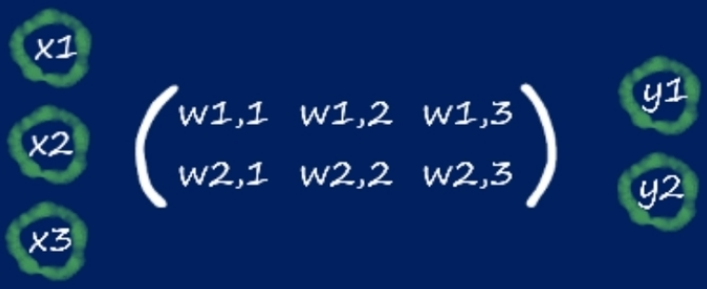
НЕЙРОННЫЕ СЕТИ ЭВОЛЮЦИЯ

$$Y = X*W = \begin{pmatrix} x1*w1,1+x2*w1,2+x3*w1,3 \\ x1*w2,1+x2*w2,2+x3*w2,3 \end{pmatrix} = \begin{pmatrix} f'(z1) \\ f'(z2) \end{pmatrix} = \begin{pmatrix} y1 \\ y2 \end{pmatrix}$$



$$E = w^{T}*E' = \begin{pmatrix} e'1*w1,1+e'2*w2,1 \\ e'1*w1,2+e'2*w2,2 \\ e'1*w1,3+e'2*w2,3 \end{pmatrix} = \begin{pmatrix} e1 \\ e2 \\ e3 \end{pmatrix}$$

Cania Can

12+

Каниа Кан

Нейронные сети. Эволюция

«ЛитРес: Самиздат»

2018

Кан К. А.

Нейронные сети. Эволюция / К. А. Кан — «ЛитРес: Самиздат», 2018

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.

Содержание

Пролог 5
Технология искусственных нейронных сетей 5
Введение 6
Цель книги. Для кого она предназначена 6
Что мы будем делать 7
Как мы будем это делать 8
ГЛАВА 1 9
Основа для создания искусственного нейрона 9
Где используются нейронные сети 9
Как устроены биологические нейронные сети 10
Уровень вычислительной мощности для моделирования ИНС 11
Почему работают нейронные сети 12
Как автоматизировать работу 13
Линейная классификация 14
ГЛАВА 2 23
Изучаем Python 23
Создаем нейронную сеть на Python 23
Установка пакета Anaconda Python 23
Простое введение в Python 24
ГЛАВА 3 36
Рождение искусственного нейрона 36
Моделирование нейрона как линейного классификатора 36
ГЛАВА 4 45
Добавляем входной параметр 45
Моделирование нейрона как линейного классификатора со всеми параметрами линейной функции 46
Обновление весовых коэффициентов 48
Применение дифференциального исчисления, понятие производной 53
Производная функции 55
Нахождение некоторых табличных производных 57
Правила дифференцирования и дифференцирование сложных функций 59
Зачем нам дифференцировать функции 63
Как мы обновляем весовые коэффициенты 68
Как работает эволюционировавший нейрон 71
ГЛАВА 5 76
Больше входных данных 76
Проблемы линейной классификации 76
Логические функции 77
Конец ознакомительного фрагмента. 79

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 Python, 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:

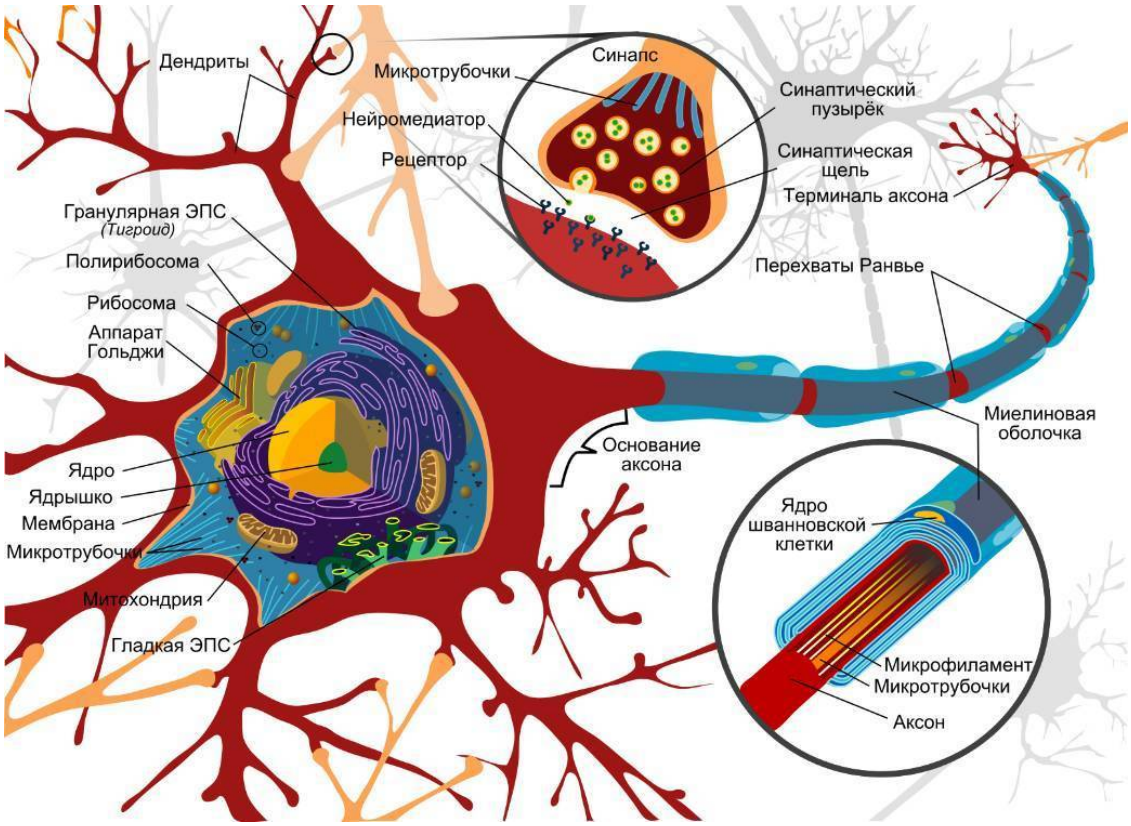
Вы без труда узнаете, что изображено на картинке, так как наш мозг идеальное средство для анализа изображения и его классификации. А вот компьютеру, напротив, очень трудно решать подобные задачи.

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.).

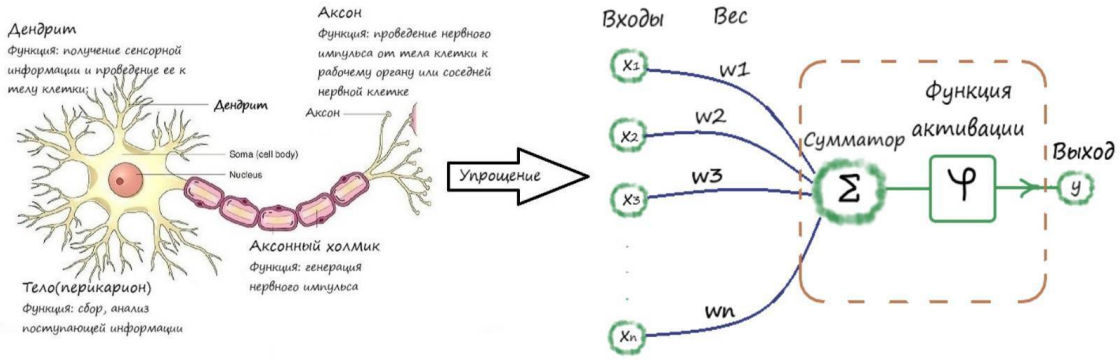
Строение биологического нейрона:



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.

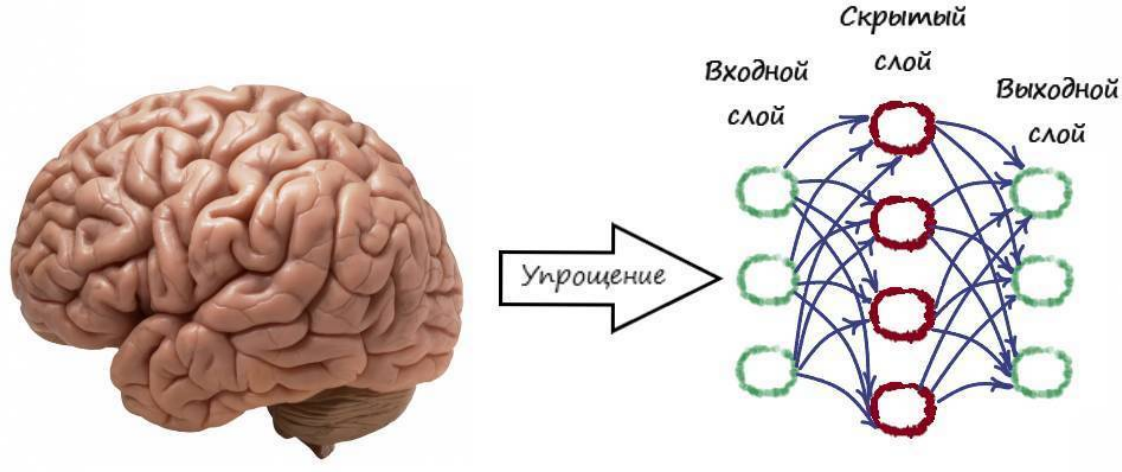
Модель искусственного нейрона (слева – биологический нейрон, справа – искусственный):



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:

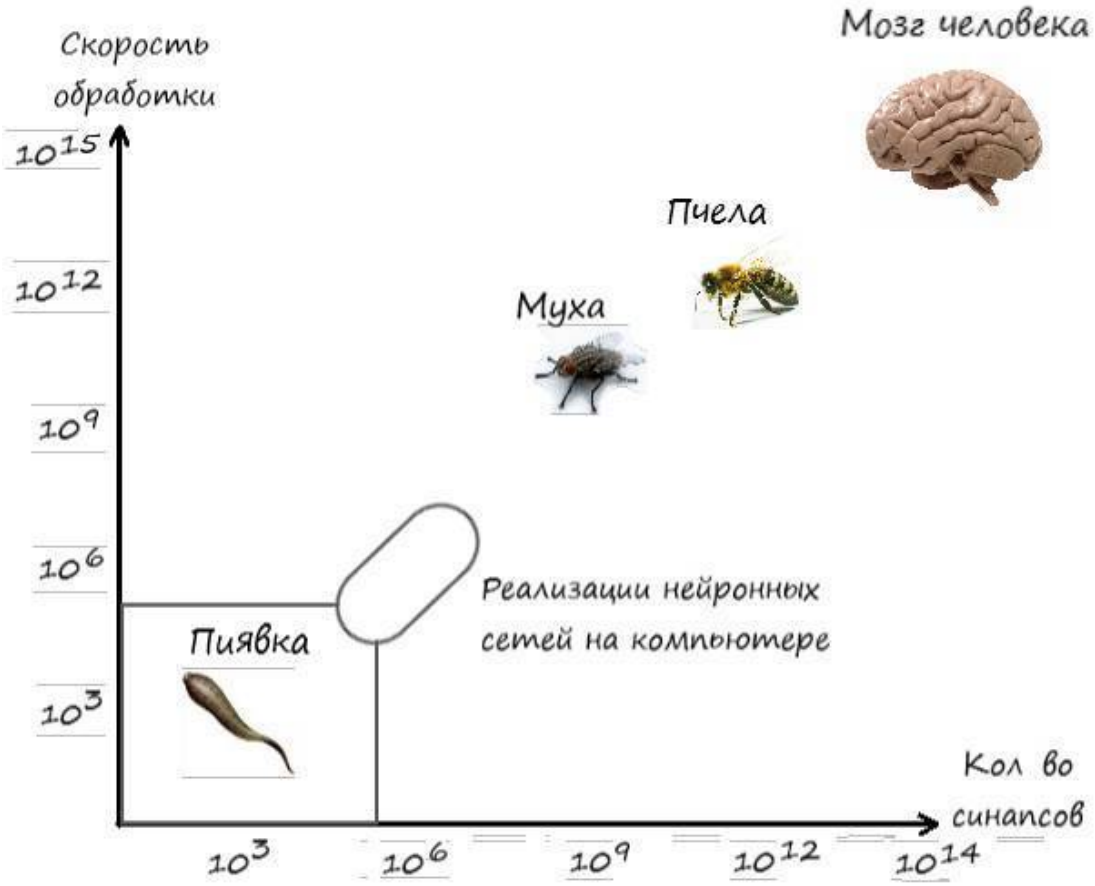


Уровень вычислительной мощности для моделирования ИНС

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.

