outputs_num4, color = 'b', linestyle = 'solid')
plt.plot(x_data,outputs_num4, color = 'c', label='wout5 * tanh(w5 * x + b5)')



plt.plot(x_data, outputs_num5, color = 'b', linestyle = 'solid')
plt.plot(x_data,outputs_num5, color = 'c', label='wout6 * tanh(w6 * x + b6)')
```

```
plt.plot(x_data, outputs_num6, color = 'b', linestyle = 'solid')
plt.plot(x_data,outputs_num7, color = 'c', label='wout7 * tanh(w7 * x + b7)')
plt.legend(loc=4) #loc – локация имени, 4 – справа внизу


# Сетка на фоне для улучшения восприятия
plt.grid(True, linestyle='-', color='0.75')
# Показать график
plt.show()
```

Я сразу привел полный текст программы, поскольку изменения по сравнению с предыдущей минимальны. Отмечу лишь функцию np.maximum(x1, 0), которая возвращает элемент равный аргументу x1, при значениях x1>0, а остальные значения равны нулю. Собственно, эта функция и реализует RELU. А условие (y1 > 0), при обновлении весов, не что иное как производная функции RELU.

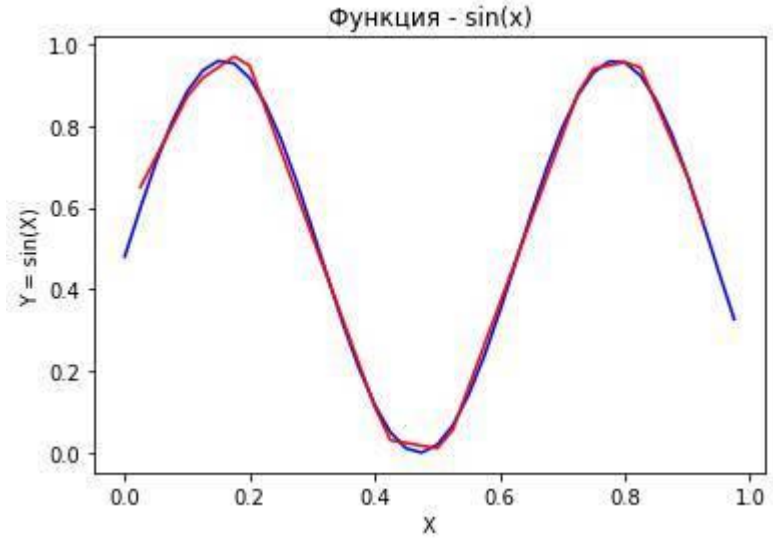
Результат работы программы:





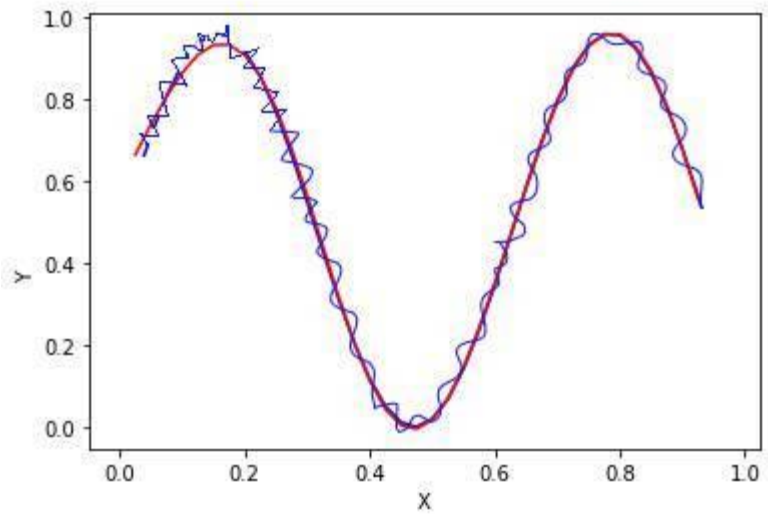
А вот на этих диаграммах очень хорошо видна не сглаженность выходных значений, относительно эталонных.

Если увеличить число нейронов в скрытом слое сразу в десять раз, тем самым увеличив их значения до семидесяти штук, то график будет выглядеть следующим образом:



На этой диаграмме мы уже можем наблюдать, большее количество ломанных линий выходных значений, тем самым делая их более сглаженными.

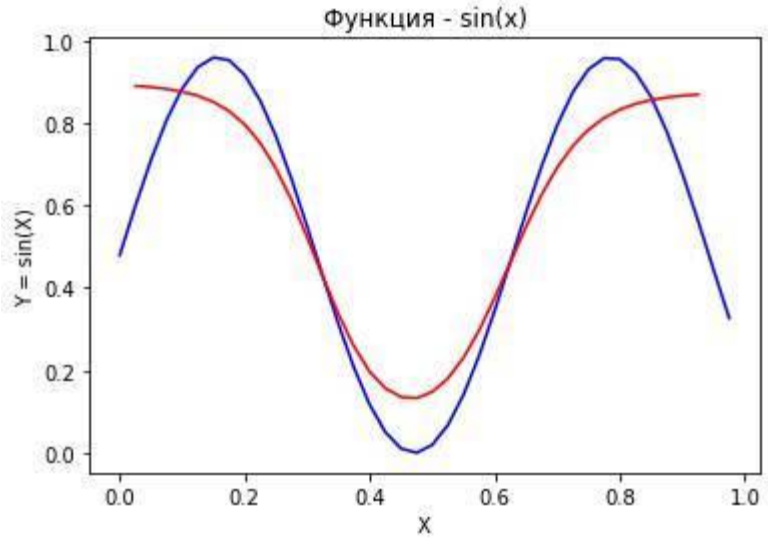
На этом фоне, стоить отметить еще одно свойство аппроксимации математических функций нейронной сетью – устранение помех:

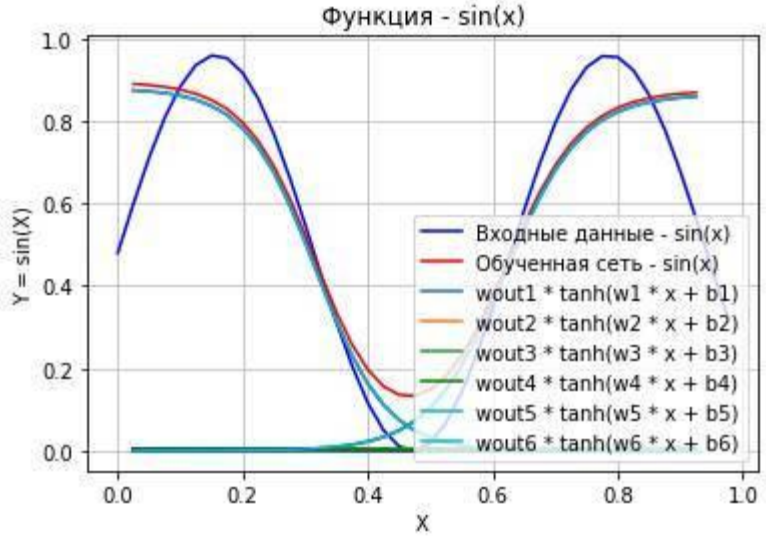


Как и ранее при линейной классификации, наши значения на выходе стараются занять такие значения, которые по возможности удовлетворяли бы всем данным, тем самым занимая некоторое среднее положение.

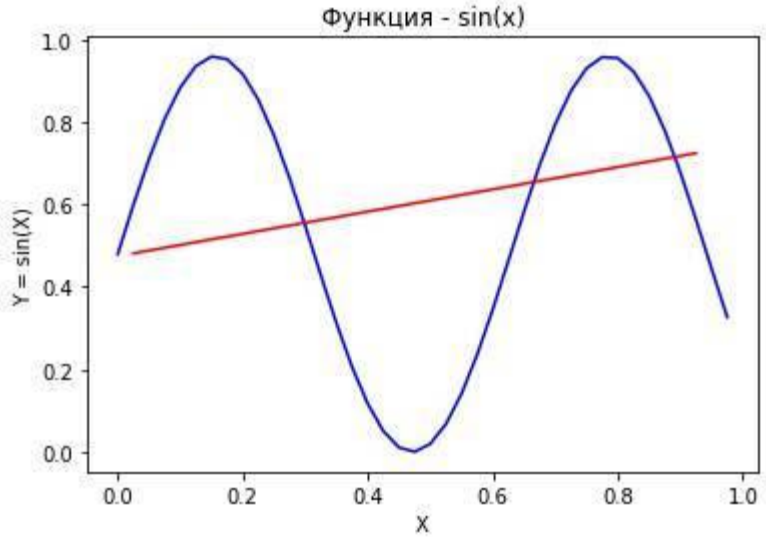
Давайте по очереди опробуем сеть с шестью нейронами скрытого слоя, с сигмоидальной функцией активации и обычной линейной функцией (без функции активации).

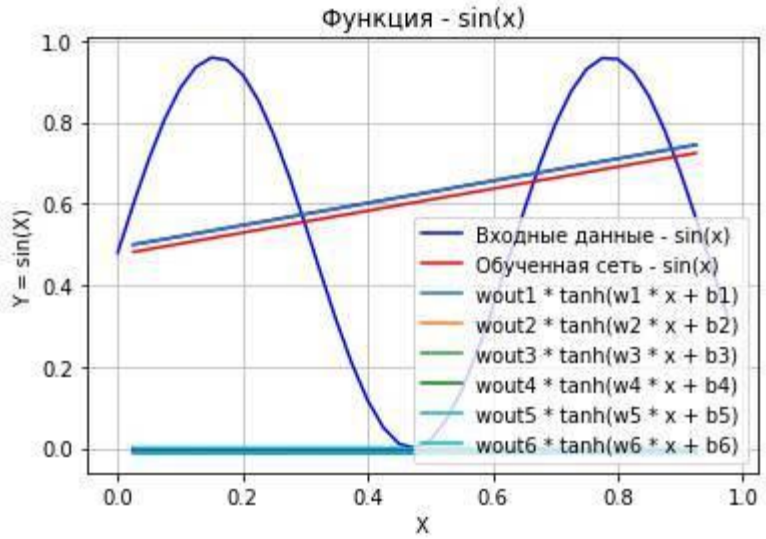
Результаты работы программы с сигмоидальной функцией:





Результаты с линейной функцией:





Даже из самого определения – линейная функция, можно сделать вывод что она не способна вносить не линейность на выходе. Что наглядно подтверждает последний график.

Набор данных рукописных цифр MNIST

Распознать текст, написанный от руки, не всегда просто даже для самого человека. Интересно было бы узнать, как с такой задачей справляется искусственный интеллект.

О трудности такого распознавания может служить следующая иллюстрация:

Даже нам трудно понять, какие цифры здесь изображены,
4
или
9
,
0
или
6
?

Некоторые из нас будут удивлены, что человек написавший их, имел в виду что первая цифра – это 9, а вторая 6.

Существует целая коллекция изображений рукописных цифр. Эта коллекция очень известна и популярна в среде исследователей искусственного интеллекта. Ей часто пользуются для тестирования своих идей и алгоритмов, заложенных в нейронную сеть.

Этим тестовым набором является набор данных рукописных цифр – “MNIST”, любезно предоставляемая известным специалистом в области искусственных нейронных сетей – Яном Лекуном. База данных “MNIST” предоставляется абсолютно бесплатно и находится по адресу: http://yann.lecun.com/exdb/mnist/.

Форматы файлов базы данных, на этой странице, не очень распространены и с ними не так легко работать. Но к счастью, другие специалисты позаботились об этой проблеме, они создали соответствующие файлы в более простом формате. Например, на странице: https://pjreddie.com/projects/mnist-in-csv/, вы можете скачать этот набор данных, в универсальном и знакомом нам формате .csv. На указанном сайте, для скачивания, предоставляются два файла:

– тренировочный набор данных (50 000 наборов цифр)

– тестовый набор данных (10 000 наборов цифр)

Все наборы промаркированы, то есть в каждом экземпляре набора указан правильный ответ. Как и ранее – это нулевой элемент строки.

Стоит отметить что данные в тестовом наборе не содержаться в тренировочном. Иными словами, при тестировании, сеть будет впервые встречать экземпляры из тестового набора.

Посмотрим, что внутри у этих файлов:

Давайте разберем что здесь понаписано. Каждый экземпляр цифры приведен в строках файла, в виде массива. В нулевом элементе этого массива, содержится целевое значение. На примере вышеизложенной иллюстрации, в нулевом элементе первой строки находится цифра 5, значит в строке зашифрована цифра 5, во второй 0, в третьей 4 и так далее. Последующие значения в строке, разделённые запятыми, – это значения пикселей рукописных цифр. Так как разрешение у всех цифр в этом наборе составляет 28x28 пикселей, то за каждым маркером следуют 784 элемента массива строки. Если присмотреться к этим значениям, то можно понять, что они находятся в пределах от 0 до 255, где минимальный 0 – белый цвет, а максимальный 255 – черный.

Используя логистическую функцию, обучая нейронную сеть на данном тренировочном наборе, нам придется привести все значения к диапазону от 0 до 1. А для того чтобы избежать ошибок, лучше всего к диапазону от 0,01 до 0,99.

Но прибегать к стандартизации данных не будем, пойдём иным путём, просто разделим значения на максимально возможное (которое нам заранее известно) 255. Таким образом сохраним нужные пропорции и область данных будет лежать в пределах от 0 до 1. Где белому цвету будет присвоено значение 0/255=0, а чёрному 255/255=1. А с учетом масштабирования, в целях уменьшения ошибок при работе программы, добавим, кроме чёрного, всем цветам значение 0,01, а чёрный уменьшим до 0,99.

Для реализации трехслойной структуры сети, с целью распознавания цифр из набора данных “MNIST”, задумаемся о количестве нейронов в каждом слое. Очевидно, что на входе их количество определяется значением входных данных, а именно 784. На выходе мы должны классифицировать изображение, целевая метка будет указывать на номер выходного нейрона. А значит на выходе нам понадобится ровно 10 нейронов. Каждый из которых будет отвечать за закрепленное за ним число. Если ответом служит 0, то должен активироваться первый нейрон, а остальные должны “молчать”. Если ответом служит 1, то должен активироваться второй нейрон, при “молчании” остальных и так далее.

А с количеством нейронов в скрытом слое определимся эмпирически, поскольку нет идеального метода для выбора скрытых узлов. Кроме того, и идеального метода выбора скрытых слоев тоже не существует.

С учётом того что на выходе имеется 10 нейронов, а на входе 784, то будем считать, что рациональным промежуточным диапазоном значений количества узлов в скрытом слое, будет значение от 100 до 400. Изменяя в этом диапазоне значения количества узлов и меняя значения скорости обучения, можно улучшать итоговые результаты.

Распознавание цифр из набора данных MNIST

Ну что же, реализуем в программе нейронную сеть распознающую рукописные цифры из набора “MNIST”.

Комментировать процесс подключения модулей, загрузки данных из файла, инициализации весов и параметров сети, алгоритма обучения, я не стану, так как всё это мы проделали ранее. Эти участки кода остаются практически не изменяемыми по сравнению с предыдущими.

```
import numpy as np
# библиотека для вывода на консоль массивов
import matplotlib.pyplot
# убедитесь, что участки находятся внутри этой записной книжки, а не внешнего окна
%matplotlib inline
#plt.show() # Вместо %matplotlib inline в других средах, не notebook
from time import time, sleep #Для замера времени выполнения функций
from tqdm import tqdm #Для вывода прогресса вычисления функций
# glob помогает выбрать несколько файлов, используя шаблоны
import glob
# помощник для загрузки данных из файлов изображений PNG
import scipy.misc



# Загрузить mnist тренировочные данные в формате CSV
training_data_file = open("MNIST_dataset/mnist_train.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data_file.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data_file.close() # закрываем фаел csv



# Определение класса нейронной сети
class neuron_Net:



# Инициализация весов нейронной сети
def __init__(self, input_num, neuron_num, output_num, learningrate): #констр.(входной слой, скрытый слой, выходной слой)
# МАТРИЦЫ ВЕСОВ
# Задаем матрицы весов как случайное
self.weights = np.random.normal(0.0, pow(input_num, -0.5), (neuron_num, input_num))
self.weights_out = np.random.normal(0.0, pow(neuron_num, -0.5), (output_num, neuron_num))
# Можно задать веса таким образом
#self.weights = (numpy.random.rand(neuron_num, input_num) -0.5)
#self.weights_out = (numpy.random.rand(output_num, neuron_num) -0.5)



# скорость обучения
self.lr = learningrate



pass



# Обучение нейронной сети
def train(self, inputs_list, targets_list): # принемает входной список данных,targets ответы
```

```
# Преобразовать список входов в вертикальный массив. .T – транспонирование
inputs_x = np.array(inputs_list, ndmin=2).T # матрица числа
targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов какое это число



# ВЫЧИСЛЕНИЕ СИГНАЛОВ ПО СЛОЯМ
# Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
x1 = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs



# вычислить сигналы, возникающие из скрытого слоя. сигмоида(Xhidden – сигнал скр.слоя)
y1 = 1/(1+np.exp(-x1))



# вычислить сигналы в окончательном выходном слое (матрица сигналов выходного слоя)
x2 = np.dot(self.weights_out, y1)
# вычислить сигналы, исходящие из конечного выходного слоя. сигмоида(Xoutputs – сигнал вых.слоя)
y2 = 1/(1+np.exp(-x2))



# ВЫЧИСЛЕНИЕ ОШИБКИ ПО СЛОЯМ
# Ошибка выходного слоя E = -(цель – фактическое значение)
E = -(targets_Y – y2)
# Ошибка скрытого слоя
E_hidden = np.dot(self.weights_out.T, E)



# ОБНОВЛЕНИЕ ВЕСОВ ПО СЛОЯМ
# Меняем веса исходящие из скрытого слоя по каждой связи
self.weights_out -= self.lr * np.dot((E * y2 * (1.0 – y2)), np.transpose(y1))



# Меняем веса исходящие из входного слоя по каждой связи
self.weights -= self.lr * np.dot((E_hidden * y1 * (1.0 – y1)), np.transpose(inputs_x))



pass



# МЕТОД ПРОГОНА СВОИХ ЗНАЧЕНИЙ ПО СЕТИ
# Метод прогона тестовых значений
def query(self, inputs_list): # Принимает свой набор тестовых данных
# Преобразовать список входов в вертикальный массив.
inputs_x = np.array(inputs_list, ndmin=2).T
```

```
    # Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
    x1 = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs
    # Вычислить сигналы, выходящие из нейрона. Функция активации – сигмоида(x)
    y1 = 1/(1+np.exp(-x1))
    # Вычислить сигналы в нейронах выходного слоя. Взвешенная сумма.
    x2 = np.dot(self.weights_out, y1)
    # вычислить сигналы, исходящие из конечного выходного слоя. Функция активации – сигмоида(x)
    y2 = 1/(1+np.exp(-x2))



    return y2



    # ЗАДАЁМ ПАРАМЕТРЫ СЕТИ
    # Количество входных данных, нейронов
    data_input = 784
    data_neuron = 220
    data_output = 10



    # Скорость обучения
    learningrate = 0.15



    # Создать экземпляр нейронной сети
    n = neuron_Net(data_input, data_neuron, data_output, learningrate)



    # ОБУЧЕНИЕ
    # Зададим количество эпох
    epochs = 3



    start = time()
    # Прогон по обучающей выборке
    for e in range(epochs):
    # Пройдите все записи в наборе тренировочных данных
    #for record in training_data_list:
    for i in tqdm(training_data_list, desc = str(e+1)): # tqdm – используем интерактив состояния прогресса вычисления
    # Получить входные данные числа
    all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    # Массив данных входа с масштабированием от 0,01 до 0,99
    inputs_x = (np.asfarray(all_values[1:])/ 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение
```

```
    # Получить целевое значение Y, (ответ – какое это число)
    targets_Y = int(all_values[0])  # перевод символов в int, 0 элемент – целевое значение



    # создать целевые выходные значения (все 0.01, кроме нужной метки, которая равна 0.99)
    targets_Y = np.zeros(data_output) + 0.01



    # Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
    targets_Y[int(all_values[0])]  = 0.99



    n.train(inputs_x,  targets_Y) # наш метод train – обучение нейронной сети



    pass
    pass



    time_out  = time()  – start
    print("Время  выполнения: ", time_out,  " сек" )
```

Теперь, после обучения, нам нужно проверить сеть на эффективность, для этого загрузим данные из файла с тестовым набором данных:

```
    # ТЕСТИРОВАНИЕ  ОБУЧЕННОЙ  СЕТИ
    # Загрузить тестовый CSV-файл
    test_data_file  = open("MNIST_dataset/mnist_test.csv",  'r') # 'r' – открываем файл для чтения
    test_data_list = test_data_file.readlines() # readlines() – читает все строки в файле в переменную test_data_list
    test_data_file.close()  # закрываем файл csv
```

А для проверки эффективности, используем формулу среднего арифметического (сумма результатов / количество данных в выборке). После чего, получим значение, лежащее в пределах от “0” до “1”, если помножить это значение на “100”, мы получаем процент предсказаний. Сумма результатов будет представлять собой сумму верных или не верных ответов сети, где верным ответом будет считаться “1”, а не верным “0”.

```
    # ПРОВЕРКА  ЭФФЕКТИВНОСТИ  НЕЙРОННОЙ  СЕТИ
    # Массив показателей эффективности сети, изначально пустой
    efficiency  = []
```

```
    # Прогон по всем записям в наборе тестовых данных
    for i in test_data_list:
    # Получить входные данные числа
    all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    # Правильный ответ, хранимый в нулевом индексе
    targets_Y = int(all_values[0])
    # Массив данных входа с масштабированием от 0,01 до 0,99
    inputs_x = (np.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение



    # Запросить ответ у сети
    outputs_y = n.query(inputs_x) # Прогон по сети тестового значения из нашего файла
    # Индекс самого высокого значения на матрице выхода, соответствует метке числа
    label_y = np.argmax(outputs_y) # argmax возвращает индекс максимального элемента в выходном массиве



    # Добавить правильный или неправильный список
    if (label_y == targets_Y): # Если индекс макс. знач. на выходе = целевому значению (0 индекс массива данных)
    # Если ответ сети соответствует целевому значению, добавляем 1 в конец массива показателей эффективности
    efficiency.append(1)
    else:
    # Если ответ сети не соответствует целевому значению, добавляем 0 в конец массива показателей эффективности
    efficiency.append(0)



    pass



    pass
```

Здесь, в процессе прогона тестовых данных по сети, создается массив для хранения ответов на выходе – efficiency = [].

В цикле прогона по тестовым данным, делаем знакомые нам манипуляции с разделением строки на символы, для последующей их подачи на вход. Получением целевых значений из данных. Последующим масштабированием входных данных и запроса ответа на выходе сети, который сохраняется в переменной, для выявления максимального значения на одном из десяти выходах сети, для сравнение индекса этого числа с целевым результатом. После чего в конец массива эффективности, добавляем значения “0”, или “1”, где “0” символизирует что сеть ошиблась, а “1” что угадала цифру. Метод append() – как раз позволяет добавлять значения в конец массива.

Выводим процент совпадений с правильными ответами:

```
    # Вычислить оценку производительности. Доля правильных ответов
    efficiency_map = np.asarray(efficiency) # asarray – преобразование списка в массив
```

print ('Производительность = ', (efficiency_map.sum() / efficiency_map.size)*100, '%') # Среднее арифметическое

Ответ:

Производительность = 97.09 %

Как видим, процент распознавания цифры из тестовой выборки, довольно высокий и составляет чуть более 97%. Это не плохой результат.

Теперь давайте проверим сеть на собственном наборе данных. Я подготовил восемь наборов цифр, один из которых я намеренно исказил, инвертировав цвета в цифре 8.

# СОБСТВЕННЫЙ НАБОР ИЗОБРАЖЕНИЙ ДЛЯ ТЕСТА
my_dataset = [] # Для хранения данных и целевых значений

Загрузить данные изображения в формате PNG, как установить тестовые данные
for image_file in glob.glob('my_image/_my_?.png'): # проход по файлам изобр. в папке my_images
#glob – из библиотеки glob, помогает выбрать сразу несколько файлов из папки

# Метка имени числа
label_y = int(image_file[-5:-4]) # хранит число в файле ?.png, -5 это ответ какое число '?'
# от -5 до -4 это будет символ '?', т.е метка числа

Загрузить данные изображения из png файлов в массив
print ('Имя файла: ', image_file) # вывод пути и имени открытого файла

```
image_list = scipy.misc.imread(image_file, flatten=True) #“flatten=True” (“выровнять=True) ”– превращает
#изображения в простой массив чисел с плавающей запятой


# Изменить формат из 28x28 в список 784 значений, инвертировать значения
image_data = 255.0 – image_list.reshape(784) #преобразует массив из квадрата 28x28 в длинный список значений
#вычитание значений массива из 255.0. т.к обычно '0' означает черное, а '255' означает белое, но набор данных MNIST
#имеет инверсные значения, поэтому мы должны их перевернуть


# Вносим данные шкалу с диапазоном от 0 до 1
image_data = (image_data / 255.0 ) # массив данных входа с масштабированием от 0 до 1


# Добавить метку числа и данные изображения к общему набору данных
my_data = np.append(label_y, image_data)
my_dataset.append(my_data)


pass
```

Здесь создаем массив для хранения данных числа из файлов – “my_dataset = []”. В цикле, проходим по всем файлам в папке “my_images”, в этом нам помогает библиотека “glob”. Далее, сохраняем в переменные значений целевых результатов и входных данных. Целевые значения получаем с помощью метки числа в имени файла, для этого используем индексы и срезы. Запись “image_file[-5:-4]”, означает что мы берем пятый элемент массива с конца. В нашем случае, на примере файла “_my_0.png” – будет значение “0”.

Функция scipy.misc.imread(image_file, flatten=True) – превращает данные из изображения в простой массив чисел с плавающей точкой.

В переменную “image_data” вносим массив значений с плавающей точкой:

```
image_data = 255.0 – image_list.reshape(784)
```

Метод “reshape” – преобразует вектор входных данных в 2D матрицу с размерностью 28x28. Вычитая из 255, значения элементов в этой матрице, мы инвертируем черные и белые цвета. Это необходимо для того, что данные MNIST представлены именно в таком формате, хотя обычно яркое свечение пикселя представляется большим значением.

Далее вносим данные в шкалу значений от 0 до 1. И с помощью метода “append”, добавляем результаты в конец массива данных.

Далее следует сам алгоритм распознавания собственного набора данных:

```
# ПРОВЕРКА СЕТИ НА СОБСТВЕННЫХ ДАННЫХ ИЗОБРАЖЕНИЙ


# запись для тестирования
room_choices = 0


# Изображение участка
matplotlib.pyplot.imshow(my_dataset[room_choices][1:].reshape(28,28), cmap='Greys', interpolation='None')
# my_dataset – наш собственный набор тестовых данных


# Правильный ответ в нулевом столбце
correct_label = my_dataset[room_choices][0] # в строках номер файла из папки собственных данных, 0 толбец – ответ какое число


# Входные значения
input_x = my_dataset[room_choices][1:]  # значение числа без ответа


# Запросить сеть
output_y = n.query(input_x)  # прогоняем тестовую выборку по сети
print (output_y) # вывод по выходу сети


print('Минимальное значение: ', np.min(output_y)) # вывод мин знач элемента на выходе
print('Максимальное значение: ', np.max(output_y)) # вывод макс знач элемента на выходе


# Индекс самого высокого значения на выходе сети, соответствует метке
number = np.argmax(output_y)
print('\nЦелевое значение: ', number)


# Вывод правильный или неправильный ответ
if (number == correct_label): # если макс знач на выходе label = ответу (0 индекс из массива) correct_label
print ('Угадал!!! :-))))')
else:
print ('Не угадал! :-(((')
```

pass

Здесь, переменная “room_choices” задает какое число из набора данных мы хотим распознать в данный момент.

После функции вывода числа на консоль – matplotlib.pyplot. imshow(my_dataset[room_choices][1:].reshape(28,28), из полученного массива данных “my_dataset” извлекаем целевые и входные данные.

Затем запрашиваем сеть и сохраняем в переменную “output_y” результаты на выходе из сети. Выводим минимальное и максимальное значение из его элементов, после чего в переменную “number” сохраняем номер индекса максимального элемента – ответ сети. В последствии сравниваем это значение с целевым и выводим сообщение “Угадал” и “Не угадал” соответственно.

Полный текст программы:

```
import numpy as np
# библиотека для вывода на консоль массивов
import matplotlib.pyplot
# убедитесь, что участки находятся внутри этой записной книжки, а не внешнего окна
%matplotlib inline
#plt.show() # Вместо %matplotlib inline в других средах, не notebook
from time import time, sleep #Для замера времени выполнения функций
from tqdm import tqdm #Для вывода прогресса вычисления функций
# glob помогает выбрать несколько файлов, используя шаблоны
import glob
# помощник для загрузки данных из файлов изображений PNG
import scipy.misc



# Загрузить mnist тренировочные данные в формате CSV
training_data_file = open("MNIST_dataset/mnist_train.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data_file.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data_file.close() # закрываем фаел csv



# Определение класса нейронной сети
class neuron_Net:



# Инициализация весов нейронной сети
def __init__(self, input_num, neuron_num, output_num, learningrate): #констр.(входной слой, скрытый слой, выходной слой)
# МАТРИЦЫ ВЕСОВ
# Задаем матрицы весов как случайное
self.weights = np.random.normal(0.0, pow(input_num, -0.5), (neuron_num, input_num))
self.weights_out = np.random.normal(0.0, pow(neuron_num, -0.5), (output_num, neuron_num))
# Можно задать веса таким образом
#self.weights = (numpy.random.rand(neuron_num, input_num) -0.5)
#self.weights_out = (numpy.random.rand(output_num, neuron_num) -0.5)
```

```
    # скорость обучения
    self.lr = learningrate


    pass


    # Обучение нейронной сети
    def train(self, inputs_list, targets_list): # принемает входной список данных,targets ответы
    # Преобразовать список входов в вертикальный массив. .T – транспонирование
    inputs_x = np.array(inputs_list, ndmin=2).T # матрица числа
    targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов какое это число


    # ВЫЧИСЛЕНИЕ СИГНАЛОВ ПО СЛОЯМ
    # Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
    x1 = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs


    # вычислить сигналы, возникающие из скрытого слоя. сигмоида(Xhidden – сигнал скр.слоя)
    y1 = 1/(1+np.exp(-x1))


    # вычислить сигналы в окончательном выходном слое (матрица сигналов выходного слоя)
    x2 = np.dot(self.weights_out, y1)
    # вычислить сигналы, исходящие из конечного выходного слоя. сигмоида(Xoutputs – сигнал вых.слоя)
    y2 = 1/(1+np.exp(-x2))


    # ВЫЧИСЛЕНИЕ ОШИБКИ ПО СЛОЯМ
    # Ошибка выходного слоя E = -(цель – фактическое значение)
    E = -(targets_Y – y2)
    # Ошибка скрытого слоя
    E_hidden = np.dot(self.weights_out.T, E)


