

# Создай свою нейросеть

Тарик Рашид

Перевел Петр Радько

Сентябрь 2017

# Оглавление

<b>1</b>	<b>Как они работают</b>	<b>1</b>
1.1	Легко для меня, тяжело для тебя . . . . .	2
1.2	Простая предсказательная машина . . . . .	4
1.3	Классификация это почти что предсказание . . . . .	10
1.4	Тренировка простого классификатора . . . . .	15
1.5	Иногда одного классификатора недостаточно . . . . .	27
1.6	Нейроны — природные вычислительные машины . . . . .	33
1.7	Проход сигнала через нейросеть . . . . .	44
1.8	Умножать матрицы полезно... Серьезно! . . . . .	48
1.9	Трехслойная нейросеть и произведение матриц . . . . .	55
1.10	Калибровка весов нескольких связей . . . . .	64
1.11	Обратное распространение ошибки от выходных нейронов . . . . .	67
1.12	Обратное распространение ошибки на множество слоев . . . . .	70
1.13	Обратное распространение ошибки и произведение матриц . . . . .	75
<b>2</b>	<b>The Second Chapter</b>	<b>79</b>

# Глава 1

## Как они работают

*Ищите вдохновение во всем, что вас окружает.*

## 1.1 Легко для меня, тяжело для тебя

Все компьютеры являются калькуляторами в душе. Они умеют очень быстро считать.

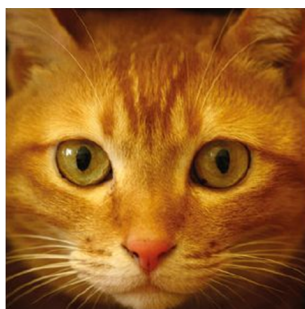
Не стоит их в этом упрекать. Они отлично выполняют свою работу: считают цену с учетом скидки, начисляют долговые проценты, рисуют графики по имеющимся данным и так далее.

Даже просмотр телевизора или прослушивание музыки с помощью компьютера представляют собой выполнение огромного количества арифметических операций снова и снова. Это может прозвучать удивительно, но отрисовка каждого кадра изображения из нулей и единиц, полученных через интернет задействует вычисления, которые не сильно сложнее тех задач, которые мы все решали в школе.

Однако, способность компьютера складывать тысячи и миллионы чисел в секунду вовсе не является искусственным интеллектом. Человеку сложно так быстро складывать числа, но согласитесь, что эта работа не требует серьезных интеллектуальных затрат. Надо придерживаться заранее известного алгоритма по складыванию чисел и ничего более. Именно этим и занимаются все компьютеры — придерживаются четкого алгоритма.

С компьютерами все ясно. Теперь давайте поговорим о том, в чем мы хороши по сравнению с ними.

Посмотрите на картинки ниже и определите, что на них изображено:



Вы видите лица людей на первой картинке, морду кошки на второй и дерево на третьей. Вы распознали объекты на этих картинках. Заметьте, что вам хватило лишь взгляда, чтобы безошибочно понять, что на них изображено. Мы редко ошибаемся в таких вещах.

Мы мгновенно и без особого труда воспринимаем огромное количество информации, которое содержат изображения и очень точно определяем объекты на них. А вот для любого компьютера такая задача встанет поперек горла.

Проблема	Компьютер	Человек
Быстро оперировать множеством больших чисел	Легко	Сложно
Найти лица на фотографии с толпой людей	Сложно	Легко

У любого компьютера вне зависимости от его сложности и скорости нет одного важного качества — интеллекта, которым обладает каждый человек.

Но мы хотим научить компьютеры решать подобные задачи, потому что они быстрые и не устают. Искусственный интеллект как раз занимается решением подобного рода задач.

Конечно компьютеры и дальше будут состоять из микросхем. Задача искусственного интеллекта — найти новые **алгоритмы** работы компьютера, которые позволят решать интеллектуальные задачи. Эти алгоритмы не всегда идеальны, но они решают поставленные задачи и создают впечатление, что компьютер ведет себя как человек.

**Ключевые мысли:**

- Есть задачи легкие для обычных компьютеров, но вызывающие трудности и людей. Например, умножение миллиона чисел друг на друга.
- С другой стороны, существуют не менее важные задачи, которые невероятно сложны для компьютера и не вызывают проблем у людей. Например, распознавание лиц на фотографиях.

## 1.2 Простая предсказательная машина

Давайте начнем с чего-нибудь очень простого. Дальше мы будем отталкиваться от материала, изученного в этом разделе.

Представьте себе машину, которая получает вопрос, "обдумывает" его и затем выдает ответ. В примере выше вы получали картинку на вход, анализировали ее с помощью мозгов и делали вывод об объекте, который на ней изображен. Выглядит это как-то так:



Компьютеры на самом деле ничего не "обдумывают". Они просто применяют заранее известные арифметические операции. Поэтому давайте будем называть вещи своими именами:

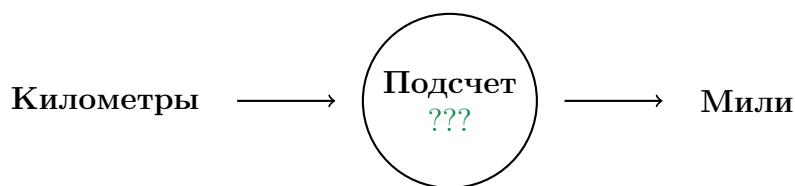


Компьютер принимает какие-то данные на вход, производит необходимые вычисления и выдает готовый результат. Рассмотрим следующий пример. Если на вход компьютеру поступает выражение  $3 \times 4$ , то оно преобразуется в более простую последовательность сложений. Как итог, получаем результат — 12.



Выглядит не слишком впечатляюще. Это нормально. С помощью этих тривиальных примеров вы увидите идею, которую реализуют нейросети.

Теперь представьте себе машину, которая преобразует километры в мили:



Теперь представьте, что мы не знаем формулу, с помощью которой километры переводятся в мили. Мы знаем только, что зависимость между двумя этими величинами **линейная**. Это означает, что если мы в два раза увеличим дистанцию в милях, то дистанция в километрах тоже увеличится в два раза. Это интуитивно понятно. Вселенная была бы очень странной, если бы это правило не выполнялось.

Линейная зависимость между километрами и милями дает нам подсказку, в какой форме надо преобразовывать одну величину в другую. Мы можем представить эту зависимость так:

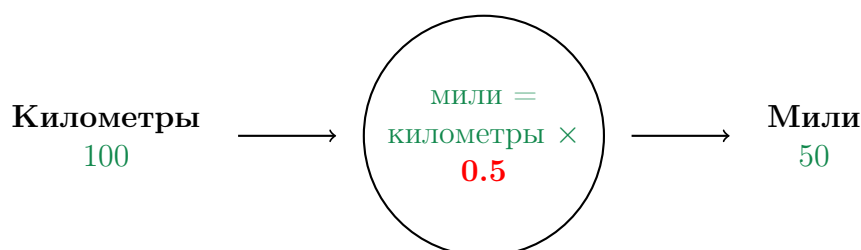
$$\text{мили} = \text{километры} \times C$$

В выражении выше  $C$  выступает в роли некоторого постоянного числа — константы. Пока мы не знаем, чему равно  $C$ .

Единственное, что нам известно — несколько заранее верно отмеренных расстояний в километрах и милях.

Номер замера	Километры	Мили
1	0	0
2	100	62.137

И как же узнать значение  $C$ ? А давайте просто придумаем **случайное** число и скажем, что ему-то и равна наша константа. Пусть  $C = 0.5$ . Что же произойдет?

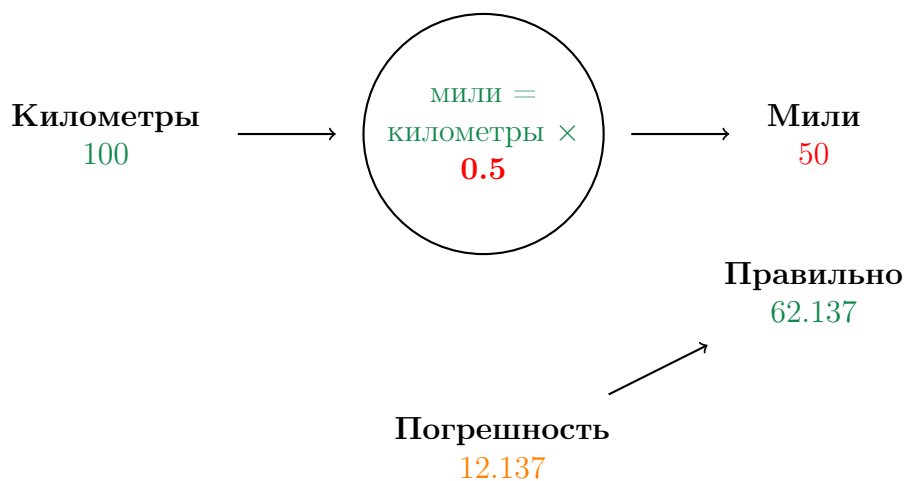


Принимая, что  $C = 0.5$  мы из 100 километров получаем 50 миль. Это отличный результат принимая во внимания тот факт, что  $C = 0.5$  мы выбрали совершенно

случайно! Но мы знаем, что наш ответ не совсем верен, потому что согласно таблице верных замеров мы должны были получить 62.137 мили.

Мы промахнулись на 12.137 миль. Это наша **погрешность** — разница между полученным ответом и заранее известным правильным результатом, который в данном случае мы имеем в таблице.

$$\begin{aligned}\text{погрешность} &= \text{правильное значение} - \text{полученный ответ} \\ &= 62.137 - 50 \\ &= 12.137\end{aligned}$$



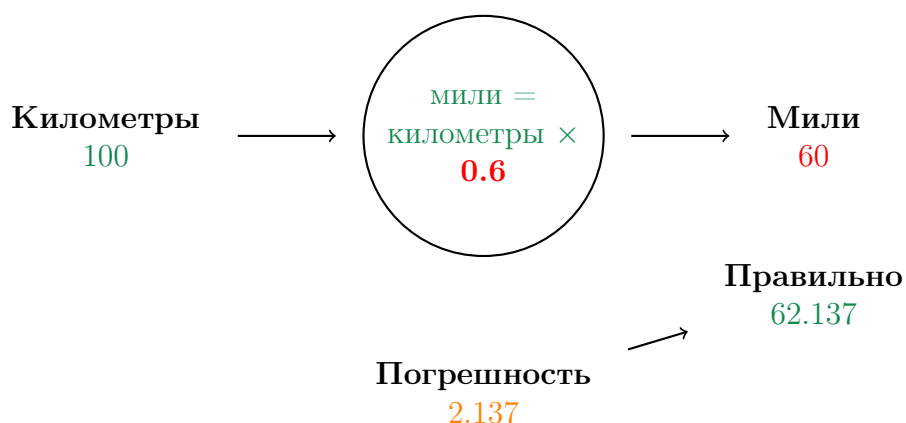
Что же дальше? Теперь мы знаем, что допустили ошибку. Более того, нам известна величина этой ошибки. Не стоит впадать в отчаяние! Вместо этого лучше пересмотреть значение константы  $C$ .

Вновь смотрим на погрешность. Полученное расстояние короче на 12.137. Так как формула по переводу километры в мили линейная (мили = километры  $\times C$ ), то увеличение значения  $C$  увеличит и выходной результат в милях.

Давайте теперь примем, что  $C = 0.6$  и посмотрим, что произойдет.

Так как  $C = 0.6$ , то для 100 километров имеем  $100 \times 0.6 = 60$  миль. Это гораздо лучше предыдущей попытки (в тот раз было 50 миль)! Теперь наша погрешность очень мала — всего 2.137 мили. Вполне себе точный результат.

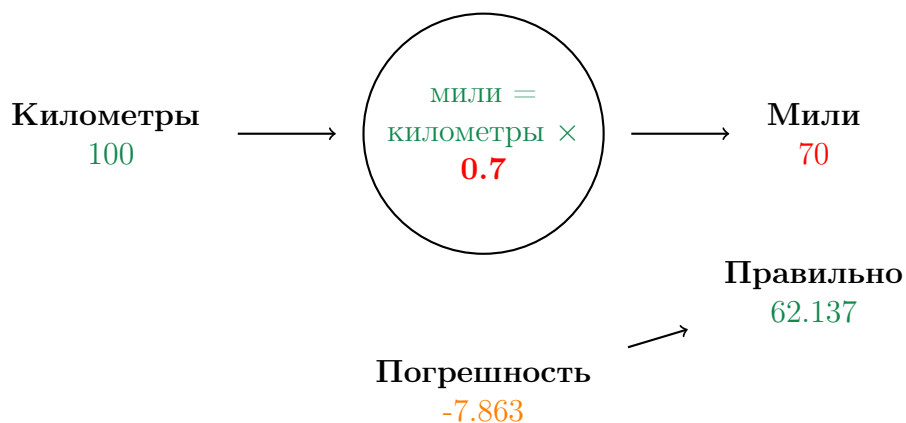




Теперь обратите внимание на то, как мы использовали полученную погрешность для корректировки значения константы  $C$ . Нам нужно было увеличить выходное число миль и мы немного увеличили значение  $C$ . Заметьте, что мы не используем алгебру для получения точного значения  $C$ , а ведь мы могли бы. Почему? Потому что на свете полно задач, которые не имеют простой математической связи между полученным входом и выдаваемым результатом.

Именно для задач, которые практически не решаются простым подсчетом нам и нужны такие изощренные штуки, как нейронные сети.

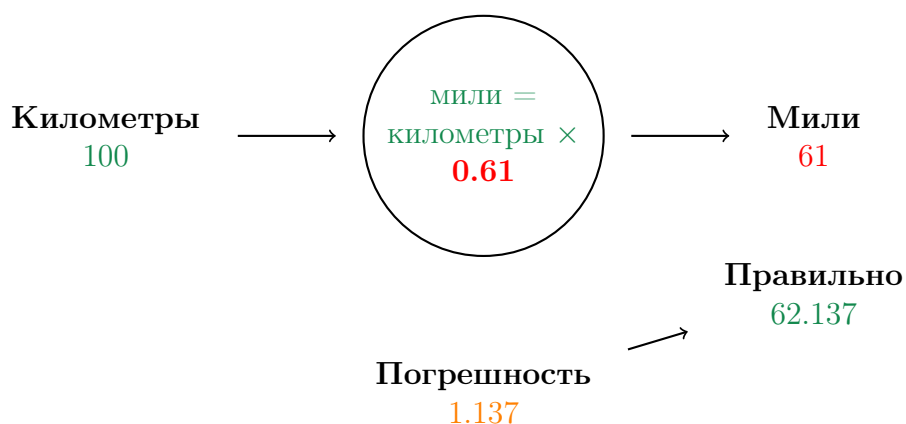
Поехали дальше. Сейчас мы на выходе имеем 60 миль. Но это все равно меньше, чем правильный результат из таблицы. Пусть  $C = 0.7$ .



Боже мой! Мы хватанули слишком много и превысили правильный результат. Наша предыдущая погрешность равнялась 2.137, а теперь она равна  $-7.863$ . Минус означает, что наш результат оказался больше правильного ответа, так как погрешность вычисляется как правильный ответ - полученный ответ.