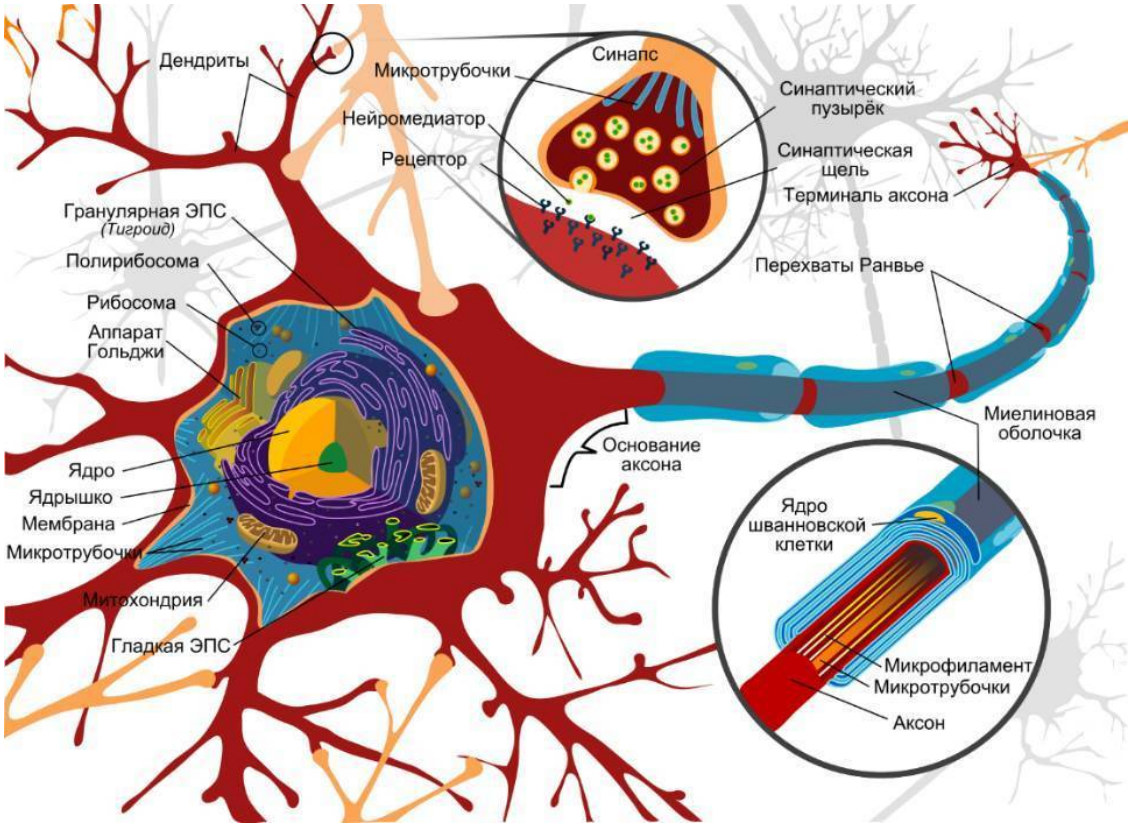
Насколько сильно мы уменьшаем количество нейронов и связей по сравнению с человеческим мозгом:



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.

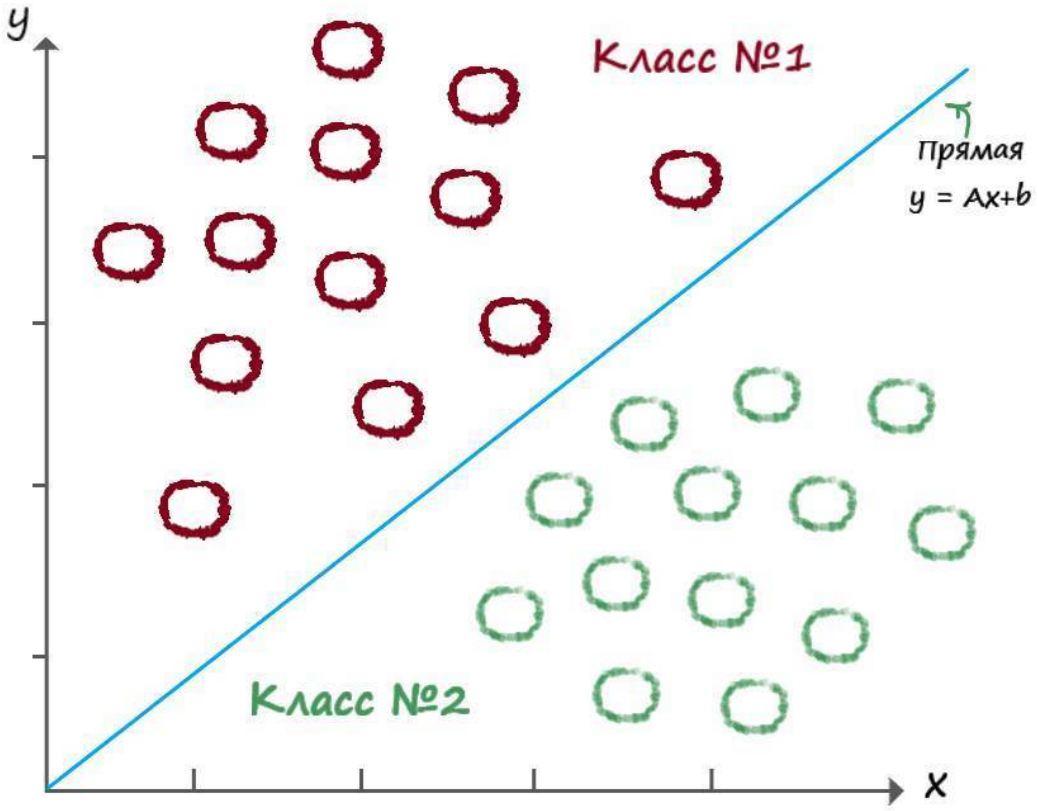
Ну как в любом начинании, нужно начать с самого простого.

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 коэффи-

циента **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:

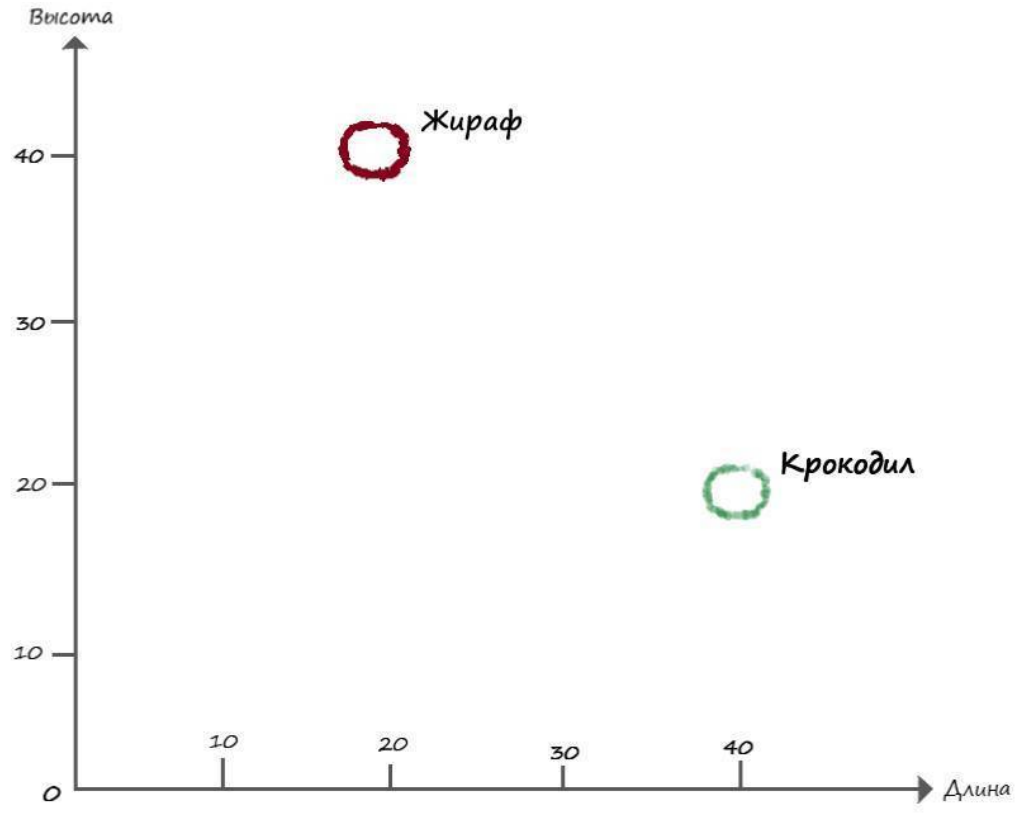
y = Ax

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:

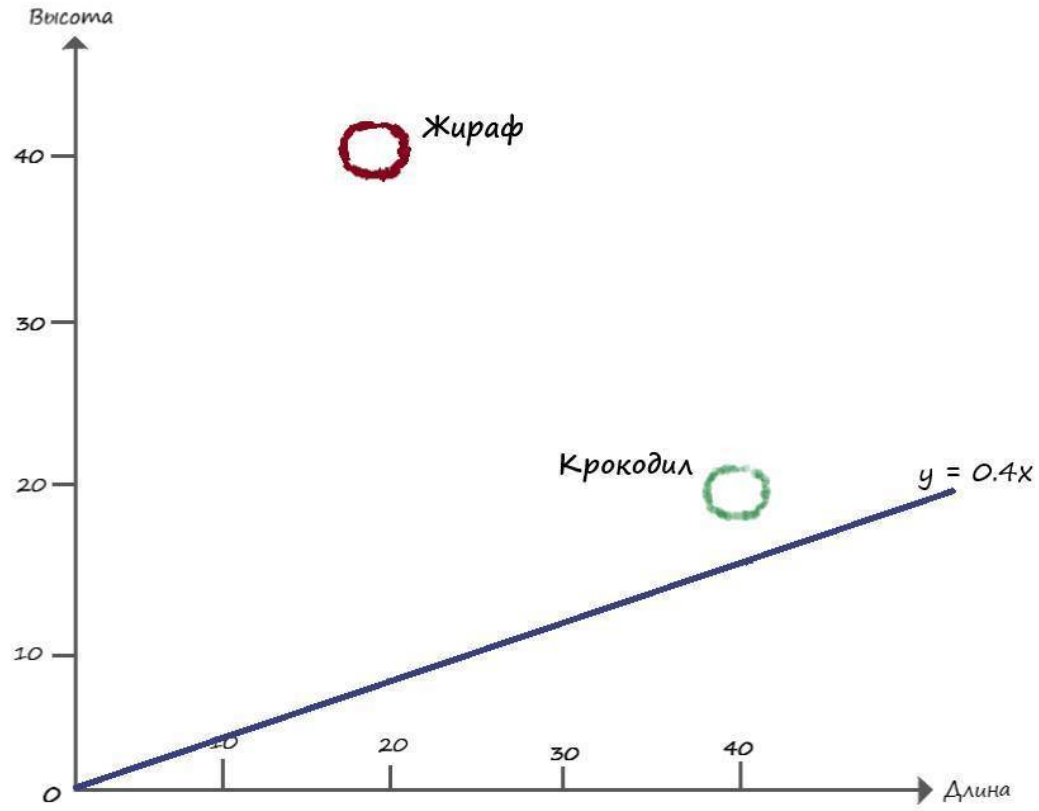
Пример№	Высота	Длинна	Вид животного
1	20	40	Крокодил
2	40	20	Жираф

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 = 0,4x**, 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**.

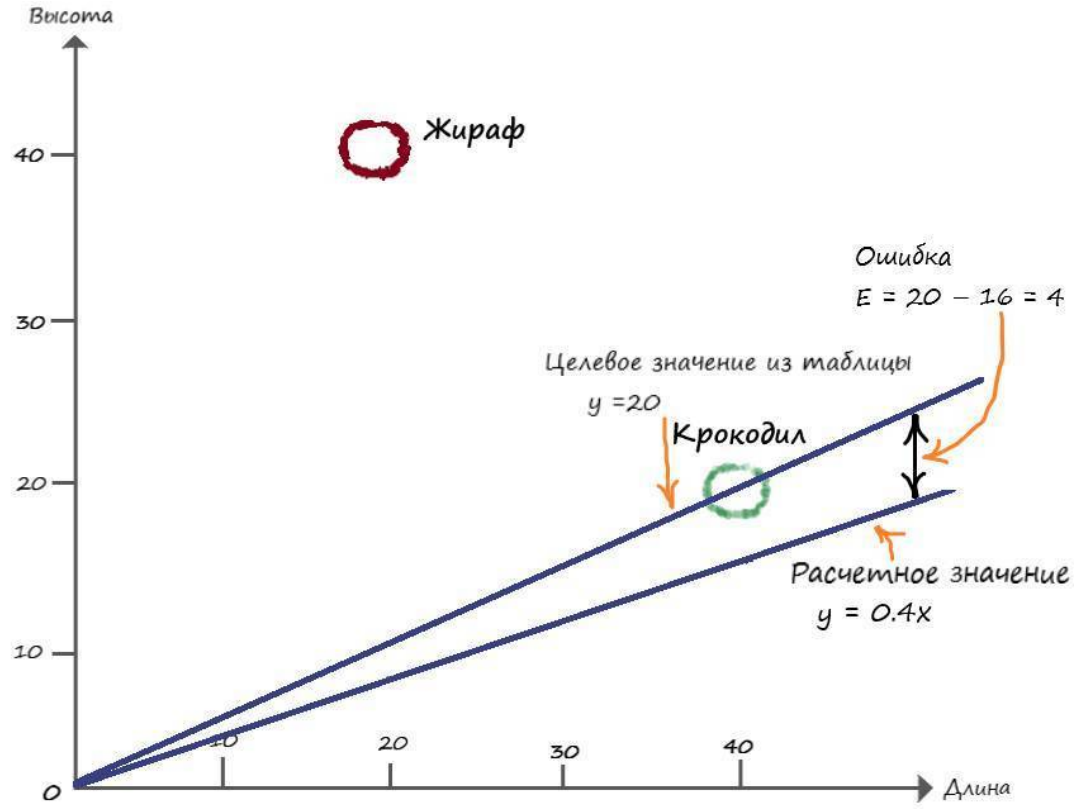
Для того чтоб исправить положение и приподнять нашу линию, введем понятие ошибки **E**, с помощью следующей формулы:

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:

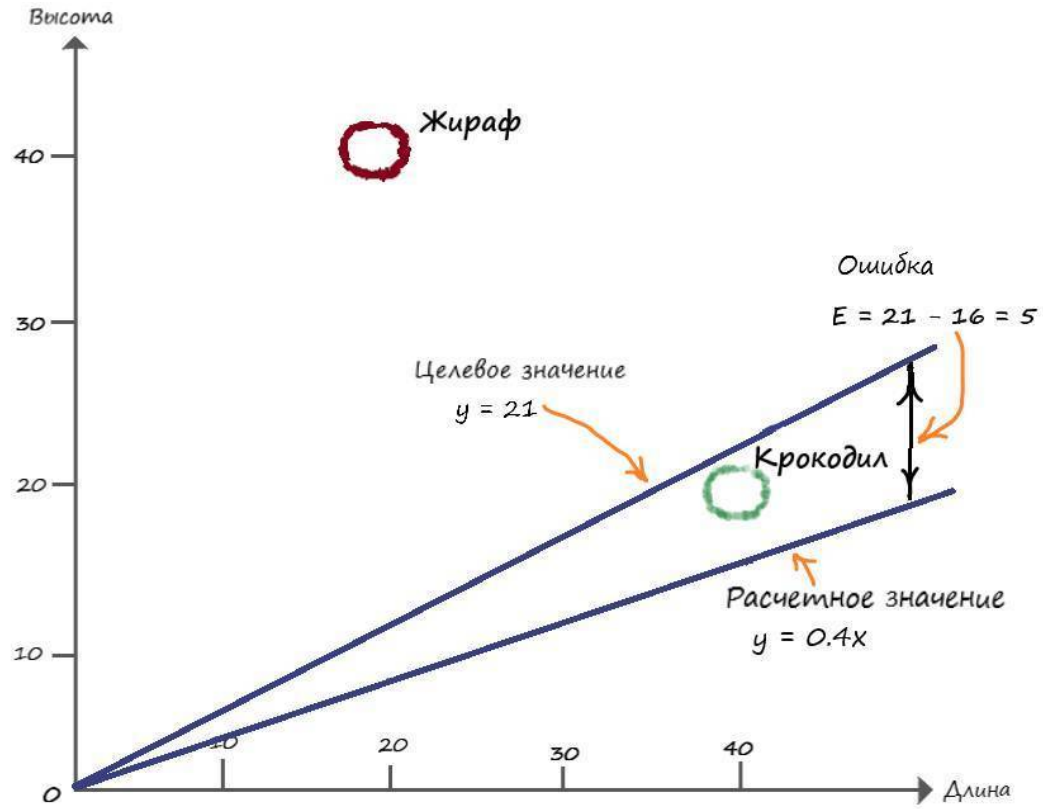


Ну и тут, как мы можем наблюдать, наша линия проходит через точку определяющую вид – крокодил, а нам надо чтобы линия лежала выше.

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:

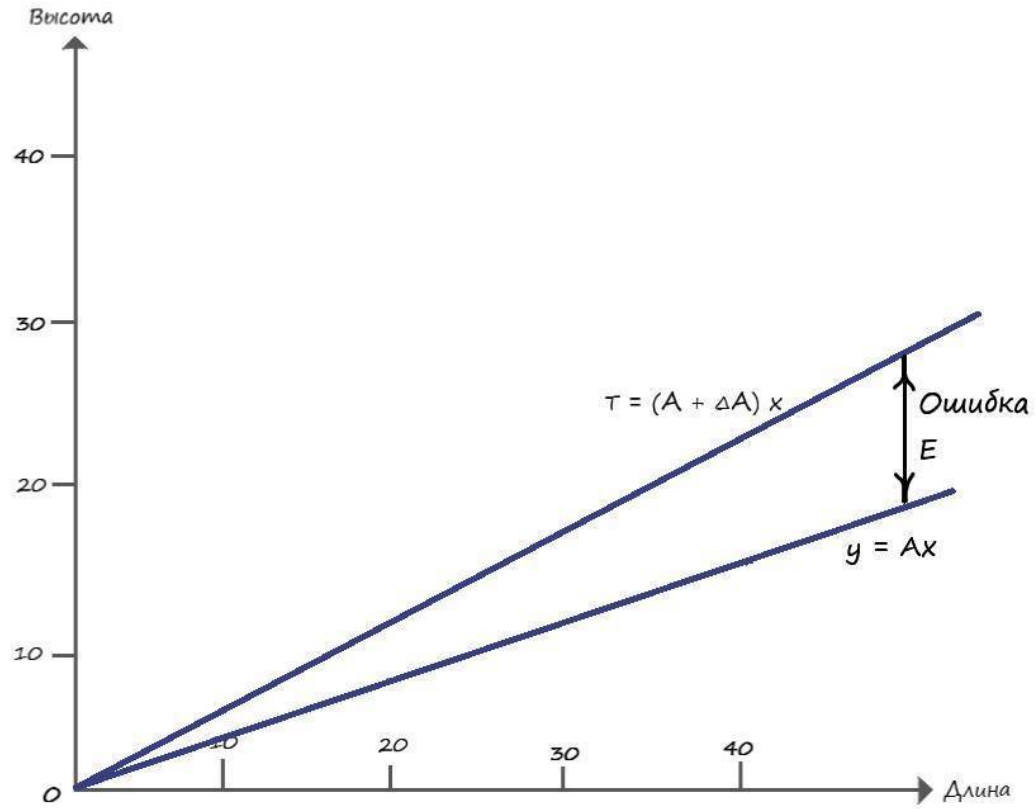
y = Ax

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 + ΔA) 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**:

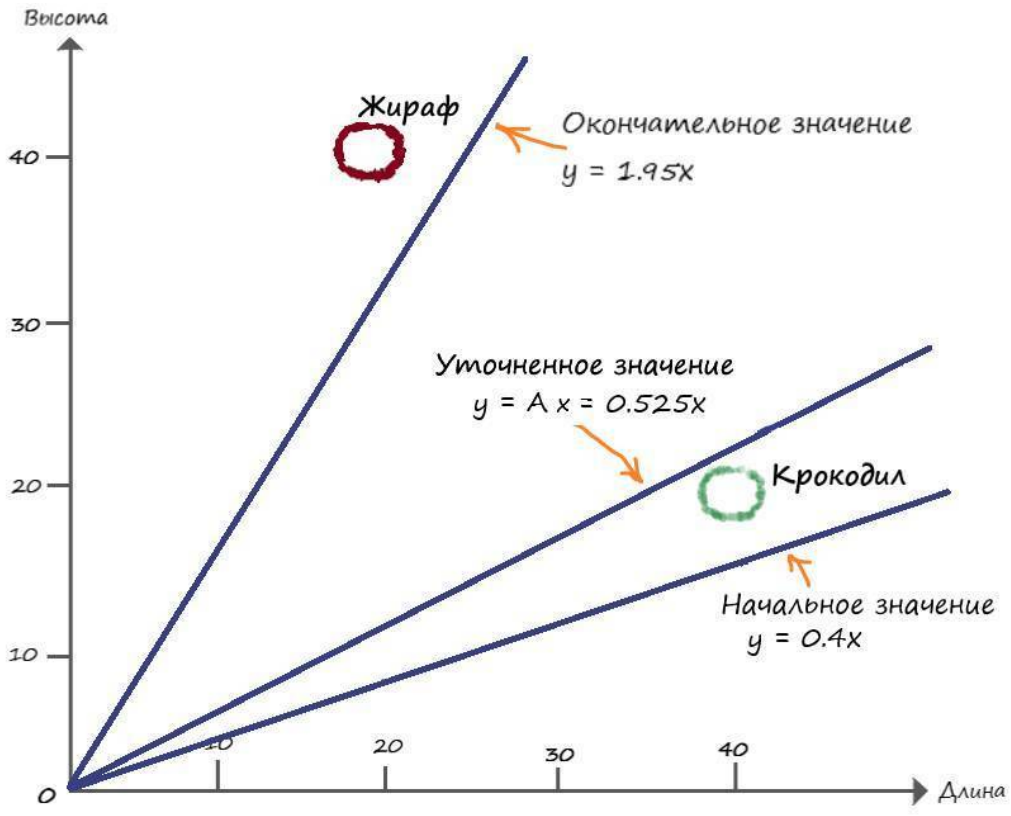
A = A+ΔA = 0,525 +1,425 = 1,95

Получим обновленный ответ:

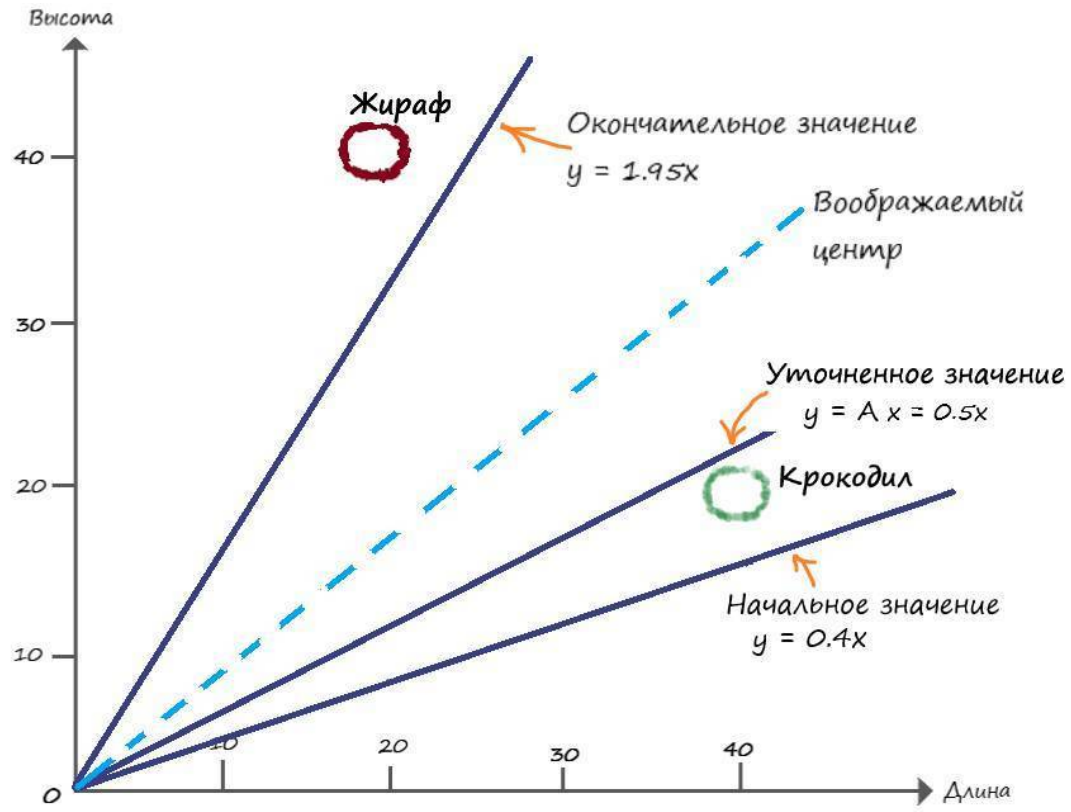
y = A x = 1,95 * 20 = 39

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.