    # ОБНОВЛЕНИЕ ВЕСОВ ПО СЛОЯМ
    # Меняем веса исходящие из скрытого слоя по каждой связи
    self.weights_out -= self.lr * np.dot((E * y2 * (1.0 – y2)), np.transpose(y1))


    # Меняем веса исходящие из входного слоя по каждой связи
    self.weights -= self.lr * np.dot((E_hidden * y1 * (1.0 – y1)), np.transpose(inputs_x))
```

```
pass


# МЕТОД ПРОГОНА СВОИХ ЗНАЧЕНИЙ ПО СЕТИ
# Метод прогона тестовых значений
def query(self, inputs_list): # Принимает свой набор тестовых данных
# Преобразовать список входов в вертикальный массив.
inputs_x = np.array(inputs_list, ndmin=2).T



# Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
x1 = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs
# Вычислить сигналы, выходящие из нейрона. Функция активации – сигмоида(x)
y1 = 1/(1+np.exp(-x1))
# Вычислить сигналы в нейронах выходного слоя. Взвешенная сумма.
x2 = np.dot(self.weights_out, y1)
# Вычислить сигналы, выходящие из нейрона. Функция активации – сигмоида(x)
y2 = 1/(1+np.exp(-x2))



return y2



# ЗАДАЁМ ПАРАМЕТРЫ СЕТИ
# Количество входных данных, нейронов
data_input = 784
data_neuron = 220
data_output = 10



# Скорость обучения
learningrate = 0.15



# Создать экземпляр нейронной сети
n = neuron_Net(data_input, data_neuron, data_output, learningrate)



# ОБУЧЕНИЕ
# Зададим количество эпох
epochs = 3



start = time()
# Прогон по обучающей выборке
for e in range(epochs):
# Пройдите все записи в наборе тренировочных данных
```

```
    #for record in training_data_list:
    for i in tqdm(training_data_list, desc = str(e+1)): # tqdm – используем интерактив состояния прогресса вычисления
    # Получить входные данные числа
    all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    # Массив данных входа с масштабированием от 0,01 до 0,99
    inputs_x = (np.asfarray(all_values[1:])/ 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение



    # Получить целевое значение Y, (ответ – какое это число)
    targets_Y = int(all_values[0]) # перевод символов в int, 0 элемент – целевое значение



    # создать целевые выходные значения (все 0.01, кроме нужной метки, которая равна 0.99)
    targets_Y = np.zeros(data_output) + 0.01



    # Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
    targets_Y[int(all_values[0])] = 0.99



    n.train(inputs_x, targets_Y) # наш метод train – обучение нейронной сети



    pass
    pass



    time_out = time() – start
    print("Время выполнения: ", time_out, " сек" )



    # ТЕСТИРОВАНИЕ ОБУЧЕННОЙ СЕТИ
    # Загрузить тестовый CSV-файл
    test_data_file = open("MNIST_dataset/mnist_test.csv", 'r') # 'r' – открываем файл для чтения
    test_data_list = test_data_file.readlines() # readlines() – читает все строки в файле в переменную test_data_list
    test_data_file.close() # закрываем файл csv



    # ПРОВЕРКА ЭФФЕКТИВНОСТИ НЕЙРОННОЙ СЕТИ
    # Массив показателей эффективности сети, изначально пустой
    efficiency = []
```

```
    # Прогон по всем записям в наборе тестовых данных
    for i in test_data_list:
    # Получить входные данные числа
    all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    # Правильный ответ, хранимый в нулевом индексе
    targets_Y = int(all_values[0])
    # Массив данных входа с масштабированием от 0,01 до 0,99
    inputs_x = (np.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение



    # Запросить ответ у сети
    outputs_y = n.query(inputs_x)  # Прогон по сети тестового значения из нашего файла
    # Индекс самого высокого значения на матрице выхода, соответствует метке числа
    label_y = np.argmax(outputs_y) # argmax возвращает индекс максимального элемента в выходном массиве



    # Добавить правильный или неправильный список
    if (label_y == targets_Y): # Если индекс макс. знач. на выходе = целевому значению (0 индекс массива данных)
    # Если ответ сети соответствует целевому значению, добавляем 1 в конец массива показателей эффективности
    efficiency.append(1)
    else:
    # Если ответ сети не соответствует целевому значению, добавляем 0 в конец массива показателей эффективности
    efficiency.append(0)
    pass
    pass



    # Вычислить оценку производительности. Доля правильных ответов
    efficiency_map  = np.asarray(efficiency)  # asarray – преобразование списка в массив
    print ('Производительность = ', (efficiency_map.sum() / efficiency_map.size)*100, '%') # Среднее арифметическое



    # СОБСТВЕННЫЙ НАБОР ИЗОБРАЖЕНИЙ ДЛЯ ТЕСТА
    my_dataset = [] # Для хранения данных и целевых значений
    # Загрузить данные изображения в формате PNG, как установить тестовые данные
    for image_file in glob.glob('my_image/_my_?.png'): # проход по файлам изобр. в папке my_images
    #glob – из библиотеки glob, помогает выбрать сразу несколько файлов из папки



    # Метка имени числа
    label_y = int(image_file[-5:-4]) # хранит число в файле ?.png, -5 это ответ какое число '?'
    # от -5 до -4 это будет символ '?', т.е метка числа
```

```
    # Загрузить данные изображения из png файлов в массив
    print ('Имя файла: ', image_file)  # вывод пути и имени открытого файла



    image_list = scipy.misc.imread(image_file, flatten=True) #“flatten=True” (“выровнять=True) ”– превращает
    #изображения в простой массив чисел с плавающей запятой



    # Изменить формат из 28x28 в список 784 значений, инвертировать значения
    image_data = 255.0 – image_list.reshape(784) #преобразует массив из квадрата 28x28 в длинный список значений
    #вычитание значений массива из 255.0. т.к обычно '0' означает черное, а '255' означает белое, но набор данных MNIST
    #имеет инверсные значения, поэтому мы должны их перевернуть



    # Вносим данные шкалу с диапазоном от 0 до 1
    image_data = (image_data / 255.0 ) # массив данных входа с масштабированием от 0 до 1



    # Добавить метку числа и данные изображения к общему набору данных
    my_data = np.append(label_y, image_data)
    my_dataset.append(my_data)



    pass



    # ПРОВЕРКА СЕТИ НА СОБСТВЕННЫХ ДАННЫХ ИЗОБРАЖЕНИЙ
    # запись для тестирования
    room_choices = 0
    # Изображение участка
    matplotlib.pyplot.imshow(my_dataset[room_choices][1:].reshape(28,28), cmap='Greys', interpolation='None')
    # my_dataset– наш собственный набор тестовых данных



    # Правильный ответ в нулевом столбце
    correct_label = my_dataset[room_choices][0] # в строках номер файла из папки собственных данных, 0 толбец – ответ какое число



    # Входные значения
    input_x = my_dataset[room_choices][1:]  # значение числа без ответа
```

```
# Запросить сеть
output_y = n.query(input_x)  # прогоняем тестовую выборку по сети
print (output_y) # вывод по выходу сети



print('Минимальное  значение: ', np.min(output_y)) # вывод мин знач элемента  на выходе
print('Максимальное  значение: ', np.max(output_y)) # вывод макс знач элемента  на выходе



# Индекс самого высокого значения на выходе сети, соответствует метке
number = np.argmax(output_y)
print('\nЦелевое  значение: ', number)



# Вывод правильный или неправильный ответ
if (number == correct_label): # если макс знач на выходе label = ответу (0 индекс из массива)
correct_label
print ('Угадал!!! :-))))')
else:
print ('Не угадал! :-(((')



pass
```

Результаты работы программы:

1)

Производительность = 97.09 %

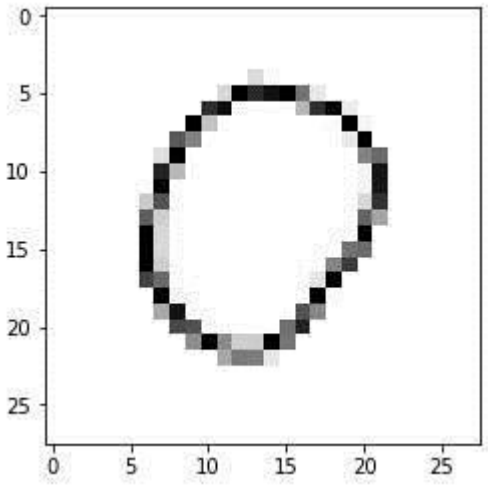
2)

```
Имя
файла
: my_image\_my_0.png
Имя файла: my_image\_my_1.png
Имя файла: my_image\_my_2.png
Имя файла: my_image\_my_3.png
Имя файла: my_image\_my_4.png
Имя файла: my_image\_my_5.png
Имя файла: my_image\_my_6.png
Имя файла: my_image\_my_7.png
Имя файла: my_image\_my_8.png
```

3)