Получается, что при  $C = 0.6$  мы имеем гораздо более точный выход. На этом можно

было бы и закончить. Но давайте все же увеличим  $C$ , но не сильно! Пусть  $C = 0.61$ .



Так-то лучше! Наша машина выдает 61 милю, что всего на 1.137 мили меньше, чем правильный ответ (62.137).

Из этой ситуации с превышением правильного ответа надо вынести важный урок. По мере приближения к правильному ответу параметры машины стоит менять все слабее и слабее. Это поможет избежать неприятных ситуаций, которые приводят к превышению правильного ответа.

Величина нашей корректировки  $C$  зависит от погрешности. Чем больше наша погрешность, тем более сильно мы меняем значение  $C$ . Но когда погрешность становится маленькой, необходимо менять  $C$  по чуть-чуть. Логично, не так ли?

Верьте или нет, но только что вы поняли самую суть работы нейронных сетей. Мы тренируем "машины" постепенно выдавать все более и более точный результат.

Важно понимать и то, как мы решали эту задачу. Мы не решали ее в один заход, хотя в данном случае так можно было бы поступить. Вместо этого, мы приходили к правильному ответу **по шагам** так, что с каждым шагом наши результаты становились лучше.ё

### Ключевые моменты

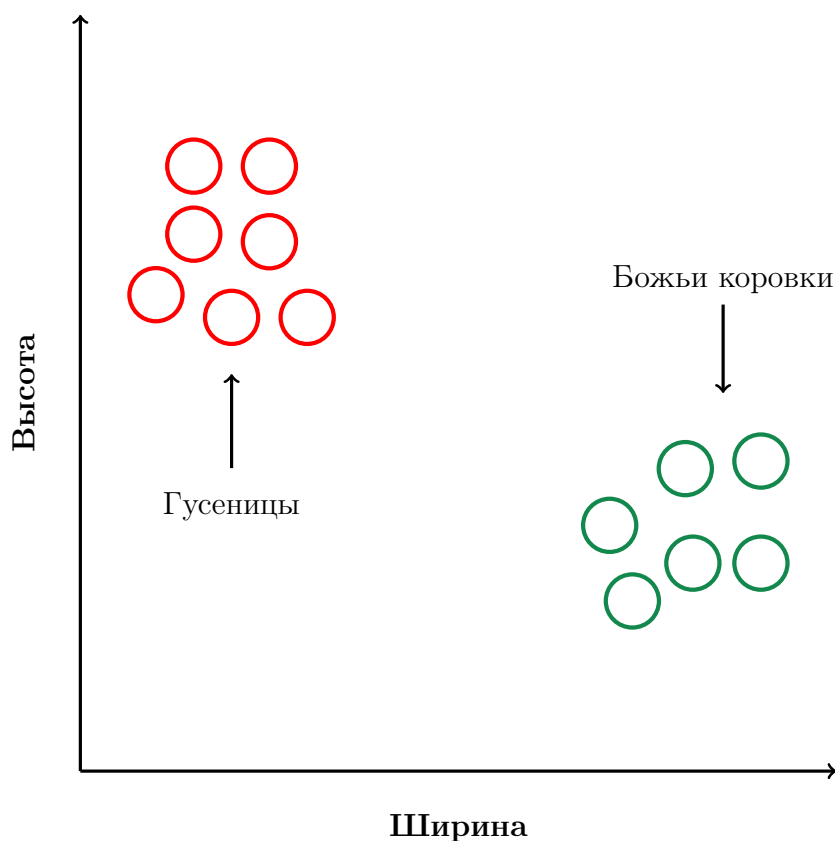
- Все вычислительные системы получают данные на вход, что-то как-то считают и возвращают результат. И нейросети в том числе.
- Когда мы не знаем точного и четкого алгоритма по решению задачи, мы можем начать изменять параметры, влияющие на результат и потихоньку приближаться к правильному ответу. Если мы не знаем, как получить мили из километров, мы можем использовать линейную зависимость как рабочую модель и менять ее постоянный коэффициент.

- Менять параметры системы можно разными способами. Отличной идеей будет регулировать силу изменения базирываясь на погрешности: чем больше погрешность, тем сильнее меняем параметр.

### 1.3 Классификация это почти что предсказание

В предыдущем разделе мы создавали предсказательную машину. Предсказательная она потому что принимает что-то на вход и предсказывает/предугадывает, какой будет выход. Мы могли влиять на ее предсказания с помощью изменения определенного параметра этой машины, основываясь на значениях получаемой погрешности.

Посмотрите на график ниже. Каждая точка на этом графике соответствует заранее замеренной длине и ширине насекомых в саду.



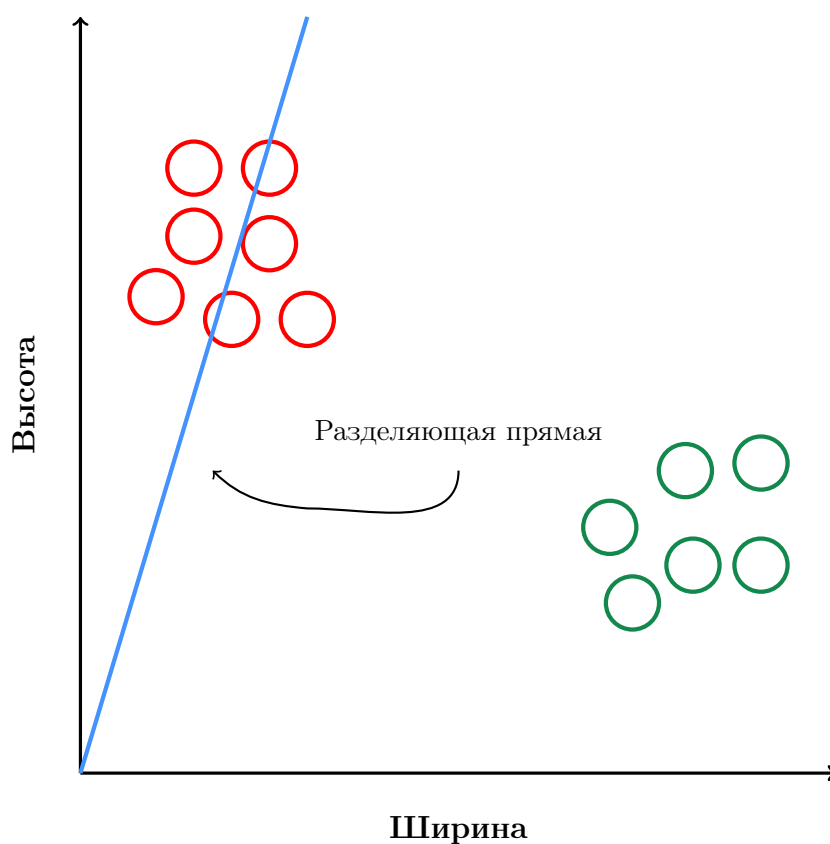
Длина и ширина жуков в саду

Нетрудно заметить увидеть две явно выделяющиеся группы. Гусеницы длинные и тоненькие, а божьи коровки широкие и короткие.

Помните предсказательную машину, которая пыталась найти правильное число миль по полученным километрам? В основе той машины лежала линейная функция.

Линейные функции оттого и называются линейными, потому что на графике они представляют собой прямые. Постоянный коэффициент  $C$  определяет наклон прямой.

Давайте нанесем какую-нибудь прямую на наш график с размерами жуков:

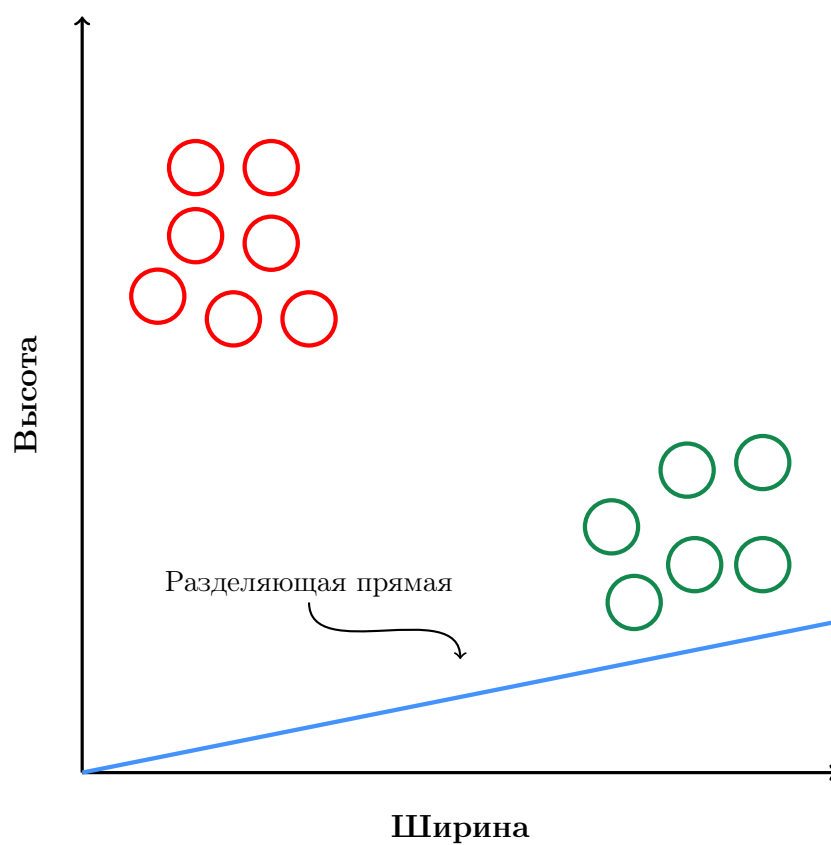


Длина и ширина жуков в саду

Очевидно, что мы не можем использовать прямую для перевода километров в мили. Но с помощью прямой мы можем попытаться что-нибудь разделить. Если бы прямая на графике выше разделяла гусениц от божьих коровок, то мы могли бы использовать ее для классификации неизвестных жуков, базируясь на их размерах.

Текущую прямую пока нельзя использовать для классификации, так как половина гусениц у нас относится к божьим коровкам (по правую сторону от прямой).

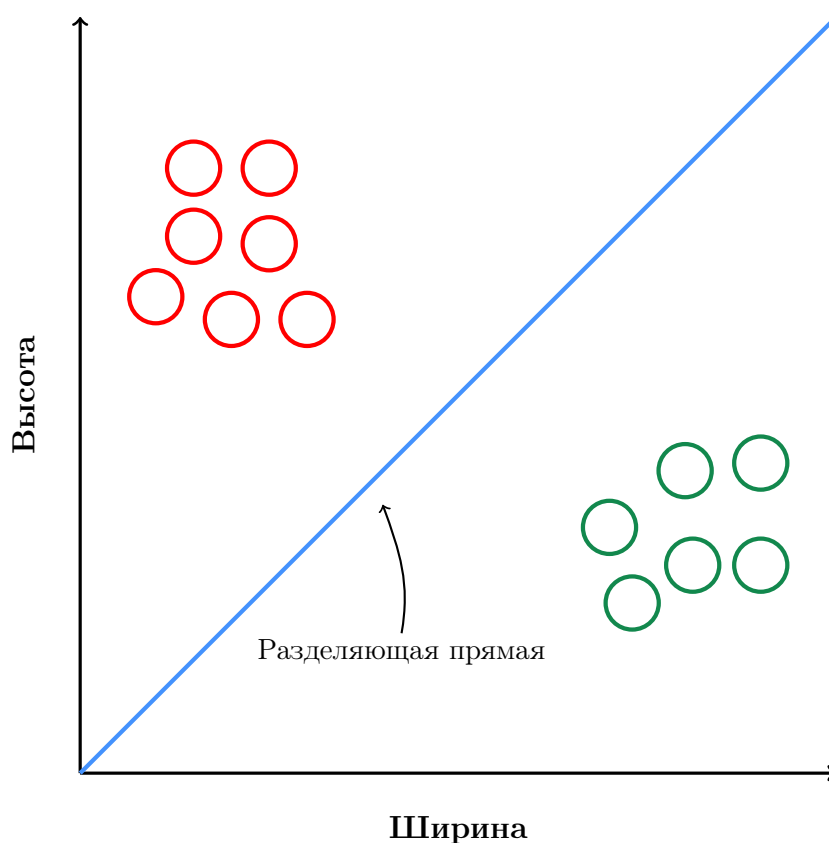
Давайте попробуем какую-нибудь другую прямую.



Длина и ширина насекомых в саду

Новая прямая не имеет никакой ценности, так как она вообще не разделяет гусениц и божьих коровок!

Еще одна попытка:



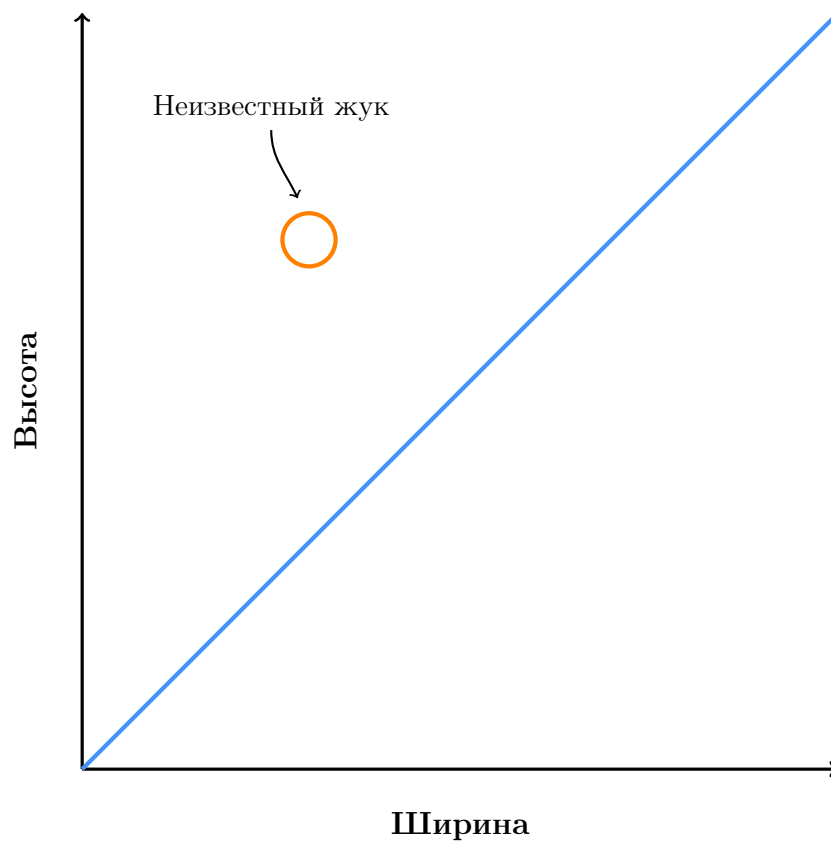
Длина и ширина насекомых в саду

Так-то лучше! Эта прямая точно разделяет гусениц и божьих коровок. Теперь эту прямую можно использовать как **классификатор** насекомых.

Предположим, что существует всего два вида насекомых: гусеницы и божьи коровки. Да, это выглядит странно, но так будет проще понять идею простейшего классификатора.

Представим, что у нас есть робот, который берет букашек своей механической рукой и замеряет их размеры. После замера он может определить вид насекомого (гусеница или божья коровка) у него в руке с помощью нашего классификатора.