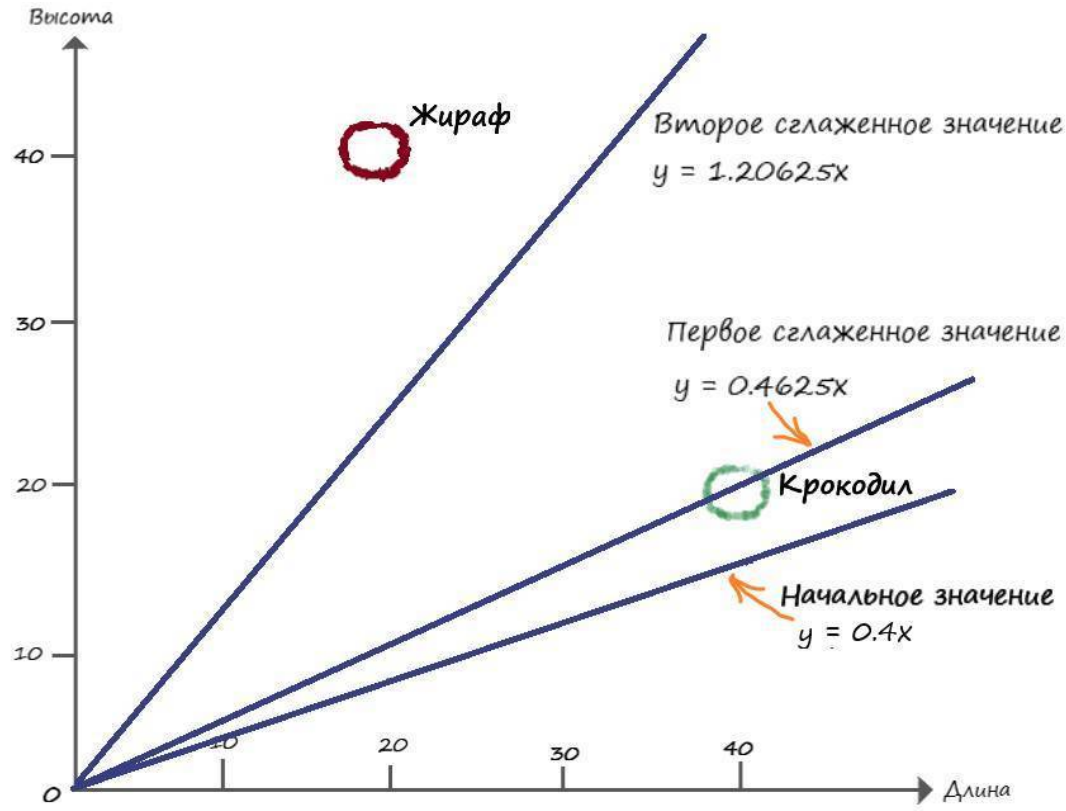
Используя обновлённое на первом прогоне значение **A**, для второго тренировочного примера **y = Ax = O,4625 * 20 = 9,25**.

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.

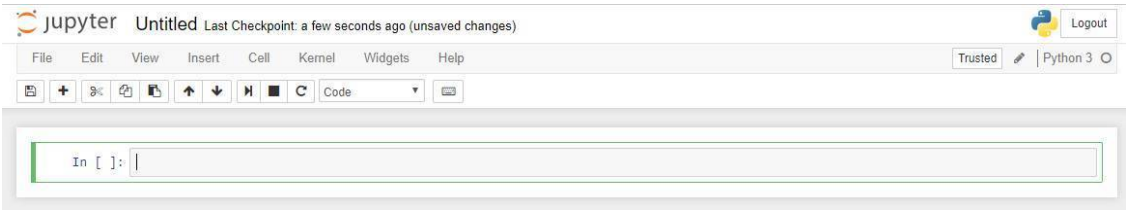
Установка пакета Anaconda Python

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)
```

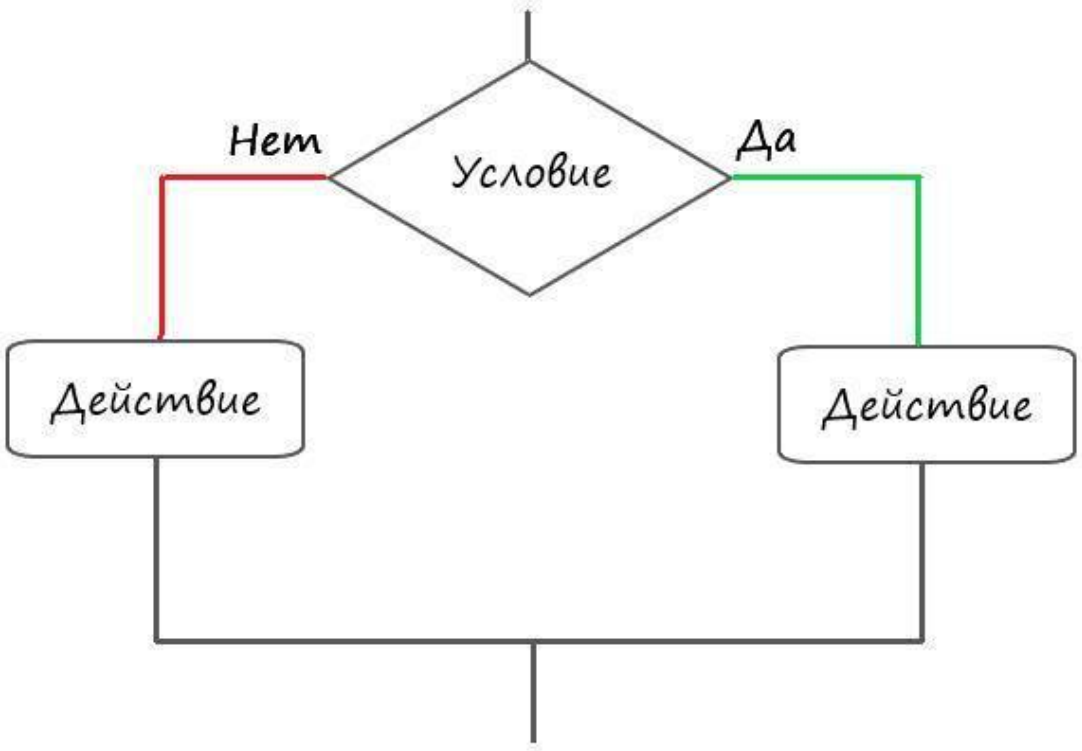
```
32
```

```
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

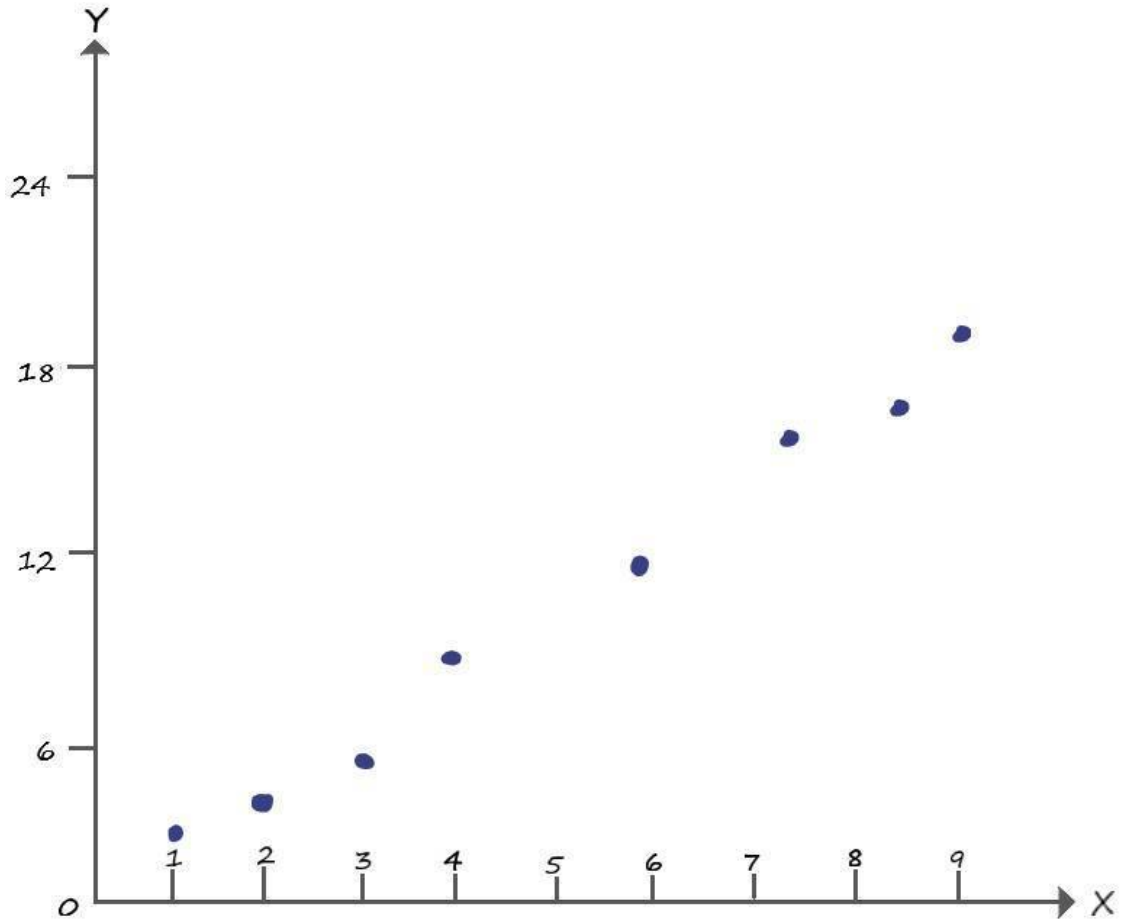
Настало время практически реализовать линейную классификацию. Для этого в Python смоделируем работу искусственного нейрона. Попробуем решить нашу задачу, найдя промежуточные значения, при заданном наборе входных и соответствующим им выходным (целевым) параметрам. Как мы помним – это были высота и длина 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**.

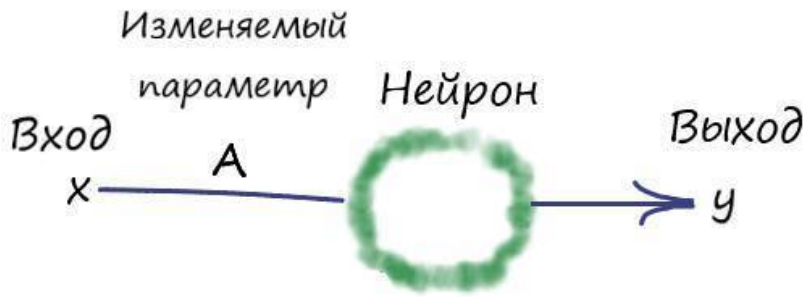
Отобразим для лучшего представления входных данных, график обучающей выборки:



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.

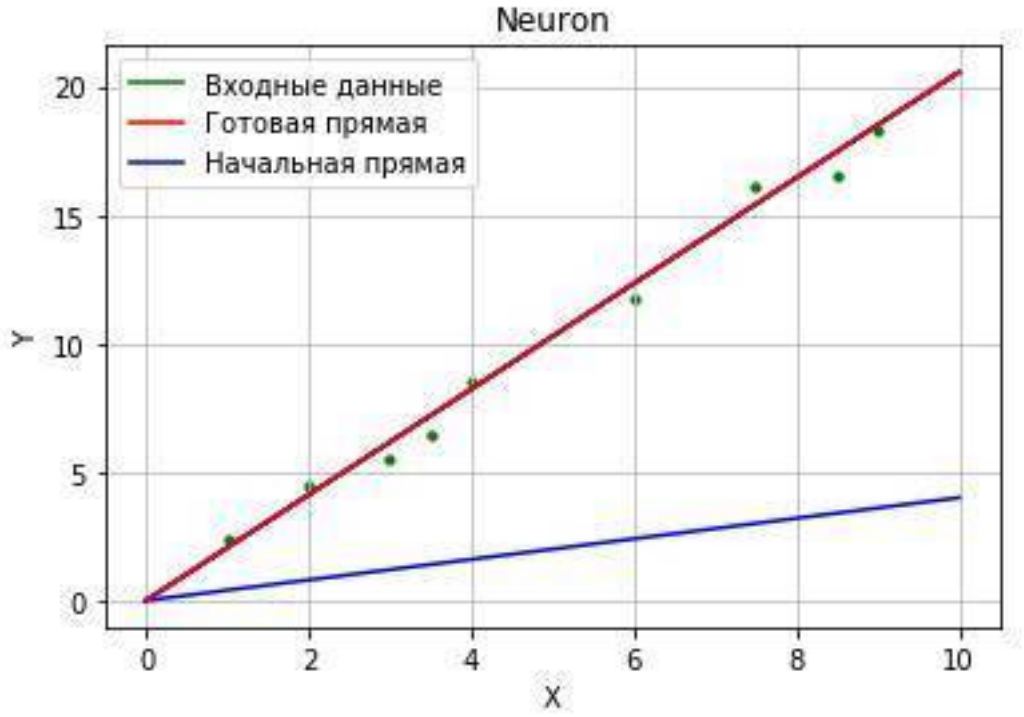
Теперь мы полностью готовы к тому, чтобы представить наши данные и функции в графическом виде.

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 > 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!