```
[[6.66671197e-01]
```

[2.56659427e-03]
[9.32429328e-03]
[6.77700446e-05]
[1.92683835e-03]
[2.02324562e-03]
[3.13702331e-03]
[8.52300221e-03]
[2.36677373e-04]
[4.81128855e-03]]
Минимальное значение: 6.77700445779e-05
Максимальное значение: 0.666671197018
Целевое значение: 0
Угадал!!! :-)))



Скачать исходники с кодом можно по ссылке:
https://github.com/CaniaCan/neuralmaster
Для того чтобы воспользоваться, данными набора “MNIST”, папку “MNIST_dataset”, необходимо будет разархивировать.

Если подвести итог по проделанной работе, можно сказать что мы достигли невероятных успехов. Пройдя путь от простейшей классификации двух видов, до распознавания цифр на большом объеме входных данных и обучающей выборке. Надо сказать, что мы использовали лишь самые простые математические концепции создания и обучения нейронных сетей. Но этого оказалось вполне достаточно чтобы показать результаты, которые сопоставимы с профессиональными.

ГЛАВА 8

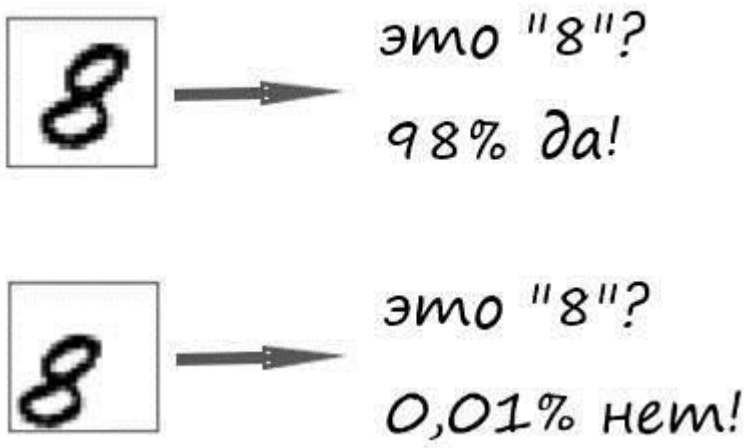
Свёрточная нейронная сеть

Если, при распознавании числа, на примере своих собственных значениях чисел, сеть часто ошибается, то это не связанно с тем что мы её плохо натренировали, ведь остальные 10000 тестовых наборов она распознаёт достаточно точно. Всё дело в том, что такого типа сети, при распознавании изображений очень чувствительны к сегментации распознаваемого изображения. Иными словами, критична к области расположения объекта на изображении, его размеру, и углу его наклона.

Чтобы избавиться от данных недостатков и тем самым улучшить качество распознавания изображений, были придуманы – **свёрточные нейронные сети**.

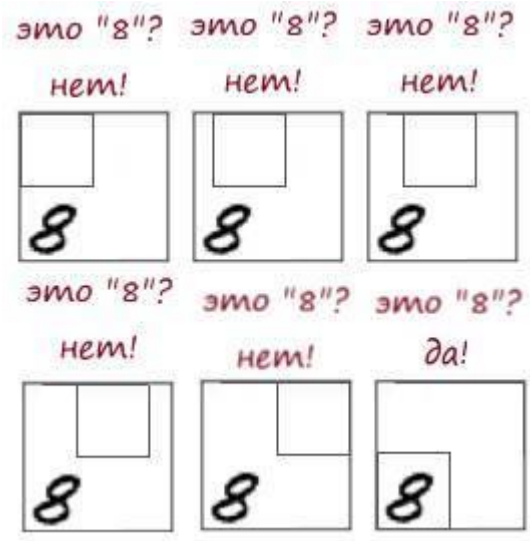
Сегментация изображений

Для начала, проиллюстрируем вышесказанное на примере цифры “8”. Наша сеть действительно хорошо работает на простых изображениях, где цифра находится прямо по середине изображения:

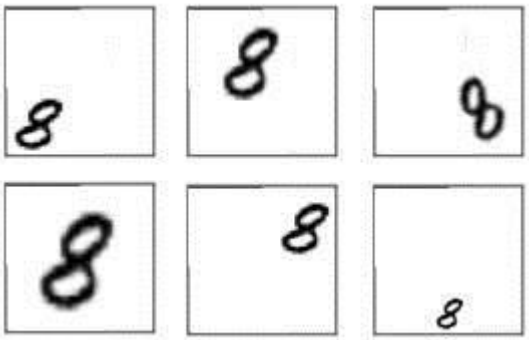


Это потому, что наша нейронная сеть обучалась только на центрированных “8”, со схожими размерами. Она совершенно не знает, что такое “8”, находящейся не в центре изображения или меньшего размера.

Мы уже создали действительно хорошую программу для распознавания рукописных чисел, когда они находится в центре изображения. Но что если мы просканируем все части изображения небольшого размера на наличие, к примеру, всё той же цифры “8”, пока не найдем её?



А что, если мы просто обучим сеть большим количеством данных, включая “8” всех размеров, под разными углами и во всех положениях на картинке? Для этого нам даже не потребуется собирать новые данные для обучения. Мы можем просто написать участок кода в программе, который будет генерировать новые изображения с “8”, во всех видах различных позиций на картинке:



Но ведь нет смысла обучать сеть распознавать “8” вверху изображения отдельно от распознавания “8” внизу изображения, как если бы это были два совершенно разных объекта.

Должен быть какой-то способ сделать нейронную сеть достаточно умной, чтобы она знала без дополнительного обучения, что “8” где-либо на картинке – это один и тот же объект. К счастью, такой способ есть!

Ответ в свёртке изображения. Пояснить этот процесс можно на примере изображения котика:

Как человек, вы интуитивно прекрасно понимаете, что это изображение имеют свою иерархию или концептуальную структуру.

Глядя на изображение, вы разделяете признаки на составные части:

– земля покрыта травой и цветами

– на картинке присутствует котик
– котик сидит на камне
– камень лежит на траве
– на заднем фоне находится кустарник

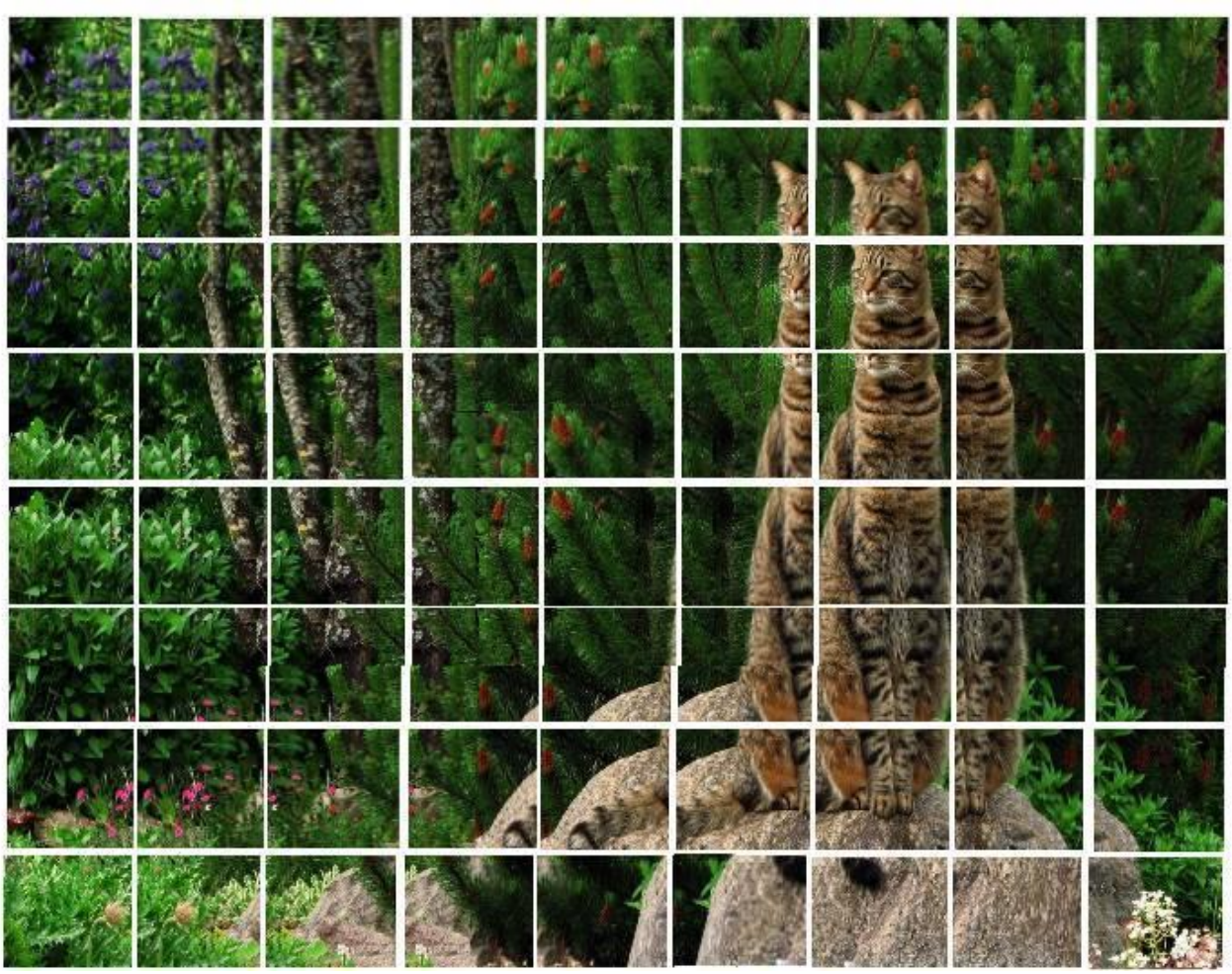
Но самое главное, что мы узнаем образ котика независимо от того, на каком фоне и поверхности он находится. Нам не нужно заново учиться узнавать образ котика, на каждом новом фоне или поверхности.

А созданная нами нейронная сеть не может этого сделать. Она думает, что “8” в новой части изображения – это совсем другой объект. Она не понимает, что перемещение объекта по изображению не делает его чем-то другим. Это означает, что она должна учиться идентифицировать каждый объект в любой возможной позиции.

Нужно сделать так, что независимо от того, где на картинке находится объект, нейронная сеть могла его распознать.

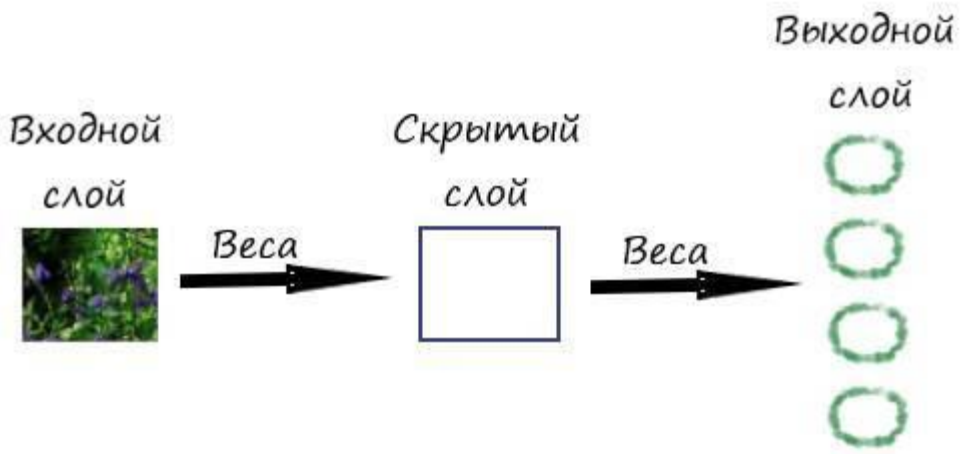
Мы сделаем это, используя процесс под названием свёртка (convolution).

Первый этап её работы будет заключаться в разбиении изображения на участки с определенным шагом:

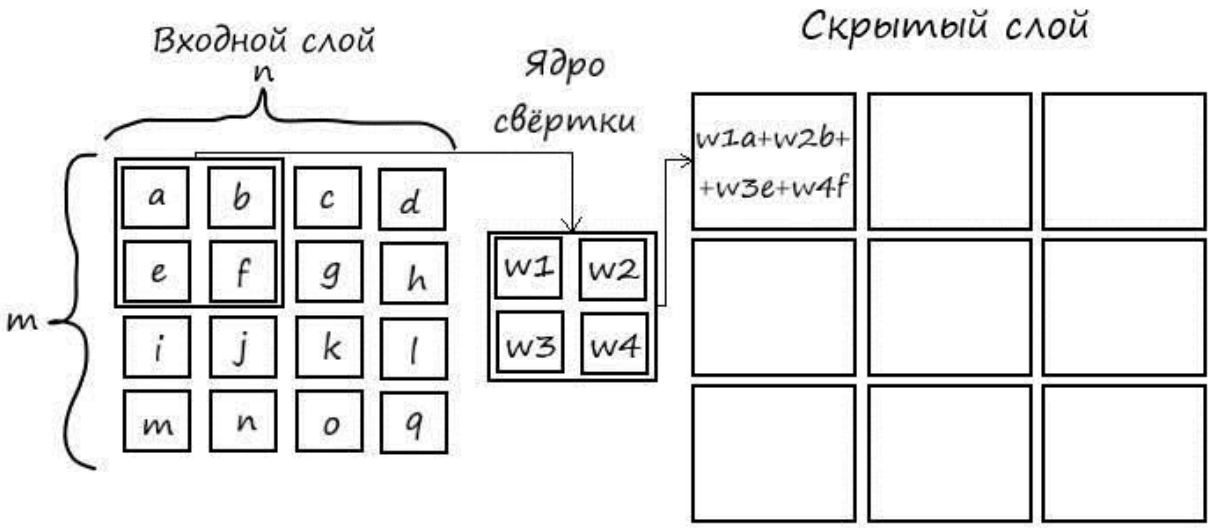


Тем самым, мы превратили одно изображение во множество более мелких.

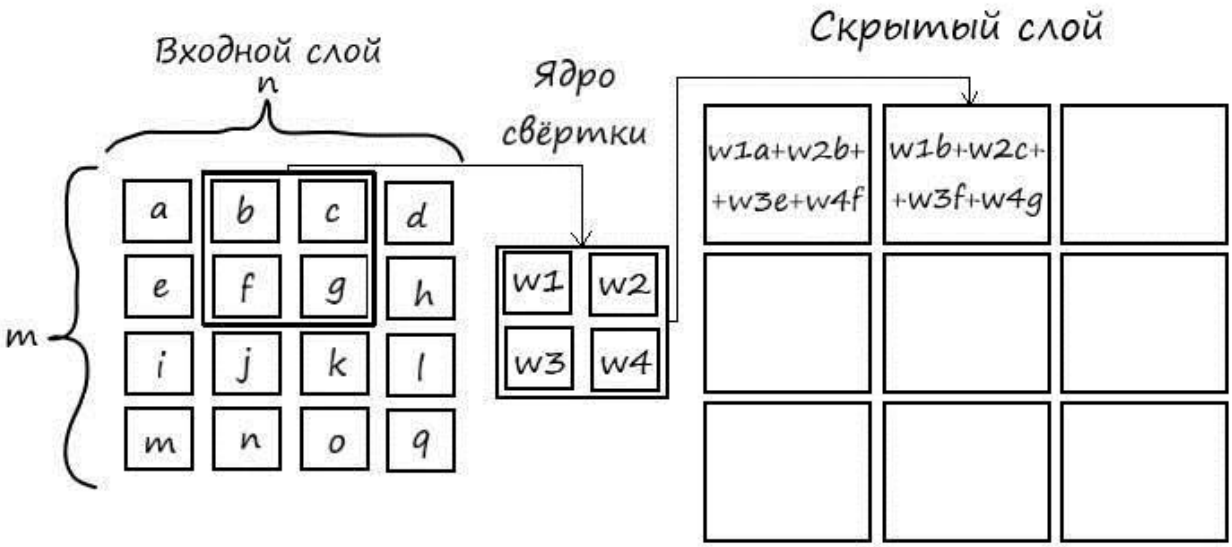
А вот теперь, мы подаем на вход нейронной сети, каждое из этих изображений, тем самым обрабатываем каждый фрагмент изображения одинаково. Если на фрагменте появляется интересующий нас класс, то этот фрагмент при обратном распространении ошибки, будет представлять наибольший интерес. При этом весовые коэффициенты будут общими для всех изображений:



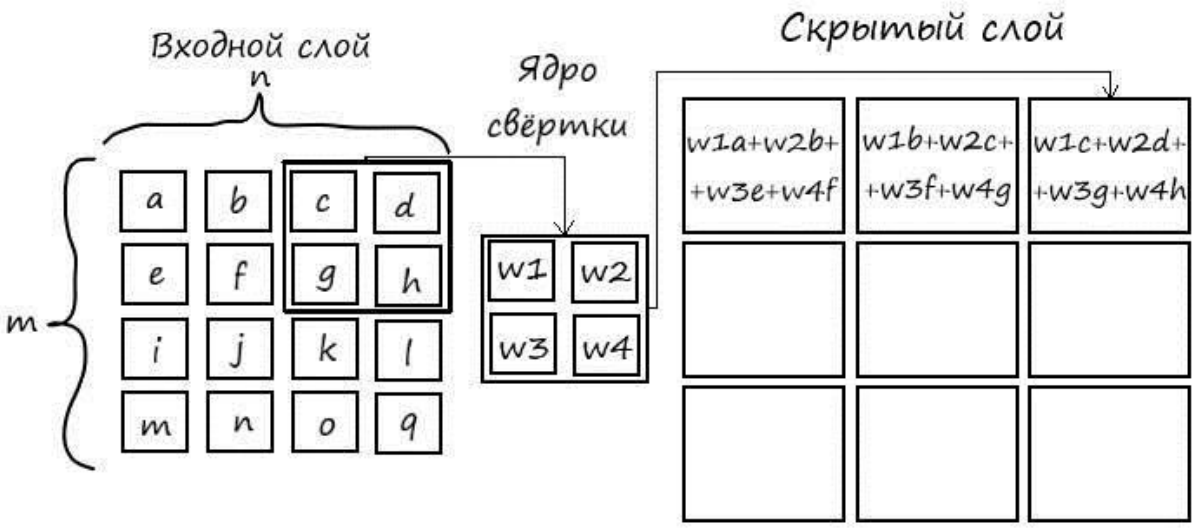
Стоит более детально описать процесс свертки между входом и скрытым слоем. Для этого представим входное изображение как 2D матрицу из пикселей, каждый из которых помечен определенной буквой. Веса так же представим в виде 2D массива. В свёрточных сетях они называются – **фильтрами или ядром свертки**. Если пройтись фильтром по всему изображению с шагом в один пиксель, то в скрытом слое получим следующие значения:



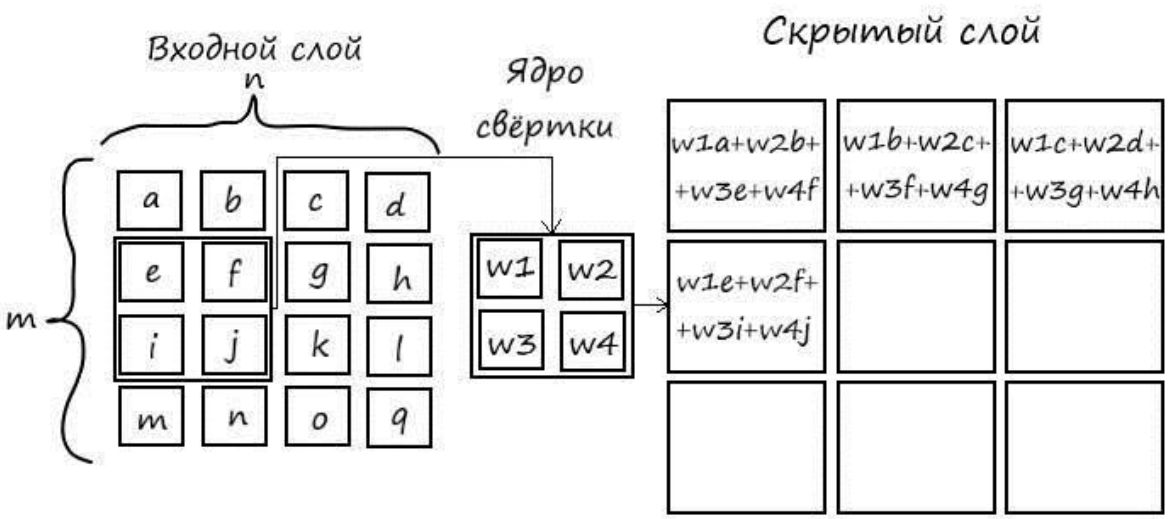
Делаем шаг на один пиксель вправо:



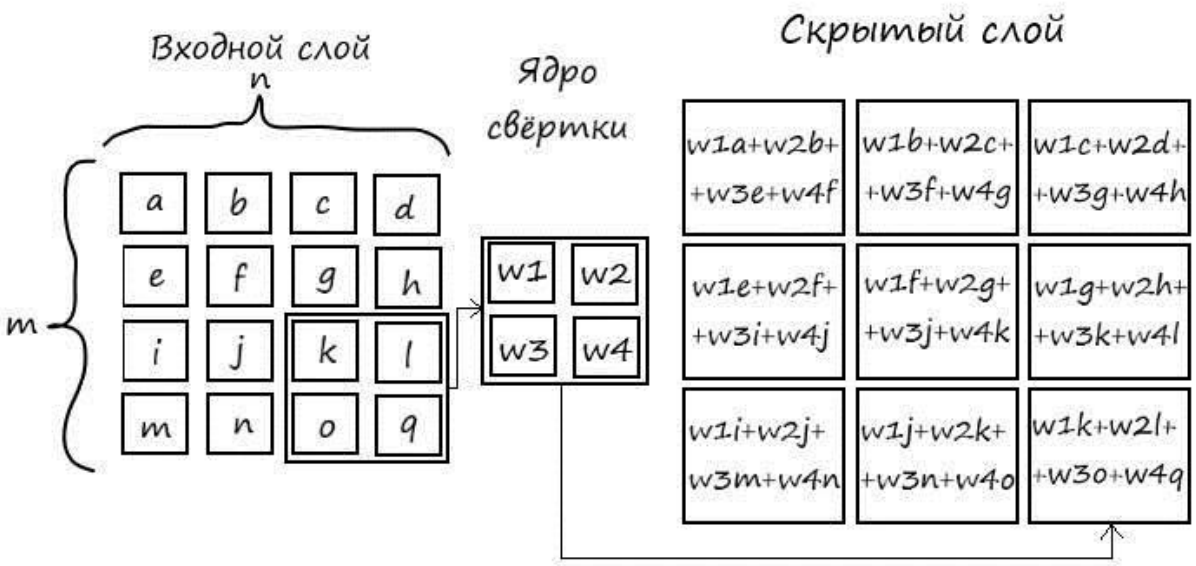
Повторяем шаги на один пиксель вправо, до окончания столбцов на входе:



Затем опускаемся вдоль строк матрицы входа на один пиксель:

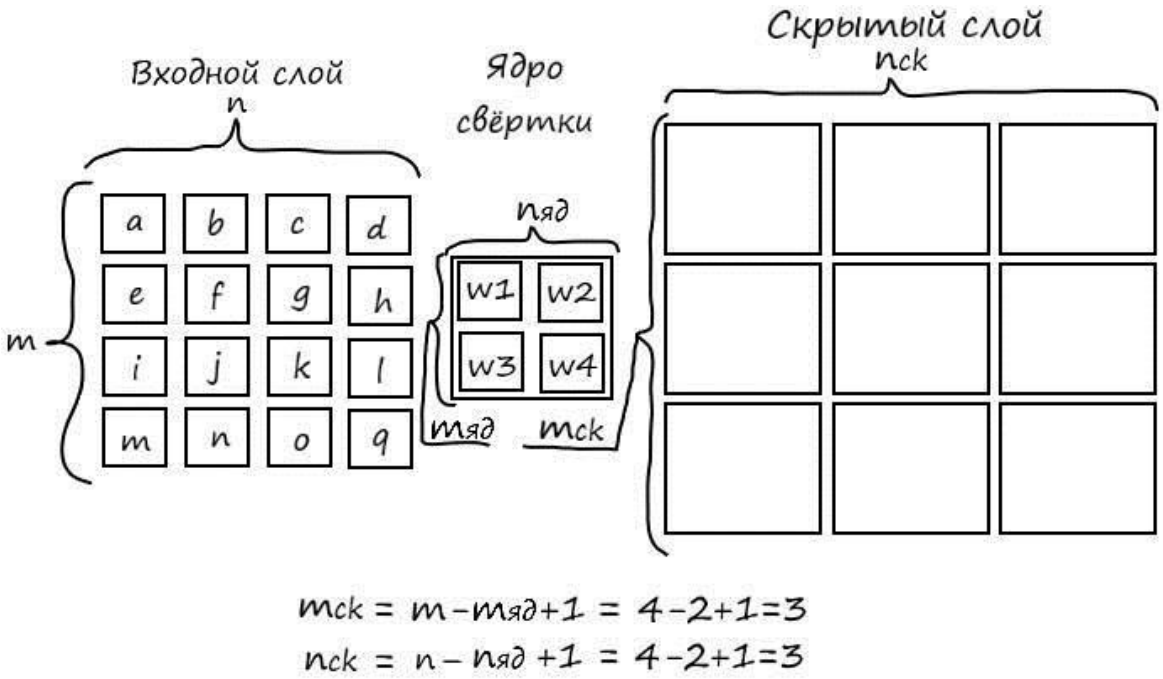


Повторяем данные процедуры до конца:



К слову – величина шага может быть иной, не обязательно в один пиксель. Но мы, в наших программах, будем использовать именно такой шаг. Так же будем рассматривать только одинаковое количество пикселей по строкам и столбцам на входе, то есть строго квадратное изображение поступающие на вход сети.

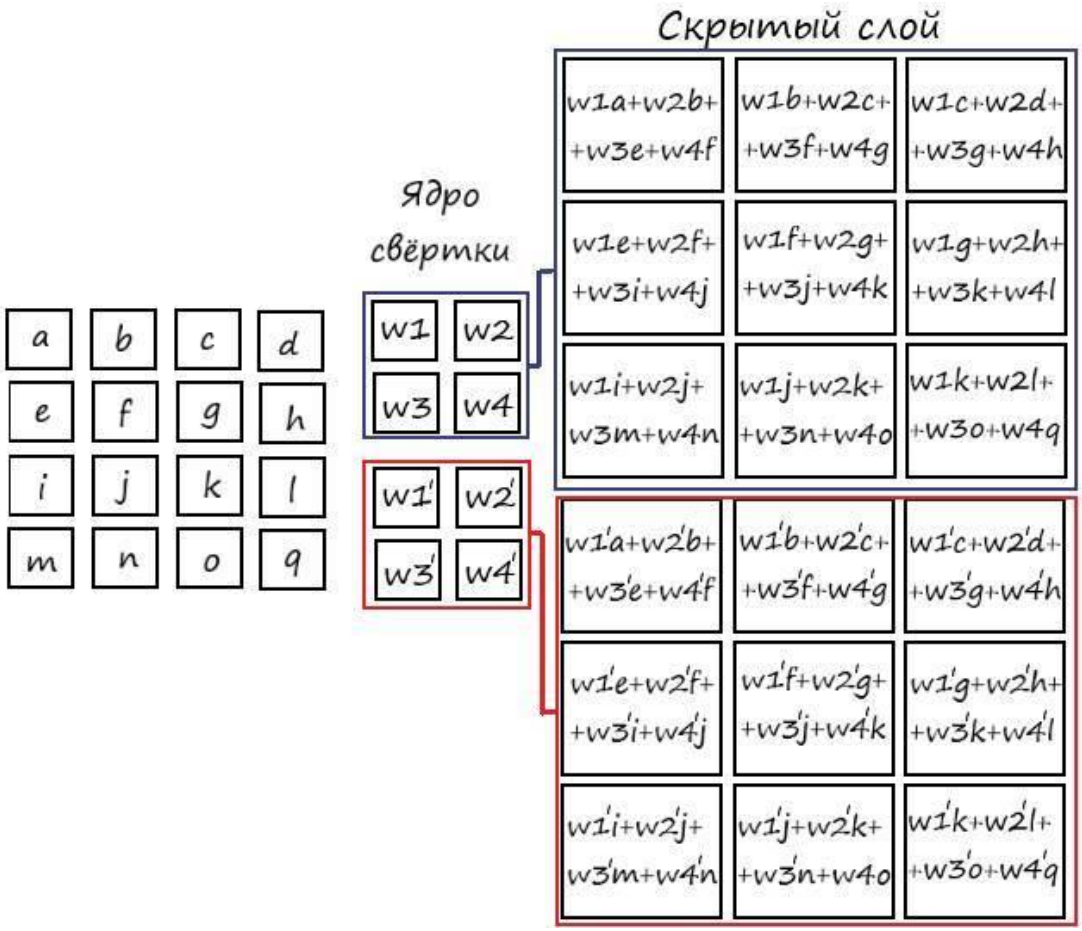
А результирующее количество пикселей в скрытом слое, определяется простым соотношением:



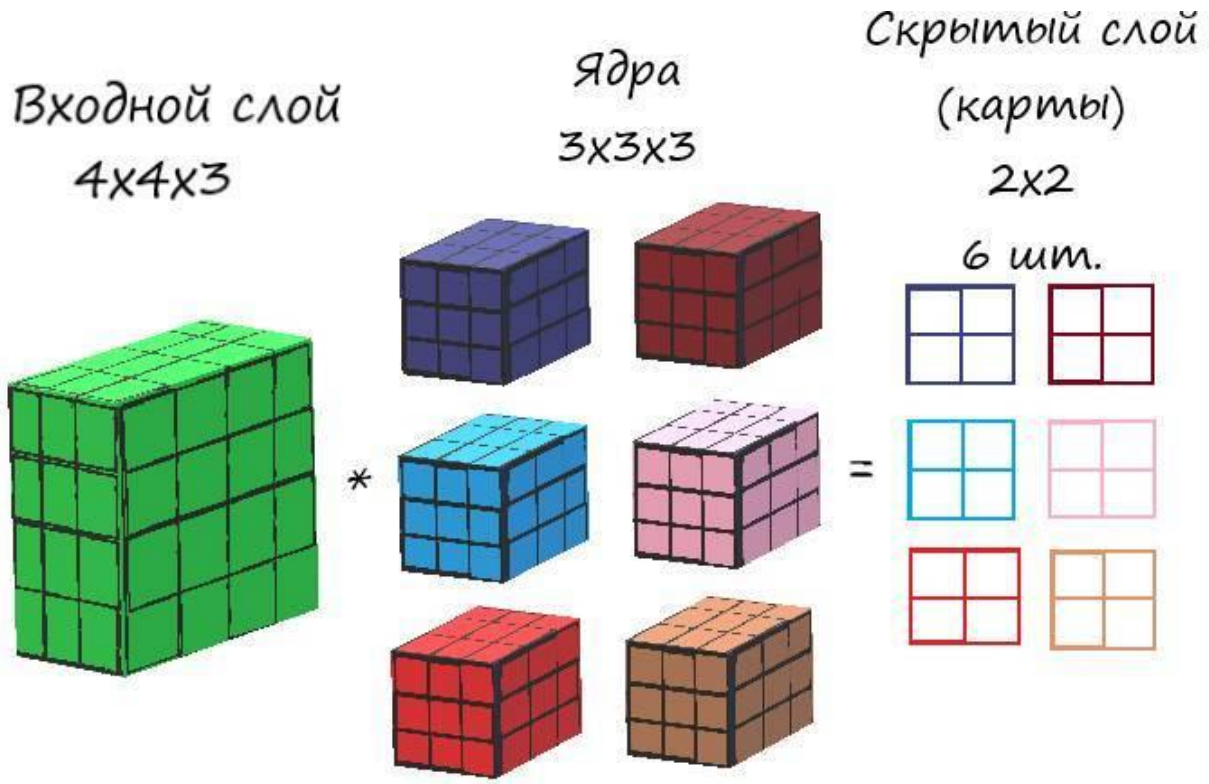
$$mck = m - mяд + 1 = 4 - 2 + 1 = 3$$
$$nck = n - nяд + 1 = 4 - 2 + 1 = 3$$

Так как мы будем рассматривать только квадратное изображение на входе, то количество строк в скрытом слое будет равно количеству столбцов.

Количество ядер свертки, может быть любым, не только одно. Но надо соблюдать некоторые правила. Так каждому ядру соответствуют свои значения в скрытом слое, не связанные с другими ядрами:



Частое явление – когда на входе трехмерный массив. Это происходит из-за того, что для обработки цветного изображения его необходимо разбить на субпиксели которые соответствуют своему цвету, а именно красный, зеленый, синий, в системе RGB. Результаты свертки в скрытом слое часто называют – **картами**:



Как видим прослеживается ряд правил, связанные с тем что если планируешь в скрытом слое получить определенное количество карт, то оно должно соответствовать количеству ядер свёртки. А также, каждое ядро должно состоять из определенного количества слоёв, которое соответствует количеству слоёв входных данных.

Каждый слой ядра сворачивается со всеми слоями входных данных, а результатом этих действий служит общая взвешенная сумма, которая помещается в отдельную карту в скрытом слое.

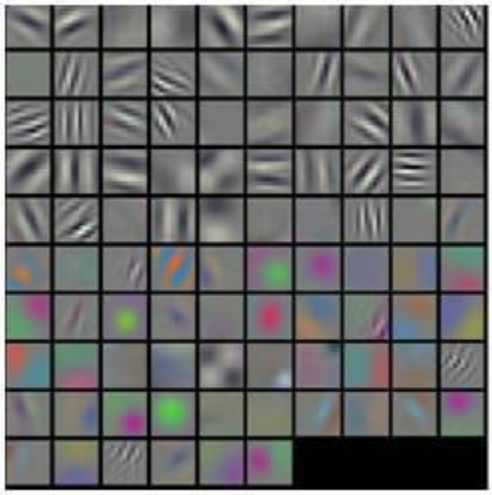
Соответственно, для следующего слоя, входные данные будут представляться в виде массива “2x2x6”:

А если мы захотим, чтобы на следующем слое было “8” карт, то размерность ядер была бы следующая – “2x2x6” или “1x1x6”, в количестве “8” штук (8x2x2x6 или 8x1x1x6). Откуда, размерность следующего слоя будет – “1x1x8” или “2x2x8” соответственно.

Что происходит при обучении

Поговорим о том, что свертка на самом деле делает на следующих слоях и как происходит обучение.

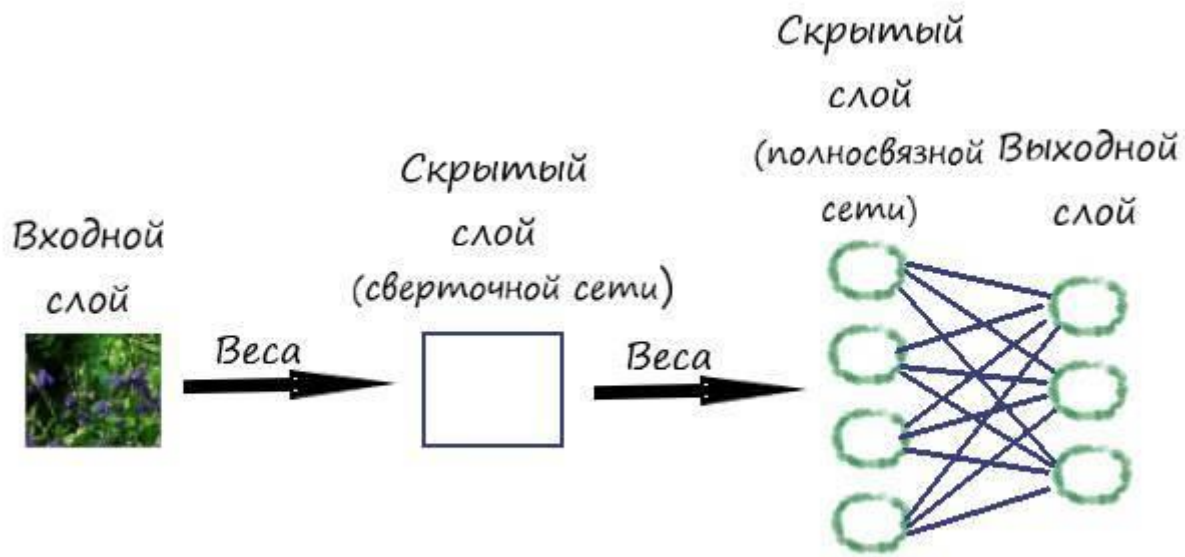
Каждый фильтр можно рассматривать как идентификатор какого-то свойства. В фильтрах, стоящих сразу после входного слоя, активизируются простые свойства, а именно прямые границы, простые фигуры и кривые:



Как это работает?

Если прикрепить полносвязный слой в конец свёрточной сети, то мы сможем идентифицировать вводные данные, путем построения N-пространственного вектора в выходном слое, где “N” – число классов, из которых программа будет выбирать нужный.

Структура такой простейшей сети:

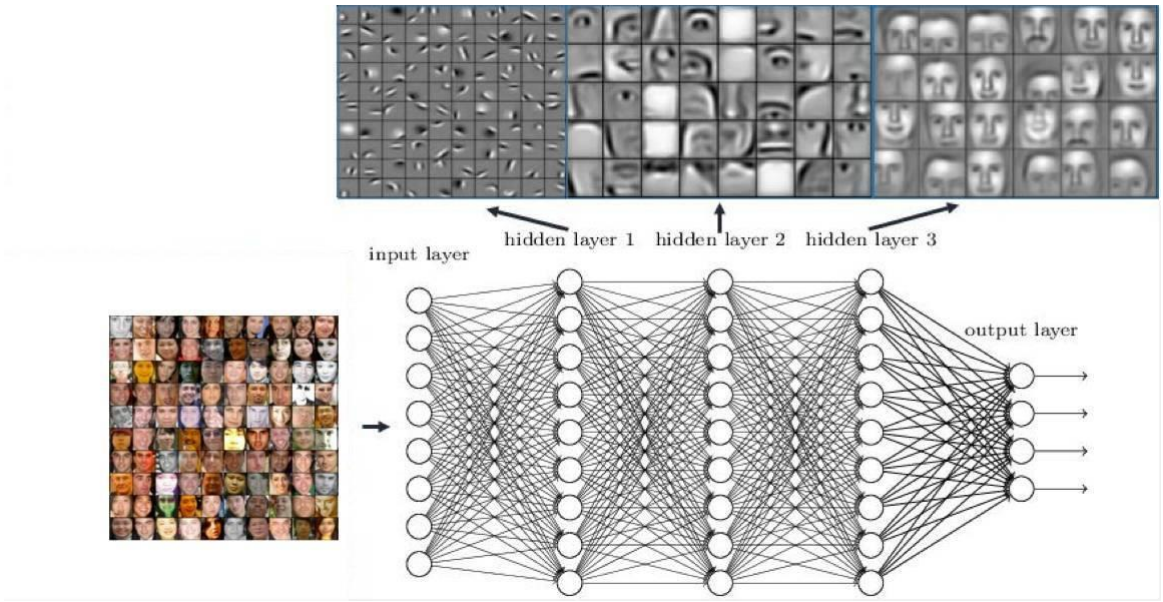


В выходном слое, количество классов задаем так же, как и ранее. Например, в программе по распознаванию цифр, у “N” будет значение “10”, потому что цифр всего “10”. Каждое число в

этом векторе представляет собой вероятность конкретного класса. Например, если результирующий вектор для программы распознавания цифр это [0 0,2 0,25 0,95 0 0 0 0 0,01], значит существует 20% вероятность, что на изображении “1”, 25% вероятность, что на изображение “2”, 95% вероятность – “3”, и 1% вероятность – “9”.

Обучение всей сети, будет выполняться всё теми же, уже нам хорошо известными методами – градиентным спуском и обратным распространением ошибки. Как это делается в полносвязных слоях мы хорошо знаем. Как происходит обновление весов в ядрах и распространение ошибки в свёрточных слоях, поговорим чуть позже.

Так вот, алгоритм схож с тем что мы делали ранее, но здесь при обучении в ядрах (весах в свёрточной сети), активируются некоторые свойства. Особенностью которых является иерархическая структура, чем ближе к выходу, тем более сложные свойства в них проявляются. Если проиллюстрировать это на примере распознавания лиц человека, то получим следующую картину:

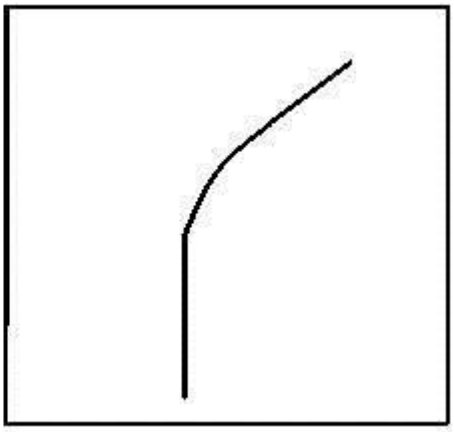


Замечаем, что в начале, в фильтрах активируются простые свойства, преимущественно наклонные линии, а ближе к выходу свойства уже представляют из себя объект распознавания.

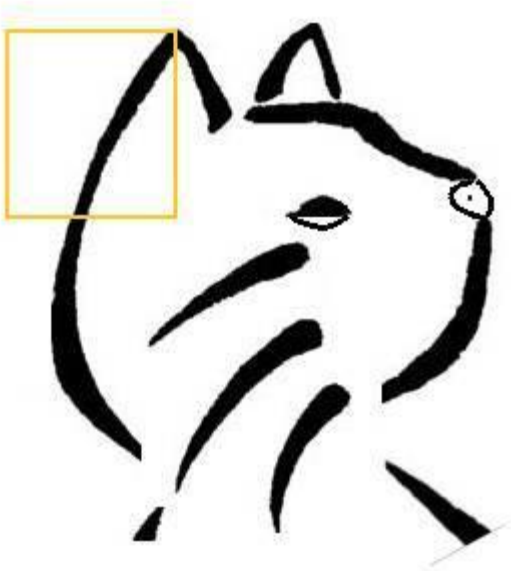
Попробую описать этот процесс на фильтрах входного слоя.

Все изображения в общем, имеют ряд простых характеристик – кривые, перекрестия, наклонные линии и так далее. Скажем, пусть наш первый фильтр будет размером 4x4x1, и он будет детектором кривых, будем считать, что изображение не цветное и у фильтра глубина 1. У фильтра пиксельная структура, в которой численные значения определяют форму свойств (в нашем случае кривой):

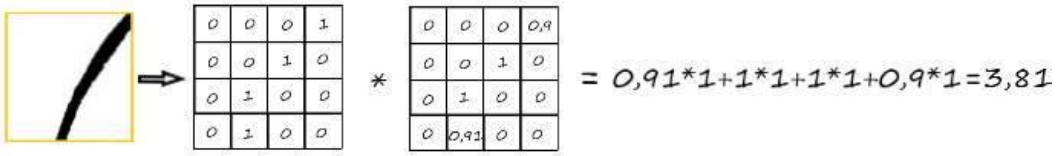
0	0	0	0,9
0	0	1	0
0	1	0	0
0	0,91	0	0



Когда у нас в левом верхнем углу вводного изображения есть фильтр, он производит умножение значений фильтра на значения пикселей в этой области. Давайте рассмотрим пример изображения, которому мы хотим присвоить класс, и установим фильтр в верхнем левом углу:

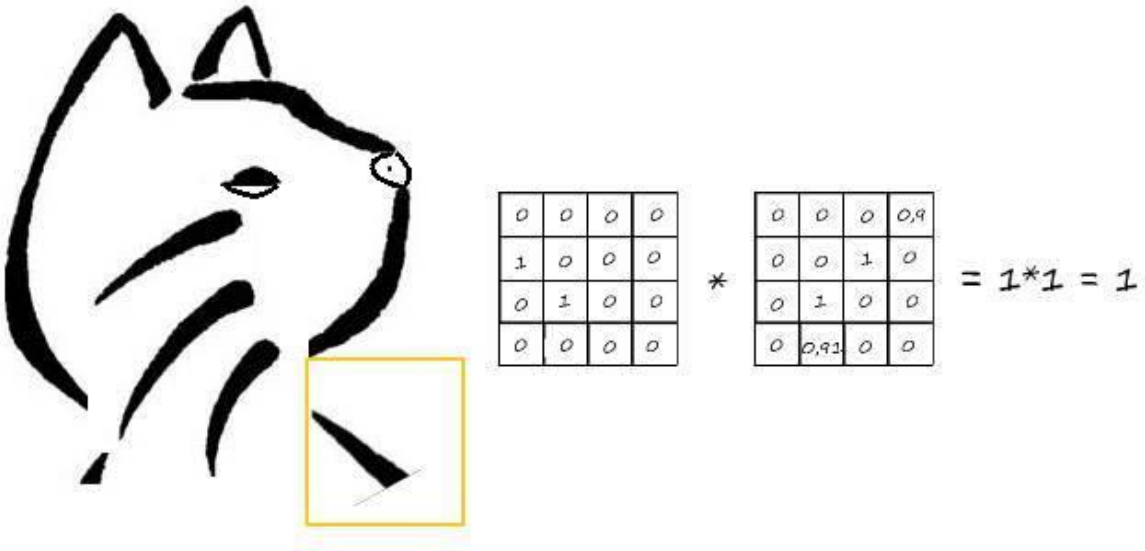


Умножаем значения фильтра на исходные значения пикселей изображения:



Как видим, если на вводном изображении есть форма напоминающая или повторяющая значения, которые представляет фильтр, то результатом будет большое значение (чем больше

они похожи, тем больше значение). Теперь давайте посмотрим, что произойдёт, когда мы переместим фильтр в крайнее правое положение:



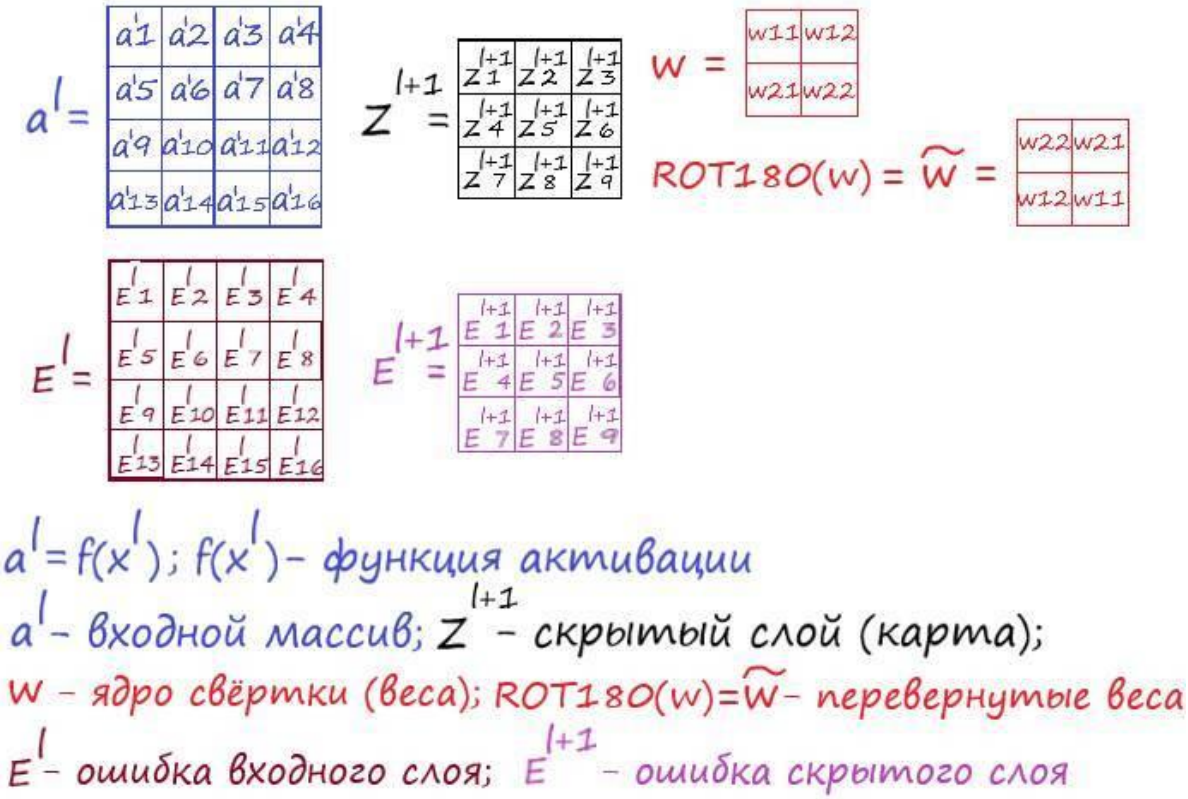
Так как, в новой области изображения мало общих черт с фильтром, то фильтр кривой линии мало мог что-либо здесь засечь. Здесь уже значение намного ниже, по сравнению с верхним левым углом изображения. Карта свойств, или вывод сверхточного слоя, в нашем самом простом случае, при наличии всего одного фильтра кривой свёртки, покажет области, в которых больше вероятности наличия кривых. Высокое значение показывает, что на изображении есть что-то похожее на кривую. Низкое значение говорит о том, что на этом участке изображения есть мало что напоминающее кривую линию.

Могут быть и другие фильтры для наклонных линий, кривых другой формы или просто прямых. Чем больше фильтров, тем больше глубина карты свойств, и тем больше информации мы имеем о вводном изображении.

Углубляясь дальше в свёрточную сеть, фильтры активирую более сложные структуры (например – глаза, нос, губы и т.д.), из которых при дальнейшем углублении активируются ещё более сложные структуры (например – лица).

Практика по свёртки значений изображения

Реализуем программною операцию свёртки. Но для начала выведем формулу по которой будем её осуществлять. Для этого проиллюстрируем все необходимые для этого входные данные:

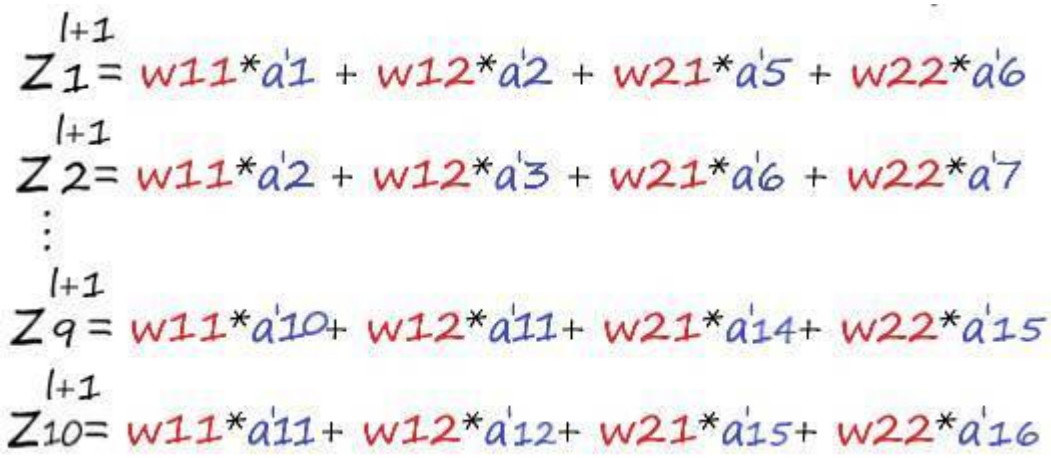


На основании уже имеющихся знаний, мы легко сможем выразить формулу самой свёртки:

$$Z^{l+1}_{i,j} = \sum_m \sum_n w_{m,n}\, a^l_{i+m,j+n}$$

$$Z^l_{i,j} = f(Z^{l+1}_{i,j})$$

Если подробней расписать для каждого элемента в карте, то получим:



Теперь запрограммируем вышесказанное.
Вводим данные для свёртки:

```
import numpy as np


                    #ВХОДНЫЕ ДАННЫЕ
m = 4 #Размерность входного массива(ДхШ)
k = 2 #Размерность ядра свертки (ДхШ)
m_k_1 = (m-k)+1 #Размерность одной карты скрытого слоя свертки (ДхШ)
stok_w = 2 #Число ядер свертки и соответственно количество карт срытого слоя (ДхШ)
stob_w = k*k #Количество элементов в ядре свертки между входным и скрытым слоями



x1 = np.arange(1,m*m+1).reshape(m,m)  #Входной 2D массив данных
w1 = np.arange(1,stok_w*stob_w +1).reshape(stok_w, k,k) #Ядра свертки (веса)
x2 = np.zeros((stok_w, m_k_1, m_k_1)) #Выходной массив (карты скрытого слоя)
#(s1, f1, f1) = x1.shape # Получаем размеры выходного массива
(s2, f2, f2) = x2.shape # Получаем размеры выходного массива
e2 = np.arange(1,s2*f2*f2+1).reshape(stok_w, m_k_1, m_k_1) #Ошибка скрытого слоя (размеры должны совпадать с картой этого слоя)
lr = 0.01 #Скорость обучения
print ('x1: ','\n', x1, '\n')
print ('w1: ','\n',w1, '\n')
```

Как говорилось ранее, мы будем рассматривать случаи с квадратными матрицами на входе (кол-во пикселей входного изображения по диагонали равны количеству пикселей по вертикали), а шаг прохода ядра во всех случаях будет равен единице. Так как размер строк и столбцов входной матрицы одинаков, то одного параметра для их выражения вполне достаточно (**m=4**). Размерности ядра свертки тоже квадратные и определяются одним параметром (**к=2**). Размерность карты скрытого слоя так же определена одним параметром (**m_k_1 = (m-k)+1**).

Не будем придерживаться строгости обозначения имен в программе с обозначениями в иллюстрации. Как видим, матрица, отвечающая за входное значение – обозначена как “**x1**”, ядро свертки как “**w1**”, а карта признаков как “**x2**”. При создании массива скрытого слоя, необходим трехмерный массив – “x2 = np.zeros((stok_w, m_k_1, m_k_1))”. Здесь значение “**stok_w**” – определяет третье измерение, а точнее количество карт, а значения “**m_k_1**” – количество строк и столбцов в картах.

Метод arange() – создаёт массив из последовательности чисел, аргументами этого метода является начало точки отсчета и конечное значение числа. Метод reshape(), определяет измерение массива, в нашем случае он переводит одномерный массив с элементами от 1 до 16, в двумерный массив 4x4.

Метод zeros() – позволяет создать массив с нулевыми значениями его элементов.

Массив ошибки скрытого слоя "**e2**" и значение скорости обучения, заданы для последующего их использования, сейчас мы их использовать не будем.

Сам алгоритм свёртки не представляет из себя чего-либо сложного:

```
# ОПЕРАЦИЯ СВЁРТКИ
print('Вход x1:\n', x1)
print('Вес w1:\n',w1)


for s in range(stok_w): # Цикл по количеству ядер свёртки
for h in range(m_k_1): # Цикл по количеству проходов ядра свёртки, по горизонтали от входных данных
for w in range(m_k_1): # Цикл по количеству проходов ядра свёртки, по вертикали от входных данных
x2 [s,h,w] = np.sum(x1[h:h+k, w:w+k] * w1[s]) # Сумма поэлементного умножения области входных данных с ядром


print('Сверточный слой x2:\n', x2, '\n')
```

Здесь в циклах последовательно проходим отдельным ядром свертки по ширине и высоте входного массива. Для определения локальной области нахождения ядра в входном массиве (для дальнейшего произведения их значений), используем срезы по строкам и столбцам в входном массиве.

А карта свойств скрытого слоя, формируется из суммы значений произведения ядра на область входного массива.

Отображение данных, при результате работы программы:

```
x1:
[[ 1 2 3 4]
[ 5 6 7 8]
[ 9 10 11 12]
[13 14 15 16]]


w1:
[[[1 2]
[3 4]]


[[5 6]
[7 8]]]
```

Отражение данных операции свертки:

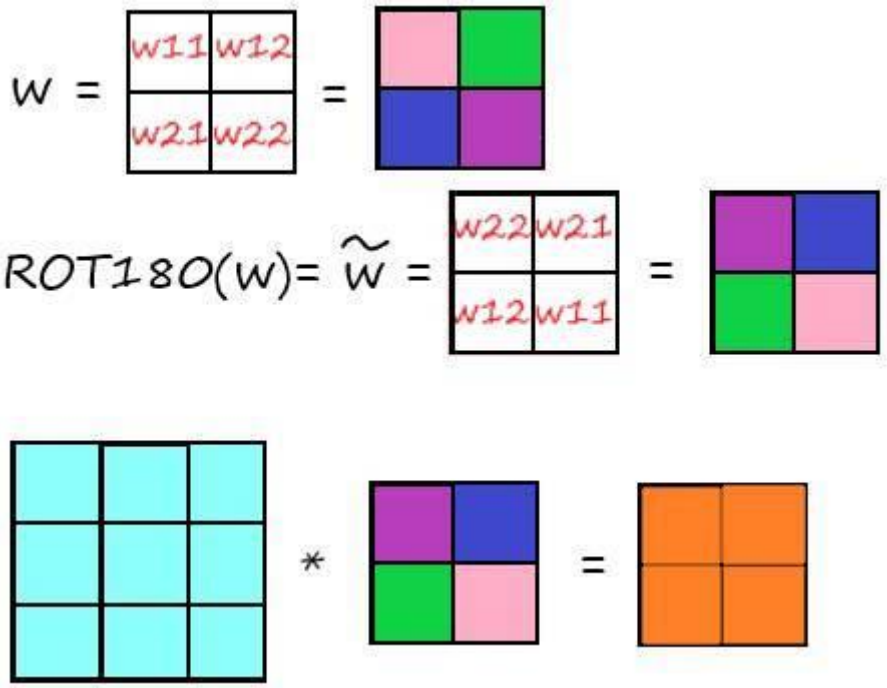
Сверточный слой x2:
[[[44. 54. 64.]
[84. 94. 104.]
[124. 134. 144.]]

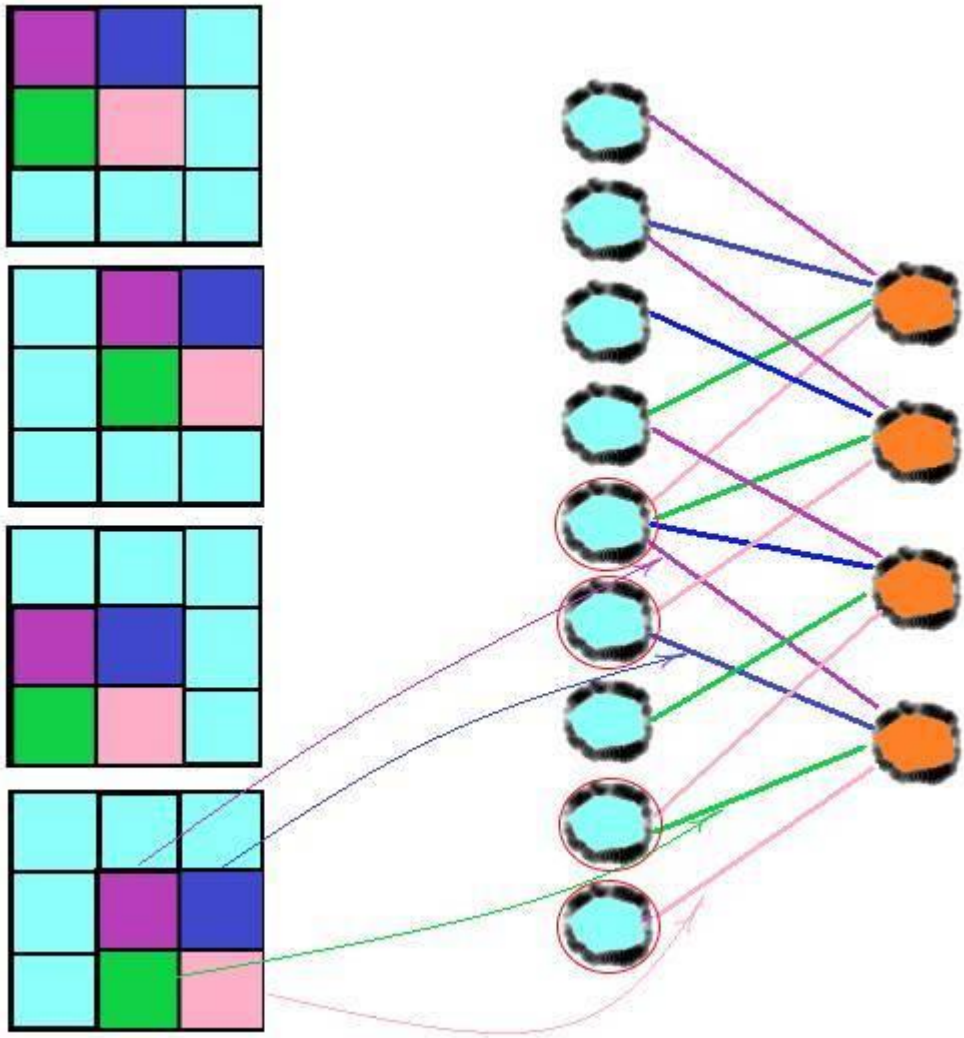
[[100. 126. 152.]
[204. 230. 256.]
[308. 334. 360.]]]

Практика обратного распространения ошибки и обновления весов ядра свёртки

Собственно, распространение ошибки, в свёрточных сетях, происходит аналогичным образом, как в полносвязной сети. Но все же, так как сеть не полносвязная, есть некоторые особенности.

Для лучшего понимания проиллюстрируем процесс прохода перевернутым ядром свёртки по входному массиву, предварительно упростив его до размерности 3x3, (для чего я перевернул ядро, станет ясно чуть позднее):

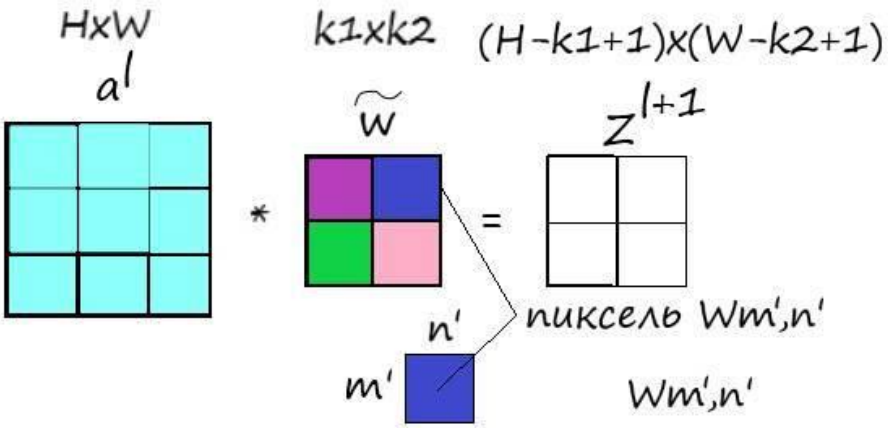




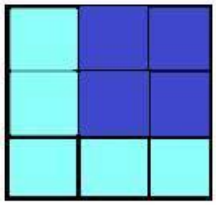
Замечаем, что при перевернутых весах, их произведение на элементы входа, отображаются зеркально. Например, относительно значения веса **w1,1**, в первом шаге вместо – **x1,1*w1,1**, будет – **x2,2*w1,1**. То есть, если смотреть относительно элементов ядра, то при произведении, локальная область входного массива переворачивается на 180 относительно не перевернутых весах в ядре.

Теперь попробуем на этих основаниях узнать, как обновляются весовые коэффициенты в ядрах свёртки.

Если выделить отдельный элемент веса в ядре свёртки и проследить с какими областями входного массива он взаимодействует (произведение входа на вес):



Область взаимодействия пикселя Wm',n' с входным массивом при свертке



Попробуем вывести формулу обновления весовых коэффициентов:

$$\frac{dE}{dw_{m',n'}} = \sum_{i=0}^{H-k1} \sum_{j=0}^{w-k2} \frac{dE}{dz_{i,j}^{l+1}} * \frac{dz_{i,j}^{l+1}}{dw_{m',n'}} =$$

$$= \sum_{i=0}^{H-k1} \sum_{j=0}^{w-k2} E_{i,j}^{l+1} * \frac{dz_{i,j}^{l+1}}{dw_{m',n'}}$$

$$\frac{dz_{i,j}^{l+1}}{dw_{m',n'}} =$$

$$= \frac{d(w_{0,0} * a^{l}_{i+0,j+0} + \ldots + w_{m',n'} * a^{l}_{i+m',j+n'})}{dw_{m',n'}} =$$

так как те элементы матрицы входа " a^{l} ", которые не попадают в область пикселя $w_{m',n'}$ равны нулю, то:

$$= \frac{d(w_{m',n'} * a^{l}_{i+m',j+n'})}{dw_{m',n'}} = a^{l}_{i+m',j+n'}$$

Откуда получаем формулу обновления весовых коэффициентов в ядре:

$$\frac{dE}{dw_{m',n'}} = \sum_{i=0}^{H-k1} \sum_{j=0}^{w-k2} \frac{dE}{dz^{l+1}_{i,j}} * \frac{dz^{l+1}_{i,j}}{dw_{m',n'}} =$$

$$= \sum_{i=0}^{H-k1} \sum_{j=0}^{w-k2} E^{l+1}_{i,j} * a^{l}_{i+m',j+n'}$$

В ходе того что при вычислении градиента мы перевернули ядро, мы выяснили, что входные значения так же перевернуты относительно весовых коэффициентов. Исходя из чего, уравнение примет окончательный вид:

$$\frac{dE}{dw_{m',n'}} = E^{l+1}_{i,j} * ROT180(a^{l}_{i+m',j+n'})$$

Как обновлять весовые коэффициенты надеюсь мы хорошо знаем, но напомнить себе еще раз не помешает:

новый wij = старый wij – L(dE/dwij)
Распишем подробней нахождение градиента, относительно наших данных:

$$gradE_1 = \frac{dE}{dw_{m',n'}} = dE_1^{l+1} * a_{11}^{l} + dE_2^{l+1} * a_{10}^{l} +$$
$$+ dE_3^{l+1} * a_9^{l} + dE_4^{l+1} * a_7^{l} + dE_5^{l+1} * a_6^{l} +$$
$$+ dE_6^{l+1} * a_5^{l} + dE_7^{l+1} * a_3^{l} + dE_8^{l+1} * a_2^{l} +$$
$$+ dE_9^{l+1} * a_1^{l}$$

$$\vdots$$

$$gradE_{16} = \frac{dE}{dw_{m',n'}} = dE_1^{l+1} * a_{16}^{l} + dE_2^{l+1} * a_{15}^{l} +$$
$$+ dE_3^{l+1} * a_{14}^{l} + dE_4^{l+1} * a_{12}^{l} + dE_5^{l+1} * a_{11}^{l} +$$
$$+ dE_6^{l+1} * a_{10}^{l} + dE_7^{l+1} * a_8^{l} + dE_8^{l+1} * a_7^{l} +$$
$$+ dE_9^{l+1} * a_6^{l}$$

Программируем:

```
print('Вход x1:\n', x1)
print('Ошибка e2:\n', e2)
```

```
def Xrot_180(X_rot_180):
#for s in range(stok_w):
X_rot_180 = np.fliplr(X_rot_180)
X_rot_180 = np.flipud(X_rot_180)
return X_rot_180
```

```
for s in range(stok_w):  # Цикл по количеству ядер свёртки
#for h in range(m-(m_k_1)+1):
for h in range(k):  # Цикл по количеству элементов ядра свёртки, по горизонтали
```

```
#for w in range(m-(m_k_1)+1):
for w in range(k): # Цикл по количеству элементов ядра свёртки, по вертикали
w1[s, h, w] = np.sum(e2[s] * Xrot_180(x1[h:h+m_k_1, w:w+m_k_1])) # Сумма поэлементного умножения области входных данных
#и ошибки скрытого слоя



print('Обновленные веса w1:\n', w1)
```

Сразу покажем результат работы программы:

```
Вход x1:
[[ 1 2 3 4]
[ 5 6 7 8]
[ 9 10 11 12]
[13 14 15 16]]
Ошибка e2:
[[[ 1 2 3]
[ 4 5 6]
[ 7 8 9]]



[[10 11 12]
[13 14 15]
[16 17 18]]]
Обновленные веса w1:
[[[ 192 237]
[ 372 417]]



[[ 678 804]
[1182 1308]]]
```

Здесь, при обновлении весов, для наглядности, я не стал вычитать градиент из старого значения.

В программе, в функции “Xrot_180()” осуществляется переворот локальной области матрицы входа, которая в качестве аргумента передается в эту функцию. Метод “fliplr()”, в теле функции, меняет местами столбцы матрицы, по принципу – конечный столбец становится начальным, предпоследний вторым и так далее:

```
>>> A = np.diag([1.,2.,3.])
>>> print(A)
array([[ 1., 0., 0.],
[ 0., 2., 0.],
```

```
[ 0., 0., 3.]])
>>> np.fliplr(A)
>>> print(A)
array([[ 0., 0., 1.],
[ 0., 2., 0.],
[ 3., 0., 0.]])
```

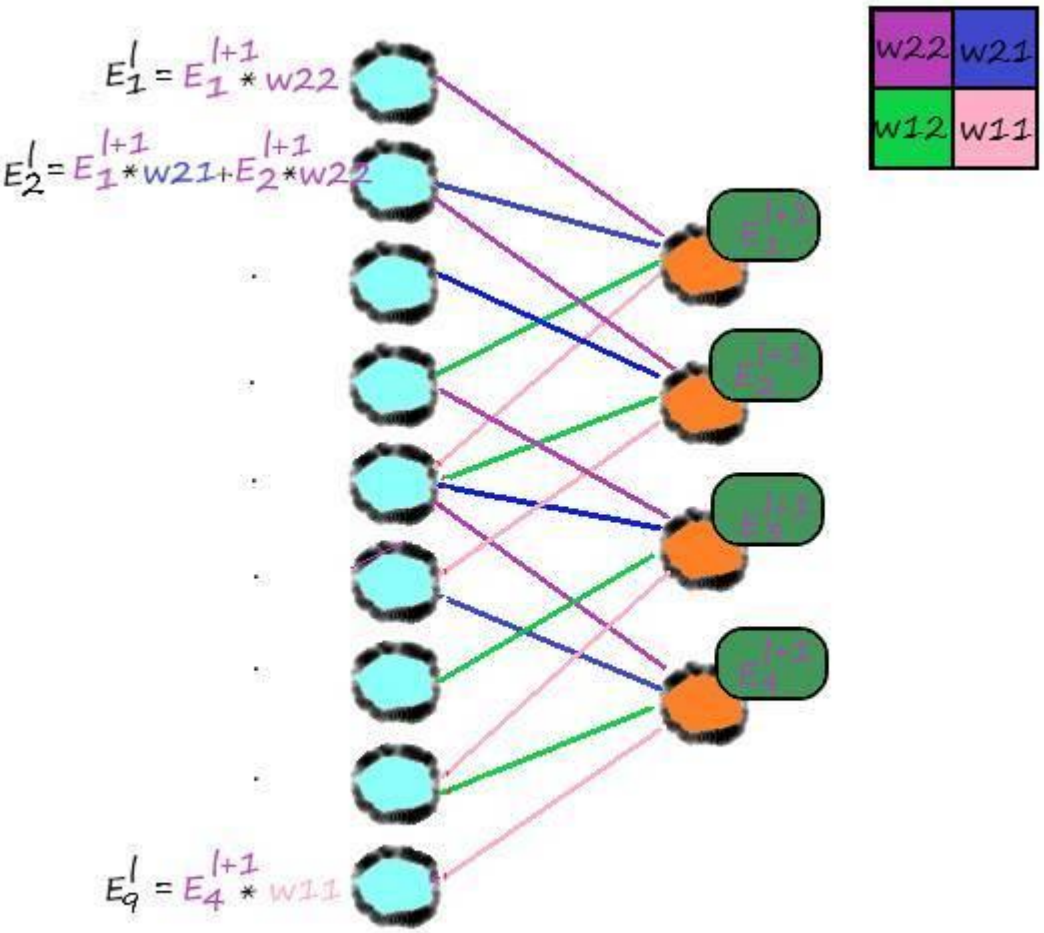
А метод “flipud()”, по тому же принципу, переставляет местами строки в массиве:

```
>>> print(A)
array([[ 0., 0., 1.],
[ 0., 2., 0.],
[ 3., 0., 0.]])
>>> np.flipud(A)
>>> print(A)
array([[ 3., 0., 0.],
[ 0., 2., 0.],
[ 0., 0., 1.]])
```

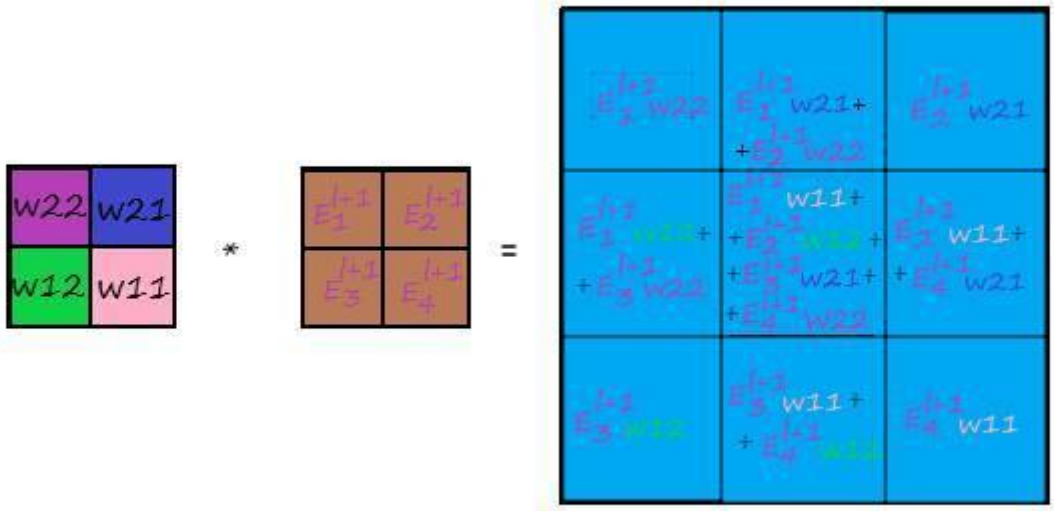
Таким образом, мы переворачиваем на 180 область входного массива.

После чего, в циклах осуществляем произведение матрицы ошибки, на каждой карте в отдельности, на локальную область входного массива, перевернутого с помощью функции “Xrot_180()” на 180. В функцию, как уже не однократно говорилось, передаем локальную область входного массива, для чего используем аппарат языка Python – “срезы”.

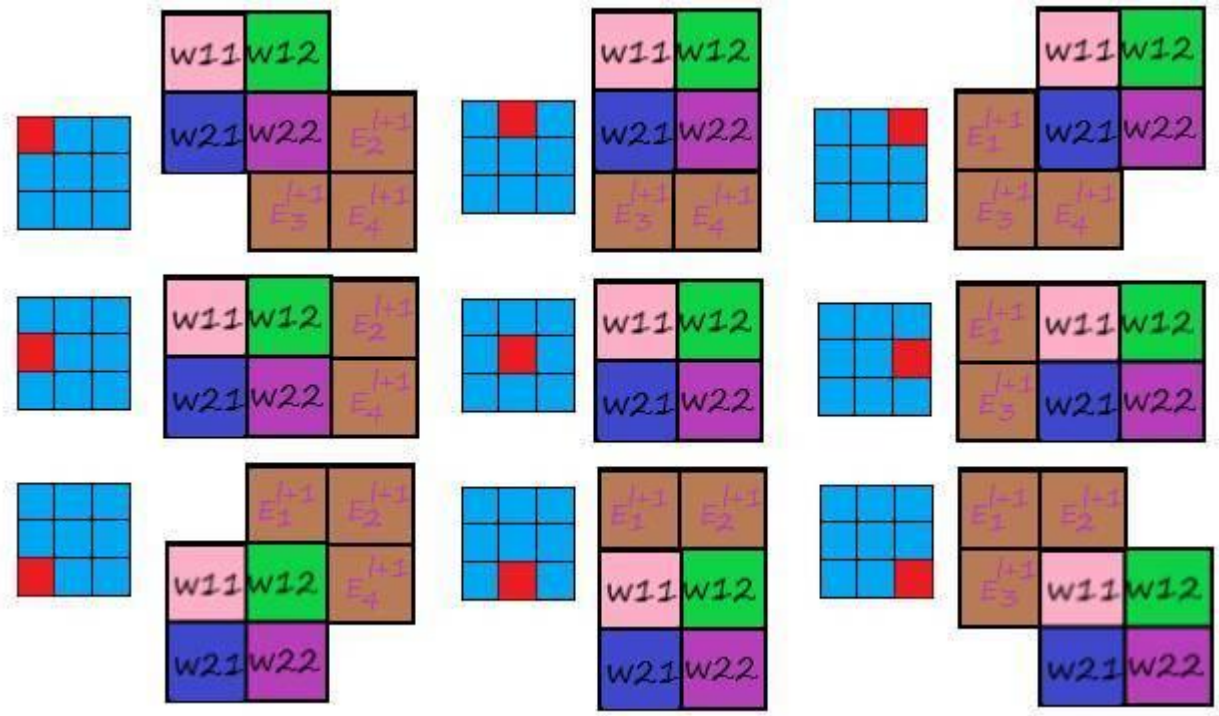
Возникает ещё один вопрос – как осуществляется обратное распространение ошибки в свёрточных сетях? Будем действовать по аналогии с тем, как мы действовали с полносвязными сетями. То есть, величина ошибки на нейронах входного слоя, будет определяться произведением ошибки нейронов и веса, с которым он связан относительно входа:



То есть, произведение связанного веса перевёрнутого ядра и ошибки скрытого слоя, даёт матрицу ошибок входного слоя:



Если отобразить это в виде прохода по матрице ошибок скрытого слоя:



Это очень наглядная иллюстрация. Теперь становится понятно, зачем мы перевернули ядро свертки при вычислении градиента.

Теперь представим всё это в математической форме:

$$\frac{dE}{da_j^l} = \sum_i \sum_j \frac{dE}{dz_i^{l+1}} \frac{dz_i^{l+1}}{da_j^l} = \sum_i \sum_j E_i^{l+1} \frac{dz_i^{l+1}}{da_j^l}$$

$$= \sum_i \sum_j E_i^{l+1} \frac{d(f(a_j^l w_{i,j}))}{da_j^l} = \sum_i \sum_j E_i^{l+1} \frac{d(f(a_j^l w_{i,j}))(a_j^l w_{i,j}))}{da_j^l} =$$

$$= E_i^{l+1} f'(x_j) w_{i,j} = \boxed{E_i^{l+1} f'(x_j)\, ROT180(w_{i,j})}$$

Соответственно, для наших первых двух и последнего входных значений ошибки:

$$E1^l = \frac{dE}{da1^l} = E1^{l+1} * w22$$

$$E2^l = \frac{dE}{da2^l} = E1^{l+1} * w21 + E2^{l+1} * w22$$

.

.

.

$$E16^l = \frac{dE}{da16^l} = E16^{l+1} * w11$$

Ну что же, теперь запрограммируем алгоритм обратного распространения ошибки:

```
# ОБРАТНОЕ РАСПРОСТРОНЕНИЕ ОШИБКИ
# Функция переворота матрицы весов на 180
e1 = np.zeros((stok_w, m, m)) #Ошибка входного слоя
def rot_180(W_rot_180):
for s in range(stok_w):
W_rot_180[s] = np.fliplr(W_rot_180[s])
W_rot_180[s] = np.flipud(W_rot_180[s])
return W_rot_180
```

```
rot180_w1 = rot_180(w1)
print('Перевернутые веса rot180_w1 :\n', rot180_w1)


# Создадим матрицу размера – для прогона весов как показано на слайде
e2_temp = np.zeros((stok_w, m+k-1, m+k-1))
# Поместим в центр ошибку на предыдущем слое
for s in range(stok_w):
e2_temp[s, k-1:m_k_1+k-1, k-1:m_k_1+k-1] = e2[s]


print('Матрица размера – для прогона весов как показано на слайде e2_temp:\n', e2_temp)


# Проходим по этой матрице перевернутыми ядрами
for s in range(stok_w):
for h in range(m):
for w in range(m):
#e1[s,h,w] = np.sum(lr * e2_temp[s, h:h+k, w:w+k] * rot180_w1[s])
e1[s,h,w] = np.sum(lr * e2_temp[s, h:h+k, w:w+k] * rot180_w1[s] * 1/1+np.exp(-x1[h, w]) * (1 –
1/1+np.exp(-x1[h, w])))
print('Ошибка до суммы – e1:\n', e1)


# Сумма ошибки (значения храним в 1м массиве)
for s in range(stok_w-1):
e1[0] = e1[s] + e1[s+1]
print('Ошибка после суммы – e1:\n', e1)
print('Ошибка входного слоя – e1:\n', e1[0])


Результат:


Перевернутые веса rot180_w1 :
[[[ 417 372]
[ 237 192]]


[[1308 1182]
[ 804 678]]]
Матрица размера – для прогона весов как показано на слайде e2_temp:
[[[ 0. 0. 0. 0. 0.]
[ 0. 1. 2. 3. 0.]
[ 0. 4. 5. 6. 0.]
[ 0. 7. 8. 9. 0.]
```

```
[ 0. 0. 0. 0. 0.]]