Посмотрите на график ниже. Оранжевая точка — размеры неизвестного насекомого. Вы сходу можете заявить, что это насекомое — гусеница, так как точка лежит выше голубой прямой.



К какому виду относится неизвестный жук?

Получается, что линейная функция, которую мы использовали в нашей предсказательной машине может быть использована для классификации новых данных.

Мы пропустили один очень важный процесс. Как именно мы настраиваем правильный наклон прямой? Как скорректировать не совсем удачный результат?

Ответ на этот вопрос, как и в случае с предсказательной машиной, относится к самой сути принципа работы нейронных сетей. И сейчас вы в этом убедитесь.



## 1.4 Тренировка простого классификатора

Сейчас мы хотим **натренировать** наш простой классификатор для определения вида жуков: либо гусеница, либо божья коровка. Нам нужно найти правильный наклон прямой, которая разделяет две группы точек на графике размеров жуков.

Как это сделать?

Вместо разработки математической теории давайте просто пойдем напролом. Так будет проще.

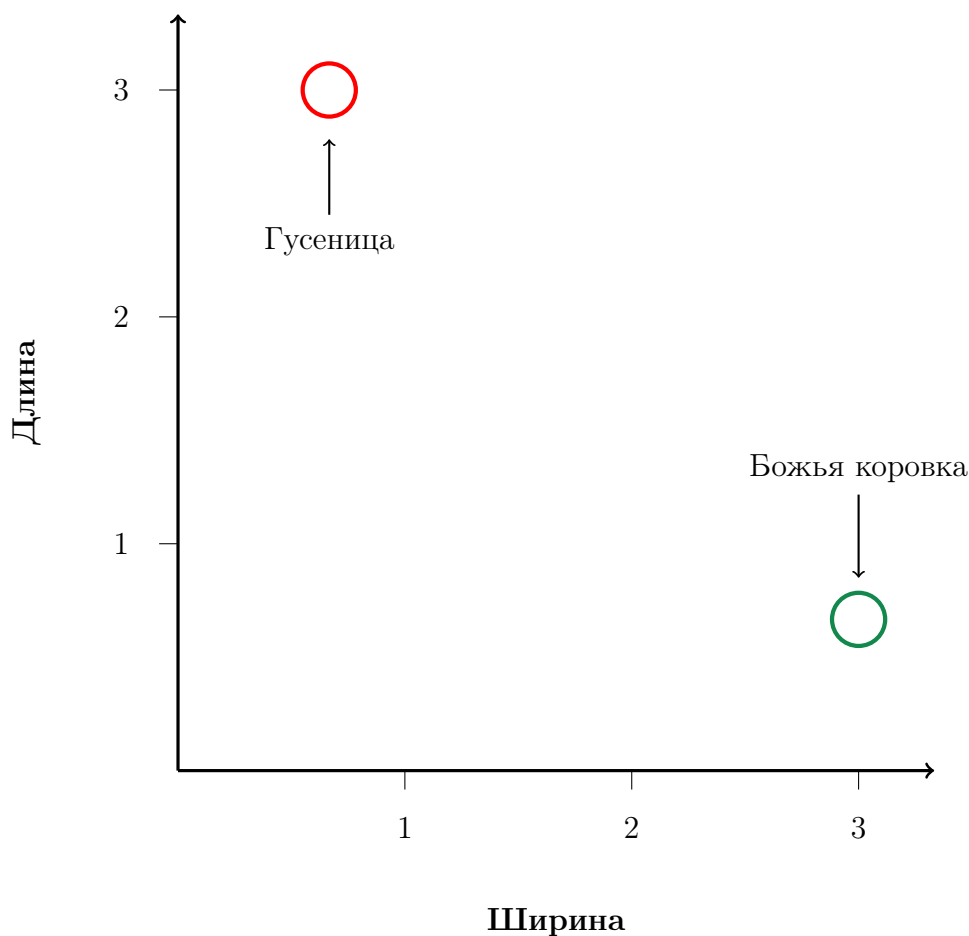
Для тренировки классификатора нам нужны какие-то примеры. Для простоты используем таблицу с двумя примерами ниже.

Пример	Ширина	Длина	Жук
1	3.0	1.0	божья коровка
2	1.0	3.0	гусеница

У нас есть божья говорка и мы знаем ее размеры: 3 в ширину и 1 в длину. У нас также есть гусеница с размерами: 1 в ширину и 3 в длину.

Мы принимаем, что оба примера — чистая правда. Именно с помощью этих двух примеров мы и настроим наклон прямой. Правильные данные, на основе которых обучают нейросети, называются **обучающей выборкой**. Таблица выше — обучающая выборка для нашего классификатора.

Давайте нанесем на график оба примера из таблицы. Визуализация данных очень часто помогает лучше понять имеющиеся данные, прочувствовать их. Это очень трудно сделать просто смотря на таблицу с цифрами.



Обучающая выборка для классификатора жуков

Давайте возьмем случайную разделяющую прямую. Надо же с чего-то начинать. Когда мы создавали машину для предсказания миль по данным километрам, мы использовали линейную функцию и подбирали постоянный коэффициент. Здесь мы тоже можем использовать линейную функцию, так как ее график есть прямая линия:

$$y = Ax$$

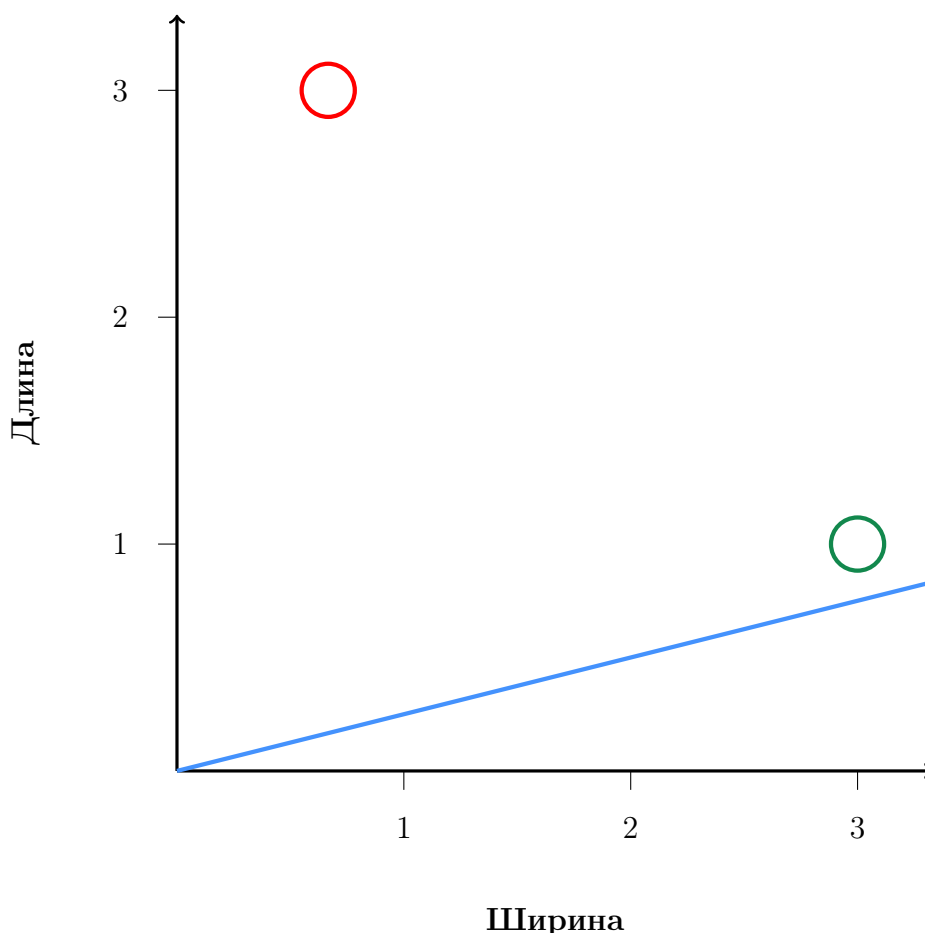
Тут мы намерено использовали обозначения  $y$  и  $x$  вместо длины и ширины, потому что, строго говоря, здесь линейная функция не выполняет предсказательной роли. Она не переводит ширину в длину, как в примере с функцией, переводящей километры в мили. Здесь эта функция определяет прямую на графике. Это классификатор.

Вы также можете заметить, что в функции  $y = Ax$  мы опустили свободный член, ведь в общем виде линейная функция должна записываться в виде  $y = Ax + B$ .  $B \neq 0$  означает, что прямая не проходит через начало координат. Но давайте не будем

усложнять нашу задачу и примем  $B = 0$ , то есть наша прямая проходит через начало координат.

Мы уже знаем, что параметр  $A$  отвечает за наклон прямой. Чем больше  $A$ , тем круче наклон.

Для начала пусть  $A = 0.25$ . Уравнение нашей разделяющей прямой следующее:  $y = 0.25x$ . Давайте нарисуем разделяющую прямую, которая соответствует этому уравнению. Рисовать будем на том же графике.



Обучающая выборка для классификатора жуков

Без всяких вычислений сразу можно сказать, что эта прямая  $y = 0.25x$  — плохой классификатор. Она не разделяет жуков в саду. Мы не можем сказать, что если точка размеров нашего жука выше прямой, то это гусеница, так как и божья коровка тоже находится выше этой прямой.

Очевидно, что нам надо немного увеличить наклон прямой. Давайте не поддаваться искушению просто приподнять прямую "на глазок". Нам нужно придумать **алгоритм**

— четкую последовательность действий (которые может выполнить и компьютер), с помощью которых мы постепенно придем к правильному положению прямой.

Давайте посмотрим на первый пример из нашей обучающей выборки: божья коровка с шириной = 3.0 и длиной = 1.0. Подставим ширину за место  $x$  в уравнение нашей разделяющей прямой:

$$y = (0.25) \times (3.0) = 0.75$$

Функция, в которой постоянный коэффициент  $A = 0.25$  мы выбрали случайным образом предполагает, что если ширина найденного в саду жука равна 3.0, то его длина должна быть равна 0.75. Но исходя из обучающей выборки мы знаем, что этого недостаточно и длина жука должна равняться 1.0.

Мы опять получаем **погрешность**, прямо как и в случае с машиной, переводящей километры в мили. Там мы подбирали коэффициент  $C$ . Здесь нам нужно подобрать коэффициент  $A$ .

Прежде чем приступить к подбору  $A$  важно прояснить один момент. Какое именно значение  $y$  (то есть длины жука) мы хотим получить? Если, мы хотим, чтобы  $y = 1.0$ , то наша прямая будет проходить прямо через точку с размерами божьей коровки (3.0, 1.0). Может показаться, что именно это нам и надо, но это не так. Нам нужно, чтобы прямая проходила выше этой точки. Почему? Потому что мы хотим, чтобы все размеры божьих коровок находились под прямой, а не на ее пути. Ведь нам нужна прямая, разделяющая гусениц и божьих коровок, а не предсказательная машина, которая выдает нам длину жука, основываясь на его ширине.

Принимая во внимание все вышесказанное давайте установим за желаемый результат  $y = 1.1$  при  $x = 3.0$ . Это чуть-чуть выше, чем 1.0. Вообще никто не мешает выбрать  $y = 1.2$  или даже 1.3. Главное не брать слишком высоко, так как в этом случае может получиться, что и божьи коровки, и гусеницы окажутся под прямой.

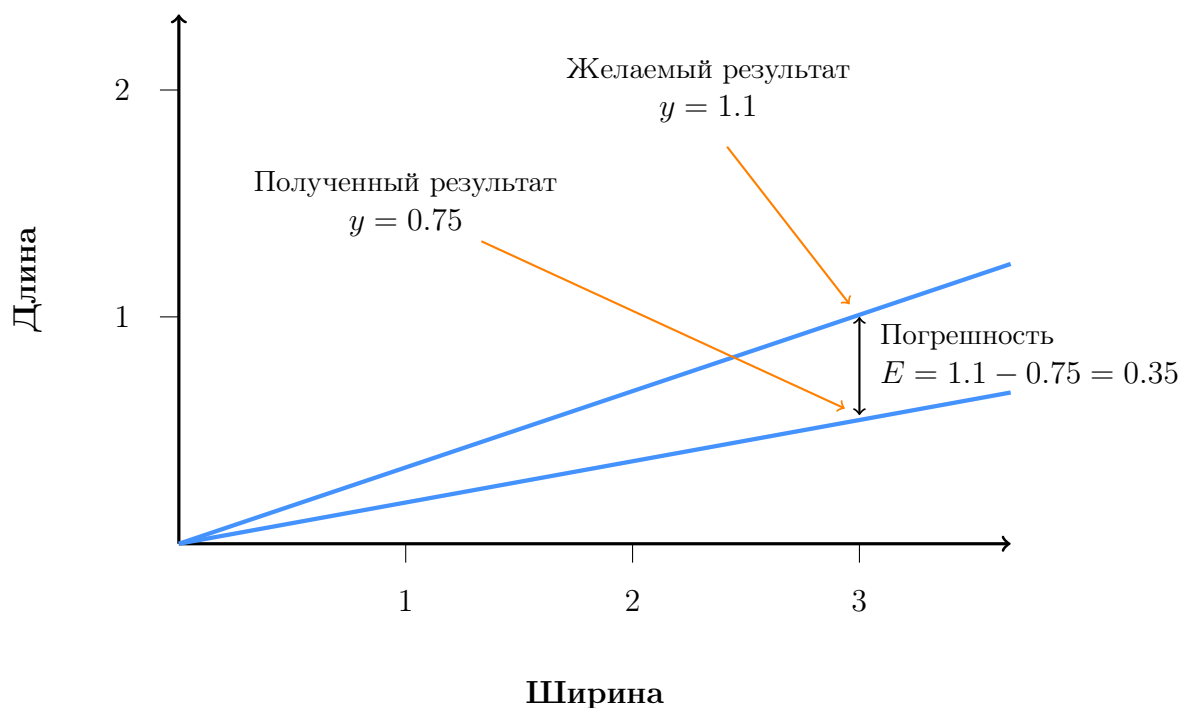
Итак, желаемый результат равен 1.1, а как вычисляется погрешность мы с вами уже знаем:

$$\text{погрешность} = \text{желаемый результат} - \text{полученный результат}$$

Подставляем числа:

$$E = 1.1 - 0.75 = 0.35$$

Сделайте паузу и внимательно изучите рисунок ниже. Убедитесь, что вы все понимаете.



Но как использовать погрешность  $E$  для получения более точного значения  $A$ ? Это важный вопрос.

Давайте рассуждать. Мы хотим использовать погрешность  $E$  для подбора значения постоянного коэффициента  $A$ . Тогда нам нужно знать связь между  $E$  и  $A$ . Если бы мы отыскивали эту связь, то смогли бы понять, как изменение одного влияет на другое.