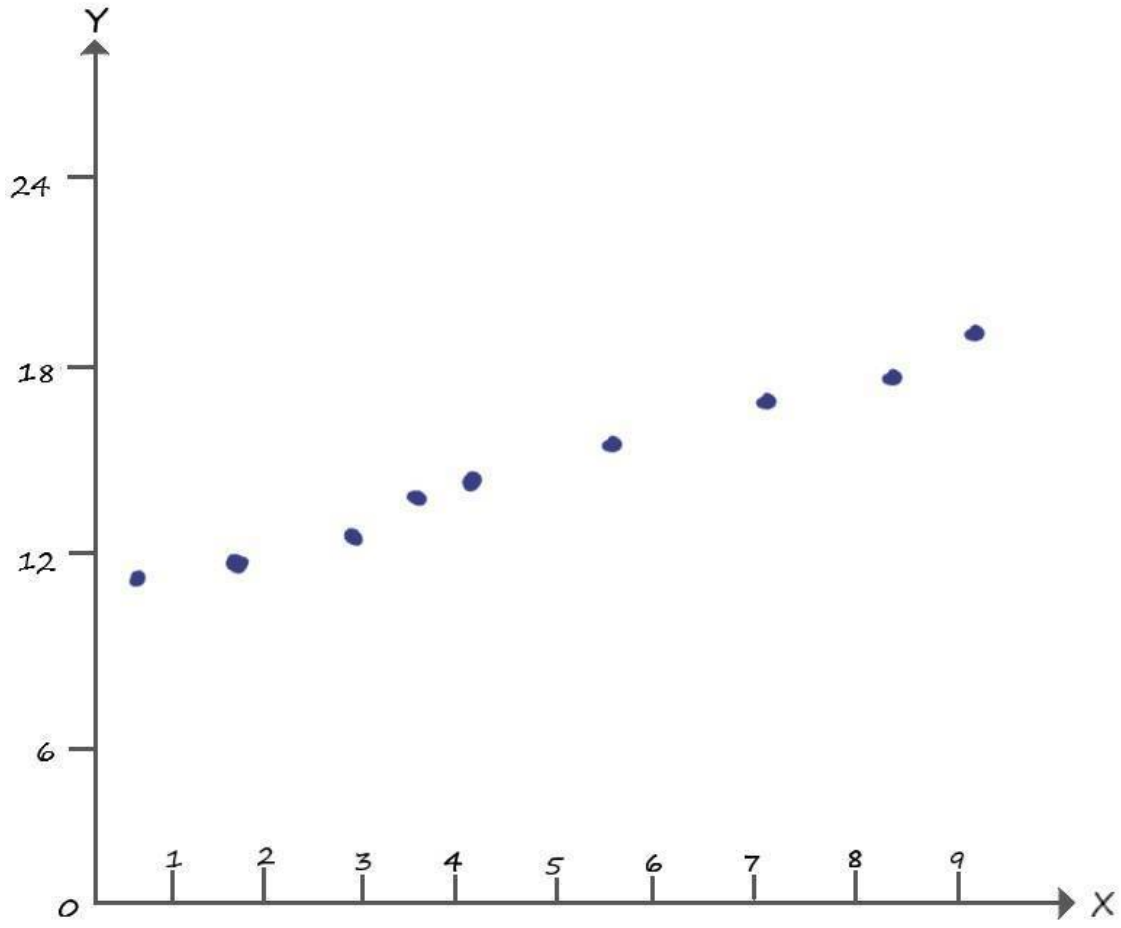
Теперь мы можем поздравить себя! Вся наша работа стала сводиться к тому, чтоб просто подавать на вход нейрона данные, не разбираясь в них самостоятельно. Нейрон сам классифицирует их и даст правильный ответ.

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**.

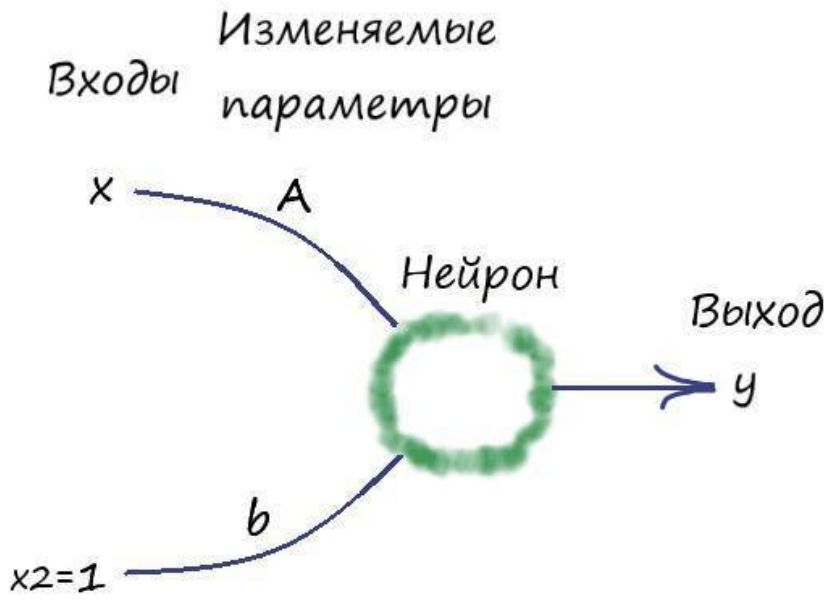
Моделирование нейрона как линейного классификатора со всеми параметрами линейной функции

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 (**x2 = 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!

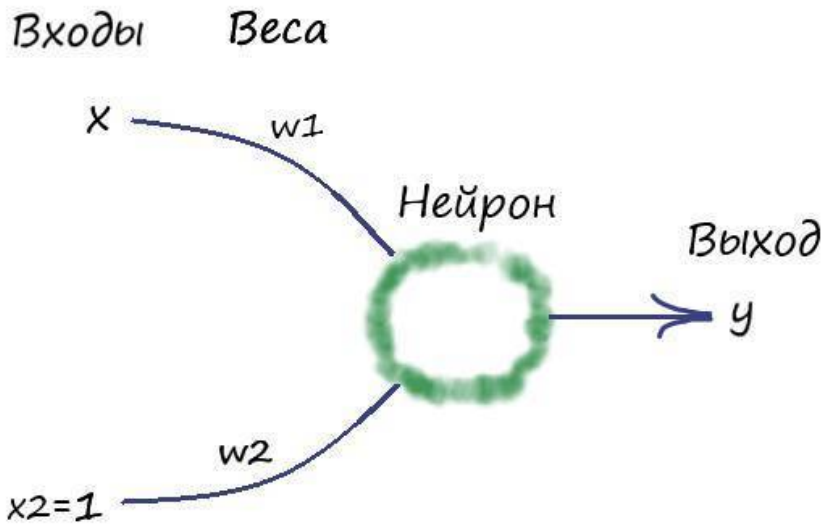
Рассмотрим следующую графическую модель искусственного нейрона:



Где, как говорилось выше, на вход нейрона поступают два входных сигнала **x** (из нашего набора данных) и **x2 = 1**. После чего, эти значения умножаются со своими изменяемыми параметрами, а далее они суммируются: **A*x+b*x2**. Значение этой суммы, а по совместительству – значение функции **y = A*x+b*x2 = 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**.

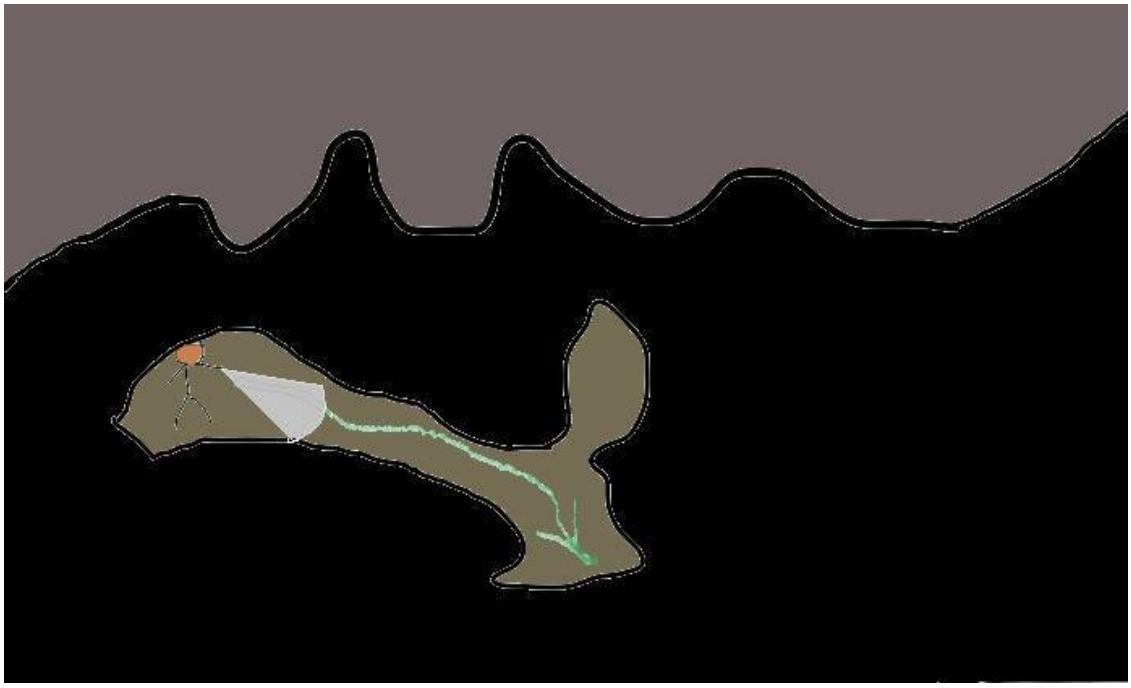
Обновление весовых коэффициентов

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.

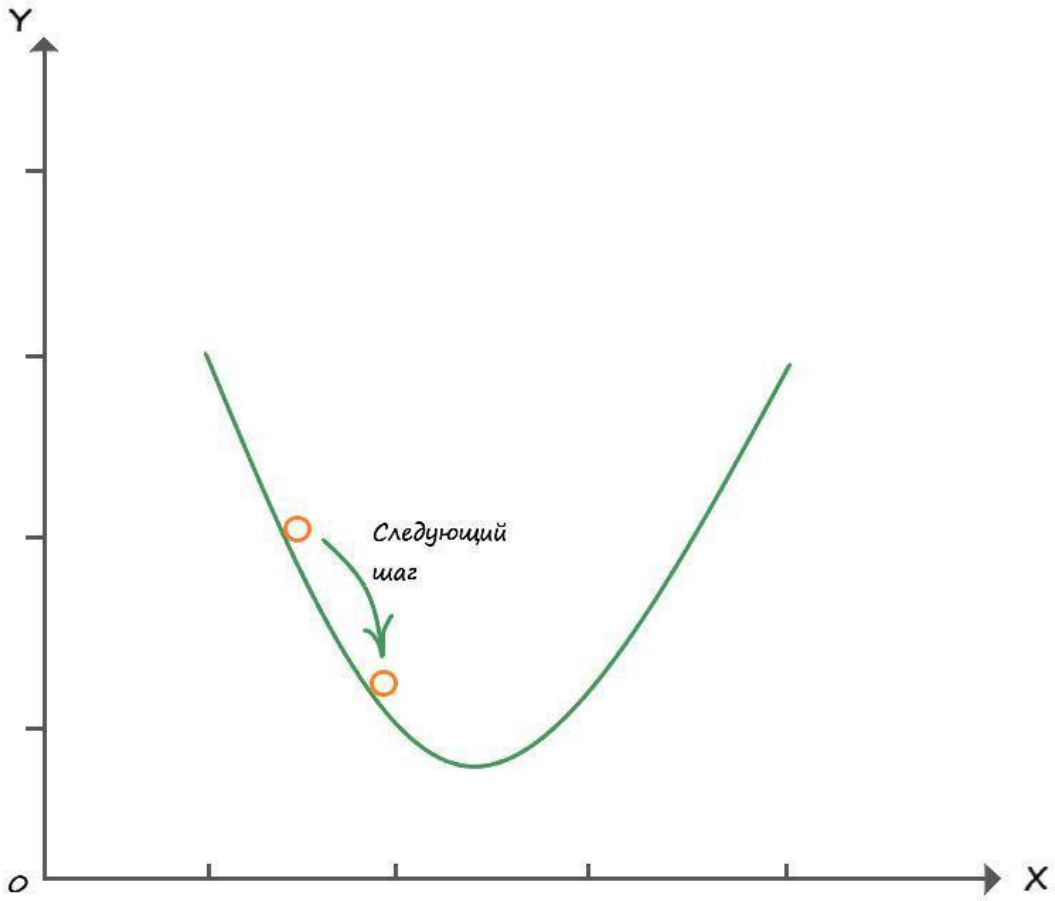
Градиентный спуск – метод нахождения локального минимума или максимума функции с помощью движения вдоль градиента – который, своим направлением указывает направление наибольшего возрастания некоторой величины, значение которой меняется от одной точки пространства к другой, а по величине (модулю) равный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



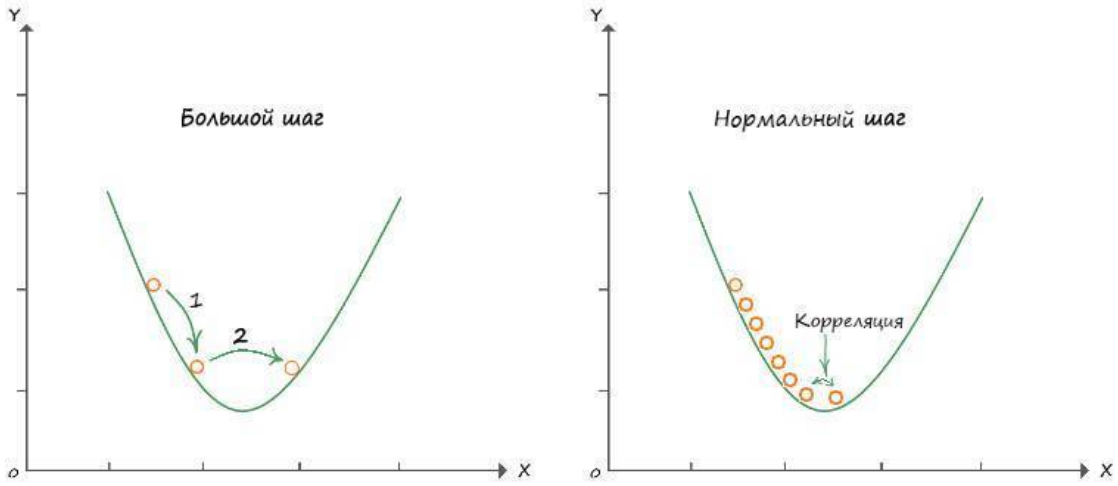
Метод градиентного спуска позволяет находить минимум, даже не располагая знаниями свойств этой функции, достаточными для нахождения минимума другими математическими методами. Если функция очень сложна, где нет простого способа нахождения минимума, мы в этом случае можем применить метод градиентного спуска. 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 (Целевое – 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 + 0,2t

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 у.