[[ 0. 0. 0. 0. 0.]
[ 0. 10. 11. 12. 0.]
[ 0. 13. 14. 15. 0.]
[ 0. 16. 17. 18. 0.]
[ 0. 0. 0. 0. 0.]]]
Ошибка до суммы – e1:
[[[ 2.46134113  6.28326256  10.50991501  7.11134185]
[ 11.4001816  30.69002458  42.87000333  26.73000045]
[ 28.32000006  67.23000001  79.41  46.35 ]
[ 26.04  58.95  66.84  37.53 ]]



[[ 68.34134113  155.05326256  169.80991501  96.48134185]
[ 206.3401816  460.26002458  499.98000333  277.56000045]
[ 262.14000006  579.42000001  619.14  340.92 ]
[ 189.12  410.22  435.12  235.44 ]]]
Ошибка после суммы – e1:
[[[ 70.80268227  161.33652511  180.31983002  103.5926837  ]
[ 217.7403632  490.95004915  542.85000665  304.2900009  ]
[ 290.46000012  646.65000002  698.55  387.27 ]
[ 215.16  469.17  501.96  272.97 ]]



[[ 68.34134113  155.05326256  169.80991501  96.48134185]
[ 206.3401816  460.26002458  499.98000333  277.56000045]
[ 262.14000006  579.42000001  619.14  340.92 ]
[ 189.12  410.22  435.12  235.44 ]]]
Ошибка входного слоя – e1:
[[ 70.80268227  161.33652511  180.31983002  103.5926837  ]
[ 217.7403632  490.95004915  542.85000665  304.2900009  ]
[ 290.46000012  646.65000002  698.55  387.27 ]
[ 215.16  469.17  501.96  272.97 ]]
```

Видим схожую функцию с переворотом области массива входных данных – переворота весов “ rot_180()”. Здесь ничего принципиально нового, так же в теле функции переставляем столбцы и строки.

Далее, создаем массив “e2_temp” для прохода по нему ядрами согласно нашему слайду. С помощью “срезов”, помещаем в его центр массив ошибок скрытого слоя – “e2_temp[s, k-1:m_k_1+k-1, k-1:m_k_1+k-1] = e2[s]”.

После чего, в циклах производим само действие обратного распространения ошибки. Здесь, для лучшего понимания, я сразу нахожу ошибки с учетом сигмоидальной функции активации входного и скрытого слоёв.

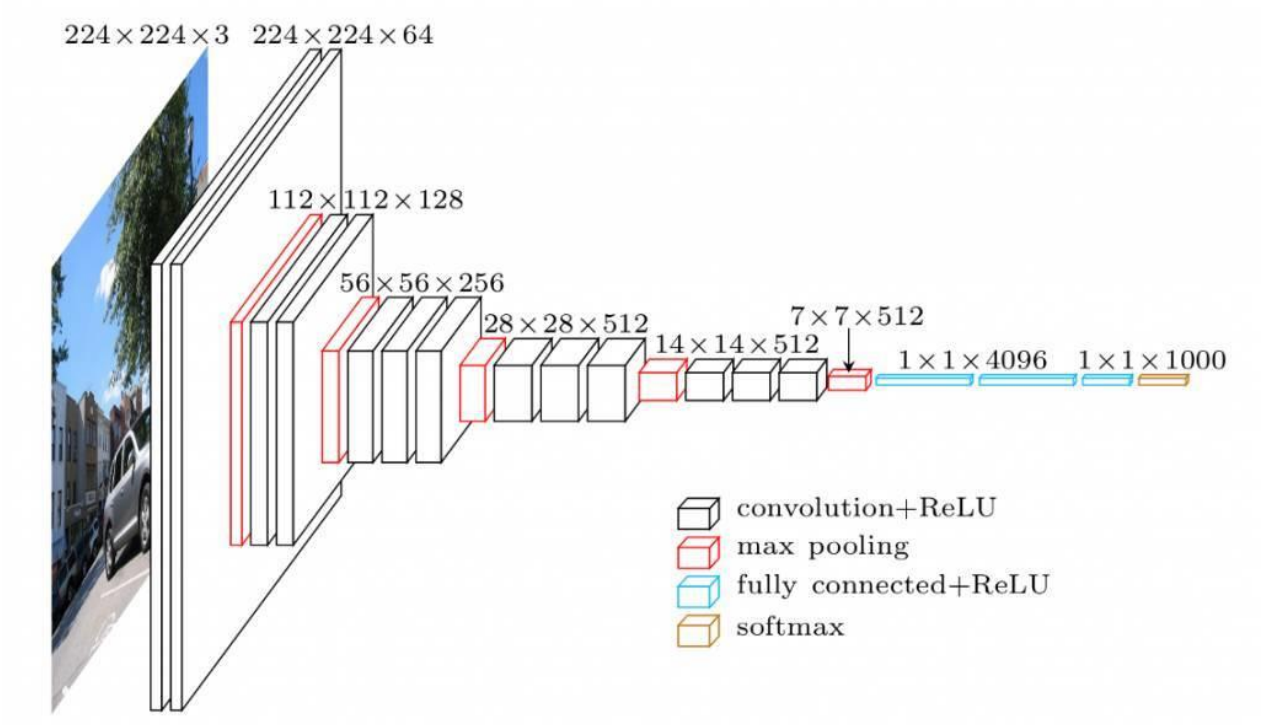
Затем, на выходе получаем два массива. Это связанно с тем, что я умножил массив ошибок скрытого слоя отдельно по своим картам, а нам нужна сумма всех ошибок. Я решил эту проблему очень просто – я просуммировал эти два массива, а результат сохранил в первом – “e1[0] = e1[s] + e1[s+1]”. После чего вывел на консоль окончательный результат – “print('Ошибка входного слоя – e1:\n', e1[0])”.

Глубокие нейронные сети и проблемы их реализации

Свёрточные нейронные сети часто называют – глубокими нейронными сетями. Такое название они получили неспроста. Как правило, свёрточные сети имеют более трех слоёв. А на практике, в большинстве случаев, и более десятка. Существуют сети с количеством слоёв более сотни.

Увеличение слоёв благотворно влияет на конечный результат.

Есть уже готовые наиболее популярные модели. Одна из самых простых таких глубокий сетей – VGG16, которая состоит из шестнадцати слоёв, крайние три из которых представляют собой полносвязные слои.



Наиболее частыми функциями активации в глубоких свёрточных сетях, являются “RELU” и “softmax”. Функция “softmax”, из-за своих свойств как правило активирует только выходной слой. Функция активации “RELU” тоже выбрана неспроста, она наименьшим образом подвержена затуханию градиента при обратном распространении ошибки, в отличие от той же сигмоиды, и всех тех функций активации, которые мы рассматривали ранее.

Большой сложностью при моделировании глубоких свёрточных сетей, является нехватка вычислительных ресурсов. Для оптимизации расчетов и в следствии более быстрого обучения сетей (в том числе и свёрточных), используются специализированные библиотеки машинного обучения. Одной из самых популярных является “TensorFlow” от корпорации “Google”. Кроме того, эти библиотеки зачастую могут выполнять расчеты на графических ускорителях, где значительно распараллеливаются необходимые операции, благодаря чему, добиваются ускоренного обучения в несколько десятков раз.

В данной книге я условился не использовать специализированные библиотеки для машинного обучения, у нас другая цель.

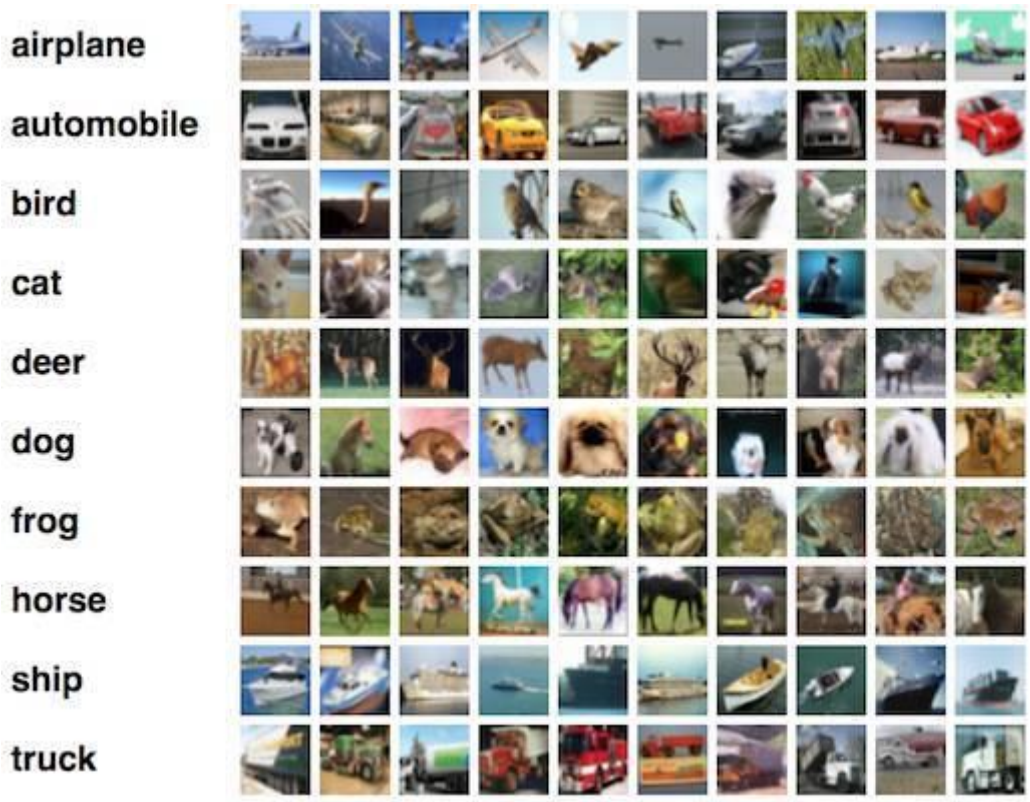
Наша цель – изучить принципы работы искусственных нейронных сетей.

Но не используя специализированные библиотеки, на обучение сети по типу VGG16 уйдут месяцы, а в реальной жизни, ваш компьютер скорее всего просто “зависнет”.

Выход из этого положения только один – значительно упрощать сеть. Главным образом снижая количество слоев и ядер свертки, подавая на вход не цветное изображение. В качестве входных данных будем использовать, всё тот же старый добрый “MNIST”.

Понимаю, что для “MNIST” нет смысла применять свёрточные сети, так как изображения в нём расположены по центру и имеют примерно одинаковые размеры. Но для проверки работоспособности наших алгоритмов он вполне даже сгодиться.

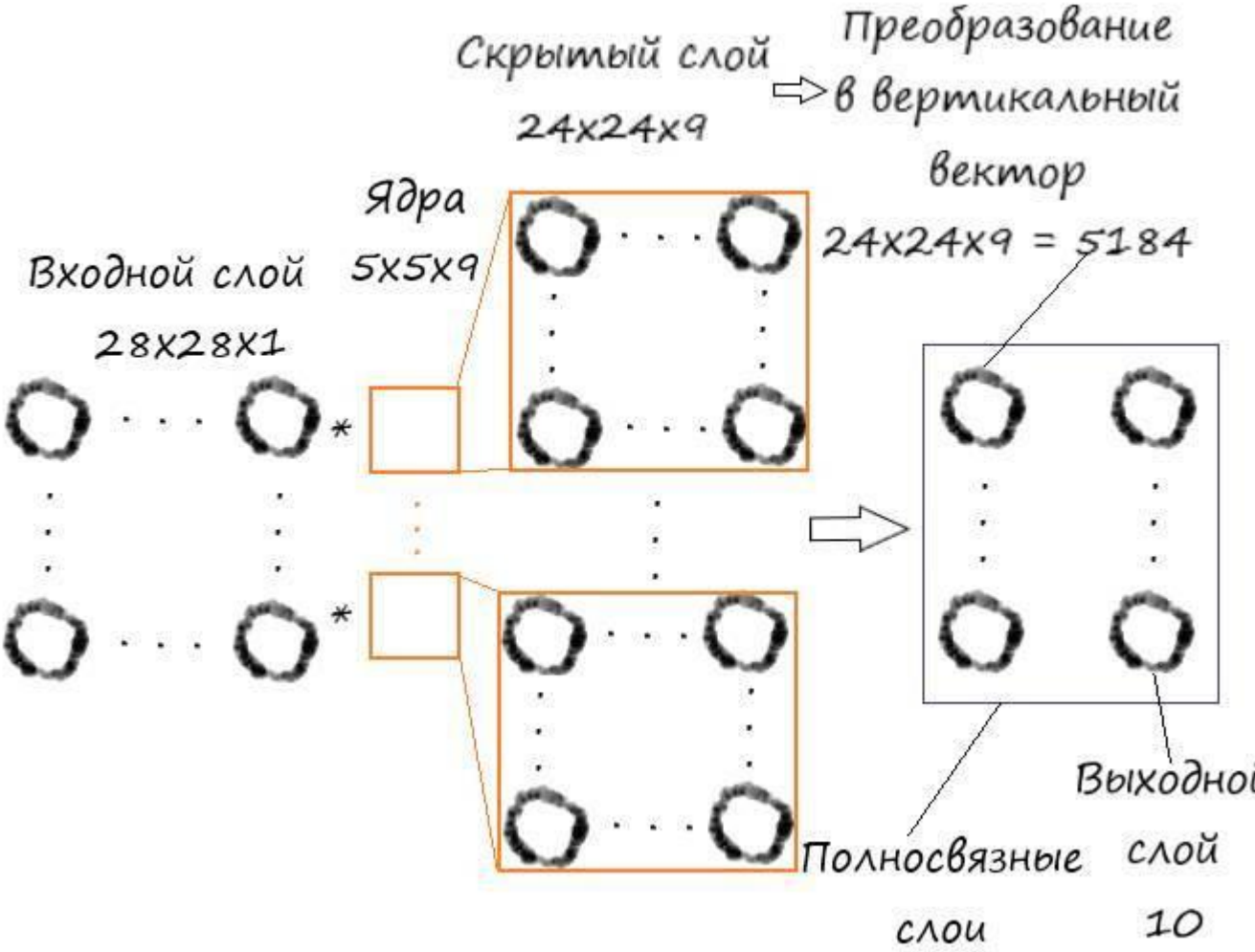
Существует еще один готовый набор данных – “CIFAR-10”. “CIFAR-10” – классификация небольших изображений по десяти классам: самолет, автомобиль, птица, кошка, олень, собака, лягушка, лошадь, корабль и грузовик:



Размер изображений в “CIFAR-10” немного больше по сравнению с “MNIST” – 32x32, но главным отличием от “MNIST”, является то, что у “CIFAR-10” изображения цветные, глубина у него на три пункта больше (32x32x3 у “CIFAR-10”, 28x28x1 у “MNIST”), что в нашем случае, значительно снизит время обучения. Поэтому мы не будем использовать этот набор данных.

Реализация сети с одним свёрточным слоем

Ну что же, за дело! Реализуем следующую структуру сети:



Так как количество слоев небольшое, то в качестве функции активации будем использовать – сигмоиду.

Внесем, по возможности, самые минимальные изменения в код программы по распознаванию цифр из набора “MNIST”.

Подключаем необходимые библиотеки и загружаем тренировочные данные:

```
import numpy as np
# библиотека для вывода на консоль массивов
import matplotlib.pyplot
# убедитесь, что участки находятся внутри этой записной книжки, а не внешнего окна
%matplotlib inline
#plt.show() # Вместо %matplotlib inline в других средах, не notebook
from time import time, sleep #Для замера времени выполнения функций
from tqdm import tqdm #Для вывода прогресса вычисления функций
# glob помогает выбрать несколько файлов, используя шаблоны
import glob
# помощник для загрузки данных из файлов изображений PNG
import scipy.misc
```

```
# Загрузить mnist тренировочные данные в формате CSV
training_data_file = open("MNIST_dataset/mnist_train.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data_file.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data_file.close() # закрываем фаел csv
```

Инициализируем параметры сети в классе:

```
# Определение класса нейронной сети
class neuron_Net:

# Инициализация весов нейронной сети
def __init__(self, input_num, hidden_num, output_num, learningrate): #констр.(входной слой, скрытый слой, выходной слой)
#РАЗМЕРНОСТЬ ВХОДНОГО МАССИВА И ПАРАМЕТРЫ ЯДЕР СВЕРТКИ
self.m = 28 #Размер входного массива(ДхШ)
self.k = 5 #Размер ядра (ДхШ)
self.m_k_1 = (self.m-self.k)+1 #Размер карты свойств скрытого слоя (ДхШ)
self.m_k = self.m-self.k
self.stok_w = 9 #Число ядер свертки
self.stob_w = self.k*self.k #Количество элементов 1го ядра свертки
self.m_k_stw = self.stok_w*self.m_k_1*self.m_k_1 #Общее кол-во элементов скрытого слоя
self.x1 = np.zeros((self.stok_w, self.m_k_1, self.m_k_1)) #Массив скрытого слоя

#Для вывода карт свойст скрытого слоя
self.hidden_outputs_image = np.zeros((self.stok_w, self.m_k_1, self.m_k_1))

# МАТРИЦЫ ВЕСОВ
self.weights = np.random.normal(0.0, pow(self.stob_w, -0.5), (self.stok_w, self.k, self.k))
self.weights_out = np.random.normal(0.0, pow(hidden_num, -0.5), (output_num, hidden_num))

# скорость обучения
self.lr = learningrate

# функция активации-функция сигмоида
self.activation_function = lambda x: scipy.special.expit(x)

pass
```

Для минимизации ошибок при обучении используем вычисление сигмоидальной функции из библиотеки “scipy” – “lambda x: scipy.special.expit(x)”.

Создаем метод обучения сети:

```
# обучение нейронной сети
def train(self, inputs_list, targets_list): # принемает входной список данных,targets ответы
# Преобразовать список входов в 2D массив
inputs_x = np.array(inputs_list.reshape(self.m , self.m)) # матрица числа
# Преобразовать список ответов в вертикальный массив. .T – транспонирование
targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов какое это число

# ВЫЧИСЛЕНИЕ СИГНАЛОВ ПО СЛОЯМ
# Вычислить сигналы в скрытом слое (карты сигналов скрытого слоя). СВЁРТКА!
for s in range(self.stok_w):
for h in range(self.m_k_1):
for w in range(self.m_k_1):
self.x1[s,h,w] = np.sum(inputs_x[h:h+self.k, w:w+self.k] * self.weights[s])

# вычислить сигналы, возникающие из скрытого слоя. сигмоида(сигнал скр.слоя)
#y1 = 1/(1+np.exp(-self.x1))
y1 = self.activation_function(self.x1)

# вычислить сигналы в окончательный выходной слой (матрица сигналов выходного слоя)
x2 = np.dot(self.weights_out, np.array(y1.flatten(), ndmin=2).T) # сигнал вых.слоя. y1.flatten() – преобразование карт скрытого слоя в вертикальный массив (5184 элемента)
# вычислить сигналы, исходящие из конечного выходного слоя. сигмоида(Xoutputs – сигнал вых.слоя)
#y2 = 1/(1+np.exp(-x2))
y2 = self.activation_function(x2)

# ВЫЧИСЛЕНИЕ ОШИБКИ ПО СЛОЯМ
# ошибка выходного слоя является (цель – фактическое)
E = -(targets_Y – y2)
# Ошибка скрытого слоя
E_hidden = np.dot(self.weights_out.T, E)
E_hidden = E_hidden.reshape(self.stok_w, self.m_k_1, self.m_k_1) # Преобразуем в 3D массив

# ОБНОВЛЕНИЕ ВЕСОВ ПО СЛОЯМ
```

```
# Меняем веса исходящие из скрытого слоя по каждой связи
self.weights_out -= self.lr * np.dot((E * y2 * (1.0 – y2)), np.transpose(np.array(y1.flatten(), ndmin=2).T))



# Меняем веса ядер свертки исходящие из входного слоя
for s in range(self.stok_w)
for h in range(self.k):
        for w in range(self.k):
#Сохраняем область входных данных в отдельную область
inputs_t = inputs_x[h:h+self.m_k_1, w:w+self.m_k_1]
# Переворачиваем отдельную область входных данныхна 180
inputs_t = np.fliplr(inputs_t)
inputs_t = np.flipud(inputs_t)
# Обновляем веса
self.weights[s, h, w] -= np.sum(E_hidden[s] * inputs_t * self.lr)



#Запоминаем карту свойст скрытого слоя для просмотра
self.hidden_outputs_image = y1



pass
```

Изменений, по сравнению с полносвязной сетью, как видите не много. Добавилась операция свёртки с использованием срезов (алгоритм мы вывели ранее) – “self.x1[s,h,w] = np.sum(inputs_x[h:h+self.k, w:w+self.k] * self.weights[s])”.

При вычислении сигналов выходного слоя – “x2 = np.dot(self.weights_out, np.array(y1.flatten(), ndmin=2).T)”, карту признаков “ y1”, трансформируем в одномерный вертикальный массив – “ np.array(y1.flatten(), ndmin=2).T”.

Для вычисления сигмоидальной функции активации, пользуемся библиотекой “scipy” (y1 = self.activation_function(self.x1) и y2 = self.activation_function(x2)).

Еще одно заметное изменение лежит в области обновления весов. Здесь, обновление весовых коэффициентов ядер свёртки, производится по алгоритму, который мы изучили чуть ранее.

Далее, следует метод прогона собственных значений через обученную сеть, после чего вбиваем необходимые параметры и передаем их для инициализации:

```
# МЕТОД ПРОГОНА СВОИХ ЗНАЧЕНИЙ ПО СЕТИ
# запросить нейронную сеть
def query(self, inputs_list): # Функция прогонки по слоям своих данных. Принемает свой набор тестовых данных
# Преобразовать список входов в 2D массив
inputs_x = np.array(inputs_list.reshape(self.m , self.m)) # матрица числа
# Вычислить сигналы в скрытом слое (карты сигналов скрытого слоя). СВЁРТКА!
for s in range(self.stok_w):
for h in range(self.m_k_1):
for w in range(self.m_k_1):
self.x1[s,h,w] = np.sum(inputs_x[h:h+self.k, w:w+self.k] * self.weights[s])
```

```
    # вычислить сигналы, возникающие из скрытого слоя. сигмоида(сигнал скр.слоя)
    #y1 = 1/(1+np.exp(-self.x1))
    y1 = self.activation_function(self.x1)



    # вычислить сигналы в окончательный выходной слой (матрица сигналов выходного слоя)
    x2 = np.dot(self.weights_out, np.array(y1.flatten(), ndmin=2).T) # сигнал вых.слоя. y1.flatten() – преобразование карт скрытого слояв вертикальный масив (5184 элемента)
    # вычислить сигналы, исходящие из конечного выходного слоя. сигмоида(Xoutputs – сигнал вых.слоя)
    #y2 = 1/(1+np.exp(-x2))
    y2 = self.activation_function(x2)



    return y2



    # количество входных, скрытых и выходных узлов
    data_input = 784
    data_hidden = 5184
    data_output = 10



    # скорость обучения
    learningrate = 0.05



    # Создать экземпляр нейронной сети
    n = neuron_Net(data_input, data_hidden, data_output, learningrate)
```

Значение – “5184” скрытого слоя, получается как произведения разрешений одной карты на их количество. Соответственно – ((m-k+1)*(m-k+1))*9=((28-5+1)*(28-5+1))*9 = 24*24*9=5184.

Затем, следует цикл обучения, здесь тоже ничего нового:

```
                    # ОБУЧЕНИЕ
    # Зададим количество эпох
    epochs = 5



    start = time()
```

```
    # Прогон по обучающей выборке
    for e in range(epochs):
    # Пройдите все записи в наборе тренировочных данных
    #for record in training_data_list:
    for i in tqdm(training_data_list, desc = str(e+1)): # tqdm – используем интерактив состояния прогресса вычисления
    # Получить входные данные числа
    all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    # Массив данных входа с масштабированием от 0,01 до 0,99
    inputs_x = (np.asfarray(all_values[1:])/ 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение



    # Получить целевое значение Y, (ответ – какое это число)
    targets_Y = int(all_values[0]) # перевод символов в int, 0 элемент – целевое значение



    # создать целевые выходные значения (все 0.01, кроме нужной метки, которая равна 0.99)
    targets_Y = np.zeros(data_output) + 0.01



    # Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
    targets_Y[int(all_values[0])] = 0.99



    n.train(inputs_x, targets_Y) # наш метод train – обучение нейронной сети



    pass
    pass



    time_out = time() – start
    print("Время выполнения: ", time_out, " сек" )
```

Загружаем тестовые данные и производим оценку качества сети:

```
    # Загрузить CSV-файл данных теста mnist в список
    test_data_file = open("mnist_dataset/mnist_test.csv", 'r') # 'r' – файл для чтения, а не для записи.
    test_data_list = test_data_file.readlines() # readlines() – читает все строки в файле в переменную test_data_list
    test_data_file.close() # закрываем фаел csv
```

```
# ПРОВЕРКА ЭФФЕКТИВНОСТИ НЕЙРОННОЙ СЕТИ
# Массив показателей эффективности сети, изначально пустой
efficiency = []



# Прогон по всем записям в наборе тестовых данных
for i in test_data_list:
# Получить входные данные числа
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
# Правильный ответ, хранимый в нулевом индексе
targets_Y = int(all_values[0])
# Массив данных входа с масштабированием от 0,01 до 0,99
inputs_x = (np.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение



# Запросить ответ у сети
outputs_y = n.query(inputs_x) # Прогон по сети тестового значения из нашего файла
# Индекс самого высокого значения на матрице выхода, соответствует метке числа
label_y = np.argmax(outputs_y) # argmax возвращает индекс максимального элемента в выходном массиве



# Добавить правильный или неправильный список
if (label_y == targets_Y): # Если индекс макс. знач. на выходе = целевому значению (0 индекс массива данных)
# Если ответ сети соответствует целевому значению, добавляем 1 в конец массива показателей эффективности
efficiency.append(1)
else:
# Если ответ сети не соответствует целевому значению, добавляем 0 в конец массива показателей эффективности
efficiency.append(0)



pass
pass



# Вычислить оценку производительности. Доля правильных ответов
efficiency_map = np.asarray(efficiency) # asarray – преобразование списка в массив



print ('Производительность = ', (efficiency_map.sum() / efficiency_map.size)*100, '%') # Среднее арифметическое
```

У меня вышло:
Производительность = 88.16 %

Меньше чем сеть с полносвязными слоями, но как я уже говорил, добиться хороших результатов от сети с одним свёрточным и одним полносвязным слоем вряд ли удастся. А экономить на их количестве вынужденная мера, по причине экономии вычислительных ресурсов, впрочем, об этом мы тоже уже говорили. Главное, что наши алгоритмы работают!

Давайте загрузим собственные данные изображений и посмотрим, как хорошо сеть справляется с таким тестом:

```
# СОБСТВЕННЫЙ НАБОР ИЗОБРАЖЕНИЙ ДЛЯ ТЕСТА
my_dataset = [] # Для хранения данных и целевых значений


# Загрузить данные изображения в формате PNG, как установить тестовые данные
for image_file in glob.glob('my_image/_my_?.png'): # проход по файлам изобр. в папке my_images
#glob – из библиотеки glob, помогает выбрать сразу несколько файлов из папки


# Метка имени числа
label_y = int(image_file[-5:-4]) # хранит число в файле ?.png, -5 это ответ какое число '?'
# от -5 до -4 это будет символ '?', т.е метка числа


# Загрузить данные изображения из png файлов в массив
print ('Имя файла: ', image_file) # вывод пути и имени открытого файла


image_list = scipy.misc.imread(image_file, flatten=True) #“flatten=True” (“выровнять=True) ”– превращает
#изображения в простой массив чисел с плавающей запятой


# Изменить формат из 28x28 в список 784 значений, инвертировать значения
image_data = 255.0 – image_list.reshape(784) #преобразует массив из квадрата 28x28 в длинный список значений
#вычитание значений массива из 255.0. т.к обычно '0' означает черное, а '255' означает белое, но набор данных MNIST
#имеет инверсные значения, поэтому мы должны их перевернуть


# Вносим данные шкалу с диапазоном от 0 до 1
image_data = (image_data / 255.0 ) # массив данных входа с масштабированием от 0 до 1
```

```
    # Добавить метку числа и данные изображения к общему набору данных
    my_data = np.append(label_y, image_data)
    my_dataset.append(my_data)
    pass



    # ПРОВЕРКА СЕТИ НА СОБСТВЕННЫХ ДАННЫХ ИЗОБРАЖЕНИЙ
    # запись для тестирования
    room_choices = 8



    # Изображение участка
    matplotlib.pyplot.imshow(my_dataset[room_choices][1:].reshape(28,28), cmap='Greys', interpolation='None')
    # my_dataset– наш собственный набор тестовых данных



    # Правильный ответ в нулевом столбце
    correct_label = my_dataset[room_choices][0] # в строках номер файла из папки собственных данных, 0 толбец – ответ какое число



    # Входные значения
    input_x = my_dataset[room_choices][1:] # значение числа без ответа



    # Запросить сеть
    output_y = n.query(input_x) # прогоняем тестовую выборку по сети
    print (output_y) # вывод по выходу сети



    print('Минимальное значение: ', np.min(output_y)) # вывод мин знач элемента на выходе
    print('Максимальное значение: ', np.max(output_y)) # вывод макс знач элемента на выходе



    # Индекс самого высокого значения на выходе сети, соответствует метке
    number = np.argmax(output_y)
    print('\nЦелевое значение: ', number)



    # Вывод правильный или неправильный ответ
    if (number == correct_label): # если макс знач на выходе label = ответу (0 индекс из массива) correct_label
    print ('Угадал!!! :-)))')
    else:
```

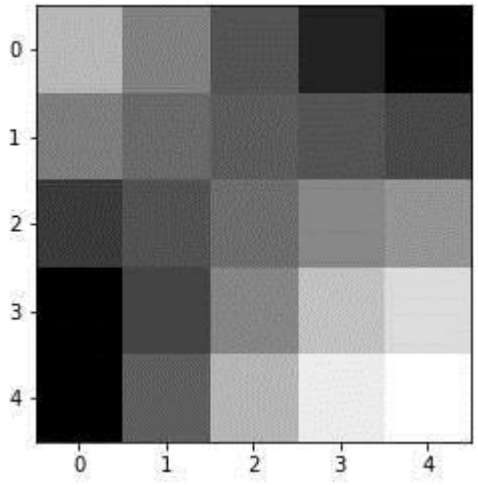
```
print ('Не угадал! :-(((')
pass
```

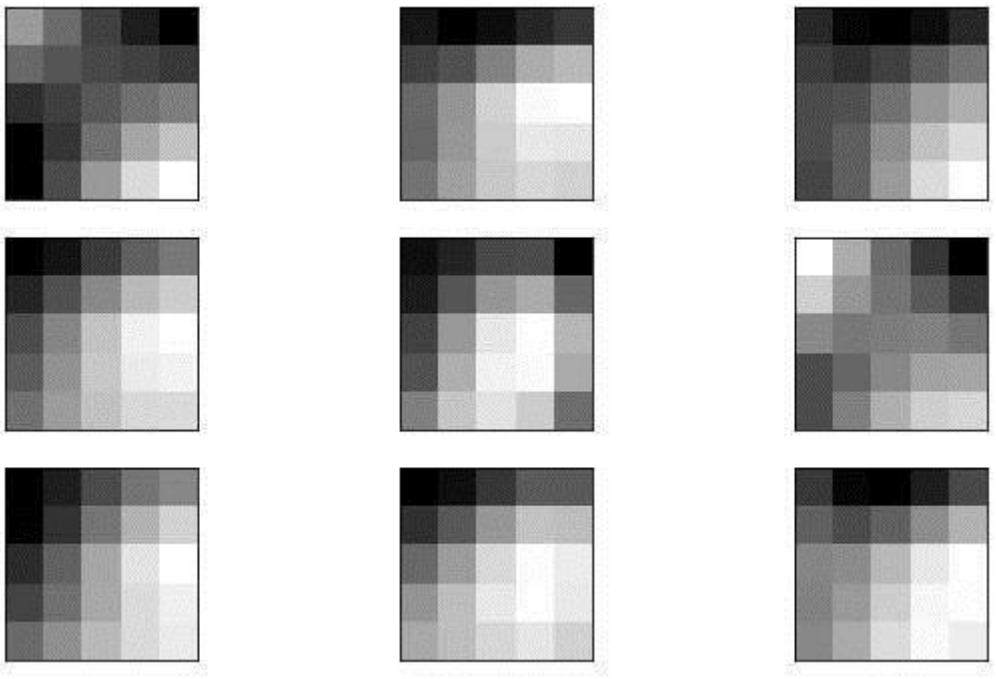
После чего визуализируем значения в ядрах:

```
# Данные изображения в ядрах свертки
# получить данные изображения с индексом "0". Для крупного масштаба.
image_y1  = n.weights[0]
# вывод данных изображения участка с индексом "0".
matplotlib.pyplot.imshow(image_y1,  cmap='Greys', interpolation='None')
```