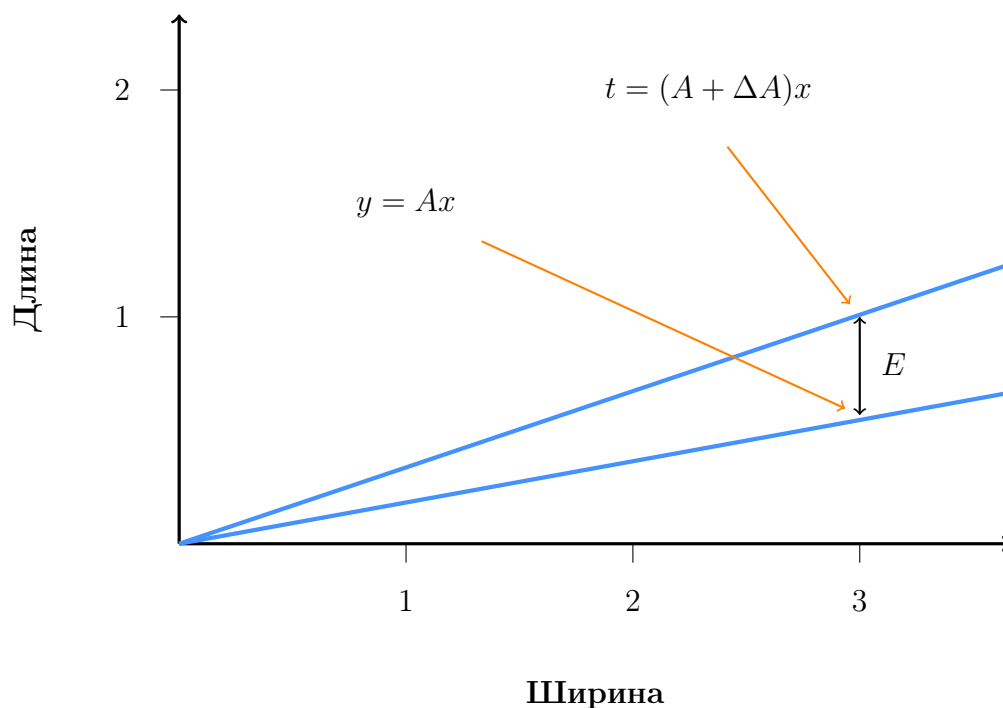
Уравнение нашего классификатора:

$$y = Ax$$

Давайте обозначим желаемое значение  $y$  за  $t$ . Для получения этого значения  $t$  нам надо как-то несильно изменить постоянный коэффициент  $A$ . Математики используют символ  $\Delta$  (дельта) для обозначения небольшого изменения. Итак:

$$t = (A + \Delta A)x$$

Проиллюстрируем вышесказанное. Мы видим, что для достижения желаемого результата  $t$  в точке с  $x = 3.0$  нам нужен новый постоянный коэффициент  $A + \Delta A$ , который отвечает за новый наклон разделяющей прямой.



Вы помните, что погрешность равна разнице между желаемым и полученным результатами. Желаемый результат мы обозначили за  $t = (A + \Delta A)x$ . Текущий получаемый результат мы получаем с помощью уравнения  $y = Ax$ . Разница  $t - y$  и есть наша погрешность:

$$E = t - y = (A + \Delta A)x - Ax$$

Раскрываем скобки и упрощаем:

$$\begin{aligned} E = t - y &= Ax + \Delta Ax - Ax \\ E &= \Delta Ax \end{aligned}$$

Вот это да! Мы получили простую зависимость между погрешностью  $E$  и небольшим изменением постоянного коэффициента  $\Delta A$ . Да, это действительно так просто.

Возможно, вся эта алгебра немного запутала вас. Давайте вспомним, для чего мы все это делали.

Мы ищем, насколько надо изменить  $A$  для улучшения наклона разделяющей прямой в зависимости от текущей погрешности  $E$ . Мы уже нашли зависимость между  $\Delta A$  и  $E$ :

$$E = \Delta Ax$$

Теперь делим обе части равенства на  $x$ :

$$\Delta A = \frac{E}{x}$$

Вот то самое магическое выражение, на поиск которого мы потратили столько сил! Теперь мы точно знаем, насколько надо изменять коэффициент  $A$  при данной ошибке  $E$ .

Давайте сделаем это — давайте улучшим наклон разделяющей прямой.

$E = 0.35$  и  $x = 3.0$ . Тогда получаем, что  $\Delta A = E/x = 0.35/3.0 = 0.1167$ . Это означает, что мы должны добавить к текущему значению  $A = 0.25$  еще  $0.1167$ . Имеем  $A = 0.25 + 0.1167 = 0.3667$ .

Теперь получаемый  $y = Ax$  с коэффициентом  $A = 0.3667$  при  $x = 3.0$  равен  $1.1$ , как мы и хотели.

Вот и все! Мы сделали это! Мы подобрали постоянный коэффициент  $A$  в зависимости от имеющейся погрешности.

Но это еще далеко не все.

У нас в обучающей выборке есть еще один пример — гусеница с шириной ( $x = 1.0$ ) и длиной ( $y = 3.0$ ). Давайте посмотрим, что мы получим при подстановке  $x = 1.0$  в нашу линейную функцию с постоянным коэффициентом  $A = 0.3667$ . Получаем  $y = 0.3667 \times 1 = 0.3667$ . Даже не близко к  $y = 3.0$ .

Исходя из того, что разделяющая прямая должна не пересекать данные нашей обучающей выборки, а проходить над/под ними, примем желаемый результат за  $2.9$ . В таком случае размеры гусеницы из обучающей выборки будут как раз немного выше разделяющей прямой.

Посчитаем текущую погрешность:

$$E = 2.9 - 0.3667 = 2.5333$$

Она может показаться большой. Но это не так странно, ведь мы настраивали коэффициент  $A$  основываясь только на первом примере из обучающей выборки.

Посчитаем обновленный коэффициент  $A$ . Для этого нам нужно вновь найти добавку  $\Delta A$ :

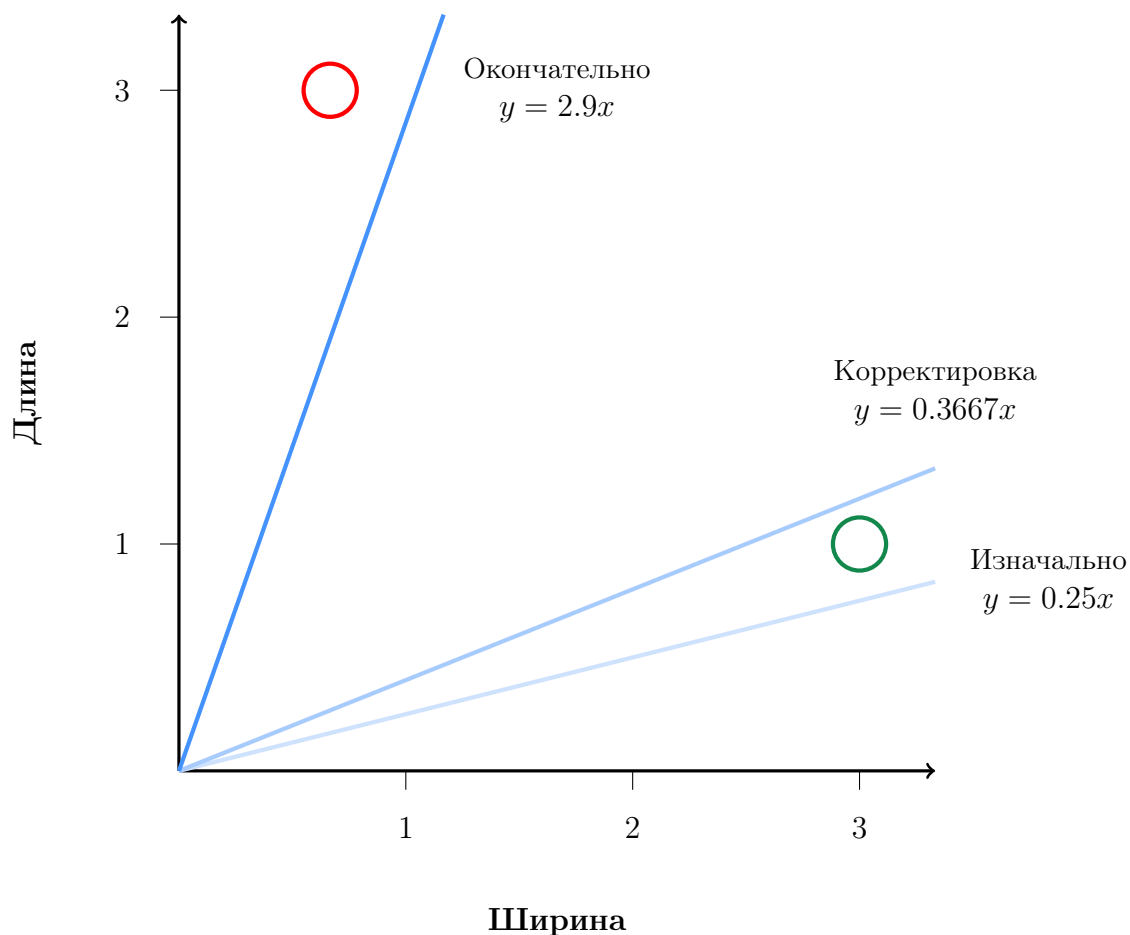
$$\Delta A = \frac{E}{x} = \frac{2.5333}{1} = 2.5333$$

Получим новое значение  $A$ :

$$A = 0.3667 + 2.5333 = 2.9$$

Теперь для  $x = 1.0$  мы будем получать  $y = 2.9$ . Именно этого мы и добивались.

Давайте теперь визуализируем результаты нашей работы. На график ниже нанесены три прямые: та, с которой мы начинали, та которую мы получили на первом примере из обучающей выборки и финальная прямая, полученная с учетом второго примера.



Стоп! Что произошло?! Какой-то странный наклон у нас получился. Он вовсе не делит область пополам между гусеницами и божьими коровками.



Ну, вообще-то мы получили то, чего добивались. Мы скорректировали наклон нашей прямой для получения желаемого  $y$ . Тогда что пошло не так? Если мы и дальше продолжим работать в таком духе, то мы всегда будем получать результат, заточенный на последний пример, под который мы проводили корректировку. Фактически, мы выбрасываем из головы нашего классификатора то, чему он научился на предыдущих примерах и учим его только на самом последнем.

Как это исправить?

Просто. И в этом решении состоит одна из основных идей **машинного обучения**. Мы **регулируем** изменения. Под регуляцией я подразумеваю их смягчение. Вместо того, чтобы бросаться сломя голову к каждому новому значению коэффициента  $A$ , мы намеренно немного уменьшаем добавку  $\Delta A$ . Таким образом мы не скачком приходим к результату, а как-бы двигаемся в его направлении, с уважением относясь к предыдущим достигнутым результатам. Мы уже встречались со смягчением изменений ранее, когда создавали машину по переводу километров в мили. Там мы понемногу меняли коэффициент  $C$ .

У регулирования изменения есть еще одно очень крутое и полезное свойство. Бывают ситуации, когда мы не можем полностью доверять даже нашей обучающей выборке. Она может содержать ошибки или всякого рода погрешности, от которых невозможно избавиться при проведении реальных измерений. С регулировкой изменений будет происходить автоматическое сглаживание всех шумов и погрешностей. Они практически не будут влиять на конечный результат.

Давайте вернемся к нашему классификатору и введем идею регулировки изменений в алгоритм его работы:

$$\Delta A = L \left( \frac{E}{x} \right)$$

Регулировку изменений часто называют **коэффициентом скорости обучения**. Примем  $L = 0.5$  для начала. Получается, что изменение коэффициента ( $\Delta A$ ) будет вдвое более слабым, чем оно было бы без регулировки изменений.

Повторим процесс обучения классификатора. Изначально  $A = 0.25$ . Первый пример из обучающей выборки дает нам следующий  $y$ :

$$y = 0.25 \times 3 = 0.75$$

Желаемый результат равен 1.1, а значит ошибка равна 0.35. Считаем добавку к  $A$ :

$$\Delta A = L \left( \frac{E}{x} \right) = 0.5 \times \frac{0.35}{3} = 0.0583$$

Считаем  $A$ :

$$A = 0.25 + 0.0583 = 0.3083$$

Давайте протестируем полученное значение  $A$ :

$$y = Ax = 0.3083 \times 3 = 0.9250$$

Теперь разделяющая прямая проходит немного ниже нашей божьей коровки. Да,  $0.9250 \neq 1.1$ , но это все равно отличный результат если относиться к нему, как к первому шагу в обучении нашего классификатора.

Теперь продолжим тренировку на втором примере из обучающей выборки, когда  $x = 1$ .

$$y = 0.3083 \times 1 = 0.3083$$

Желаемый результат для второго примера из выборки был равен 2.9. Считаем ошибку:

$$E = 2.9 - 0.3083 = 2.5917$$

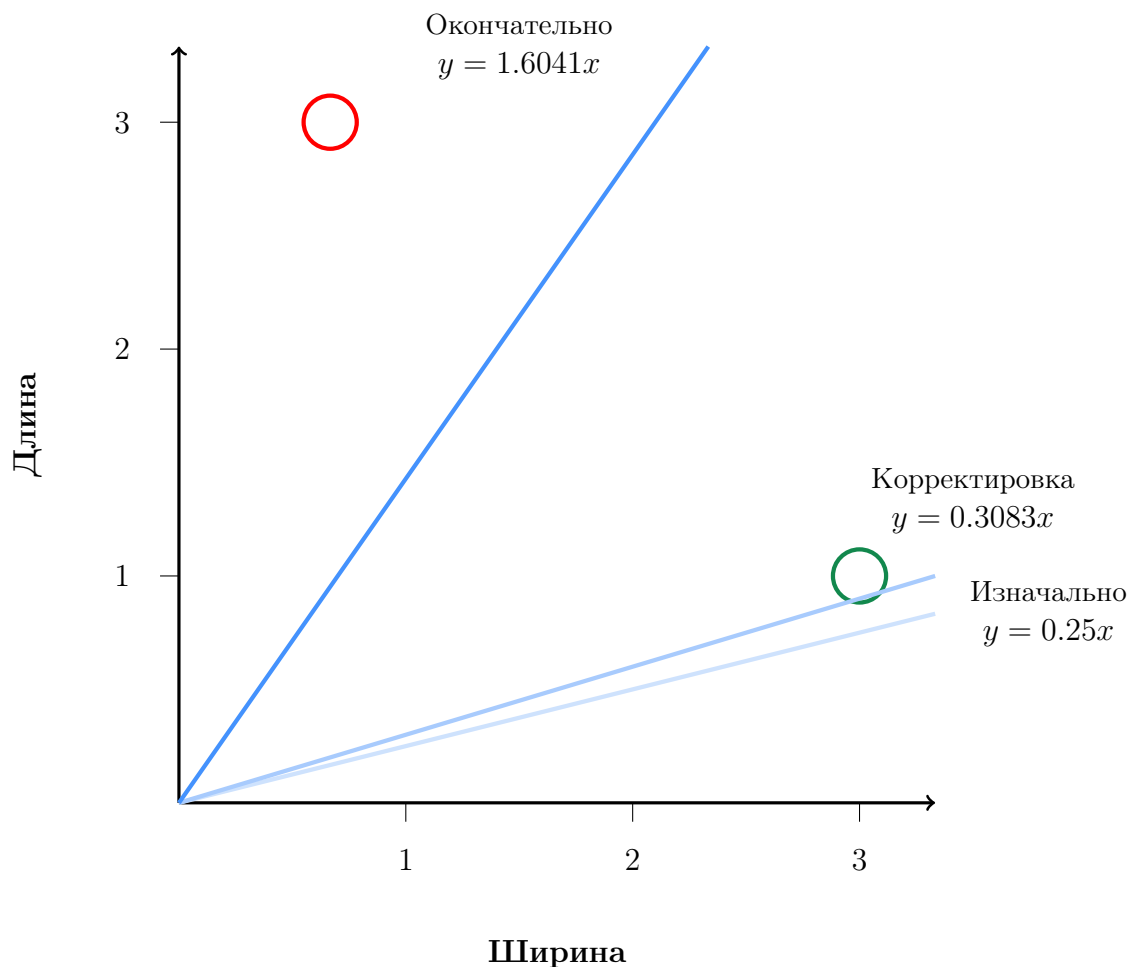
Считаем добавку к  $A$ :

$$\Delta A = L \left( \frac{E}{x} \right) = 0.5 \times \frac{2.5917}{1} = 1.2958$$

Более совершенное значение  $A$ :

$$A = 0.3083 + 1.2958 = 1.6041$$

Теперь давайте вновь отобразим все шаги обучения на графике и проверим, работает ли концепция регулирования изменений.