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?

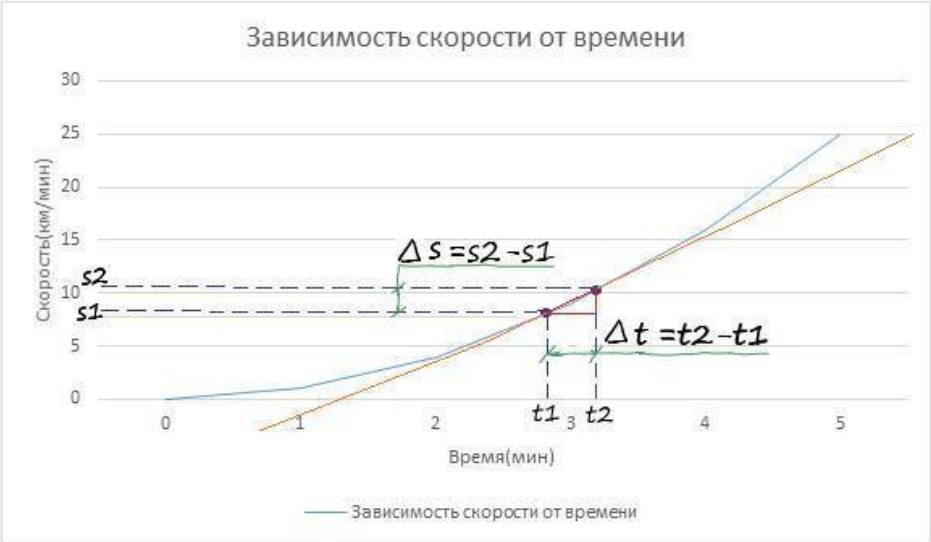
Применение дифференциального исчисления, понятие производной

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:

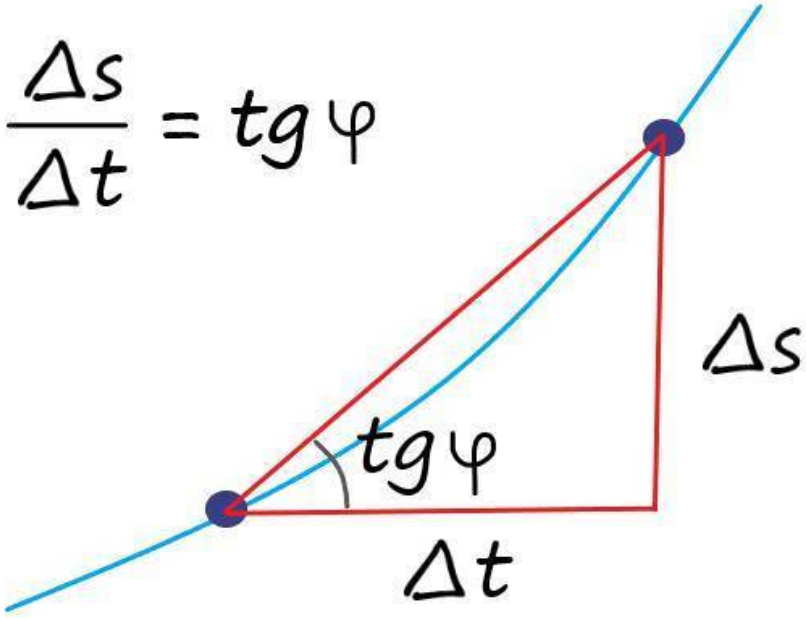
Вы видите, что чем больше скорость в точке касательной, тем её наклон круче. Оба наклона представляют искомую скорость изменения скорости движения. Можно сравнить с вторым примером – линейное изменение.

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:

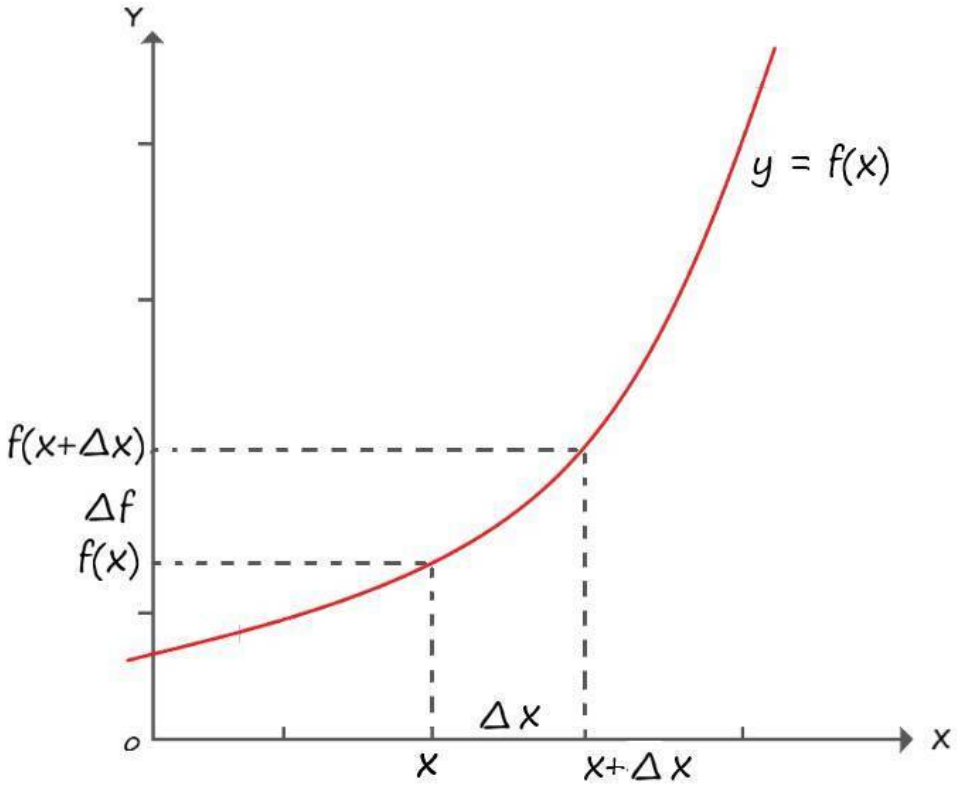
$$\frac{\Delta s}{\Delta t} = \frac{11-7}{3{,}33 - 2{,}67} = 6{,}06$$

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)**, **Δf** и **f(x+Δx)**. Приращение аргумента **Δx**, есть аналог промежутка времени **Δt**, а соответствующее приращение функции – это аналог пути **Δs**, пройденного за время **Δt**.

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:

$$\lim_{x \to 0} \frac{\Delta f}{\Delta x} = \lim_{x \to 0} tg\,\varphi = tg\,\varphi$$

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 (**Δs = 0**), и соответственно тангенса угла не существует:

Δs = s(t+Δt) – s(t) = s(t) – s(t) = 0

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

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 = 0,2t + 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:

$s(t) = t^2$

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

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

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:

$s(t) = t^3$

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

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

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

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:

Δs = s(t+Δt) – s(t) = t + Δt – t = Δt

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:
Δf = f(x+Δx) – f(x) = u(x+Δx) + v(x+Δx) – u(x) – v(x) = u(x) + Δu + v(x) + Δv – u(x) – v(x) = Δu + Δv
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 **Δu/Δx** и **Δv/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:

Δf = f(x+Δx) – f(x) = u(x+Δx) * v(x+Δx) – u(x) * v(x) = (u(x) + Δu) * (v(x) + Δv) – u(x) * v(x) = u(x)v(x) + v(x)Δu + u(x)Δv + ΔuΔv – u(x)v(x) = v(x)Δu + u(x)Δv + ΔuΔv

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 **Δu/Δx**, 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 = y^2

y = x^2+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	$a^x \ln a$
$\sqrt[n]{x}$	$\frac{1}{n\sqrt[n]{x^{n-1}}}$
$\sin x$	$\cos x$
$\cos x$	$-\sin x$
$td\ x$	$\frac{1}{\cos^2 x}$
$ctd\ x$	$\frac{1}{-\sin^2 x}$
$\ln x$	$\frac{1}{x}$
$\log_a x$	$\frac{1}{x}\log_a e$
$ld\ x$	$\frac{1}{x} ld\ e$

$\arcsin x$	$\frac{1}{\sqrt{1-x^2}}$
$\arccos x$	$-\frac{1}{\sqrt{1-x^2}}$
$\operatorname{arctd} x$	$\frac{1}{1+x^2}$
$\operatorname{arcctd} x$	$-\frac{1}{1+x^2}$

Основные свойства

$$(cu)' = cu'$$

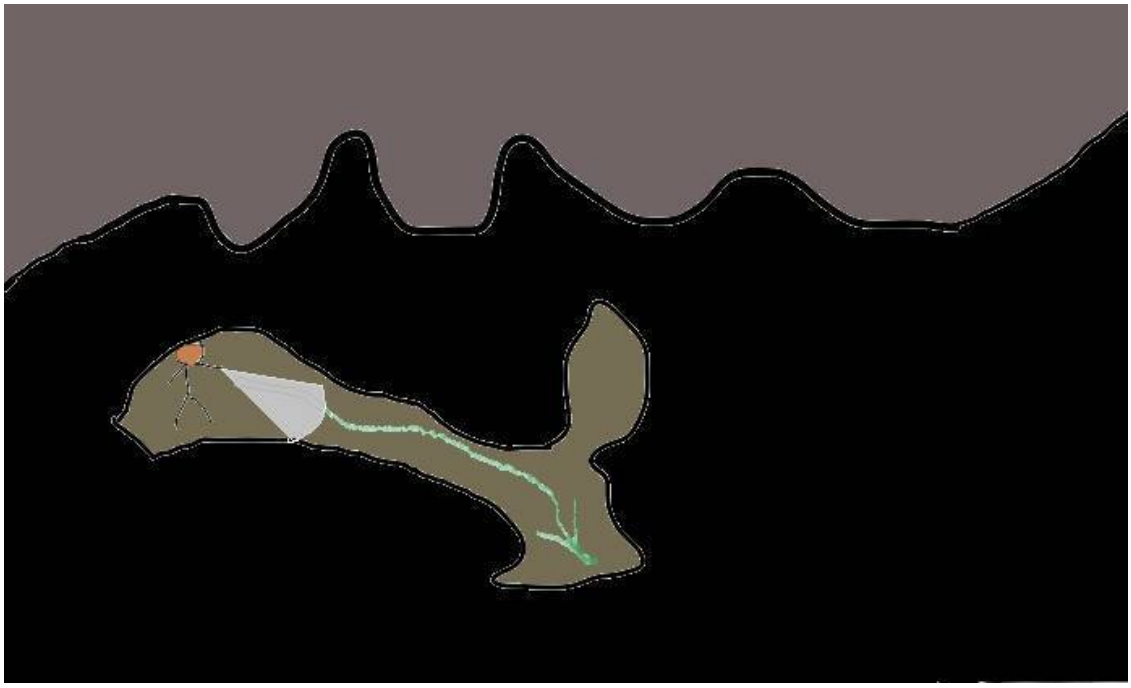
$$(u \pm v)' = u' \pm v'$$

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

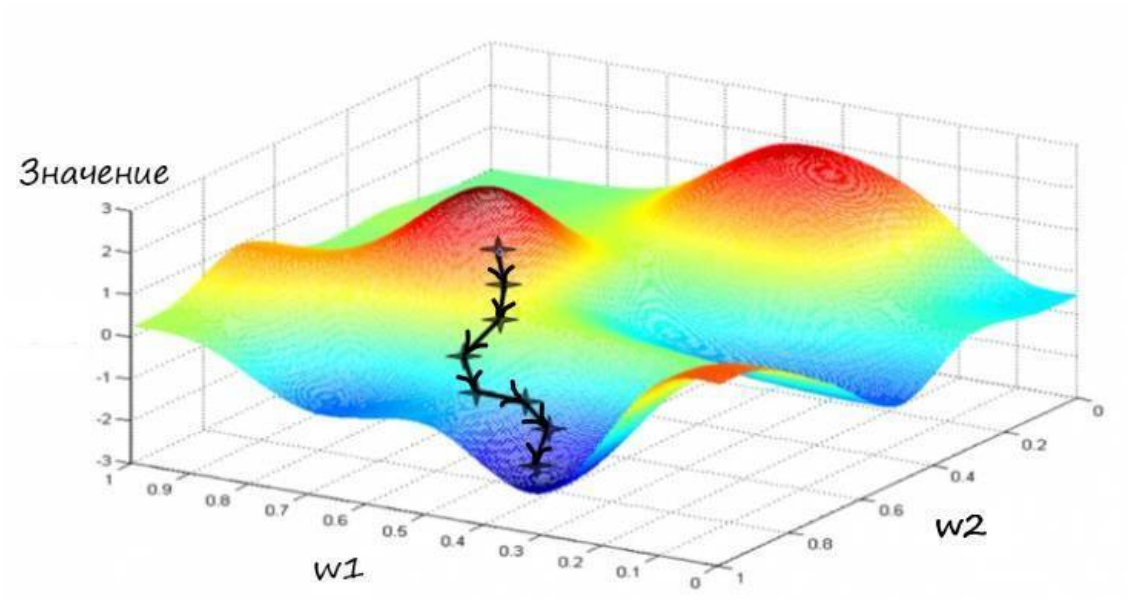
$$\left(\frac{u}{v}\right)' = \frac{u' * v - u * v'}{v^2}$$

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 коорди-

наты весов, которые необходимо изменять в соответствии с координатами точки – которая движется вниз.