```
fig = matplotlib.pyplot.figure(figsize=(10,6))
for j in range(9):
ax = fig.add_subplot(3,  3, j+1)
ax.imshow(n.weights[j],
cmap=matplotlib.cm.binary,  interpolation='none')
matplotlib.pyplot.xticks(np.array([]))
matplotlib.pyplot.yticks(np.array([]))
matplotlib.pyplot.show()
```

Результат данного участка кода:





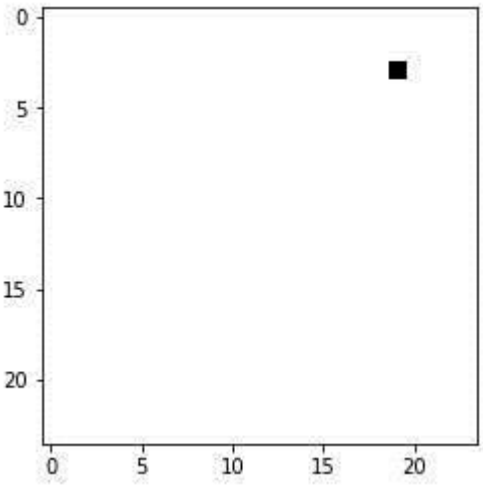
Первое изображение визуализирует ядро с индексом “0” крупным планом. Затем, следует изображения всех ядер в сети, с индексами от “0” до “8”.

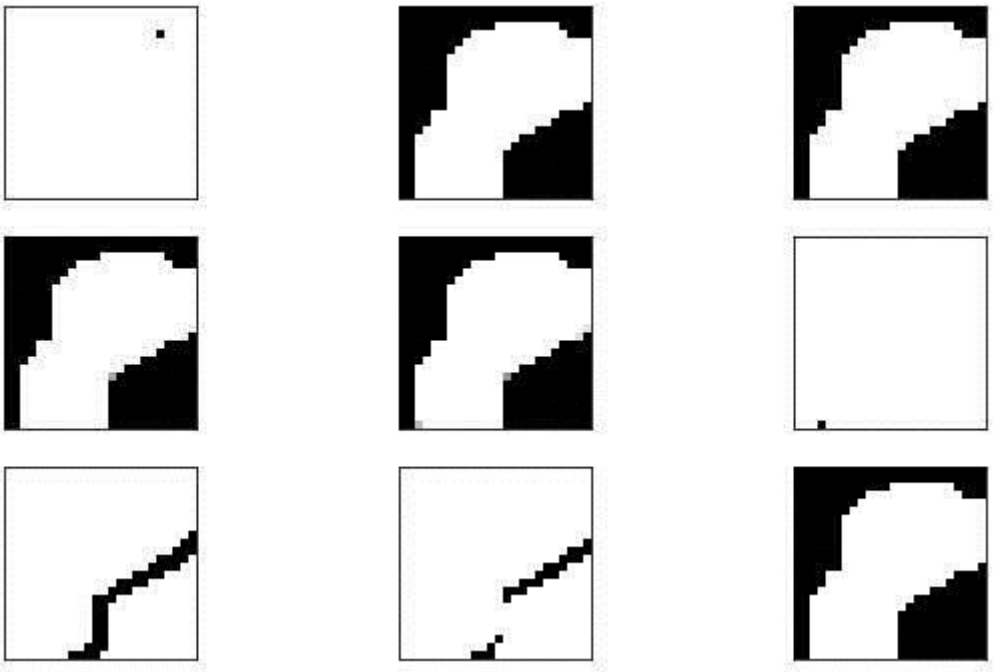
Для вывода подобной визуализации необходимо создать массив координатных сеток. С помощью функции – “fig.add_subplot(3, 3, j+1)”, мы создаем такой массив, где значения аргумента – разрешение одной сетки по вертикали, разрешение сетки по горизонтали, количество координатных сеток.

Видим, что на первом и единственном в данном случае слое свёртки, в фильтрах активизируются простые свойства, а именно наклонные границы, вертикальные и горизонтальные прямые.

Теперь посмотрим, что твориться в картах скрытого слоя:

```
# Данные изображения в признаках(в свертночном слое)
# получить данные изображения с индексом "0". Для крупного масштаба.
image_y1 = n.hidden_outputs_image[0] # Карта запомнила последний тренировочный пример при обучении сети!
# вывод данных изображения участка с индексом "0".
matplotlib.pyplot.imshow(image_y1, cmap='Greys', interpolation='None')


fig = matplotlib.pyplot.figure(figsize=(10,6))
for j in range(9):
ax = fig.add_subplot(3, 3, j+1)
ax.imshow(n.hidden_outputs_image[j],
cmap=matplotlib.cm.binary, interpolation='none')
matplotlib.pyplot.xticks(np.array([]))
matplotlib.pyplot.yticks(np.array([]))
matplotlib.pyplot.show()
```





На некоторых картах мы видим силуэт, напоминающий цифру “8”. Дело в том, что значения карт, с помощью переменной “hidden_outputs_image”, хранят в себе результаты после свёртки последнего тренировочного значения (если открыть файл mnist_train.csv, то можно убедиться, что крайним его значением будет цифра 8).

Полный код программы:

```
import numpy as np
# библиотека для вывода на консоль массивов
import matplotlib.pyplot
# убедитесь, что участки находятся внутри этой записной книжки, а не внешнего окна
%matplotlib inline
#plt.show() # Вместо %matplotlib inline в других средах, не notebook
from time import time, sleep #Для замера времени выполнения функций
from tqdm import tqdm #Для вывода прогресса вычисления функций
# glob помогает выбрать несколько файлов, используя шаблоны
import glob
# помощник для загрузки данных из файлов изображений PNG
import scipy.misc
```

```
# Загрузить mnist тренировочные данные в формате CSV
training_data_file = open("MNIST_dataset/mnist_train.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data_file.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data_file.close() # закрываем фаел csv



# Определение класса нейронной сети
class neuron_Net:



# Инициализация весов нейронной сети
def __init__(self, input_num, hidden_num, output_num, learningrate): #констр.(входной слой, скрытый слой, выходной слой)
#РАЗМЕРНОСТЬ ВХОДНОГО МАССИВА И ПАРАМЕТРЫ ЯДЕР СВЕРТКИ
self.m = 28 #Размер входного массива(ДхШ)
self.k = 5 #Размер ядра (ДхШ)
self.m_k_1 = (self.m-self.k)+1 #Размер карты свойств скрытого слоя (ДхШ)
self.m_k = self.m-self.k
self.stok_w = 9 #Число ядер свертки
self.stob_w = self.k*self.k #Количество элементов 1го ядра свертки
self.m_k_stw = self.stok_w*self.m_k_1*self.m_k_1 #Общее кол-во элементов скрытого слоя
self.x1 = np.zeros((self.stok_w, self.m_k_1, self.m_k_1)) #Массив скрытого слоя



#Для вывода карт свойст скрытого слоя
self.hidden_outputs_image = np.zeros((self.stok_w, self.m_k_1, self.m_k_1))



# МАТРИЦЫ ВЕСОВ
self.weights = np.random.normal(0.0, pow(self.stob_w, -0.5), (self.stok_w, self.k, self.k))
self.weights_out = np.random.normal(0.0, pow(hidden_num, -0.5), (output_num, hidden_num))



# скорость обучения
self.lr = learningrate



# функция активации-функция сигмоида
self.activation_function = lambda x: scipy.special.expit(x)
pass



# обучение нейронной сети
def train(self, inputs_list, targets_list): # принемает входной список данных,targets ответы
# Преобразовать список входов в 2D массив
```

```
inputs_x = np.array(inputs_list.reshape(self.m  , self.m)) # матрица числа
# Преобразовать список ответов в вертикальный массив. .T – транспонирование
targets_Y = np.array(targets_list,  ndmin=2).T  # матрица ответов какое это число



# ВЫЧИСЛЕНИЕ  СИГНАЛОВ  ПО СЛОЯМ
# Вычислить сигналы в скрытом слое (карты сигналов скрытого слоя). СВЁРТКА!
for s in range(self.stok_w):
for h in range(self.m_k_1):
for w in range(self.m_k_1):
self.x1[s,h,w]  = np.sum(inputs_x[h:h+self.k,  w:w+self.k]  * self.weights[s])



# вычислить сигналы, возникающие из скрытого слоя. сигмоида(сигнал скр.слоя)
#y1 = 1/(1+np.exp(-self.x1))
y1 = self.activation_function(self.x1)



# вычислить сигналы в окончательный выходной слой (матрица сигналов выходного слоя)
x2 = np.dot(self.weights_out, np.array(y1.flatten(), ndmin=2).T) # сигнал вых.слоя. y1.flatten() – преобразование карт скрытого слоя в вертикальный массив (5184 элемента)
# вычислить сигналы, исходящие из конечного выходного слоя. сигмоида(Xoutputs – сигнал вых.слоя)
#y2 = 1/(1+np.exp(-x2))
y2 = self.activation_function(x2)



# ВЫЧИСЛЕНИЕ  ОШИБКИ ПО СЛОЯМ
# ошибка выходного слоя является (цель – фактическое)
E = -(targets_Y  – y2)
# Ошибка скрытого слоя
E_hidden = np.dot(self.weights_out.T,  E)
E_hidden = E_hidden.reshape(self.stok_w,  self.m_k_1,  self.m_k_1) # Преобразуем в 3D массив



# ОБНОВЛЕНИЕ  ВЕСОВ ПО СЛОЯМ
# Меняем веса исходящие из скрытого слоя по каждой связи
self.weights_out  -= self.lr  *  np.dot((E  *  y2  *  (1.0  –  y2)),  np.transpose(np.array(y1.flatten(), ndmin=2).T))



# Меняем веса ядер свертки исходящие из входного слоя
for s in range(self.stok_w):
#for h in range(self.m-(self.m_k_1)+1):
for h in range(self.k):
#for w in range(self.m-(self.m_k_1)+1):
for w in range(self.k):
#Сохраняем область входных данных в отдельную область
inputs_t = inputs_x[h:h+self.m_k_1,  w:w+self.m_k_1]
```

```
    # Переворачиваем отдельную область входных данныхна 180
    inputs_t = np.fliplr(inputs_t)
    inputs_t = np.flipud(inputs_t)
    # Обновляем веса
    self.weights[s, h, w] -= np.sum(E_hidden[s] * inputs_t * self.lr)



    #Запоминаем карту свойст скрытого слоя для просмотра
    self.hidden_outputs_image  = y1
    pass



    # МЕТОД ПРОГОНА СВОИХ ЗНАЧЕНИЙ  ПО СЕТИ
    # запросить нейронную сеть
    def query(self, inputs_list): # Функция прогонки по слоям своих данных. Принемает свой набор тестовых данных
    # Преобразовать список входов в 2D массив
    inputs_x = np.array(inputs_list.reshape(self.m  , self.m)) # матрица числа
    # Вычислить сигналы в скрытом слое (карты сигналов скрытого слоя). СВЁРТКА!
    for s in range(self.stok_w):
    for h in range(self.m_k_1):
    for w in range(self.m_k_1):
    self.x1[s,h,w]  = np.sum(inputs_x[h:h+self.k,  w:w+self.k]  * self.weights[s])



    # вычислить сигналы, возникающие из скрытого слоя. сигмоида(сигнал скр.слоя)
    #y1 = 1/(1+np.exp(-self.x1))
    y1 = self.activation_function(self.x1)



    # вычислить сигналы в окончательный выходной слой (матрица сигналов выходного слоя)
    x2 = np.dot(self.weights_out, np.array(y1.flatten(), ndmin=2).T) # сигнал вых.слоя. y1.flatten() – преобразование карт скрытого слояв вертикальный масив (5184 элемента)
    # вычислить сигналы, исходящие из конечного выходного слоя. сигмоида(Xoutputs – сигнал вых.слоя)
    #y2 = 1/(1+np.exp(-x2))
    y2 = self.activation_function(x2)
    return y2



    # количество входных, скрытых и выходных узлов
    data_input  = 784
    data_hidden = 5184
    data_output = 10



    # скорость обучения
    learningrate  = 0.05
```

```
# Создать экземпляр нейронной сети
n = neuron_Net(data_input, data_hidden, data_output, learningrate)



# ОБУЧЕНИЕ
# Зададим количество эпох
epochs = 5



start = time()
# Прогон по обучающей выборке
for e in range(epochs):
# Пройдите все записи в наборе тренировочных данных
#for record in training_data_list:
for i in tqdm(training_data_list, desc = str(e+1)): # tqdm – используем интерактив состояния прогресса вычисления
# Получить входные данные числа
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
# Массив данных входа с масштабированием от 0,01 до 0,99
inputs_x = (np.asfarray(all_values[1:])/ 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение



# Получить целевое значение Y, (ответ – какое это число)
targets_Y = int(all_values[0]) # перевод символов в int, 0 элемент – целевое значение



# создать целевые выходные значения (все 0.01, кроме нужной метки, которая равна 0.99)
targets_Y = np.zeros(data_output) + 0.01



# Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
targets_Y[int(all_values[0])] = 0.99



n.train(inputs_x, targets_Y) # наш метод train – обучение нейронной сети
pass
pass



time_out = time() – start
print("Время выполнения: ", time_out, " сек" )
```

```
    # Загрузить CSV-файл данных теста mnist в список
    test_data_file = open("mnist_dataset/mnist_test.csv", 'r') # 'r' – файл для чтения, а не для записи.
    test_data_list = test_data_file.readlines() # readlines() – читает все строки в файле в переменную test_data_list
    test_data_file.close() # закрываем фаел csv



    # ПРОВЕРКА ЭФФЕКТИВНОСТИ НЕЙРОННОЙ СЕТИ
    # Массив показателей эффективности сети, изначально пустой
    efficiency = []



    # Прогон по всем записям в наборе тестовых данных
    for i in test_data_list:
    # Получить входные данные числа
    all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    # Правильный ответ, хранимый в нулевом индексе
    targets_Y = int(all_values[0])
    # Массив данных входа с масштабированием от 0,01 до 0,99
    inputs_x = (np.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение



    # Запросить ответ у сети
    outputs_y = n.query(inputs_x) # Прогон по сети тестового значения из нашего файла
    # Индекс самого высокого значения на матрице выхода, соответствует метке числа
    label_y = np.argmax(outputs_y) # argmax возвращает индекс максимального элемента в выходном массиве



    # Добавить правильный или неправильный список
    if (label_y == targets_Y): # Если индекс макс. знач. на выходе = целевому значению (0 индекс массива данных)
    # Если ответ сети соответствует целевому значению, добавляем 1 в конец массива показателей эффективности
    efficiency.append(1)
    else:
    # Если ответ сети не соответствует целевому значению, добавляем 0 в конец массива показателей эффективности
    efficiency.append(0)
    pass
    pass



    # Вычислить оценку производительности. Доля правильных ответов
    efficiency_map = np.asarray(efficiency) # asarray – преобразование списка в массив
```

```
print ('Производительность = ', (efficiency_map.sum() / efficiency_map.size)*100, '%') # Среднее арифметическое


# СОБСТВЕННЫЙ НАБОР ИЗОБРАЖЕНИЙ ДЛЯ ТЕСТА
my_dataset = [] # Для хранения данных и целевых значений


# Загрузить данные изображения в формате PNG, как установить тестовые данные
for image_file in glob.glob('my_image/_my_?.png'): # проход по файлам изобр. в папке my_images
#glob – из библиотеки glob, помогает выбрать сразу несколько файлов из папки


# Метка имени числа
label_y = int(image_file[-5:-4]) # хранит число в файле ?.png, -5 это ответ какое число '?'
# от -5 до -4 это будет символ '?', т.е метка числа


# Загрузить данные изображения из png файлов в массив
print ('Имя файла: ', image_file) # вывод пути и имени открытого файла


image_list = scipy.misc.imread(image_file, flatten=True) #“flatten=True” (“выровнять=True) ”– превращает
#изображения в простой массив чисел с плавающей запятой


# Изменить формат из 28x28 в список 784 значений, инвертировать значения
image_data = 255.0 – image_list.reshape(784) #преобразует массив из квадрата 28x28 в длинный список значений
#вычитание значений массива из 255.0. т.к обычно '0' означает черное, а '255' означает белое, но набор данных MNIST
#имеет инверсные значения, поэтому мы должны их перевернуть


# Вносим данные шкалу с диапазоном от 0 до 1
image_data = (image_data / 255.0 ) # массив данных входа с масштабированием от 0 до 1


# Добавить метку числа и данные изображения к общему набору данных
my_data = np.append(label_y, image_data)
my_dataset.append(my_data)


pass
```

```
# ПРОВЕРКА СЕТИ НА СОБСТВЕННЫХ ДАННЫХ ИЗОБРАЖЕНИЙ
# запись для тестирования
room_choices = 8



# Изображение участка
matplotlib.pyplot.imshow(my_dataset[room_choices][1:].reshape(28,28), cmap='Greys', interpolation='None')
# my_dataset– наш собственный набор тестовых данных



# Правильный ответ в нулевом столбце
correct_label = my_dataset[room_choices][0] # в строках номер файла из папки собственных данных, 0 толбец – ответ какое число



# Входные значения
input_x = my_dataset[room_choices][1:] # значение числа без ответа



# Запросить сеть
output_y = n.query(input_x) # прогоняем тестовую выборку по сети
print (output_y) # вывод по выходу сети



print('Минимальное значение: ', np.min(output_y)) # вывод мин знач элемента на выходе
print('Максимальное значение: ', np.max(output_y)) # вывод макс знач элемента на выходе



# Индекс самого высокого значения на выходе сети, соответствует метке
number = np.argmax(output_y)
print('\nЦелевое значение: ', number)



# Вывод правильный или неправильный ответ
if (number == correct_label): # если макс знач на выходе label = ответу (0 индекс из массива) correct_label
print ('Угадал!!! :-))))')
else:
print ('Не угадал! :-(((')



pass
```

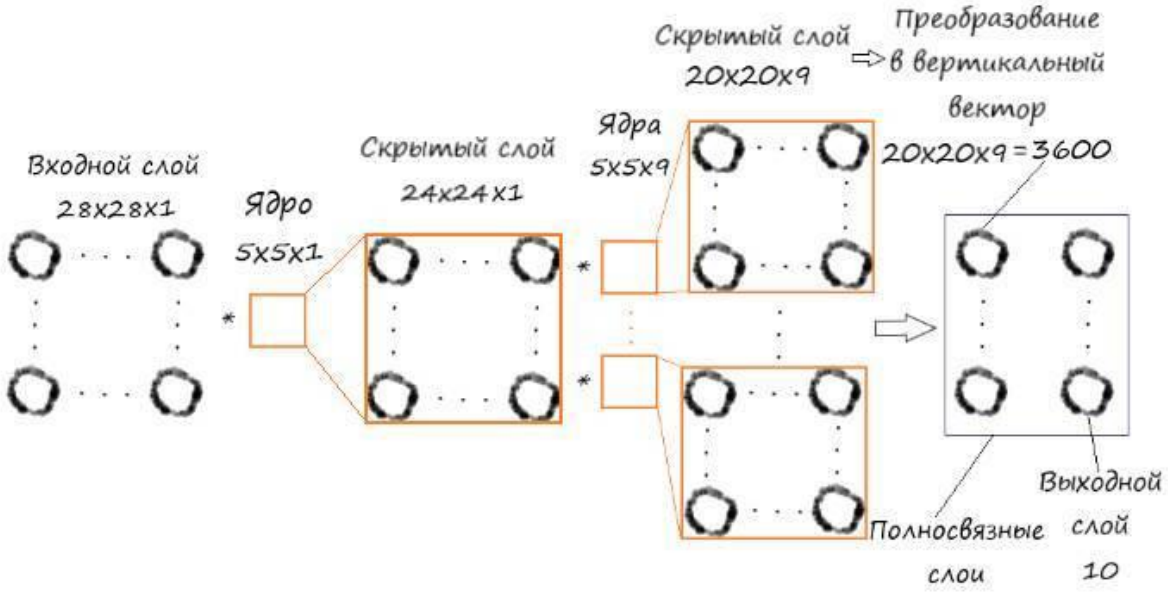
```
# Данные изображения в ядрах свертки
# получить данные изображения с индексом "0". Для крупного масштаба.
image_y1 = n.weights[0]
# вывод данных изображения участка с индексом "0".
matplotlib.pyplot.imshow(image_y1, cmap='Greys', interpolation='None')



fig = matplotlib.pyplot.figure(figsize=(10,6))
for j in range(9):
ax = fig.add_subplot(3, 3, j+1)
ax.imshow(n.weights[j],
cmap=matplotlib.cm.binary, interpolation='none')
matplotlib.pyplot.xticks(np.array([]))
matplotlib.pyplot.yticks(np.array([]))
matplotlib.pyplot.show()



# Данные изображения в признаках(в сврточ ном слое)
# получить данные изображения с индексом "0". Для крупного масштаба.
image_y1 = n.hidden_outputs_image[0] # Карта запомнила последний тренировочный пример при обучении сети!
# вывод данных изображения участка с индексом "0".
matplotlib.pyplot.imshow(image_y1, cmap='Greys', interpolation='None')



fig = matplotlib.pyplot.figure(figsize=(10,6))
for j in range(9):
ax = fig.add_subplot(3, 3, j+1)
ax.imshow(n.hidden_outputs_image[j],
cmap=matplotlib.cm.binary, interpolation='none')
matplotlib.pyplot.xticks(np.array([]))
matplotlib.pyplot.yticks(np.array([]))
matplotlib.pyplot.show()
```

Скачать коды программ можно по следующей ссылке:
https://github.com/CaniaCan/neuralmaster

Реализация сети с двумя свёрточными слоями

Осталось проверить еще один алгоритм, который мы разобрали ранее – обратное распространение ошибки в свёрточном слое. Для этого создадим бессмысленный с точки зрения эффективности, а даже ухудшающий её, входной слой с одним ядром свертки (одно ядро – более наглядно и экономит вычислительные ресурсы). Нам лишь важно убедиться в работоспособности алгоритма обратного распространения ошибки, который будет реализован через этот слой. Скрытых слоёв будет уже два, второй будет трансформироваться в полносвязный с выходным, как мы уже это делали ранее.

Визуализируем описанную структуру сети:



Запрограммируем данную структуру. За основу возьмем предыдущий код, инициализировать параметры ядер и скрытые слои будем прямо в теле инициализации параметров класса “neuron_Net”, таким образом минимизируем изменения с предыдущим кодом.

Думаю, вы уже без труда смогли разобрать все перипетии предыдущих кодов программ, поэтому длительные комментарии по внесению изменений в программу по сравнению с предыдущей, будут излишни. Поэтому можно сразу дать полный текст кода программы:

```
import numpy as np
# библиотека для вывода на консоль массивов
import matplotlib.pyplot
# убедитесь, что участки находятся внутри этой записной книжки, а не внешнего окна
%matplotlib inline
#plt.show() # Вместо %matplotlib inline в других средах, не notebook
from time import time, sleep #Для замера времени выполнения функций
from tqdm import tqdm #Для вывода прогресса вычисления функций
# glob помогает выбрать несколько файлов, используя шаблоны
import glob
# помощник для загрузки данных из файлов изображений PNG
import scipy.misc
```

```
# Загрузить mnist тренировочные данные в формате CSV
training_data_file = open("MNIST_dataset/mnist_train.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data_file.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data_file.close() # закрываем фаел csv
```

```
# Определение класса нейронной сети
```

```
class neuron_Net:



    # инициализация нейронной сети
    def __init__(self, input_num, hidden_num, output_num, learningrate): #констр.(входной слой, скрытый слой, выходной слой)
    # ВХОДНЫЕ ДАННЫЕ
    self.m = 28 #Размер входного массива(ДхШ)
    self.k = 5 #Размер ядер (ДхШ)
    self.m_k_1 = (self.m-self.k)+1 #Размер карты свойств скрытого слоя1 (ДхШ)
    self.m_k = self.m-self.k
    self.stob_w = self.k*self.k #Количество элементов 1го ядра свертки
    self.m_k1_1 = (self.m_k_1 – self.k)+1 #Размер карты свойств скрытого слоя2(ДхШ)
    self.stok_w1 = 9 #Число ядер свертки1



    self.x1 = np.zeros((self.m_k_1, self.m_k_1)) #Скрытый слой1
    self.x2 = np.zeros((self.stok_w1, self.m_k1_1, self.m_k1_1)) #Скрытый слой2



    #Для вывода карт свойст скрытого слоя (для их визуального просмотра)
    self.hidden_outputs_image = np.zeros((self.m_k_1, self.m_k_1))
    self.hidden_outputs_image1 = np.zeros((self.stok_w1, self.m_k1_1, self.m_k1_1))



    # МАТРИЦЫ ВЕСОВ
    # Значения в ядре1
    self.weights = np.random.normal(0.0, pow(self.stob_w, -0.5), (self.k, self.k))
    # Значения в ядрах2
    self.weights2 = np.random.normal(0.0, pow(self.stob_w, -0.5), (self.stok_w1, self.k, self.k))
    self.weights2_rot_180 = self.weights2 # Шаблон для перевернутой матрицы ядер весов2
    # Значения в весов выходного слоя
    self.weights_out = np.random.normal(0.0, pow(hidden_num, -0.5), (output_num, hidden_num)) #20*20*9



    # Создадим матрицу размера ошибки1 – как показано на слайде
    # Нулевая матрица размера (для помещения значений ошибок в ее центр)
    self.hidden_errors0_temp = np.zeros((self.stok_w1, self.m_k_1+self.k-1, self.m_k_1+self.k-1))
    self.hidden_errors0 = np.zeros((self.stok_w1, self.m_k_1, self.m_k_1))



    # скорость обучения
    self.lr = learningrate



    # функция активации-функция сигмоида
    self.activation_function = lambda x: scipy.special.expit(x)
```

```
    pass



    # обучение нейронной сети
    def train(self, inputs_list, targets_list): # принемает входной список данных,targets ответы
    # Преобразовать список входов в 2D массив
    inputs_x = np.array(inputs_list.reshape(self.m , self.m)) # матрица числа
    # Преобразовать список ответов в вертикальный массив. .T – транспонирование
    targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов какое это число



    # ВЫЧИСЛЕНИЕ СИГНАЛОВ ПО СЛОЯМ
    # вычислить сигналы в скрытом слое1 (карты сигналов скрытого слоя1). СВЁРТКА!
    for h in range(self.m_k_1):
    for w in range(self.m_k_1):
    self.x1[h,w] = np.sum(inputs_x[h:h+self.k, w:w+self.k] * self.weights)



    # вычислить сигналы, возникающие из скрытого слоя0. сигмоида(сигнал скр.слоя)
    y1 = self.activation_function(self.x1)



    # вычислить сигналы в скрытом слое2 (карты сигналов скрытого слоя2). СВЁРТКА!
    for s in range(self.stok_w1):
    for h in range(self.m_k1_1):
    for w in range(self.m_k1_1):
    self.x2[s,h,w] = np.sum(y1[h:h+self.k, w:w+self.k] * self.weights2[s])



    # вычислить сигналы, возникающие из скрытого слоя. сигмоида(сигнал скр.слоя)
    y2 = self.activation_function(self.x2)



    # вычислить сигналы в окончательный выходной слой (матрица сигналов выходного слоя)
    x3 = np.dot(self.weights_out, np.array(y2.flatten(), ndmin=2).T) # сигнал вых.слоя = вес скр. слоя
* значение сигнала скр.слоя
    # вычислить сигналы, исходящие из конечного выходного слоя. сигмоида(сигнал вых.слоя)
    y3 = self.activation_function(x3)



    # ВЫЧИСЛЕНИЕ ОШИБКИ ПО СЛОЯМ
    # ошибка выходного слоя является (цель – фактическое)
    E = -(targets_Y – y3)
    # Ошибка скрытого слоя2
    E_hidden = np.dot(self.weights_out.T, E)
```

```
E_hidden = E_hidden.reshape(self.stok_w1, self.m_k1_1, self.m_k1_1) # преобразование ошибки скрытого слоя2 в 3D



# Запоминаем значения перевернутых ядер весов2
for s in range(self.stok_w1):
self.weights2_rot_180[s] = np.fliplr(self.weights2[s])
self.weights2_rot_180[s] = np.flipud(self.weights2_rot_180[s])



# Поместим в центр нулевой матрицы большего размера, ошибку на предыдущем слое
for s in range(self.stok_w1):
self.hidden_errors0_temp[s, self.k-1:self.m_k1_1+self.k-1, self.k-1:self.m_k1_1+self.k-1] = E_hidden[s]



# Проходим по этой матрице перевернутыми весами
for s in range(self.stok_w1):
for h in range(self.m_k_1):
for w in range(self.m_k_1):
self.hidden_errors0[s,h,w] = np.sum(self.hidden_errors0_temp[s, h:h+self.k, w:w+self.k] * self.weights2_rot_180[s]
* (1 – y1[h, w]) * (y1[h, w]))



# Сумма ошибки (значения храним в 1м массиве)
for s in range(self.stok_w1-1):
self.hidden_errors0[0] = self.hidden_errors0[s] + self.hidden_errors0[s+1]
# Запоминаем в отдельную переменную сумму ошибок
hidden_errors0 = self.hidden_errors0[0]



# ОБНОВЛЕНИЕ ВЕСОВ ПО СЛОЯМ
# Обновления весов ядер связей между скрытым и выходным слоями(скрытый слой2)
self.weights_out -= self.lr * np.dot((E * y3 * (1.0 – y3)), np.transpose(np.array(y2.flatten(), ndmin=2).T))
# обновления весов связей между скрытыми слоями
for s in range(self.stok_w1):
for h in range(self.m_k_1-(self.m_k1_1)+1):
for w in range(self.m_k_1-(self.m_k1_1)+1):
#Запоминаем локальную область скрытого слоя1
inputs_t = y1[h:h+self.m_k1_1, w:w+self.m_k1_1]
#Переворачиваем на 180 локальную область скрытого слоя1
inputs_t = np.fliplr(inputs_t)
inputs_t = np.flipud(inputs_t)
#Обновляем веса скрытого слоя1
self.weights2[s, h, w] -= np.sum(E_hidden[s] * inputs_t * self.lr)
```

```
# Обновления весов ядра связи между скрытым и входным слоем
for h in range(self.m-(self.m_k_1)+1):
for w in range(self.m-(self.m_k_1)+1):
#Запоминаем локальную область входных данных
inputs_t = inputs_x[h:h+self.m_k_1,  w:w+self.m_k_1]
#Переворачиваем на 180 локальную область входных данных
inputs_t = np.fliplr(inputs_t)
inputs_t = np.flipud(inputs_t)
#Обновляем веса входных данных
self.weights[h,  w] -= np.sum(hidden_errors0  * inputs_t * self.lr)



#Запоминаем карту свойст скрытого слоя для их визуального просмотра
self.hidden_outputs_image  = y1
self.hidden_outputs_image1  = y2
pass



# МЕТОД ПРОГОНА СВОИХ ЗНАЧЕНИЙ  ПО СЕТИ
# запросить нейронную сеть
def query(self, inputs_list): # Функция прогонки по слоям своих данных. Принемает свой набор тестовых данных
# Преобразовать список входов в 2D массив
inputs_x = np.array(inputs_list.reshape(self.m  , self.m)) # матрица числа



# вычислить сигналы в скрытом слое1 (карты сигналов скрытого слоя1). СВЁРТКА!
for h in range(self.m_k_1):
for w in range(self.m_k_1):
self.x1[h,w]  = np.sum(inputs_x[h:h+self.k,  w:w+self.k]  * self.weights)



# вычислить сигналы, возникающие из скрытого слоя0. сигмоида(сигнал скр.слоя)
y1 = self.activation_function(self.x1)



# вычислить сигналы в скрытом слое2 (карты сигналов скрытого слоя2). СВЁРТКА!
for s in range(self.stok_w1):
for h in range(self.m_k1_1):
for w in range(self.m_k1_1):
self.x2[s,h,w]  = np.sum(y1[h:h+self.k,  w:w+self.k]  * self.weights2[s])