А вот это уже круто!

2 примеров из обучающей выборки и относительно простого правила по регулировке изменений с применением **коэффициента скорости обучения** нам с лихвой хватило для получения отличной разделяющей прямой с постоянным коэффициентом  $A = 1.6041$ .

Не преуменьшайте значение достигнутого. Только что мы разработали автоматический алгоритм обучения классификатора. Причем наш алгоритм демонстрирует замечательные результаты и при этом остается очень простым внутри.

Великолепно!

### Ключевые моменты

- С помощью достаточно простой математики мы можем установить взаимосвязь между погрешностью линейного классификатора и коэффициентом наклона пря-

мой. Другими словами, мы точно знаем, насколько надо изменить наклон для уменьшения погрешности.

- Слишком резкие изменения параметра приводят к тому, что наша модель подстраивается под последний обучающий пример, "забывая" все предыдущие примеры. Отличный способ избавиться от этой проблемы — регулировка изменений с помощью коэффициента скорости обучения. В таком случае можно быть уверенным, что какой-то конкретный элемент из обучающей выборки не станет доминировать над общей картиной.
- Элементы обучающей выборки, созданной по реальным данным могут быть зашумлены или содержать ошибки. Изменение с регулировкой не позволяет таким неправильным данным испортить все конечный результат.

## 1.5 Иногда одного классификатора недостаточно

Мы уже умеем создавать простые предсказатели и классификаторы, которые принимают что-то на вход, проводят какие-то подсчеты и выводят ответ. И хотя они отлично решают поставленные задачи, они не всесильны. Есть множество других проблем, с которыми они не могут справиться, но которые мы надеемся решить с помощью нейросетей.

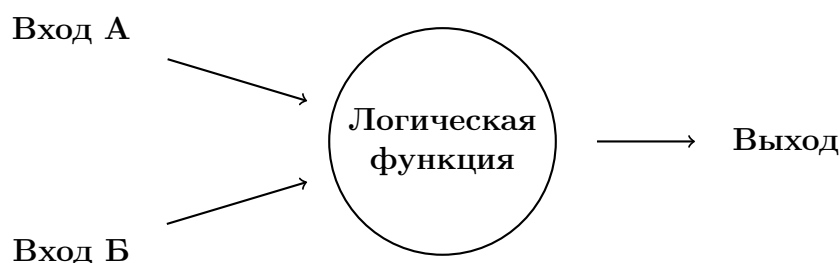
Сейчас мы покажем пример такой простой проблемы, которую не способны решить линейные классификаторы. А зачем? Не лучше ли сразу перейти к нейросетям? В физике, математике, машинном обучении и любой другой теории очень важно понимать границы применимости моделей. Знание границ позволяет правильно понять проблему и выбрать подходящий метод ее решения.

Давайте оставим несчастных жуков в своем саду и рассмотрим **логические (булевы) функции**. Если вы не имеете понятия, что это такое — не волнуйтесь. Булевыми эти функции называются в честь английского математика Джорджа Буля. В число логических функций входят функции **И** и **ИЛИ**.

Логические функции оперируют не числами, а утверждениями. Например, в фразе "ты можешь съесть мороженое, если ты поел овощи **И** ты еще голоден" мы используем булеву функцию **И**. Логическое **И** всегда верно, когда оба условия истины. Если хотя бы одно из условий ложное, то значение логического **И** тоже ложь. Если я голоден, но не поел овощей, то я не могу съесть мороженое.

В фразе "Ты можешь съездить на дачу, если сейчас выходные **ИЛИ** ты в отпуске" мы используем булеву функцию **ИЛИ**. Логическое **ИЛИ** верно, если одно условие или оба сразу истины. Если я сейчас не в отпуске, но сегодня суббота, то я могу съездить на дачу.

Предсказательная машина из первого раздела принимала что-то на вход, что-то считала и выводила ответ. Булевы функции похожи на эту машину, но они принимают сразу два входа и выдают один ответ:



Принято сокращенно обозначать **истину** 1, а **ложь** за 0. На таблице ниже отображены все возможные пары входов *A* и *B*, а также значения логического **И** и **ИЛИ** для

каждой пары.

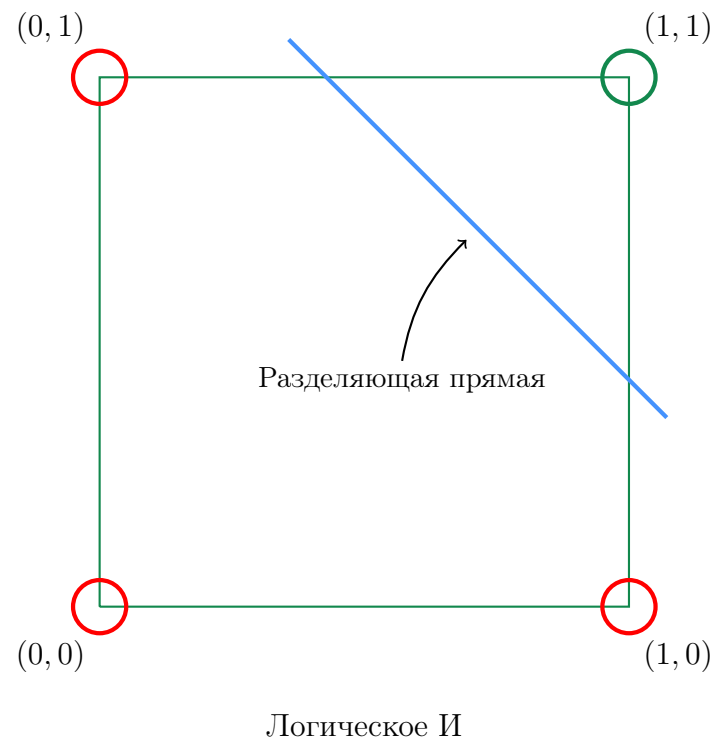
Вход А	Вход Б	Логическое И	Логическое ИЛИ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Сразу видно, что логическое И истинно только тогда, когда и  $A$  и  $B$  истинны. А вот логическое ИЛИ истинно всегда, когда хотя бы один из входов истинный.

Булева логика составляет основу всей теории вычислительных машин. Самые первые компьютеры собирались из маленьких электрических схем, которые как раз и вычисляли значения логических функций. Даже арифметические действия производились с помощью комбинаций состояний этих схем.

Представьте, что вы используете линейный классификатор, обученный на данных, которые основываются на какой-то логической функции. Такой подход часто используют учены для выявления связей между полученными связями. Например, заболевание малярия распространено в регионах с частыми дождями И с высокой температурой? Может быть малярия распространяется сильнее когда хотя бы одно (логическое ИЛИ) из этих условий верно?

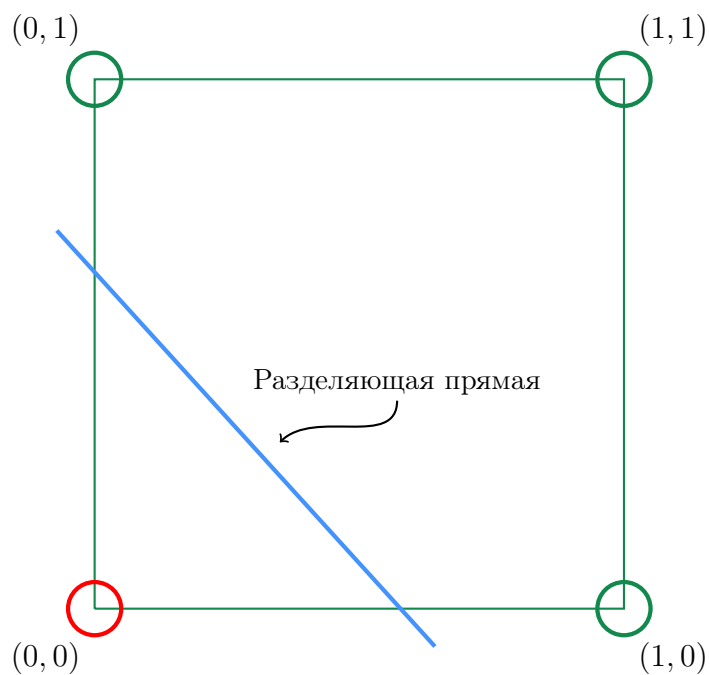
Посмотрите на график ниже. Кружками на нем изображены возможные комбинации входов  $A$  и  $B$ . Мы видим, только когда оба входа есть истина (равны 1), то и значение функции тоже истинно (зеленый цвет кружка). Кружки, приводящие к ложным значениям логической функции окрашены в красный цвет.



На графика имеется прямая, отделяющая регион красных точек от региона зеленых точек. Это линейный классификатор, который можно обучить точно также, как мы это делали в предыдущем разделе. Никаких особых отличий здесь не будет.

Итак мы знаем, что простой линейный классификатор, описывающийся уравнением  $y = Ax + B$  может научиться работать с логическим И.

Проверим, способен ли линейный классификатор справиться с логическим ИЛИ?



Логическое ИЛИ

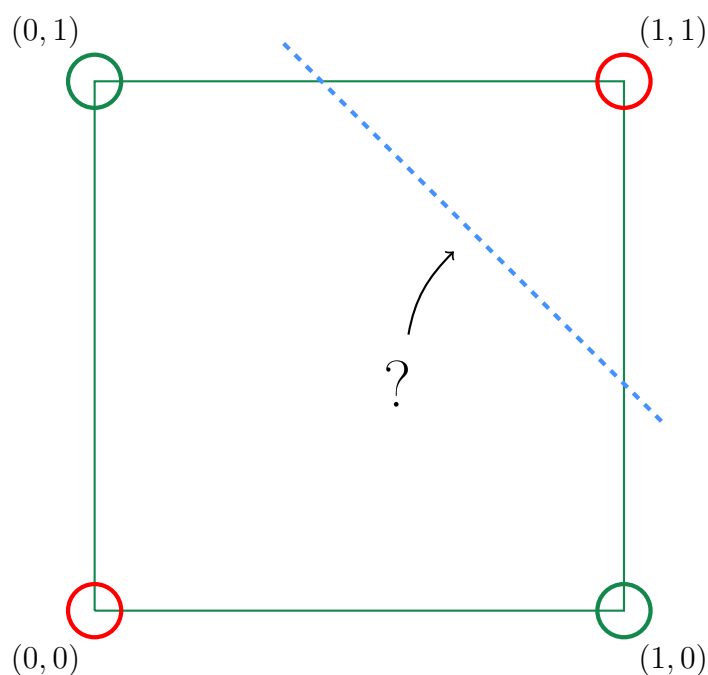
Сейчас на графике только один красный кружок, так как логическое ИЛИ всегда возвращает истину, кроме того случая, когда оба входа ложны. Как и в случае с логическим И мы сразу видим, что мы можем отделить истинные значения функции от ложных с помощью простого простого линейного классификатора.

Но существует еще одна логическая функция. Ее называют XOR или исключающим ИЛИ. Она возвращает истину только тогда, когда один из входов истинный, но не оба сразу. Когда оба входа ложны или истины, функция возвращает 0. Исключающее ИЛИ можно описать с помощью таблицы ниже:

Вход А	Вход Б	Исключающее ИЛИ
0	0	0
0	1	1
1	0	1
1	1	0

Визуализируем эту таблицу:





Исключающее ИЛИ (XOR)

А вот это уже задача посложнее! Судя по всему, мы не можем отделить зеленые кружки от красных с помощью одной прямой.

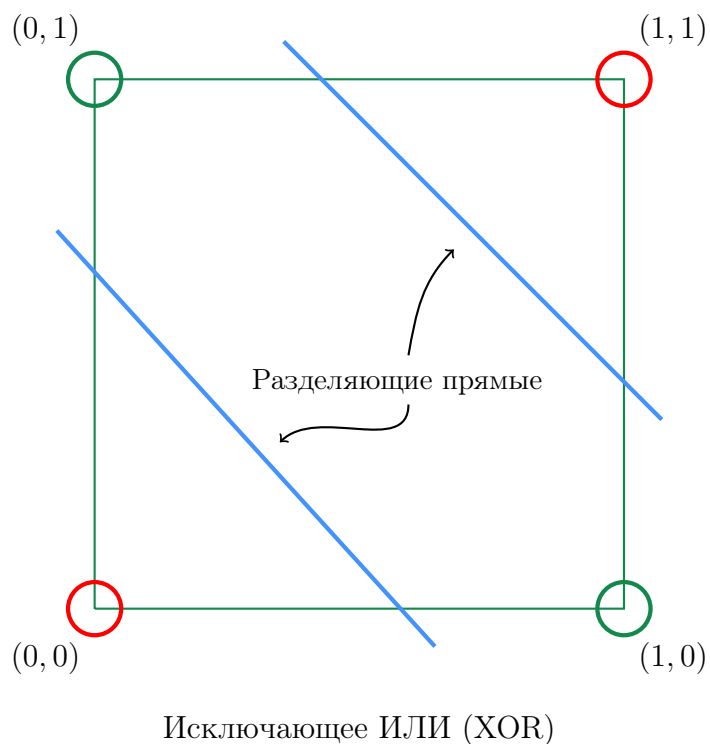
На самом деле это доказанный факт. Невозможно провести одну прямую таким образом, чтобы разделить значения исключающего ИЛИ. Сколько вы не тренируйте простой линейный классификатор — ничего из этого не выйдет. Если ваши научные данные по своей сути распределяются по принципу XOR, то вы не сможете построить корректный классификатор.

Вот мы и нашли границу применимости линейного классификатора. Линейный классификатор можно применить только для данных, которые можно как-то разделить прямой.

Но мы хотим, чтобы нейронные сети решали проблемы и с линейно неразделимыми данными — там, где одна прямая линия нам не поможет.

Как же решить эту проблему?

К счастью, решение достаточно простое. Надо всего лишь добавить еще одну прямую — еще один линейный классификатор. Эта идея тоже лежит в основе нейросетей. Вы можете представить множество линий, которые вместе разделяют даже самые хитроумные регионы данных.



Прежде чем мы приступим к рассмотрению нейросетей, которые состоят из совокупности нескольких линейных классификаторов, давайте посмотрим, как работают нейронные сети, созданные самой природой — мозги.