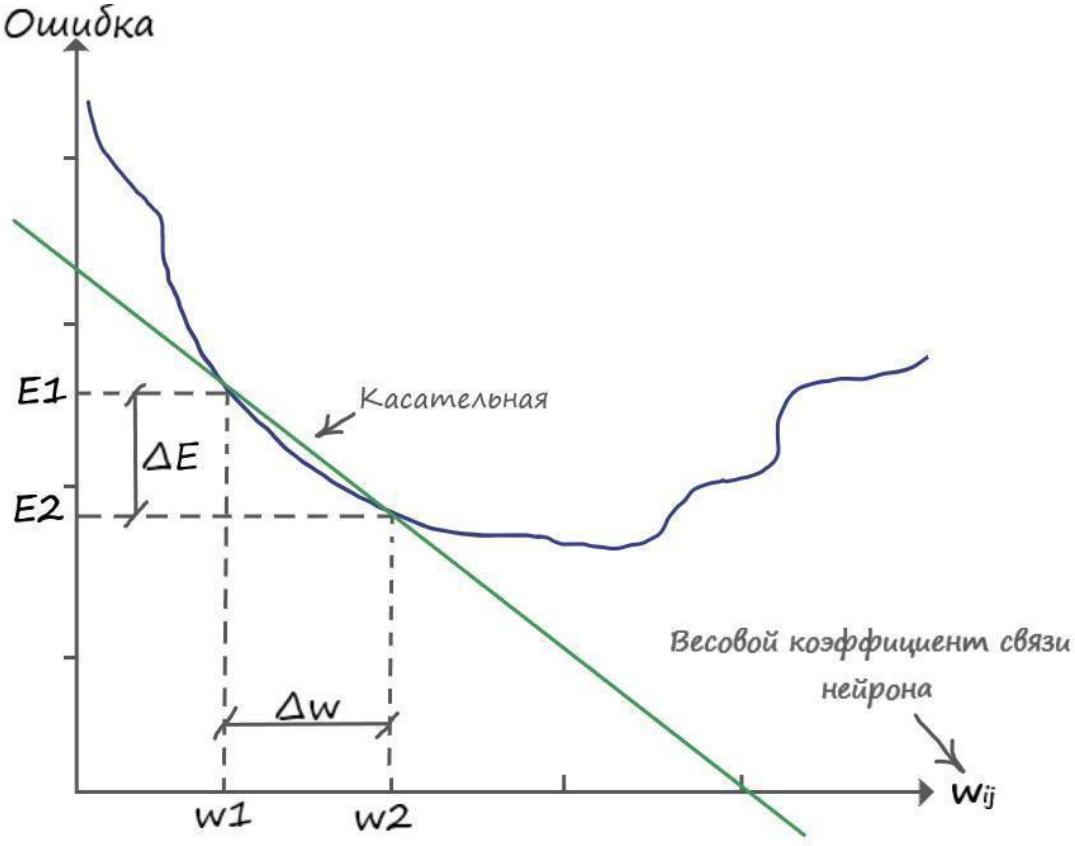
Представим ось значение, как ось ошибка. Очень хорошо видно, что функция ошибки общая для всех значений весов. Соответственно – координаты точки значения ошибки, при определенных значениях весовых коэффициентов, тоже общие.

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 wij*xi**, где **xi** – известная величина (константа), а весовые коэффициенты **wij** – переменная, производная по которой, дает как мы знаем единицу, то взвешенную сумму можно разбить на сумму простых множителей:

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

Откуда нетрудно найти:

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

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

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

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:

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

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

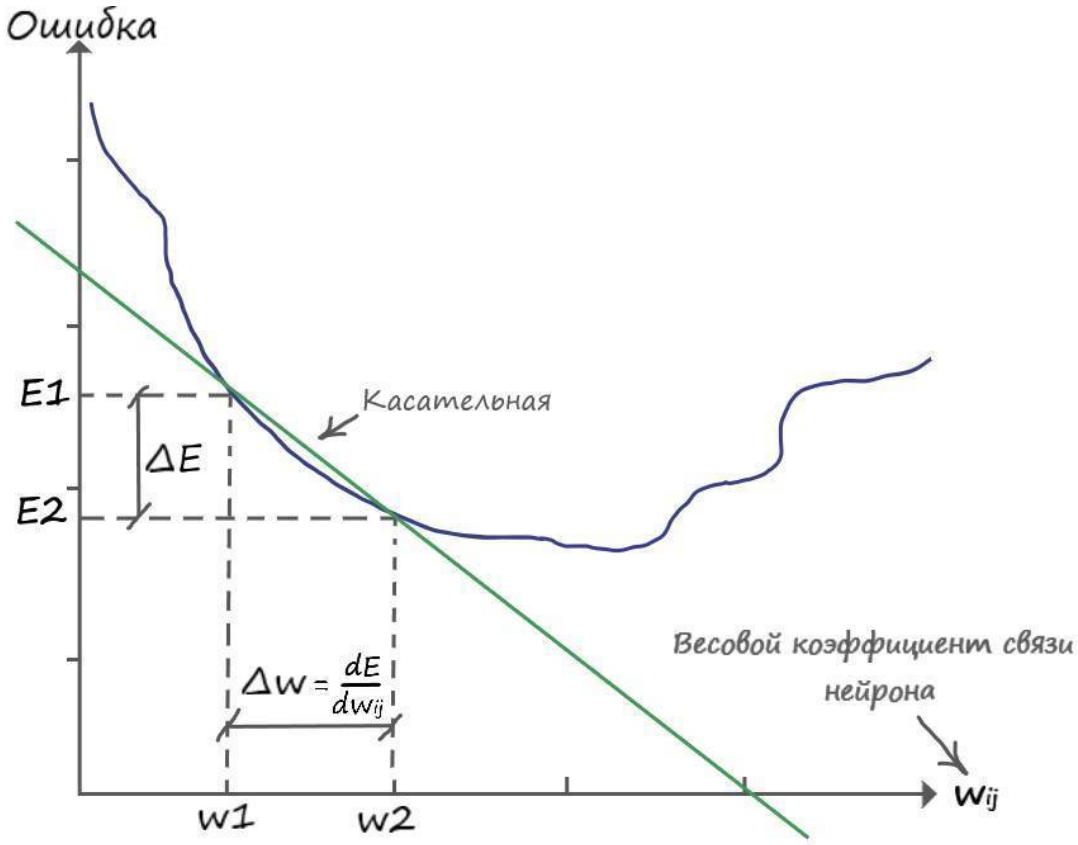
Как мы обновляем весовые коэффициенты

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 (**Δw**), откуда: (**w2 = =w1+Δw**). Приравняв (**Δw**) к производной ошибки (величину которой уже знаем), мы спускаемся на эту величину в сторону уменьшения ошибки.

Так же замечаем, что (**E2 – E1 = -ΔE**) и (**w2 – w1 = Δw**), откуда делаем вывод:

Δw = -ΔE/Δw

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 wij = старый wij -(– ΔE/Δw)

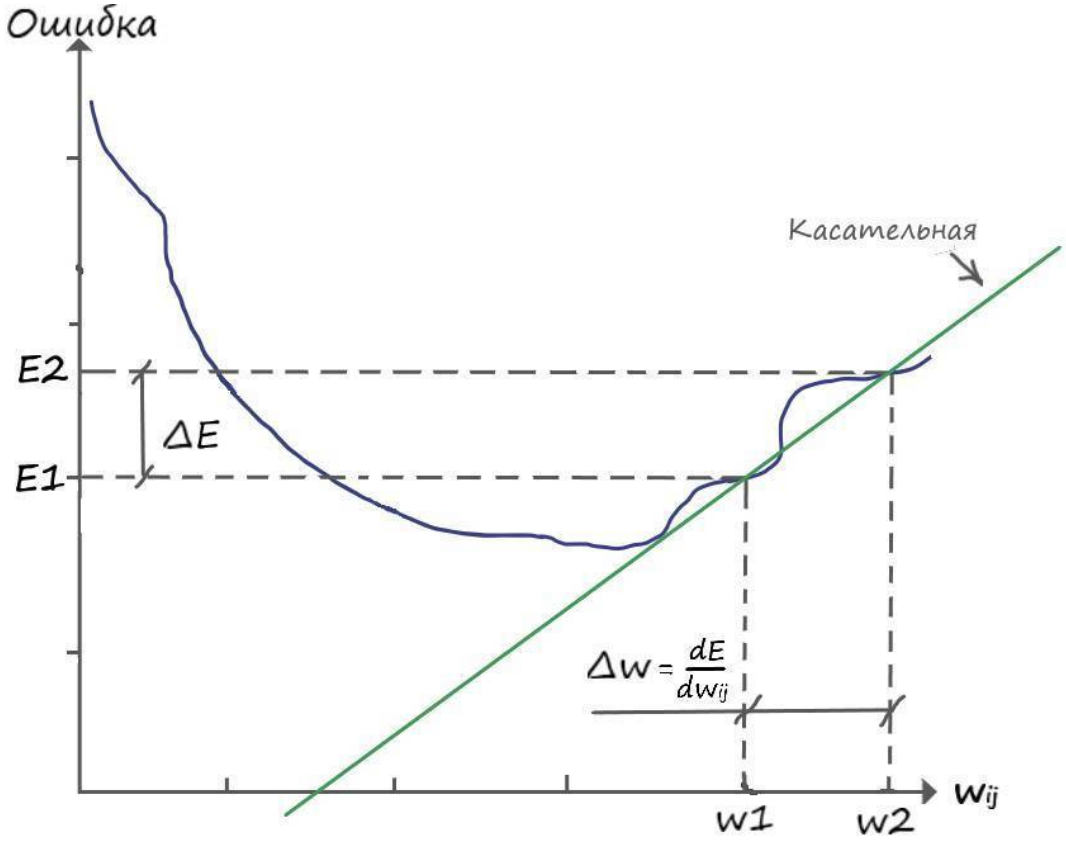
Знак минус, для того чтобы обновить вес в большую сторону, для уменьшения ошибки. На примере графика – от **w1** до **w2**.

В общем виде выражение записывается как:

новый wij = старый wij – dE/dwij

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 (**E2 – E1 = ΔE**) и (**w2 – w1 = Δw**), откуда делаем вывод:

Δw = ΔE/Δw

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 wij = старый wij - ΔE/Δw

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 wij = старый wij – dE/dwij

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.