# вычислить сигналы, возникающие из скрытого слоя. сигмоида(сигнал скр.слоя)
y2 = self.activation_function(self.x2)



# вычислить сигналы в окончательный выходной слой (матрица сигналов выходного слоя)
```

```
    x3 = np.dot(self.weights_out, np.array(y2.flatten(), ndmin=2).T) # сигнал вых.слоя = вес скр. слоя * значение сигнала скр.слоя
    # вычислить сигналы, исходящие из конечного выходного слоя. сигмоида(сигнал вых.слоя)
    y3 = self.activation_function(x3)



    return y3



    # количество входных, скрытых и выходных узлов
    data_input = 784
    data_hidden = 3600 #20x20x9
    data_output = 10



    # скорость обучения
    learningrate = 0.05



    # Создать экземпляр нейронной сети
    n = neuron_Net(data_input, data_hidden, data_output, learningrate)



    # ОБУЧЕНИЕ
    # Зададим количество эпох
    epochs = 1



    start = time()
    # Прогон по обучающей выборке
    for e in range(epochs):
    # Пройдите все записи в наборе тренировочных данных
    #for record in training_data_list:
    for i in tqdm(training_data_list, desc = str(e+1)): # tqdm – используем интерактив состояния прогресса вычисления
    # Получить входные данные числа
    all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    # Массив данных входа с масштабированием от 0,01 до 0,99
    inputs_x = (np.asfarray(all_values[1:])/ 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение



    # Получить целевое значение Y, (ответ – какое это число)
    targets_Y = int(all_values[0]) # перевод символов в int, 0 элемент – целевое значение



    # создать целевые выходные значения (все 0.01, кроме нужной метки, которая равна 0.99)
```

```
    targets_Y = np.zeros(data_output) + 0.01



    # Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
    targets_Y[int(all_values[0])]  = 0.99



    n.train(inputs_x,  targets_Y) # наш метод train – обучение нейронной сети



    pass
    pass



    time_out  = time()  – start
    print("Время  выполнения: ", time_out,  " сек" )



    # Загрузить CSV-файл данных теста mnist в список
    test_data_file  = open("mnist_dataset/mnist_test.csv",  'r') # 'r' – файл для чтения, а не для записи.
    test_data_list = test_data_file.readlines() # readlines() – читает все строки в файле в переменную test_data_list
    test_data_file.close()  # закрываем фаел csv



    # ПРОВЕРКА ЭФФЕКТИВНОСТИ  НЕЙРОННОЙ  СЕТИ
    # Массив показателей  эффективности сети, изначально пустой
    efficiency  = []



    # Прогон по всем записям в наборе тестовых данных
    #for i in test_data_list:
    for i in tqdm(test_data_list):
    # Получить входные данные числа
    all_values  = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    # Правильный ответ, хранимый в нулевом индексе
    targets_Y = int(all_values[0])
    # Массив данных входа с масштабированием от 0,01 до 0,99
    inputs_x = (np.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение



    # Запросить ответ у сети
    outputs_y = n.query(inputs_x)  # Прогон по сети тестового значения из нашего файла
    # Индекс самого высокого значения на матрице выхода, соответствует метке числа
```

```
label_y = np.argmax(outputs_y) # argmax возвращает индекс максимального элемента в выходном массиве



# Добавить правильный или неправильный список
if (label_y == targets_Y): # Если индекс макс. знач. на выходе = целевому значению (0 индекс массива данных)
# Если ответ сети соответствует целевому значению, добавляем 1 в конец массива показателей эффективности
efficiency.append(1)
else:
# Если ответ сети не соответствует целевому значению, добавляем 0 в конец массива показателей эффективности
efficiency.append(0)



pass



pass



# Вычислить оценку производительности. Доля правильных ответов
efficiency_map  = np.asarray(efficiency)  # asarray – преобразование списка в массив



print ('Производительность = ', (efficiency_map.sum() / efficiency_map.size)*100, '%') # Среднее арифметическое



# Данные изображения в ядрах свертки скрытого слоя2
# получить данные изображения с индексом "0". Для крупного масштаба.
image_y1  = n.weights2[0]
# вывод данных изображения участка с индексом "0".
matplotlib.pyplot.imshow(image_y1,  cmap='Greys', interpolation='None')



fig = matplotlib.pyplot.figure(figsize=(10,6))
for j in range(9):
ax = fig.add_subplot(3,  3, j+1)
ax.imshow(n.weights2[j],
cmap=matplotlib.cm.binary,  interpolation='none')
matplotlib.pyplot.xticks(np.array([]))
matplotlib.pyplot.yticks(np.array([]))
matplotlib.pyplot.show()
```

```
# Данные изображения в признаках скрытого слоя2
# получить данные изображения признака скрытого слоя2
image_o2 = n.hidden_outputs_image1[0]
# данные изображения участка
matplotlib.pyplot.imshow(image_o2, cmap='Greys', interpolation='None')
```

```
# получить данные изображения признака скрытого слоя1
fig = matplotlib.pyplot.figure(figsize=(10,6))
for j in range(9):
ax = fig.add_subplot(3, 3, j+1)
ax.imshow(n.hidden_outputs_image1[j],
cmap=matplotlib.cm.binary, interpolation='none')
matplotlib.pyplot.xticks(np.array([]))
matplotlib.pyplot.yticks(np.array([]))
matplotlib.pyplot.show()
```

```
# Данные изображения в ядрах свертки скрытого слоя1
# получить данные изображения с индексом "0". Для крупного масштаба.
image_y1 = n.weights
# вывод данных изображения
matplotlib.pyplot.imshow(image_y1, cmap='Greys', interpolation='None')
```

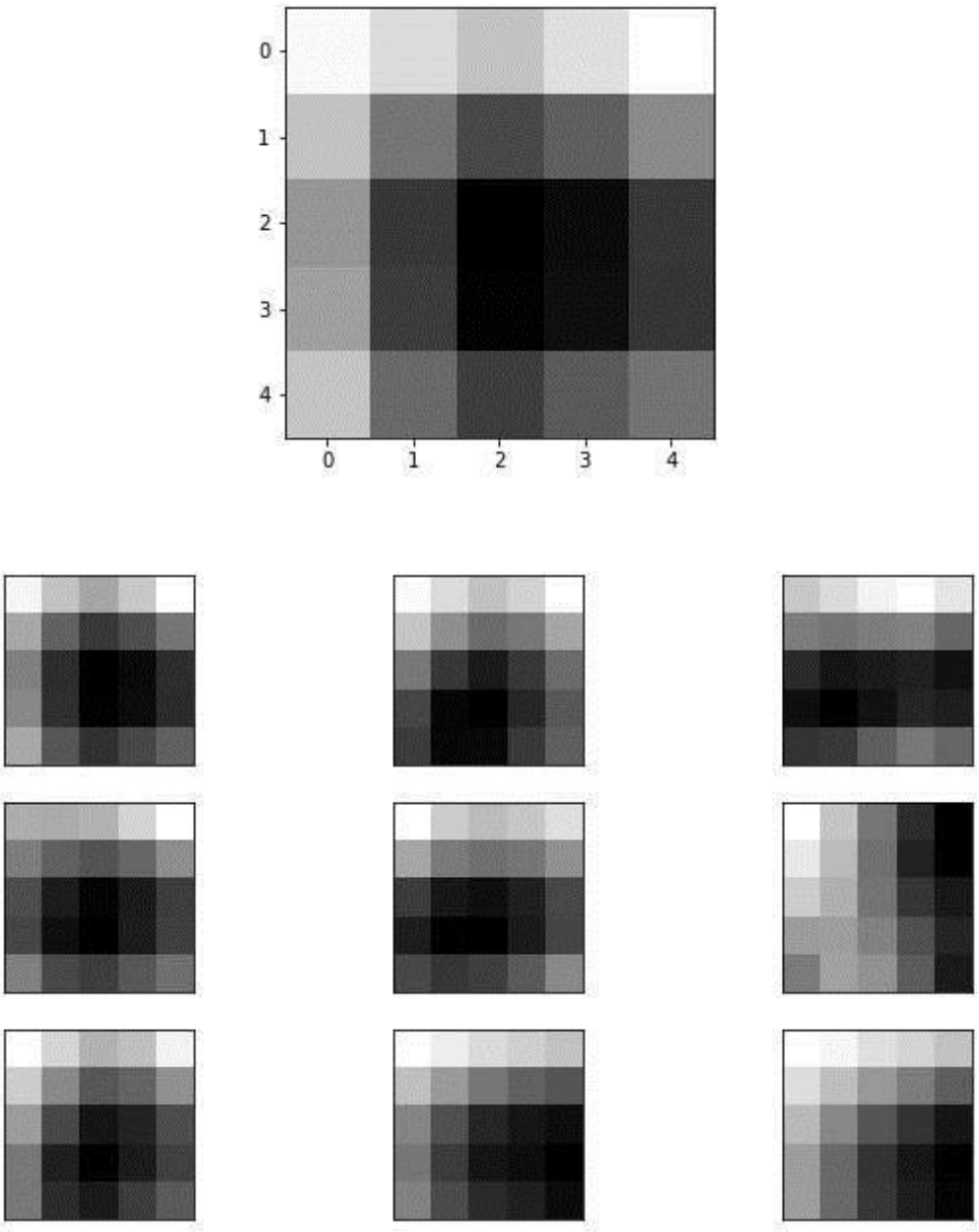
```
# Данные изображения в признаках скрытого слоя1
# получить данные изображения признака скрытого слоя1
image_o2 = n.hidden_outputs_image
# данные изображения участка
matplotlib.pyplot.imshow(image_o2, cmap='Greys', interpolation='None')
```

Результаты работы программы:

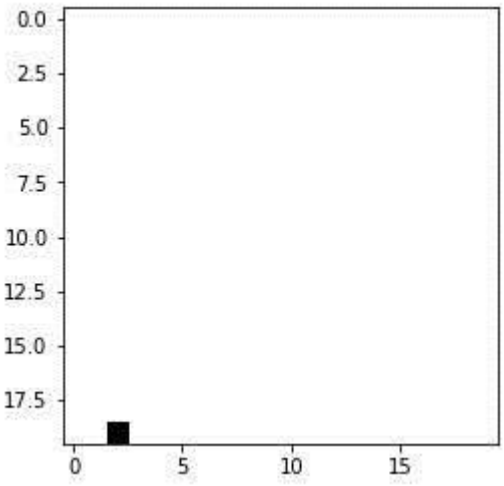
Время выполнения: 3932.404888153076 сек

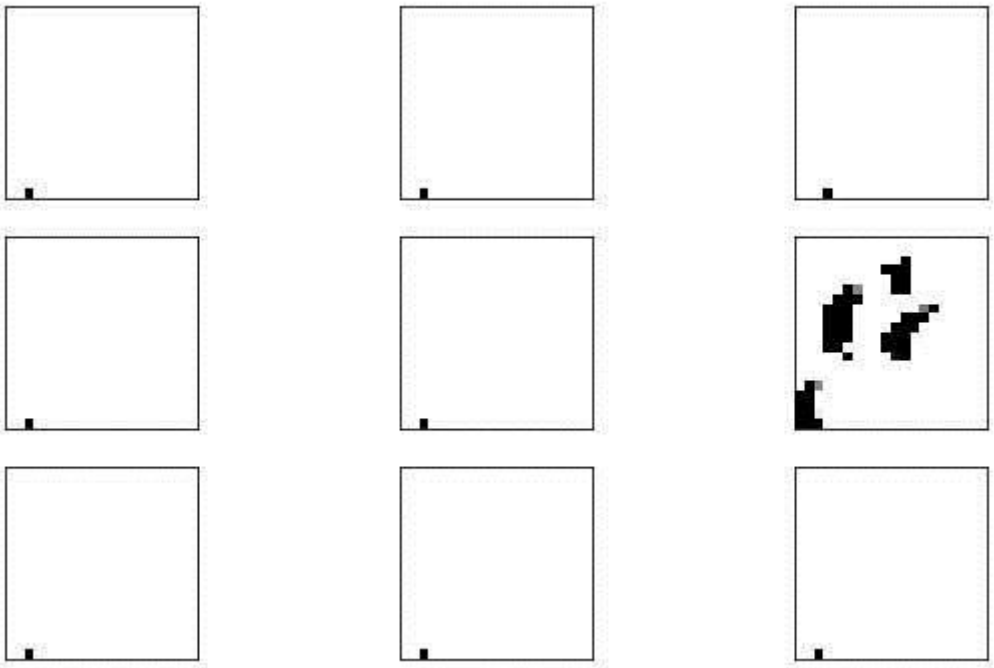
Производительность = 70.76 %

Визуализация значений нулевого ядра второго скрытого слоя и значений всех его ядер:

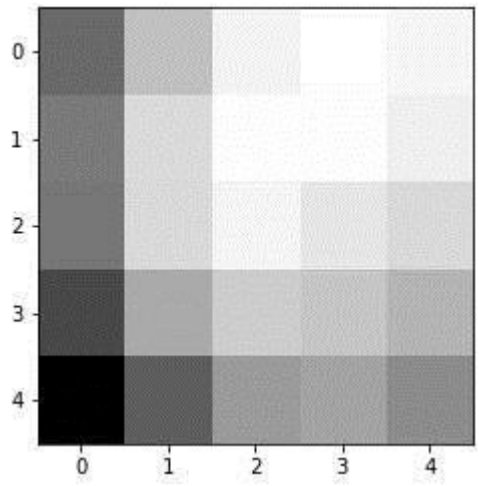


Визуализация значений нулевой карты признаков второго скрытого слоя и значений всех карт:

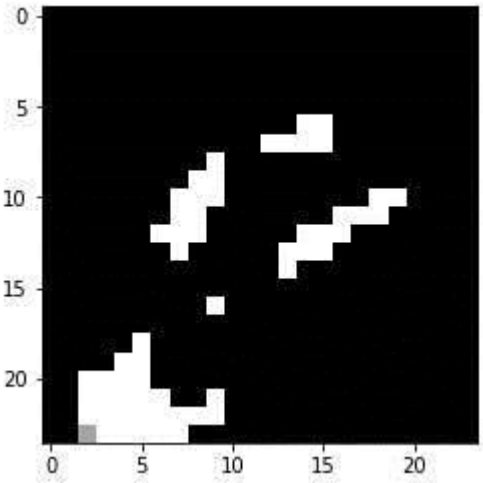




Визуализация значений ядра первого скрытого слоя:



Визуализация значений карты первого скрытого слоя:



В этом коде я не стал тестировать сеть на собственных значениях.

Поскольку проход одного ядра по входу, в результате чего появился дополнительный скрытый слой, довольно бессмысленное занятие ухудшающее итоговый результат, то производительность оказалась ожидаемо ниже. Хотя, если бы мы прошли не одну, а пять эпох, как в предыдущем коде, то производительность могла бы стать лучше.

По программе стоит ещё отметить, что кроме инициализации скрытых слоев, дополнительных ядер свертки и их размерности, ключевым дополнением является алгоритм обратного распространения ошибки через первый свёрточный слой – “self.x2[s,h,w] = np.sum(y1[h:h+self.k, w:w+self.k] * self.weights2[s])”, с последующей суммой ошибок (данный алгоритм был описан ранее).

В данных примерах не ставился приоритет производительности. Главный итог для нас – разобраться в принципах работы и убедится в работоспособности разработанных нами ранее алгоритмов свёртки, обновления ядер свёртки и обратного распространения ошибки через свёрточный слой. Чего мы успешно добились!

Слой макспулинга

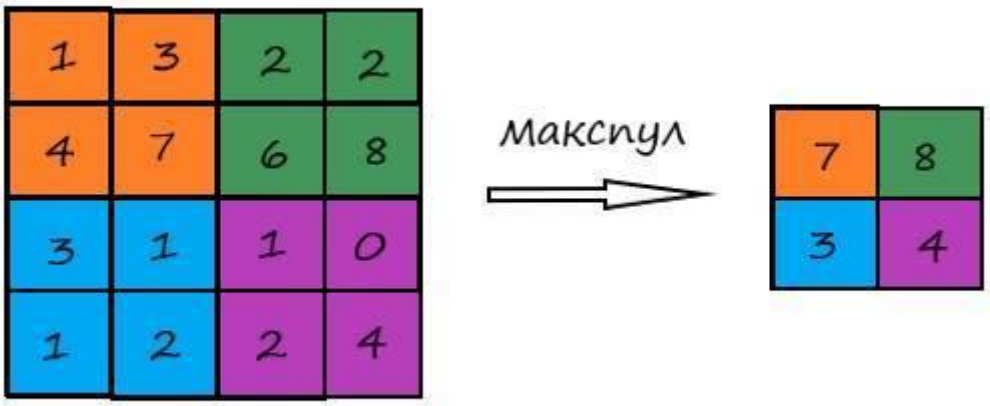
Этот слой позволяет выделять важные особенности на картах признаков, дает инвариантность к нахождению объекта на картах, и также снижает размерность карт, ускоряя время работы сети.

Принцип его работы похож на функцию свертки. Но здесь не происходит поэлементного перемножения матриц, а только лишь выбор максимального значения из заданного окна. Таким образом, выделяются ключевые значения в картах, с дальнейшим их занесением в слой макспулинга, и тем самым уменьшая количество параметров, что, как говорилось ранее, приводит к ускорению вычислений снижая нагрузку на вычислительные блоки.

На всё той же “VGG16”, вы можете разглядеть этот слой, после каждого слоя свёртки.

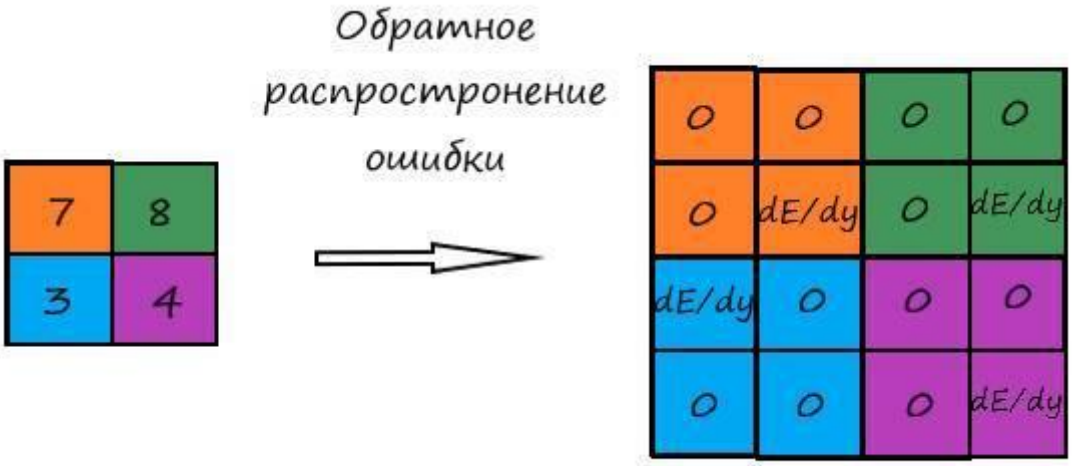
Ну что же, давайте разберемся в работе этого слоя и выработаем алгоритм его решения в Python.

Для этого, по традиции, визуализируем процесс макспула:



Сразу что приходит на ум, а как действовать при обратном распространении ошибки через этот слой?

Разумно было бы предположить, что ошибка проходит только через те значения исходной матрицы, которые были выбраны максимальными на шаге макспулинга. Остальные значения ошибки для этой матрицы будут равны нулю, так как значения по этим элементам не были задействованы функцией макспулинга во время прямого прохождения через сеть, а значит они никак не повлияли на итоговый результат:



Теперь, на этих основаниях, попробуем выработать алгоритмы макспула и обратного распространения ошибки через него. Начнем с самого процесса макспулинга:

```
print('ПРОГОН ВПЕРЕД:\n ')
import re # Для извлечения дробной и целой части


x22 = np.arange(stok_w*m*m).reshape(stok_w,m,m) #выходной слой
print('Скрытый слой x22:\n', x22)


# Пулинг данные
pool_m = 2 # Размер матрицы ядра пулинга (ДхШ)
m_k_1_pool = int(m/pool_m) # Размерность слоя пулинга (целая часть)
x22_maxpool = np.zeros((stok_w, m_k_1_pool, m_k_1_pool)) # Матрица выходного массива пулинга
x22_el = np.zeros((stok_w, m_k_1_pool, m_k_1_pool), dtype='<U32') # хранит адрес макс элемента


# Операция пулинга
for s in range(stok_w):
for h in range(m_k_1_pool):
for w in range(m_k_1_pool):
temp2 = x22[s, h*pool_m:h*pool_m+pool_m, w*pool_m:w*pool_m+pool_m] # Для хранения подматрицы с нулевыми и макс знач.
x22_maxpool[s,h,w] = x22[s, h*pool_m:h*pool_m+pool_m, w*pool_m:w*pool_m+pool_m].max() # Матрица выходного массива после пулинга
# Здесь же, в циклах пулинга запоминаем координаты максимальных элементов в своих областях
for i in range(pool_m):
for j in range(pool_m):
if temp2[i, j] == x22_maxpool[s,h,w]:
temp3 = str(i+h*pool_m) + ',' + str(j+w*pool_m) #Запоминаем,через запятую, координаты по строкам и столбцам


x22_el[s, h, w] = temp3


print('x2_maxpool слой:\n ',x22_maxpool)
print('Адрес (индекс) максимального элемента x22_el:\n ',x22_el)
```

Здесь, подвергаем макспулу слой “x22”. Результатом операция макспулинга будет храниться в переменной “x22_maxpool”.

Ключевым действием в этой части кода является сама операция макспулинга – temp2 = x22[s, h*pool_m:h*pool_m+pool_m, w*pool_m:w*pool_m+pool_m]

x22_maxpool[s,h,w]=x22[s,h*pool_m:h*pool_m+pool_m,w*pool_m:w*pool_m+pool_m].max(). Здесь, при проходе области макспулинга по слою “x2” мы просто находим максимальный элемент в локальной области и заносим его в массив слоя макспула. А переменная “temp2”, служит для временного хранения текущей области, чтобы в последующем сравнить её в цикле с максимальным элементом в этой области – “ if temp2[i, j] == x22_maxpool[s,h,w]:”. Если условие выполняется, то сохраняем координаты, в виде строк, через запятую, в переменной “ temp3”. Массив “x22_el” и будет хранить эти координаты.

Результат работы данного участка программы:

ПРОГОН ВПЕРЕД:

Скрытый слой x22:
[[[0 1 2 3]
[4 5 6 7]
[8 9 10 11]
[12 13 14 15]]

[[16 17 18 19]
[20 21 22 23]
[24 25 26 27]
[28 29 30 31]]]
x2_maxpool слой:
[[[5. 7.]
[13. 15.]]

[[21. 23.]
[29. 31.]]]
Адрес (индекс) максимального элемента x22_el:
[[['1,1' '1,3']
['3,1' '3,3']]

[['1,1' '1,3']
['3,1' '3,3']]]

Теперь, на основании полученных результатов координат значений максимальных элементов в скрытом слое “x22”, выработаем алгоритм обратного распространения ошибки через макспул.

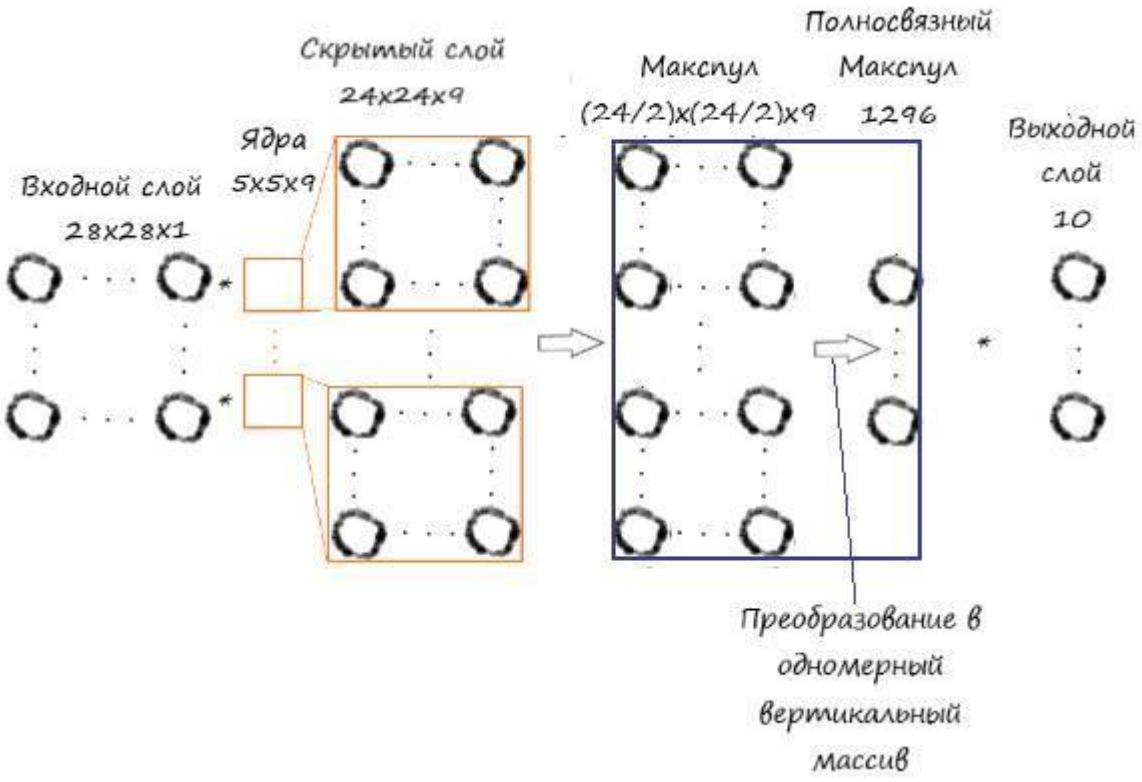
```
print('\n\nПРОГОН НАЗАД:\n ')
#После вычисления ошибки в слое пулинга, распростроняем эту ошибку на слой перед пулингом согласно индексам, остальные нули
# Обнулим ошибки перед пулингом
e1 = np.zeros((stok_w, m, m))
# Запишем сюда ошибки на пулинге согласно индексам x22_el
x22e_maxpool = x22_maxpool # Ошибка на слое пулинга
print('Ошибка на слое пулинга x22e_maxpool:\n ',x22e_maxpool)
for s in range(stok_w):
for h in range(m_k_1_pool):
for w in range(m_k_1_pool):
result = re.split(r'[,]', x22_el[s,h,w]) # Разбивает строку по разделителю ',' можно сразу несколько разделителей
i = int(result[0])
j = int(result[1])
e1[s,i,j] = x22e_maxpool[s,h,w]
print('Ошибка на слое перед пулингом e1:\n ',e1)
```

Здесь, в общих чертах, в нулевой массив ошибки e1, в циклах, по полученным координатам, вносим значения из слоя макспул. С помощью библиотеки “re” и её методу “split”, через разделитель запятой, поочередно извлекаем из массива “ x22_el ”, координаты максимального элемента. После чего, по полученным координатам, ставим максимальный элемент, на своё место в слое ошибки скрытого слоя “e1”.

Результат работы данного участка программы:

```
ПРОГОН НАЗАД:
Ошибка на слое пуллинга x22e_maxpool:
[[[ 5. 7.]
[ 13. 15.]]


[[ 21. 23.]
[ 29. 31.]]]
Ошибка на слое перед пулингом e1:
[[[ 0. 0. 0. 0.]
[ 0. 5. 0. 7.]
[ 0. 0. 0. 0.]
[ 0. 13. 0. 15.]]