### Ключевые моменты

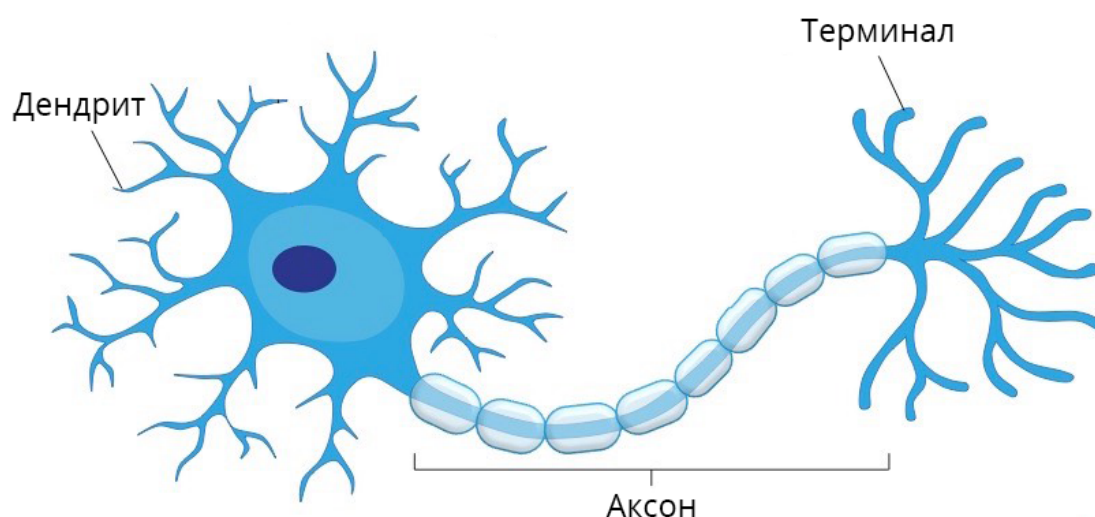
- Простой линейный классификатор не способен правильно разделить данные, которые по своей сути неразделимы с помощью прямой. Например, данные, которые распределяются по принципу исключающего ИЛИ.
- Однако, эти проблемы решаются добавлением дополнительных классификаторов, которые вместе разделяют данные на регионы разной формы.

## 1.6 Нейроны — природные вычислительные машины

Животные всегда вводили ученых в ступор, тем что даже голуби оказывались более интеллектуальными, чем цифровые компьютеры, состоящие из немереного количества вычислительных элементов, способные хранить огромное количество данных и производящие вычисления гораздо быстрее и точнее, чем биологические мозги.

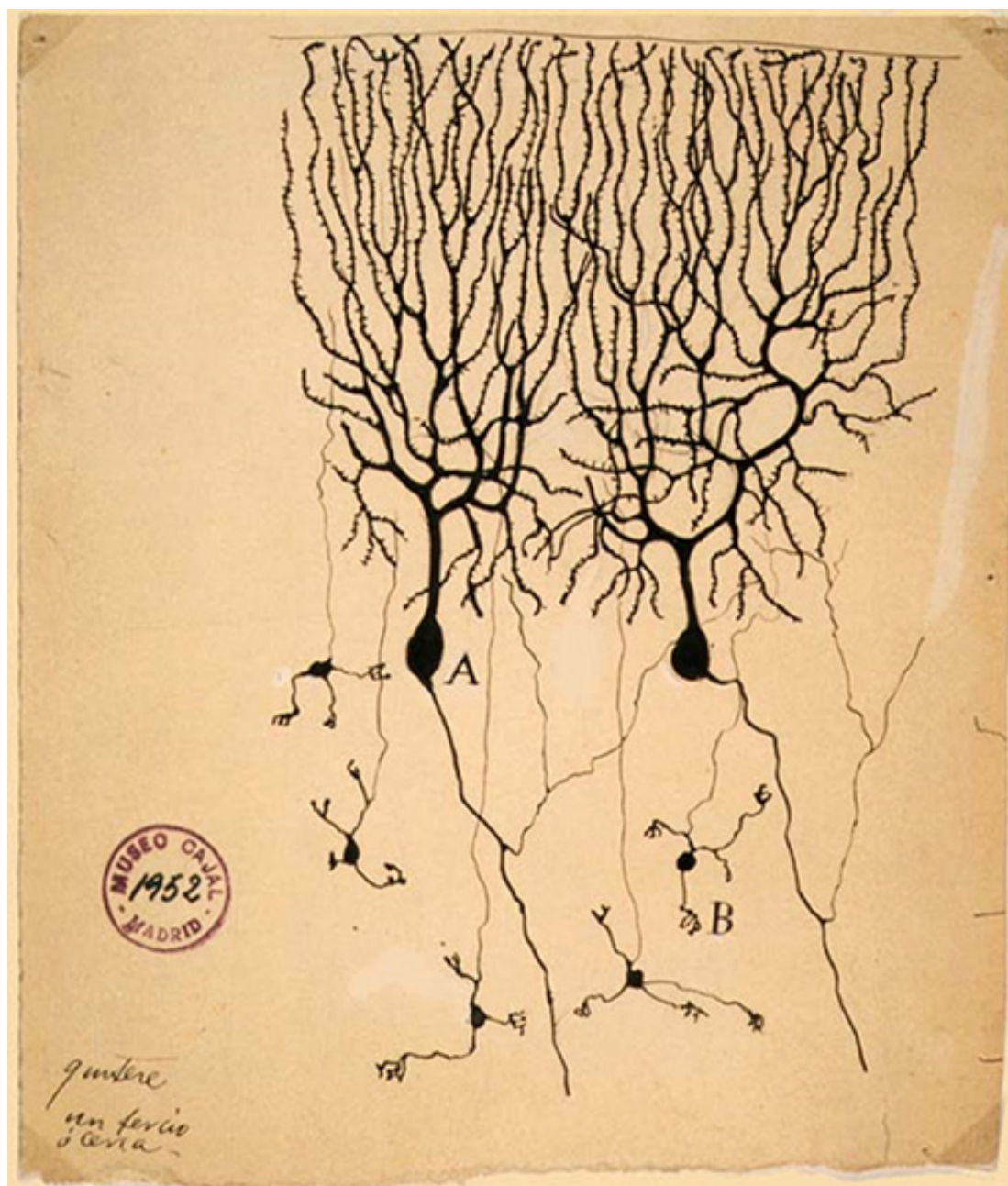
Традиционные компьютеры выполняют конкретные и четкие команды. В сердцах этих калькуляторов нет места неопределенности или двойственности. А вот мозги животных работают совсем по-другому. Они медленнее, но обрабатывают несколько сигналов одновременно, что часто приводит к разного рода неопределенностям.

Самый основной вычислительный элемент любого биологического мозга — **нейрон**.



Существует много разных видов нейронов, но все они переносят электрический сигнал с одного своего конца, к другому. От дендритов к терминалам аксона. Дальше сигнал переходит к другому нейрону и все повторяется. Именно таким образом ваше тело чувствует свет, звуки, прикосновения, тепло и так далее. Сигналы принимаются из внешней среды специальными сенсорными нейронами. Затем эти сигналы передаются через нервную систему в мозг, который почти полностью состоит из нейронов.

Ниже вы можете увидеть набросок некоторых видов нейронов мозга голубя, сделанные испанским нейробиологом в 1889. Основные части нейрона сразу бросаются в глаза: дендриты и терминалы аксона.



Как много нейронов необходимо для выполнения интересных и сложных задач? Человеческий мозг в среднем содержит около 100 тысяч миллиардов нейронов! У плодовых мушек около 100 000 нейронов, позволяющих ей летать, питаться, избегать опасности, находить еду и выполнять много других сложных задач. Современные компьютеры сейчас могут работать симулировать работу 100 000 нейронов. У круглых червей всего 302 нейрона — сущий пустяк по сравнению с компьютерами. Однако, эти черви могут выполнять такие действия, от которых никак не справятся традиционные компьютерные программы.

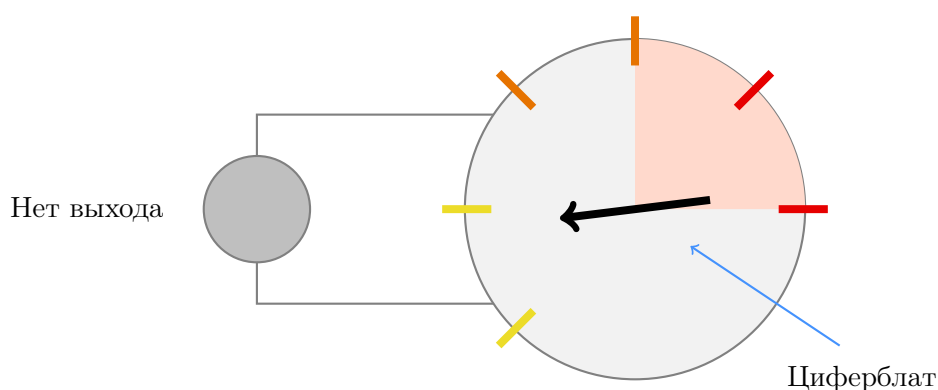
Так в чем же секрет? Почему биологические мозги так умны учитывая, что они работают медленнее и зачастую состоят из меньшего количества самих вычислительных элементов в сравнении с компьютерами? Мы точно не знаем, как работает мозг в целом. Природа сознания для нас все еще покрыта тайной. Но мы достаточно хорошо изучили то, из чего мозг состоит — нейроны. И на основе этих данных мы можем построить различные модели для решения разного рода проблем.

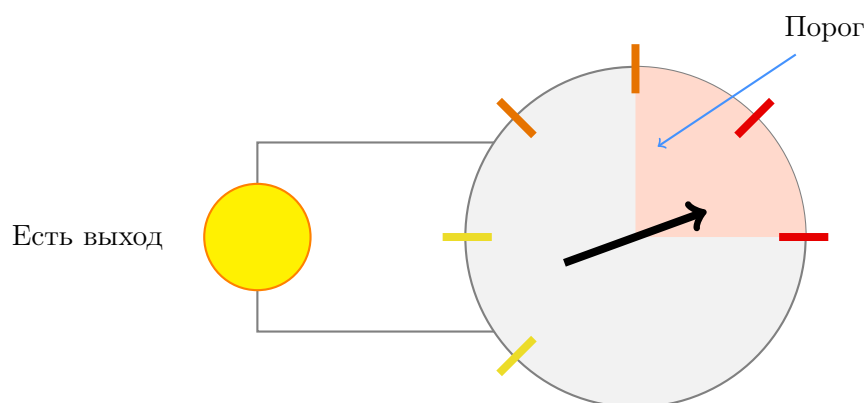
Давайте посмотрим, как работает нейрон. Он принимает электрический сигнал, прогоняет его через себя и выдает измененный сигнал. Звучит точь-в-точь как предсказательная машина или классификатор, которые мы рассматривали раньше. Они тоже что-то принимали на вход, что-то делали и выдавали ответ.

Может быть надо представлять нейроны в виде линейных функций, прямо как мы делали раньше? Идея хорошая, но нет. Ответ биологического нейрона вовсе не является результатом применения линейной функции ко входу. То есть ответ нейрона не считается так:

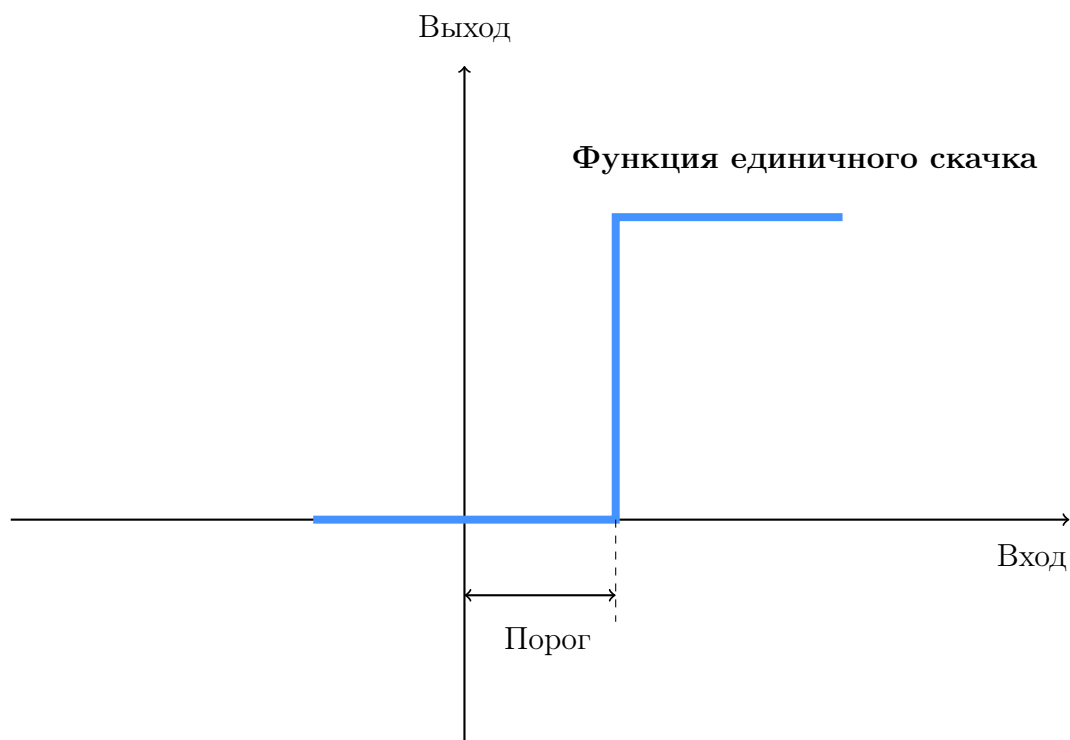
$$\text{выход нейрона} = (\text{константа} \times \text{вход}) + (\text{возможно, еще константа})$$

Наблюдения показывают, что нейроны реагируют на вход не сразу. Они подавляют входные сигналы до тех пор, пока они не станут настолько большими, что спровоцируют испускание выходного сигнала. Можно представить это в виде какой-то границы, которую необходимо преодолеть для того, чтобы нейрон дал ответ. Это интуитивно понятно. Нейроны не должны реагировать на всякие шумы — только достаточно сильный сигнал вызывает реакцию. Картинка ниже демонстрирует идею того, что нейрон выдает ответ только тогда, когда входной сигнал превысил некоторый **порог**.