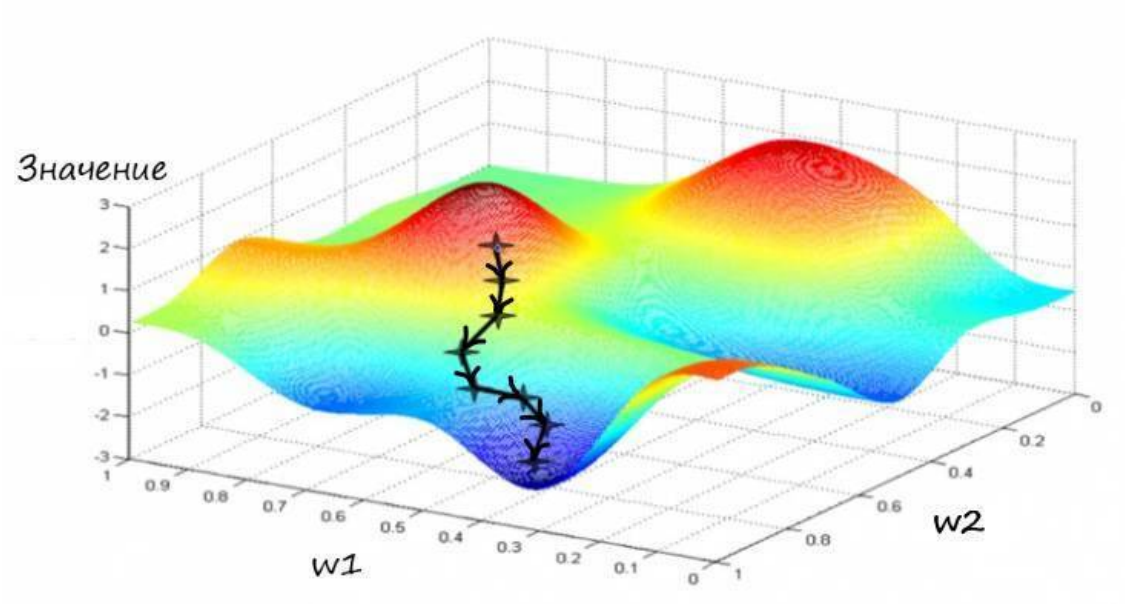
Ну и наконец, давайте запишем окончательный вариант выражения при обновлении весовых коэффициентов:

новый wij = старый wij – L*(dE/dwij)

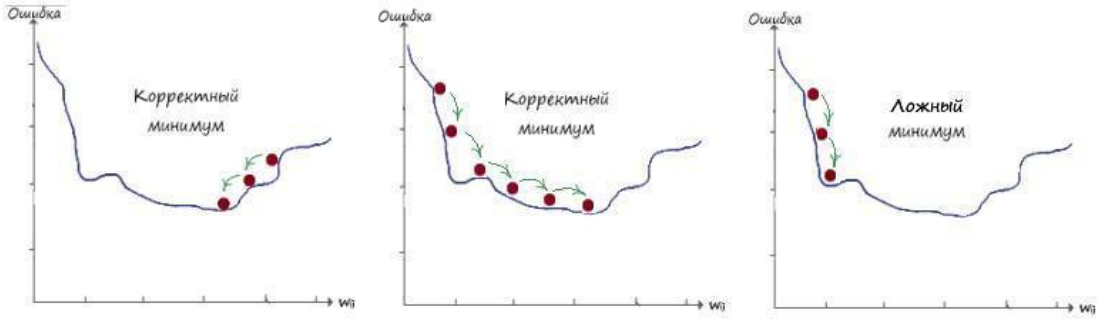
Еще раз можем убедиться, в постепенном улучшении свойств, в ходе эволюции искусственного нейрона. Много из того что реализовывали ранее остается, лишь небольшая часть подверглась эволюционному улучшению.

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 = 2x + 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/neuralmaster

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)
```

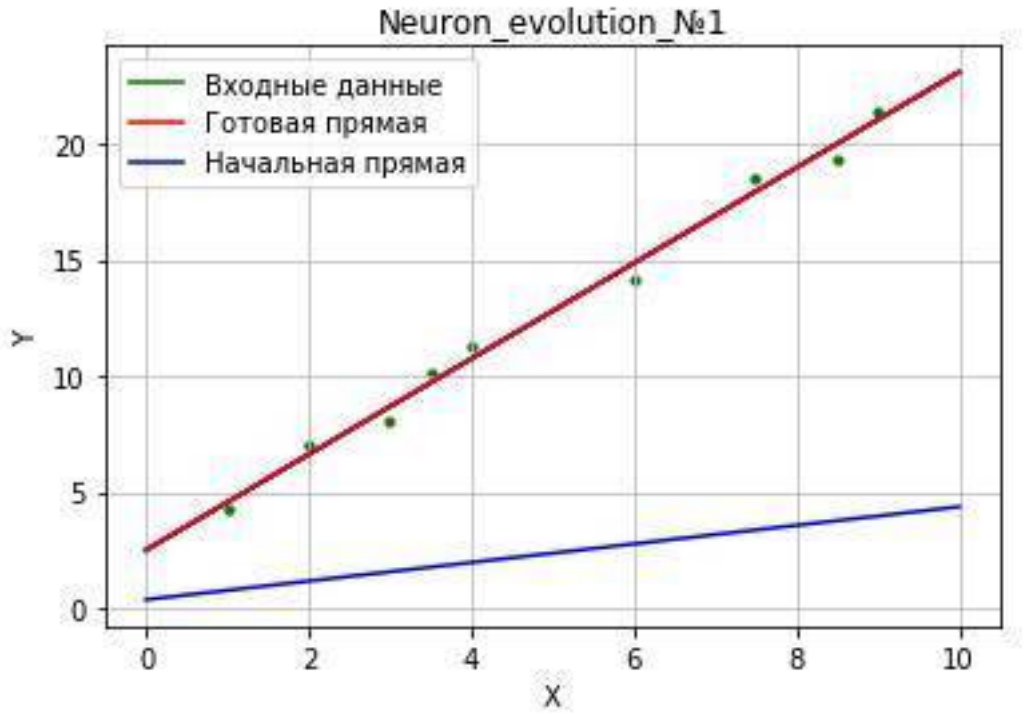
Метод модуля random – uniform(from, to), генерирует случайное вещественное число от from до to включительно.

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 > 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 = Ax1+b**), нет произведения на значение переменной, то на второй вход, в качестве данных, всегда поступает единица (**x2 = 1**). Откуда на выходе получаем взвешенную сумму: **y = Ax1+bx2**. При **x2 = 1**, на выходе получаем **y = Ax1+b**. И наконец, назвав коэффициенты, при входных данных – весовыми коэффициентами, изменили их обозначение – **w1 = A**, а **w2 = b**, в итоге: **y = w1x1+w2**.

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.

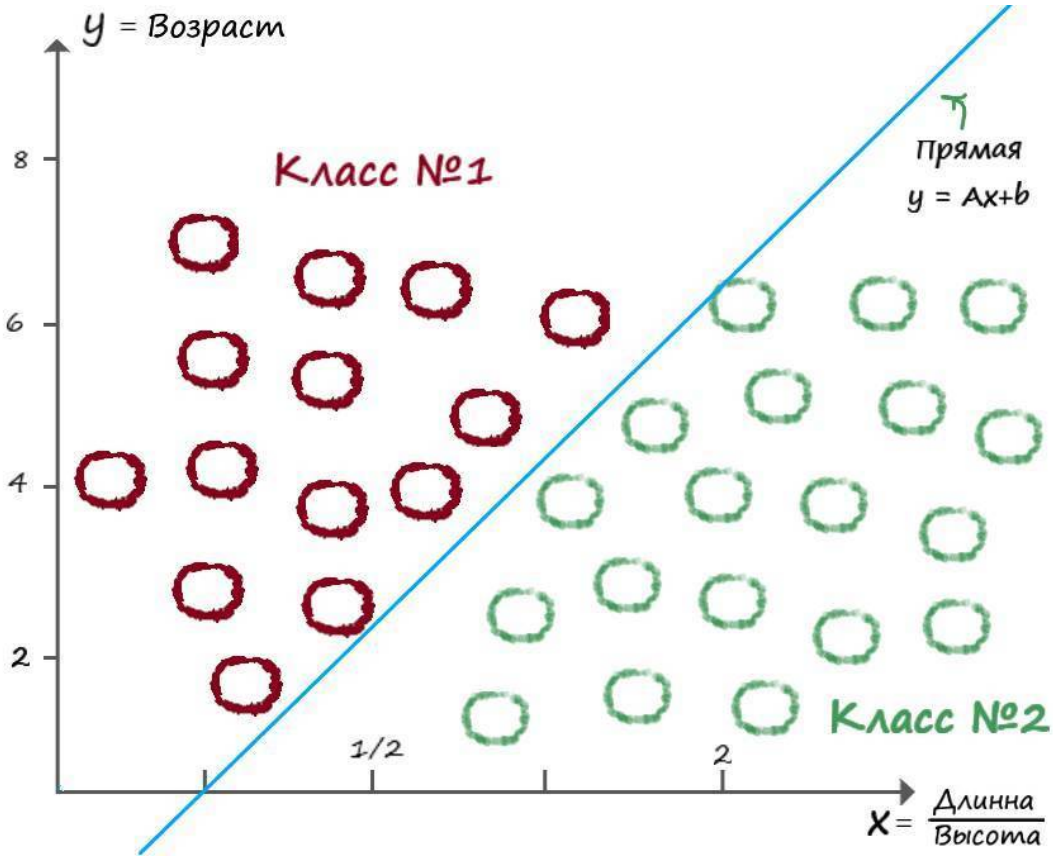
ГЛАВА 5

Больше входных данных

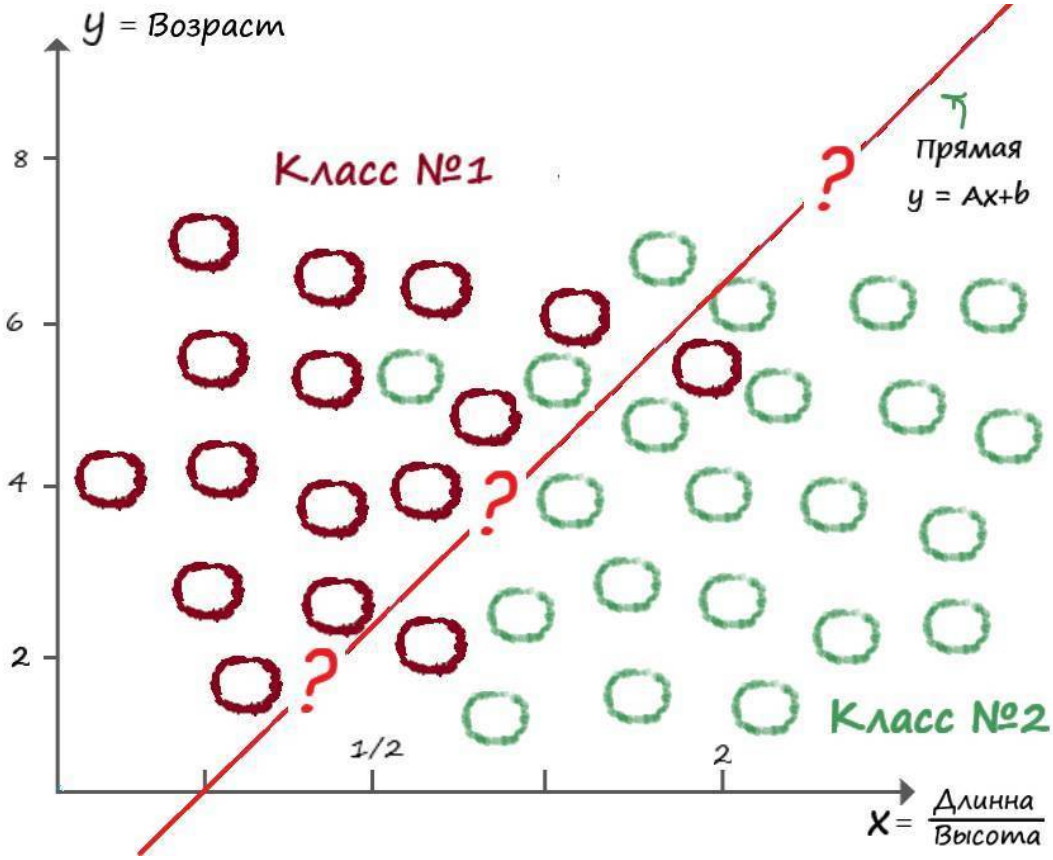
А что будет если добавить на вход искусственного нейрона, еще больше данных? Для начала, хотя бы еще один…

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).

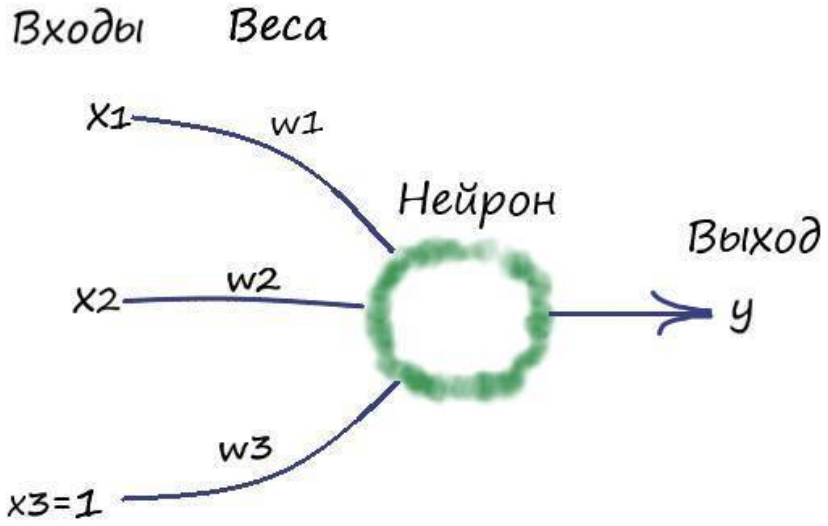
Рассмотрим логическую функцию (И). Такая функция равна нулю для любого набора входных аргументов, кроме набора (x1 = 1, x2 = 1):

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:

Конец ознакомительного фрагмента.