[[ 0. 0. 0. 0.]
[ 0. 21. 0. 23.]
[ 0. 0. 0. 0.]
[ 0. 29. 0. 31.]]]
```

Остается проверить наш алгоритм на практике. Для этого реализуем в Python следующую структуру:



Код программы:

```
import numpy as np
# библиотека для вывода на консоль массивов
import matplotlib.pyplot
# убедитесь, что участки находятся внутри этой записной книжки, а не внешнего окна
%matplotlib inline
#plt.show() # Вместо %matplotlib inline в других средах, не notebook
from time import time, sleep #Для замера времени выполнения функций
from tqdm import tqdm #Для вывода прогресса вычисления функций
# glob помогает выбрать несколько файлов, используя шаблоны
import glob
# помощник для загрузки данных из файлов изображений PNG
import scipy.misc
import re # Для извлечения дробной и целой части
```

```
# Загрузить mnist тренировочные данные в формате CSV
training_data_file = open("MNIST_dataset/mnist_train_100.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data_file.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data_file.close() # закрываем фаел csv
```

```
# Определение класса нейронной сети
class neuron_Net:



# инициализация нейронной сети
def __init__(self, input_num, hidden_num, maxpul_num, output_num, learningrate): #констр.(входной слой, скрытый слой, макспул, выходной слой)



self.m = 28 #Размер входного массива(ДхШ)
self.k = 5 #Размер весов (ДхШ)
self.m_k_1 = (self.m-self.k)+1
self.stok_w = 9 #Число ядер свертки (весов)
self.stob_w = self.k*self.k #Количество элементов 1го ядра свертки
self.x1 = np.zeros((self.stok_w, self.m_k_1, self.m_k_1)) #Массив скрытого слоя



# Пулиинг данные
self.pool_m = 2 # Матрица ядра пуллинга
self.m_k_1_pool = int(self.m_k_1/self.pool_m) # Размерность слоя пуллинга
self.hidden_outputs_mp = np.zeros((self.stok_w, self.m_k_1_pool, self.m_k_1_pool)) # Массива пулинга
self.hidden_outputs_el = np.zeros((self.stok_w, self.m_k_1_pool, self.m_k_1_pool), dtype='<U32') # хранит адрес макс. элемента



#Для вывода карт свойст скрытого слоя
self.hidden_outputs_image = np.zeros((self.stok_w, self.m_k_1, self.m_k_1))
# МАТРИЦЫ ВЕСОВ
self.weights = np.random.normal(0.0, pow(self.stob_w, -0.5), (self.stok_w, self.k, self.k))
self.weights_out = np.random.normal(0.0, pow(maxpul_num, -0.5), (output_num, maxpul_num)) # После пуллинг слоя



# скорость обучения
self.lr = learningrate



# функция активации-функция сигмоида
self.activation_function = lambda x: scipy.special.expit(x)



pass



# обучение нейронной сети
```

```
def train(self, inputs_list, targets_list): # принемает входной список данных,targets ответы
# Преобразовать список входов в 2D массив
inputs_x = np.array(inputs_list.reshape(self.m , self.m)) # матрица числа



targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов какое это число



# ВЫЧИСЛЕНИЕ СИГНАЛОВ ПО СЛОЯМ
# вычислить сигналы в скрытом слое (матрица сигналов скрытого слоя)
for s in range(self.stok_w):
for h in range(self.m_k_1):
for w in range(self.m_k_1):
self.x1[s,h,w] = np.sum(inputs_x[h:h+self.k, w:w+self.k] * self.weights[s])



# вычислить сигналы, возникающие из скрытого слоя.
y1 = self.activation_function(self.x1) #Сигмоида



# Макспуллинг
for s in range(self.stok_w):
for h in range(self.m_k_1_pool):
for w in range(self.m_k_1_pool):
temp2 = y1[s, h*self.pool_m:h*self.pool_m+self.pool_m, w*self.pool_m:w*self.pool_m+self.pool_m] # Для хранения подматрицы с нулевыми и макс знач.
self.hidden_outputs_mp[s,h,w] = y1[s, h*self.pool_m:h*self.pool_m+self.pool_m, w*self.pool_m:w*self.pool_m+self.pool_m].max() # Матрица массива пулинга
for i in range(self.pool_m):
for j in range(self.pool_m):
if temp2[i, j] == self.hidden_outputs_mp[s,h,w]:
temp3 = str(i+h*self.pool_m) + ',' + str(j+w*self.pool_m)



self.hidden_outputs_el[s, h, w] = temp3



# вычислить сигналы в окончательный выходной слой (матрица сигналов выходного слоя)
x2 = np.dot(self.weights_out, np.array(self.hidden_outputs_mp.flatten(), ndmin=2).T) # сигнал вых.слоя = вес скр. слоя * значение сигнала скр.слоя
# вычислить сигналы, исходящие из конечного выходного слоя
y2 = self.activation_function(x2) # Сигмоида



# ВЫЧИСЛЕНИЕ ОШИБКИ ПО СЛОЯМ
# ошибка выходного слоя является (цель – фактическое)
E = -(targets_Y – y2)
# Скрытая ошибка слоя макспулинга
```

```
    hidden_errors_mp = np.dot(self.weights_out.T,  E) # Одномерный, вертикальный
    hidden_errors_mp        =        self.hidden_outputs_mp.reshape(self.stok_w,        self.m_k_1_pool,
self.m_k_1_pool)# 3D



    # Обнулим ошибки перед пулингом
    E_hidden = np.zeros((self.stok_w, self.m_k_1, self.m_k_1))
    for s in range(self.stok_w):
    for h in range(self.m_k_1_pool):
    for w in range(self.m_k_1_pool):
    result = re.split(r'[,]', self.hidden_outputs_el[s,h,w]) # Разбивает строку по разделителю ',' можно
сраху несколько разделителей
    i = int(result[0])
    j = int(result[1])
    E_hidden[s,i,j]  = hidden_errors_mp[s,h,w]



    # ОБНОВЛЕНИЕ  ВЕСОВ ПО СЛОЯМ
    # обновления весов связей между скрытым пуллинг слоем и выходным слоями
    self.weights_out  -= self.lr  * np.dot((E * y2
    * (1.0 – y2)), np.transpose(np.array(self.hidden_outputs_mp.flatten(),  ndmin=2).T))  # Сигмоида



    # обновления весов связей между входным и скрытым слоями
    for s in range(self.stok_w):
    for h in range(self.k):
    for w in range(self.k):
    inputs_t = inputs_x[h:h+self.m_k_1,  w:w+self.m_k_1]
    inputs_t = np.fliplr(inputs_t)
    inputs_t = np.flipud(inputs_t)
    self.weights[s,  h,  w] -= np.sum(E_hidden[s] * inputs_t * self.lr) #Для софтмакс и без функции
активации  и RRELU на выходе



    #Запоминаем карту свойст скрытого слоя перед пулингом  для просмотра
    self.hidden_outputs_image  = y1
    pass



    # МЕТОД ПРОГОНА СВОИХ ЗНАЧЕНИЙ  ПО СЕТИ
    # запросить нейронную сеть
    def query(self, inputs_list): # Функция прогонки по слоям своих данных. Принемает свой набор
тестовых данных
    # Преобразовать список входов в 2D массив
    inputs_x = np.array(inputs_list.reshape(self.m  , self.m)) # матрица числа



    # вычислить сигналы в скрытом слое (матрица сигналов скрытого слоя)
    for s in range(self.stok_w):
```

```
for h in range(self.m_k_1):
for w in range(self.m_k_1):
self.x1[s,h,w] = np.sum(inputs_x[h:h+self.k, w:w+self.k] * self.weights[s])



# вычислить сигналы, возникающие из скрытого слоя.
y1 = self.activation_function(self.x1) #Сигмоида



# Макспуллинг
for s in range(self.stok_w):
for h in range(self.m_k_1_pool):
for w in range(self.m_k_1_pool):
temp2 = y1[s, h*self.pool_m:h*self.pool_m+self.pool_m, w*self.pool_m:w*self.pool_m+self.pool_m] # Для хранения подматрицы с нулевыми и макс знач.
self.hidden_outputs_mp[s,h,w] = y1[s, h*self.pool_m:h*self.pool_m+self.pool_m, w*self.pool_m:w*self.pool_m+self.pool_m].max() # Матрица выходного массива после пулинга
for i in range(self.pool_m):
for j in range(self.pool_m):
if temp2[i, j] == self.hidden_outputs_mp[s,h,w]:
temp3 = str(i+h*self.pool_m) + ',' + str(j+w*self.pool_m)



self.hidden_outputs_el[s, h, w] = temp3



# вычислить сигналы в окончательный выходной слой (матрица сигналов выходного слоя)
x2 = np.dot(self.weights_out, np.array(self.hidden_outputs_mp.flatten(), ndmin=2).T) # сигнал вых.слоя = вес скр. слоя * значение сигнала скр.слоя
# вычислить сигналы, исходящие из конечного выходного слоя
y2 = self.activation_function(x2) # Сигмоида



return y2



# количество входных, скрытых и выходных узлов
data_input = 784
data_hidden = 5184
data_maxpul = 1296 # 24/2 * 24/2 * 9
data_output = 10



# скорость обучения
learningrate= 0.008
```

```
# Создать экземпляр нейронной сети
n = neuron_Net(data_input, data_hidden, data_maxpul, data_output, learningrate)



# ОБУЧЕНИЕ
# Зададим количество эпох
epochs = 10



start = time()
# Прогон по обучающей выборке
for e in range(epochs):
# Пройдите все записи в наборе тренировочных данных
#for record in training_data_list:
for i in tqdm(training_data_list, desc = str(e+1)): # tqdm – используем интерактив состояния прогресса вычисления
# Получить входные данные числа
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
# Массив данных входа с масштабированием от 0,01 до 0,99
inputs_x = (np.asfarray(all_values[1:])/ 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение



# Получить целевое значение Y, (ответ – какое это число)
targets_Y = int(all_values[0]) # перевод символов в int, 0 элемент – целевое значение



# создать целевые выходные значения (все 0.01, кроме нужной метки, которая равна 0.99)
targets_Y = np.zeros(data_output) + 0.01



# Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
targets_Y[int(all_values[0])] = 0.99



n.train(inputs_x, targets_Y) # наш метод train – обучение нейронной сети



pass
pass



time_out = time() – start
print("Время выполнения: ", time_out, " сек" )
```

```
    # Загрузить CSV-файл данных теста mnist в список
    test_data_file = open("mnist_dataset/mnist_test_10.csv", 'r') # 'r' – файл для чтения, а не для записи.
    test_data_list = test_data_file.readlines() # readlines() – читает все строки в файле в переменную test_data_list
    test_data_file.close()  # закрываем фаел csv



    # ПРОВЕРКА ЭФФЕКТИВНОСТИ НЕЙРОННОЙ СЕТИ
    # Массив показателей эффективности сети, изначально пустой
    efficiency = []



    # Прогон по всем записям в наборе тестовых данных
    #for i in test_data_list:
    for i in tqdm(test_data_list):
    # Получить входные данные числа
    all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    # Правильный ответ, хранимый в нулевом индексе
    targets_Y = int(all_values[0])
    # Массив данных входа с масштабированием от 0,01 до 0,99
    inputs_x = (np.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01 # Игнорируем нулевой индекс, где целевое значение



    # Запросить ответ у сети
    outputs_y = n.query(inputs_x)  # Прогон по сети тестового значения из нашего файла
    # Индекс самого высокого значения на матрице выхода, соответствует метке числа
    label_y = np.argmax(outputs_y) # argmax возвращает индекс максимального элемента в выходном массиве



    # Добавить правильный или неправильный список
    if (label_y == targets_Y): # Если индекс макс. знач. на выходе = целевому значению (0 индекс массива данных)
    # Если ответ сети соответствует целевому значению, добавляем 1 в конец массива показателей эффективности
    efficiency.append(1)
    else:
    # Если ответ сети не соответствует целевому значению, добавляем 0 в конец массива показателей эффективности
    efficiency.append(0)



    pass
    pass
```

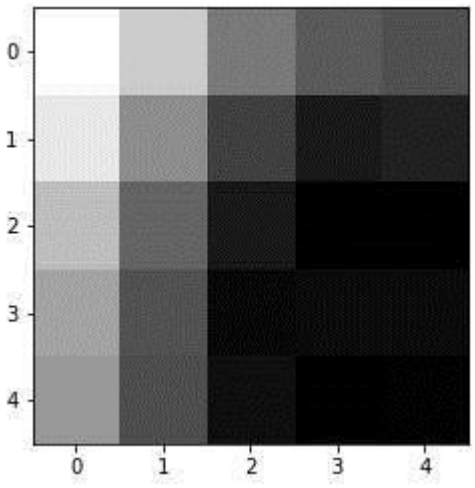
```
# Вычислить оценку производительности. Доля правильных ответов
efficiency_map  = np.asarray(efficiency)  # asarray – преобразование списка в массив



print ('Производительность = ', (efficiency_map.sum() / efficiency_map.size)*100, '%') # Среднее арифметическое



# Данные изображения в ядрах свертки
# получить данные изображения с индексом "0". Для крупного масштаба.
image_y1  = n.weights[0]
# вывод данных изображения участка с индексом "0".
matplotlib.pyplot.imshow(image_y1,  cmap='Greys', interpolation='None')



fig = matplotlib.pyplot.figure(figsize=(10,6))
for j in range(9):
ax = fig.add_subplot(3,  3, j+1)
ax.imshow(n.weights[j],
cmap=matplotlib.cm.binary,  interpolation='none')
matplotlib.pyplot.xticks(np.array([]))
matplotlib.pyplot.yticks(np.array([]))
matplotlib.pyplot.show()



# Данные изображения в признаках(в сврточ ном слое)
# получить данные изображения с индексом "0". Для крупного масштаба.
image_y1 = n.hidden_outputs_image[0] # Карта запомнила последний тренировочный пример при обучении сети!
# вывод данных изображения участка с индексом "0".
matplotlib.pyplot.imshow(image_y1,  cmap='Greys', interpolation='None')



fig = matplotlib.pyplot.figure(figsize=(10,6))
for j in range(9):
ax = fig.add_subplot(3,  3, j+1)
ax.imshow(n.hidden_outputs_image[j],
cmap=matplotlib.cm.binary,  interpolation='none')
matplotlib.pyplot.xticks(np.array([]))
matplotlib.pyplot.yticks(np.array([]))
matplotlib.pyplot.show()



# Данные изображения в признаках(слой макспул)
# получить данные изображения с индексом "0". Для крупного масштаба.
image_y1 = n.hidden_outputs_mp[0] # Карта запомнила последний тренировочный пример при обучении сети!
# вывод данных изображения участка с индексом "0".
matplotlib.pyplot.imshow(image_y1,  cmap='Greys', interpolation='None')
```

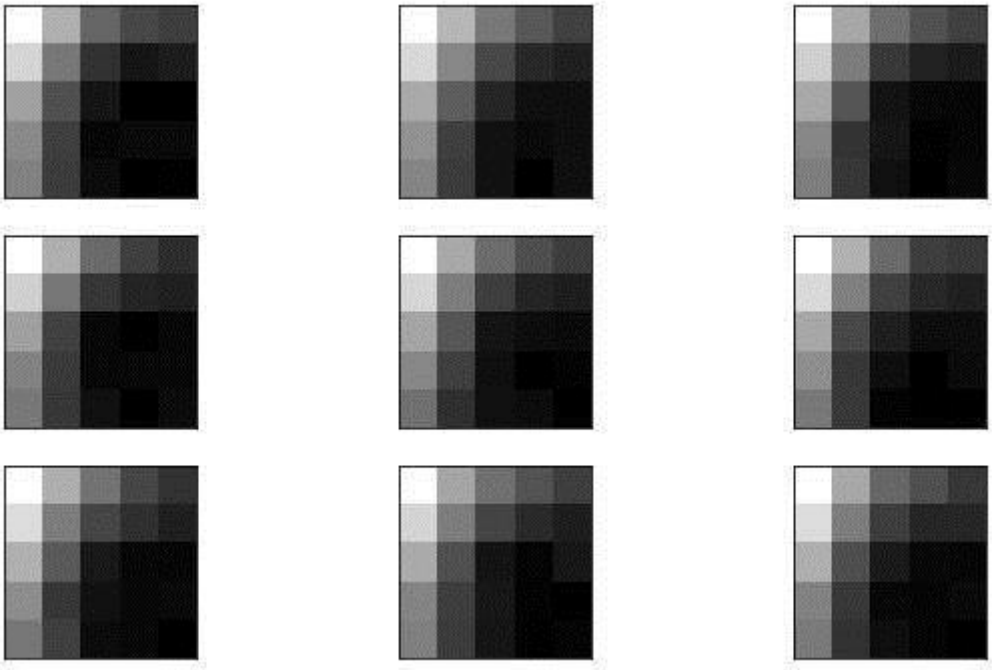
```
fig = matplotlib.pyplot.figure(figsize=(10,6))
for j in range(9):
ax = fig.add_subplot(3, 3, j+1)
ax.imshow(n.hidden_outputs_mp[j],
cmap=matplotlib.cm.binary, interpolation='none')
matplotlib.pyplot.xticks(np.array([]))
matplotlib.pyplot.yticks(np.array([]))
matplotlib.pyplot.show()
```

Результат работы программы:

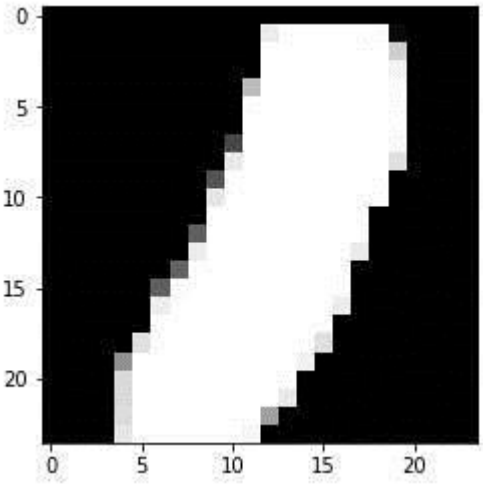
Производительность = 60.0 %

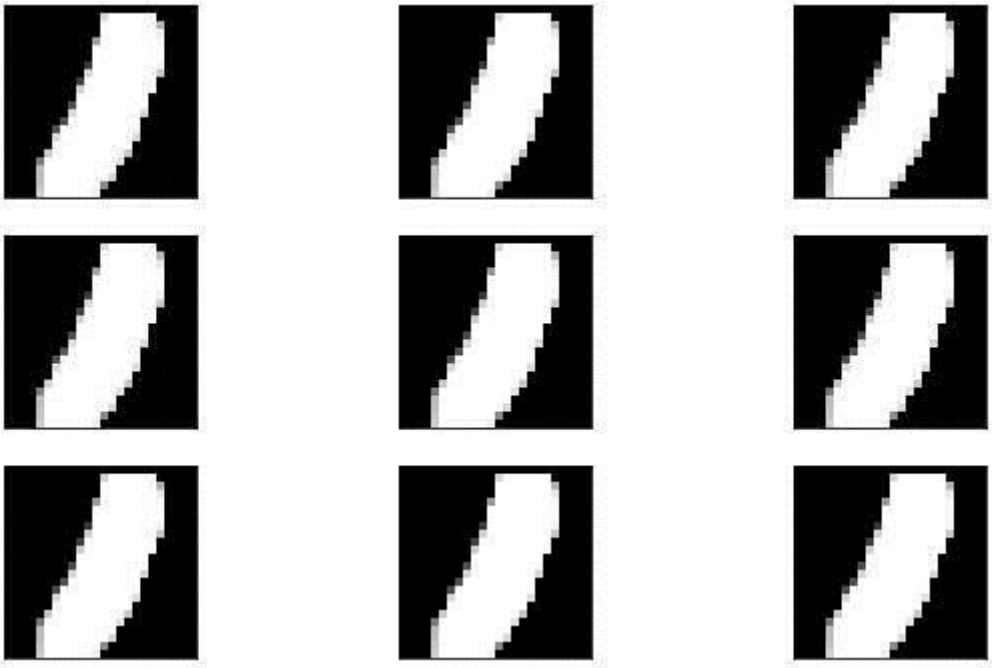
Данные изображения в ядрах свертки:



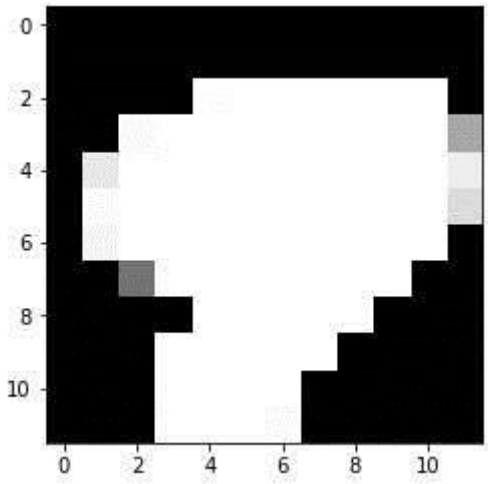


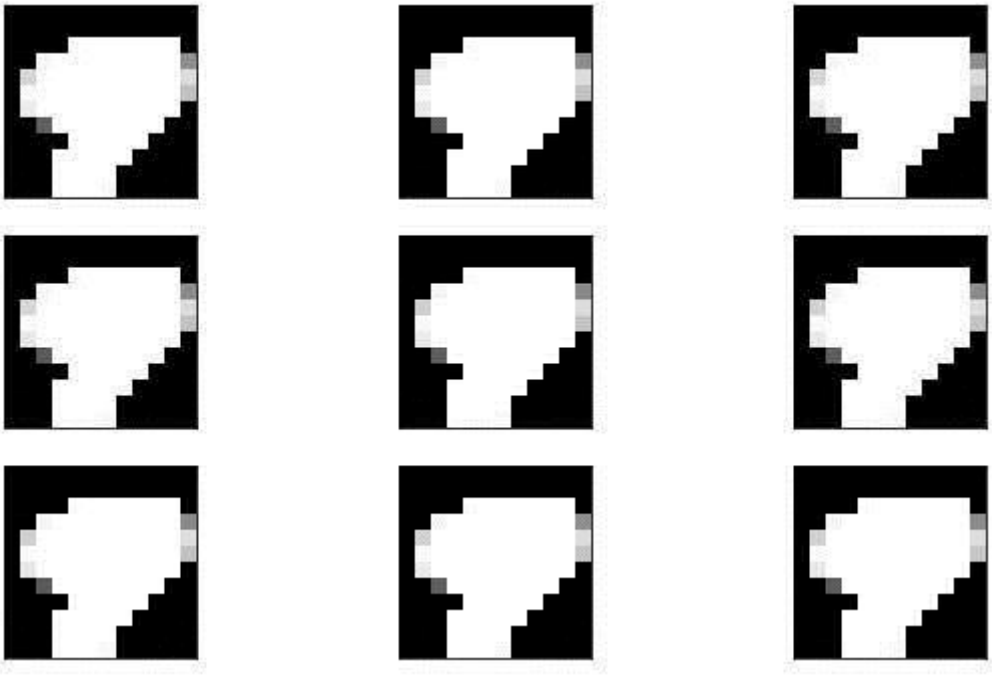
Данные изображения в признаках (в свёточном слое):





Данные изображения слое макспул:





Чтобы избавиться от томительного ожидания обучения сети, я тренировал её на данных всего из ста выборок.

Производительность, из-за малого количества слоев, упала. Это произошло потому, что макспул слой не может выявить значимые признаки из первого свёрточного слоя, сразу подавая их на выходной слой. Для увеличения производительности с применением макспул слоёв, необходимо значительно увеличить количество скрытых слоев. Но опять же, главное в том, что мы убедились, что наши алгоритмы работают.

Функция активации Softmax (софтмакс)

В приведённой выше, структуры сети “VGG16”, можем видеть не знакомую нам функцию активации, а именно – “ Softmax”. Давайте посмотрим, что она из себя представляет.

Softmax – преобразует вектор z размерности j в вектор y той же размерности, где каждая координата yi полученного вектора представлена вещественным числом в интервале [0,1] и сумма этих координат равна единице.

Координаты yi полученного вектора при этом трактуются как вероятности того, что объект принадлежит к классу i. Благодаря этим свойствам, функцию активации софтмакс, в подавляющем большинстве случаев, применяют в выходном слое нейронной сети.

Координаты yi вычисляются следующим образом:

$$y_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}};$$

$$z_i = (x_i * w)$$

Ошибка слоя софтмакс:

$$E = -\sum_{j=1}^{n} ti * \log yi$$

ti - целевое значение выхода сети

yi - реальное значение выхода сети

log - натуральный логорифм (по основанию "e")

Градиент по одной из выходных размерностей или нейрону:

$$\frac{dE}{dzi} = \sum_{j} \frac{dE}{dy_j} \frac{dy_j}{dzi};$$

$$\frac{dE}{dy_j} = -\frac{d(tj * \log y_j)}{dy_j} = -\frac{tj}{y_j}$$

$$\frac{dy_i}{dzi} = \frac{e^{Zi} \sum_{j=1}^{n} e^{Zj} - e^{Zi} e^{Zi}}{\left(\sum_{j=1}^{n} e^{Zj}\right)^2} = y_i(1-y_i), \text{ т.к } \frac{de^{Zi}}{dzi} = e^{Zi}$$

$$\frac{dy_i}{dz_m} = \frac{e^{Zi} e^{Zm}}{\left(\sum_{j=1}^{n} e^{Zj}\right)^2} = -y_i y_m, \text{ т.к } \frac{de^{Zi}}{dz_m} = 0 \ (const' = 0)$$

$$\frac{dE}{dzi} = \frac{dE}{dyj}\frac{dyj}{dzi} = \begin{cases} -\frac{tj}{yi} yi(1-yi) = -tj(1-yi), \text{ если } j=i \\ (-yi\ yj)\left(-\frac{tj}{yj}\right), \text{ если } j\neq i \end{cases}$$

$$\frac{dE}{dzi} = \sum_j \frac{dE}{dyj}\frac{dyj}{dzi} = \sum_{j;\, j\neq i} yi\,tj - ti(1-yi) = -ti + tiyi + yi\sum_{j;\, j\neq i} tj =$$

$$= -ti + yi\left(ti + \sum_{j;\, j\neq i} tj\right)$$

Так как сумма отдельного выхода **yi** с остальными значениями (кроме **yi**) дает единицу, то имеем:

$$\left(ti + \sum_{j;\, j\neq i} tj\right) = 1, \text{ то } \frac{dE}{dzi} = \sum_j \frac{dE}{dyj}\frac{dyj}{dzi} = yi - ti$$

Градиент по связям:

$$\frac{dE}{dw} = \frac{dE}{dzi}\frac{dzi}{dw} = x^T(yi - ti)$$

Применять данную функцию, в целях экономии времени вычислений и лучшей наглядности, на свёрточной сети я не буду. Рассмотрим её применение, на сети с полносвязными слоями, которую мы рассматривали в седьмой главе. В ней, функцию софтмакс поместим на выход сети, а активационной функцией скрытого слоя будет RELU. Распознавать будем всё тот же набор данных “MNIST”.

Полный текст программы:

```
import numpy as np
# библиотека для вывода на консоль массивов
import matplotlib.pyplot
# убедитесь, что участки находятся внутри этой записной книжки, а не внешнего окна
%matplotlib inline
#plt.show() # Вместо %matplotlib inline в других средах, не notebook
from time import time, sleep #Для замера времени выполнения функций
```

```
from tqdm import tqdm #Для вывода прогресса вычисления функций
# glob помогает выбрать несколько файлов, используя шаблоны
import glob
# помощник для загрузки данных из файлов изображений PNG
import scipy.misc



# Загрузить mnist тренировочные данные в формате CSV
training_data_file = open("MNIST_dataset/mnist_train.csv", 'r') # 'r' – открываем файл для чтения
training_data_list = training_data_file.readlines() # readlines() – читает все строки в файле в переменную training_data_list
training_data_file.close() # закрываем фаел csv



# Определение класса нейронной сети
class neuron_Net:



# Инициализация весов нейронной сети
def __init__(self, input_num, neuron_num, output_num, learningrate): #констр.(входной слой, скрытый слой, выходной слой)
# МАТРИЦЫ ВЕСОВ
# Задаем матрицы весов как случайное
self.weights = np.random.normal(0.0, pow(input_num, -0.5), (neuron_num, input_num))
self.weights_out = np.random.normal(0.0, pow(neuron_num, -0.5), (output_num, neuron_num))
# Можно задать веса таким образом
#self.weights = (numpy.random.rand(neuron_num, input_num) -0.5)
#self.weights_out = (numpy.random.rand(output_num, neuron_num) -0.5)



# скорость обучения
self.lr = learningrate
pass



# Обучение нейронной сети
def train(self, inputs_list, targets_list): # принемает входной список данных,targets ответы
# Преобразовать список входов и ответов в вертикальный массив. .T – транспонирование
inputs_x = np.array(inputs_list, ndmin=2).T # матрица числа
targets_Y = np.array(targets_list, ndmin=2).T # матрица ответов какое это число



# ВЫЧИСЛЕНИЕ СИГНАЛОВ ПО СЛОЯМ
# Вычислить сигналы в нейронах скрытого слоя. Взвешенная сумма.
x1 = np.dot(self.weights, inputs_x) # dot – умножение матриц X = W*I = weights * inputs



# вычислить сигналы, возникающие из скрытого слоя. RELU
```

```
y1 = np.maximum(x1,  0) # RELU



# вычислить сигналы в окончательном выходном слое (матрица сигналов выходного слоя)
x2 = np.dot(self.weights_out,  y1)
# вычислить сигналы, исходящие из конечного выходного слоя. Softmax
y2 = np.exp(x2)/np.sum(np.exp(x2),  axis=0) # Softmax



# ВЫЧИСЛЕНИЕ  ОШИБКИ ПО СЛОЯМ
# Ошибка выходного слоя
E = y2 – targets_Y
# Ошибка скрытого слоя
E_hidden = np.dot(self.weights_out.T,  E)



# ОБНОВЛЕНИЕ  ВЕСОВ ПО СЛОЯМ
# Меняем веса исходящие из скрытого слоя по каждой связи
self.weights_out  -= self.lr  * np.dot((E), np.transpose(y1)) # Softmax



# Меняем веса исходящие из входного слоя по каждой связи
self.weights  -= self.lr  * np.dot((E_hidden  * (y1 > 0)), np.transpose(inputs_x)) # RELU
pass



# МЕТОД ПРОГОНА СВОИХ ЗНАЧЕНИЙ  ПО СЕТИ
# Метод прогона тестовых значений
def query(self,  inputs_list):  # Принимает  свой набор тестовых данных
# Преобразовать список входов в вертикальный массив.
inputs_x = np.array(inputs_list,  ndmin=2).T



# Вычислить сигналы в нейронах скрытого слоя. Взвешенная  сумма.
x1 = np.dot(self.weights,  inputs_x) # dot – умножение матриц X = W*I = weights  * inputs
# вычислить сигналы, возникающие из скрытого слоя. RELU
y1 = np.maximum(x1,  0) # RELU



# вычислить сигналы в окончательном выходном слое (матрица сигналов выходного слоя)
x2 = np.dot(self.weights_out,  y1)
# вычислить сигналы, исходящие из конечного выходного слоя. Softmax
y2 = np.exp(x2)/np.sum(np.exp(x2),  axis=0) # Softmax
return  y2



# ЗАДАЁМ ПАРАМЕТРЫ  СЕТИ
```

```
# Количество входных данных, нейронов
data_input = 784
data_neuron = 220
data_output = 10



# Скорость обучения
learningrate = 0.01



# Создать экземпляр нейронной сети
n = neuron_Net(data_input, data_neuron, data_output, learningrate)



# ОБУЧЕНИЕ
# Зададим количество эпох
epochs = 1



start = time()
# Прогон по обучающей выборке
for e in range(epochs):
# Пройдите все записи в наборе тренировочных данных
#for record in training_data_list:
for i in tqdm(training_data_list, desc = str(e+1)): # tqdm – используем интерактив состояния прогресса вычисления
# Получить входные данные числа
all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
# Массив данных входа
inputs_x = (np.asfarray(all_values[1:])/ 255.0) # Игнорируем нулевой индекс, где целевое значение



# Получить целевое значение Y, (ответ – какое это число)
targets_Y = int(all_values[0]) # перевод символов в int, 0 элемент – целевое значение



# создать целевые выходные значения
targets_Y = np.zeros(data_output)



# Получить целевое значение Y, (ответ – какое это число). all_values[0] – целевая метка для этой записи
targets_Y[int(all_values[0])] = 1



n.train(inputs_x, targets_Y) # наш метод train – обучение нейронной сети
```

```
    pass
    pass



    time_out = time() – start
    print("Время выполнения: ", time_out, " сек" )



    # ТЕСТИРОВАНИЕ ОБУЧЕННОЙ СЕТИ
    # Загрузить тестовый CSV-файл
    test_data_file = open("MNIST_dataset/mnist_test.csv", 'r') # 'r' – открываем файл для чтения
    test_data_list = test_data_file.readlines() # readlines() – читает все строки в файле в переменную test_data_list
    test_data_file.close() # закрываем файл csv



    # ПРОВЕРКА ЭФФЕКТИВНОСТИ НЕЙРОННОЙ СЕТИ
    # Массив показателей эффективности сети, изначально пустой
    efficiency = []



    # Прогон по всем записям в наборе тестовых данных
    for i in test_data_list:
    # Получить входные данные числа
    all_values = i.split(',') # split(',') – раздел строку на символы где запятая "," символ разделения
    # Правильный ответ, хранимый в нулевом индексе
    targets_Y = int(all_values[0])
    # Массив данных входа
    inputs_x = (np.asfarray(all_values[1:]) / 255.0) # Игнорируем нулевой индекс, где целевое значение



    # Запросить ответ у сети
    outputs_y = n.query(inputs_x) # Прогон по сети тестового значения из нашего файла
    # Индекс самого высокого значения на матрице выхода, соответствует метке числа
    label_y = np.argmax(outputs_y) # argmax возвращает индекс максимального элемента в выходном массиве



    # Добавить правильный или неправильный список
    if (label_y == targets_Y): # Если индекс макс. знач. на выходе = целевому значению (0 индекс массива данных)
    # Если ответ сети соответствует целевому значению, добавляем 1 в конец массива показателей эффективности
    efficiency.append(1)
    else:
```

```
    # Если ответ сети не соответствует целевому значению, добавляем 0 в конец массива показателей эффективности
    efficiency.append(0)
    pass
    pass



    # Вычислить оценку производительности. Доля правильных ответов
    efficiency_map  = np.asarray(efficiency)  # asarray – преобразование списка в массив



    print ('Производительность = ', (efficiency_map.sum() / efficiency_map.size)*100, '%') # Среднее арифметическое
```

Результат работы программы:

```
Производительность = 96.01 %
```

ЭПИЛОГ

Надеюсь, мне удалось дать базовые знания о работе искусственных нейронов и нейронных сетей. Хочется верить, что мне удалось, доступно для понимания, продемонстрировать теорию нейронных сетей, и объяснить, что в их основе нет ничего сложного.

Возможно данная книга пробудит в вас интерес к дальнейшему изучению нейронных сетей, их разновидностям и методам обучения. Если она кого-то побудит к дальнейшей работе в этом направлении – значит моя цель достигнута.