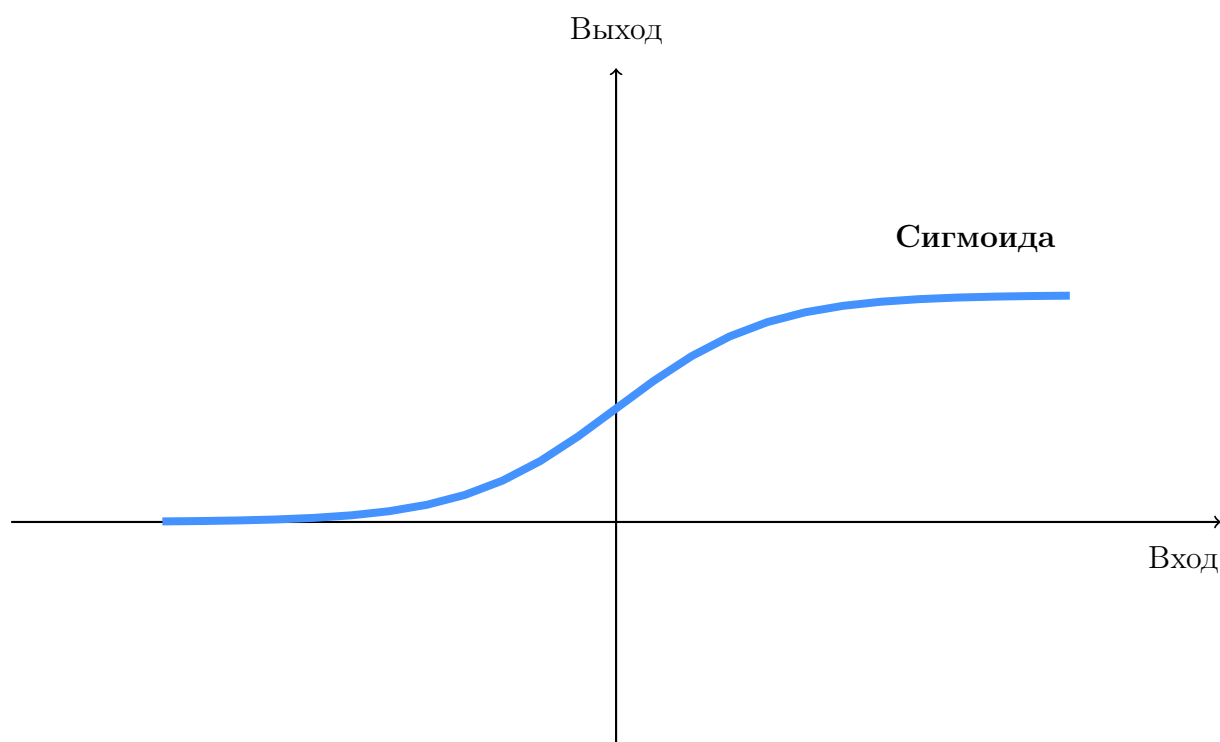
Функция, которая принимает входной сигнал и создает выходной, принимая во внимание некоторый порог называется **функцией активации**. Существует много разных функций активации. Одна из таких — **функция единичного скачка**:



Если значение входа маленькое, то выход равен 0. Однако, как только входное значение достигло или превысило пороговое значение, то сразу же появляется выходной сигнал. Искусственный нейрон с такой функцией активации уже больше похож на реальный биологический нейрон.

Мы можем улучшить функцию единичного скачка. S-образная функция на графике ниже называется **сигмоида**. Она гораздо более гладкая, чем угловатая функция

единичного скачка. Природа редко использует острые углы.



Имеено такую S-образную сигмоиду мы будем использовать в наших нейронных сетях. Специалисты в сфере искусственного интеллекта иногда используют другие, но похожие на нее функции. Но сигмоида очень простая и используется повсеместно.

Вот так выглядит уравнение сигмоиды, которую также часто называют **логистической функцией**:

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

На первый взгляд уравнение выглядит страшно. Буква  $e$  это математическая константа, которая равна 2.71828... Это очень интересное число, которое встречается в любых областях математики и физики. Точки в конце 2.71828 я поставил потому что цифры дробной части этого числа никогда не заканчиваются. Такие числа в математике трансцендентными. Это очень интересная тема, но для наших целей это не имеет значения и вы можете считать, что  $e = 2.71828$ .

Интересный факт. Если  $x = 0$ , то  $e^{-x} = 1$ , так как любое число в степень 0 равно 1. Имеем  $1/(1 + 1) = 1/2$ . Это означает, что сигмоида пересекает ось  $y$  в точке  $y = 1/2$ .

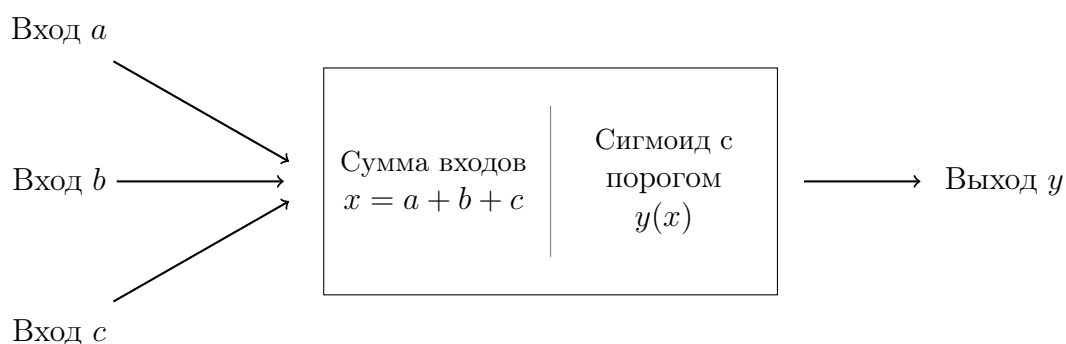
На самом деле есть еще одна серьезная причина использовать именно сигмоиду вместо множества других S-образных функций, которые тоже могут отвечать за выход

нейрона. Причина в том, что сигмоиду очень удобно использовать для вычислений, которые мы вскоре будем делать. С другими функциями процесс серьезно усложняется.

Давайте теперь попробуем построить модель искусственного нейрона.

Во-первых, биологические нейроны имеют много входов. Для нас эта идея уже не супер новая, так как логические функции принимают два аргумента.

И что нам делать с таким количеством входов? Мы просто складываем их друг с другом и полученную сумму передаем в качестве аргумента сигмоиду. Значение функции и является выходом нейрона. Так и работают биологические нейроны. Диаграмма ниже прекрасно иллюстрирует то, как работает нейрон:

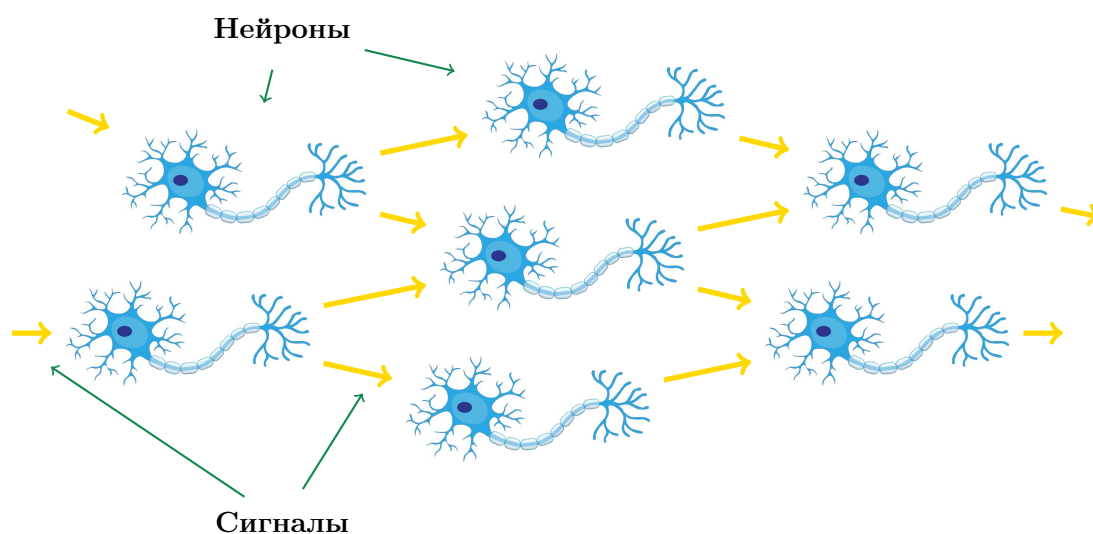


Если сумма всех сигналов оказалась меньше порога, то нейрон подавляет проходящий через него сигнал. А вот если  $x$  оказался больше или равен порогу, то нейрон пропускает сигнал через сигмоид и выдает результат. Важно заметить, что нейрон может отреагировать в том случае, если на некоторые входы поступил очень мощные сигналы, а на остальные небольшой. А может получиться, что на все входы поступили не очень мощные сигналы, но их суммы все же хватило для преодоления порога и активации нейрона. Именно в этом и кроется двойственность и неопределенность, присущая нейронам, а значит и нейронным сетям.

В реальности нейронные принимают электрический сигнал с помощью дендритов и если сумма сигналов ото всех дендритов выше порога, то нейрон передает измененный сигнал по аксону к дендритам других нейронов.

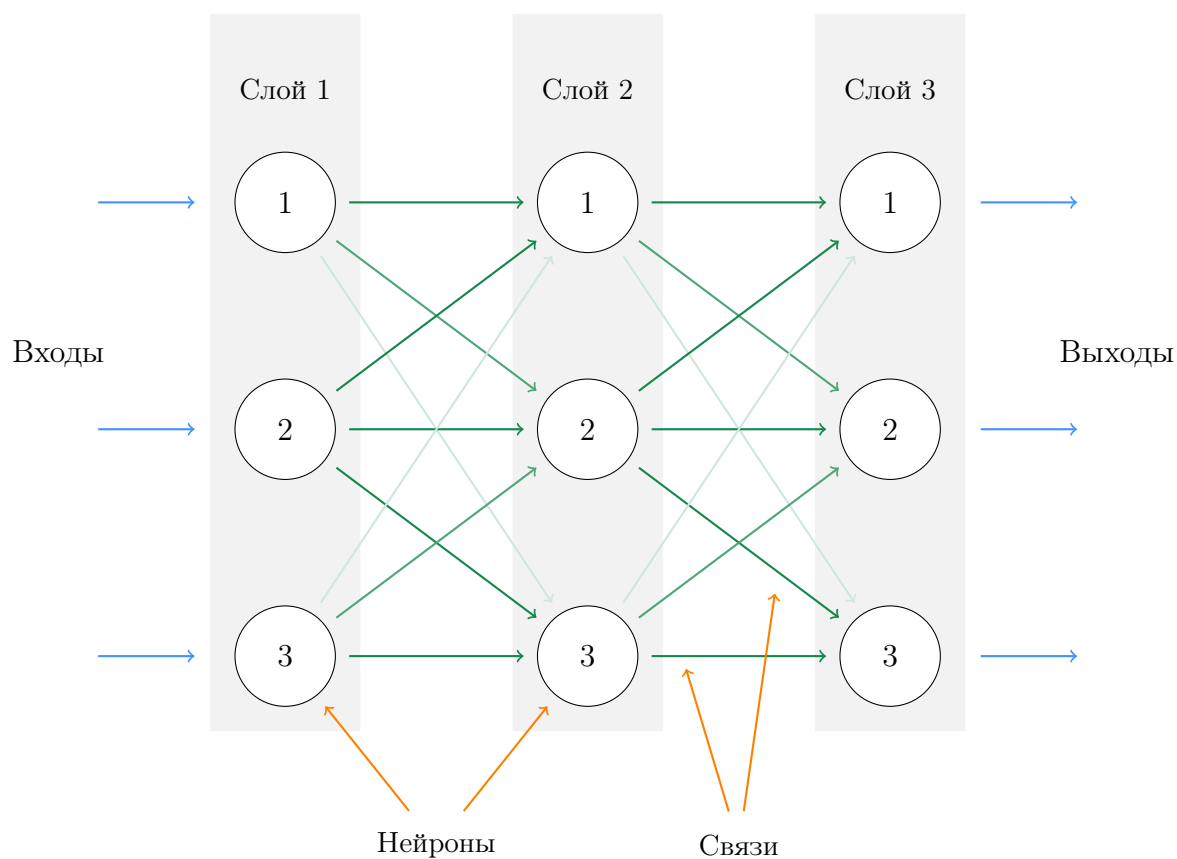
Рисунок ниже демонстрирует этот принцип связи нейронов друг с другом:





Заметьте, что каждый нейрон получает сигналы от предыдущих и, если превышен порог, сам отдает сигнал множеству следующих.

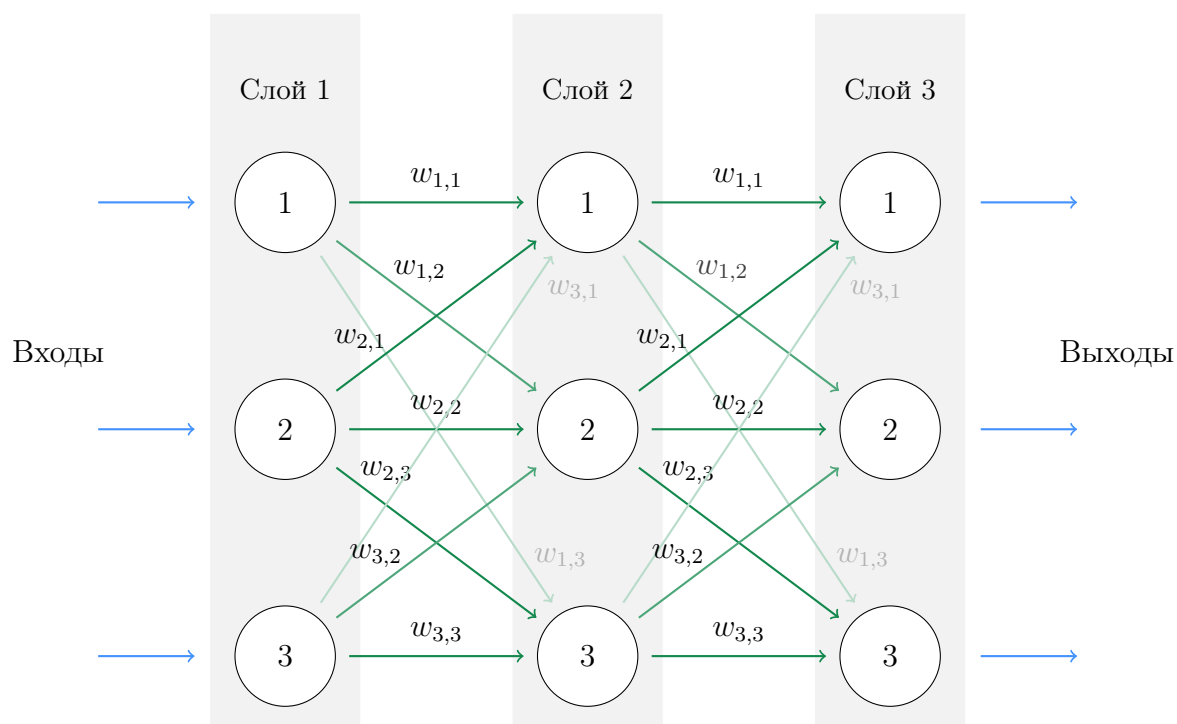
На рисунке выше много лишних деталей. Для нашей искусственной модели нет разницы, какую форму имеет каждый нейрон и где они расположены. Давайте будем считать, что в нашей искусственной нейросети есть слои нейронов и каждый нейрон в одном слое связан со всеми нейронами из следующего и предыдущего слоев:



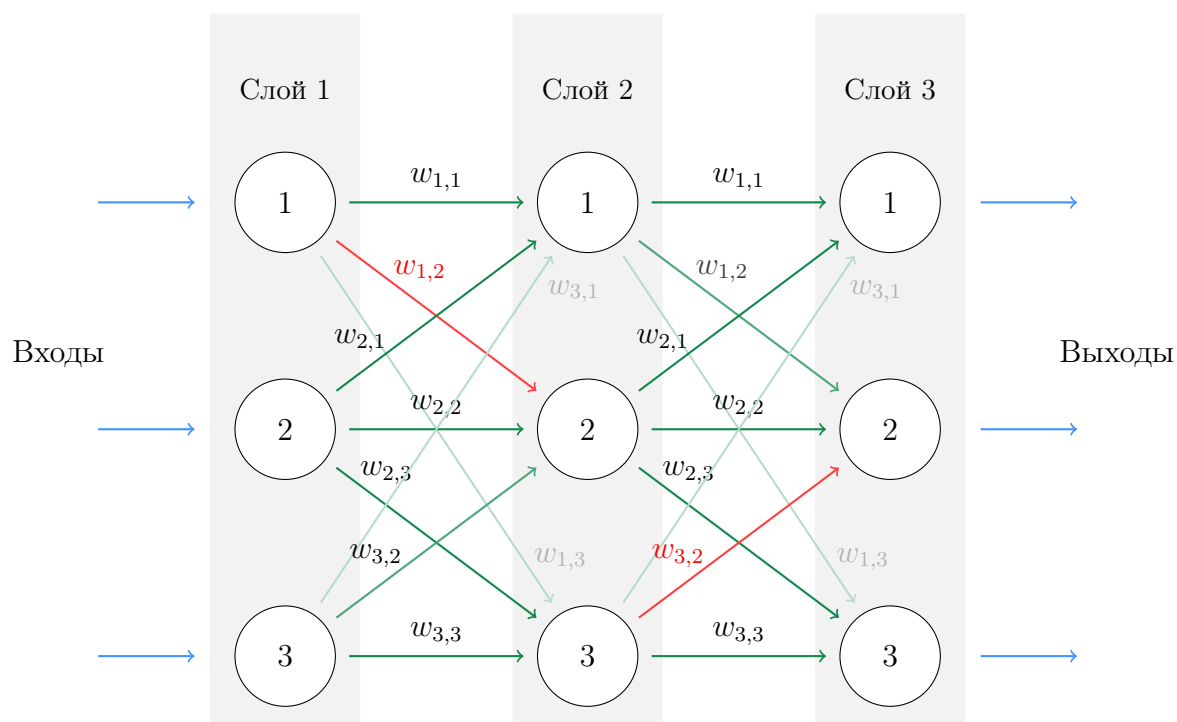
На диаграмме выше изображено три слоя, в каждом из которых находятся по 3 искусственных нейрона. Каждый нейрон имеет связи со всеми нейронами в предыдущем и следующем слоях. Великолепно! Но что из диаграммы выше отвечает за процесс обучения? У нас ведь должны быть какие-то параметры, которые мы можем изменять для минимизации погрешности. Есть ли на этой диаграмме что-то, что можно произвольно менять?

Можно пытаться менять сумму входящих в нейрон сигналов. Можно также искать подходящий наклон сигмоиды. На практике заниматься такими расчетами очень сложно. Самый простой способ — менять силу связей между искусственными нейронами.

Сила связи выражается ее **весом**. Вес связи умножается на значение проходящего по связи сигнала. Маленький вес будет уменьшать проходящей по этой связи сигнал. Большой вес дополнительно усилит сигнал.



Поясню, что означают эти маленькие буквы с цифрами в индексах. Вес  $w_{3,2}$  означает вес связи от искусственного нейрона 3 к нейрону 2 следующего слоя. Например,  $w_{1,2}$  есть значение, на которое уменьшается или увеличивается сигнал, который идет от первого нейрона в слое ко второму нейрону в следующем слое. На диаграмме ниже подсвечены рассмотренные выше связи и их веса:



У вас может возникнуть совершенно правильный вопрос: а какой смысл связывать нейрон со всеми нейронами в следующем и предыдущем слоях? Можно ли обойтись меньшим количеством связей? На самом деле можно соединять нейроны друг с другом любыми способами. Но чаще всего этого не делают, так как очень трудно сказать, сколько связей потребуется для решения конкретной задачи. Более того, в процессе обучения сеть сама избавится от ненужных связей.

Это как? Дело в том, что в процессе обучения веса некоторых связей станут равны 0 или очень близко к нему. А это означает, что они просто подавят проходящий через связь сигнал, так как любое число при умножении на 0 тоже равно 0. Такие "мертвые" связи никакой роли играть не будут. По мере обучения, нейросеть сама определяет, какие связи ей нужны, а какие нет. Лучшим шагом от нас тут будет просто связать все нейроны друг с другом и предоставить сети решать самой, что нужно, а что нет.

### Ключевые моменты

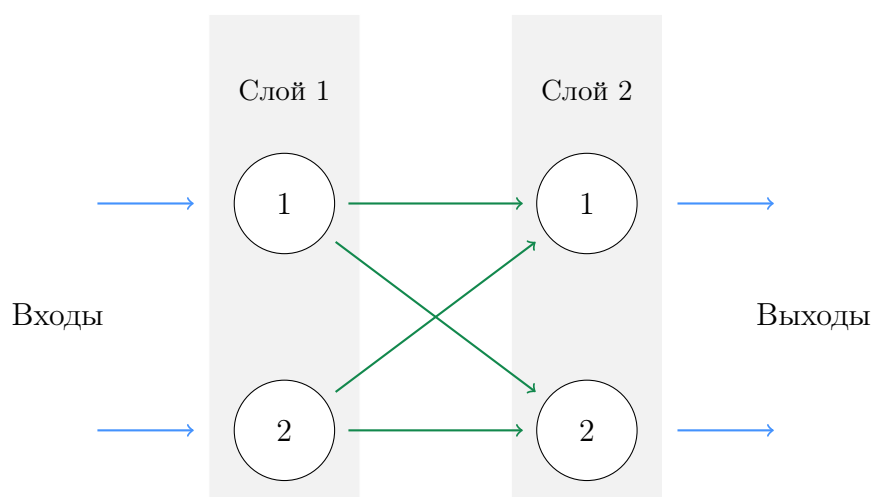
- Биологические мозги способны решать сложнейшие задачи полета, пропитания, изучения других языков и избегания хищников несмотря на то, что они работают медленнее и имеют меньше памяти, чем современные компьютеры.
- Биологические мозги не позволяют испорченным или зашумленным сигналам серьезно повлиять на результаты в сравнении с традиционными компьютерами.

- Искусственные нейронные сети являются упрощенными моделями биологических мозгов.

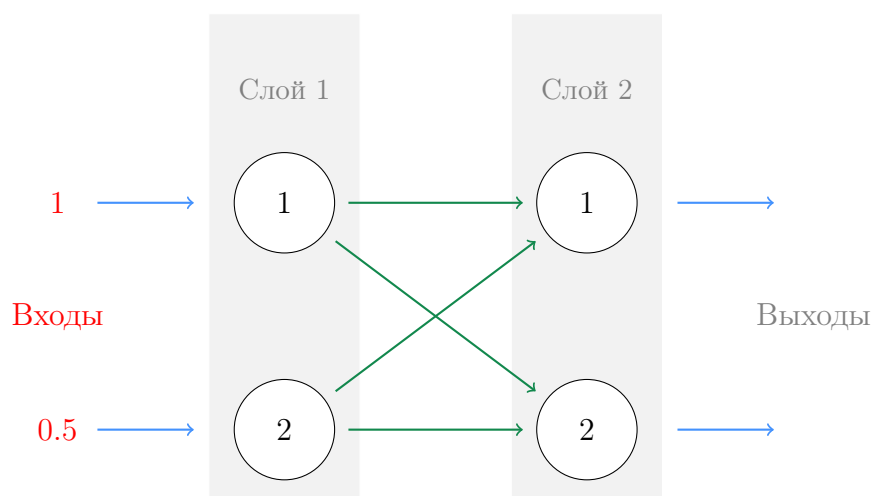
## 1.7 Проход сигнала через нейросеть

Согласитесь, картинка с изображением трехслойной нейросети, в который каждый нейрон связан со всеми нейронами в следующем и предыдущем слоях, выглядит классно.

Но проводить столько вычислений ради получения выхода, да еще по несколько раз по мере обучения сети! Это очень тяжелая работа! Да, тяжелая. Но в дальнейшем эту работу будет выполнять компьютер. Сейчас же наша задача — научиться понимать, что происходит с сигналами внутри нейронной сети. Для простоты мы будем рассматривать простую нейросеть, состоящую из 2 слоев, в каждом из которых есть по 2 нейрона:



Давайте представим, что на входы подаются 1 и 0.5:



Как я писал выше, мы складываем поступающие на входы нейрона сигналы и используем эту сумму в качестве аргумента функции активации. В качестве активации мы используем следующий сигмоид:

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

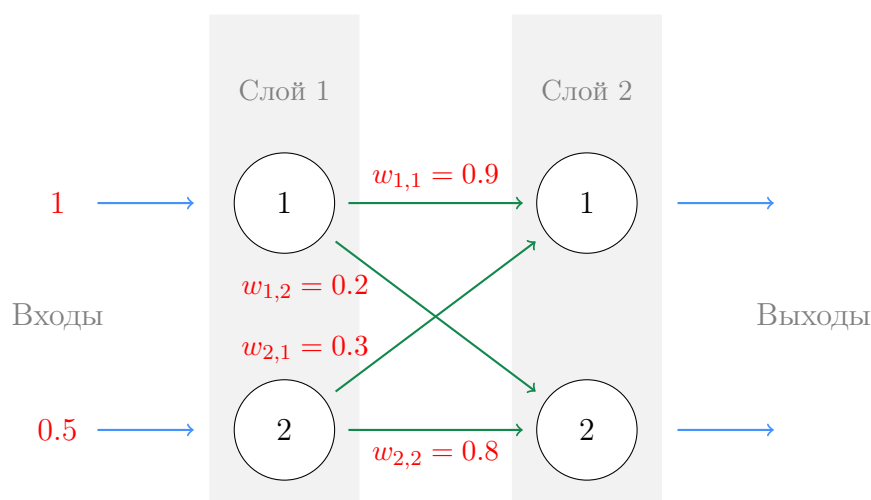
где  $y$  является выходом нейрона, а  $x$  — сумма поступивших сигналов.

А что с весами связей? Это отличный вопрос. Для начала возьмем случайные веса:

- $w_{1,1} = 0.9$
- $w_{1,2} = 0.2$
- $w_{2,1} = 0.3$
- $w_{2,2} = 0.8$

Брать случайные веса не такая уж и плохая идея. Мы брали случайные коэффициенты для простого линейного классификатора и машины, предсказывающей количество миль по километрам.

Отметим на диаграмме нашей нейросети веса связей:



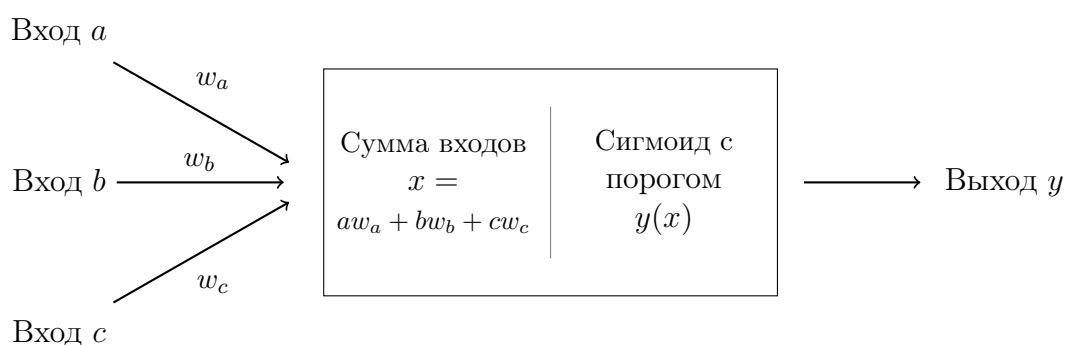
Давайте считать!

Исторически сложилось, что первый слой в искусственных нейросетях просто передает сигнал нейронам следующего слоя. Так что никак вычислений на первом слое проводить не требуется.

На втором слое уже нужно считать. Для каждого нейрона нужно что-то сделать с суммой входящих в него сигналов. Модифицировать сумму входящих сигналов мы будем с помощью сигмоиды:

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

Но входящие сигналы у нас не простые, а модифицированные весами связей. Это тоже надо учесть. На диаграмме ниже приведена модель искусственного нейрона. Мы уже видели ее ранее, но теперь мы учитываем веса связей.



Проведем все эти вычисления для первого нейрона во втором слое. Оба нейрона из первого слоя связаны с ним. Они получают на вход 1 и 0.5. Вес связи первого нейрона — 0.9. Вес связи второго — 0.3. Значит сумма взвешенных входов:

$$x = (\text{выход первого нейрона} \times \text{вес}) + (\text{выход второго нейрона} \times \text{вес})$$

$$x = (1 \times 0.9) + (0.5 \times 0.3)$$

$$x = 0.9 + 0.15$$

$$x = 1.05$$

Еще раз напомню, что веса связей и являются теми параметрами, которые мы будем в менять для минимизации погрешности.

Мы получили сумму взвешенных входов нейрона  $x$ . Теперь надо пропустить эту сумму через функцию активации, роль которой выполняет сигмоид.

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



Тут можно воспользоваться калькулятором. Получаем  $y = 0.7408$ .

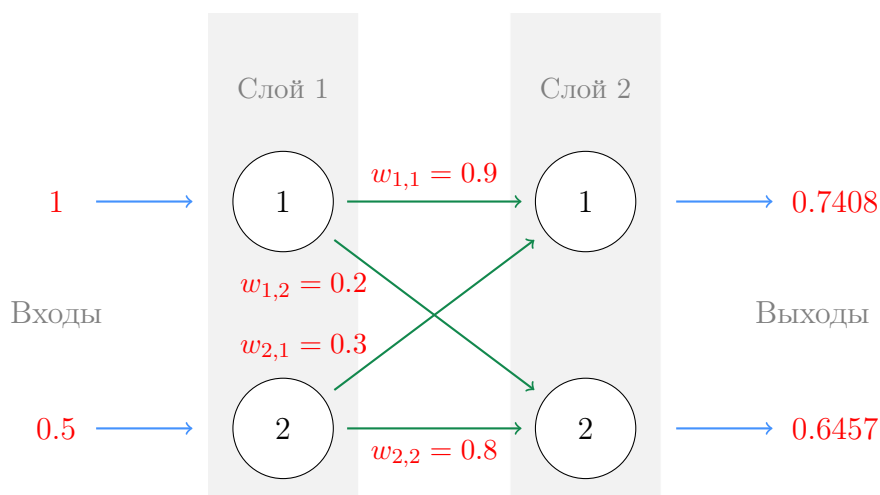
Отличная работа! Теперь мы имеем реальный выход одного из нейронов нашей нейросети.

Займемся теперь вторым нейроном второго слоя. Считаем сумму взвешенных входов:

$$\begin{aligned}x &= (1 \times 0.2) + (0.5 \times 0.8) \\x &= 0.2 + 0.4 \\x &= 0.6\end{aligned}$$

Мы получили  $x$ . Выход этого нейрона равен  $y = 0.6457$ .

Отобразим результаты работы нейросети на диаграмме:



Мы честно посчитали выходы двухслойной нейросети. Лично мне бы не хотелось проводить все эти вычисления для более больших нейросетей. К счастью, у нас есть никогда не устающие компьютеры, которые отлично подходят для быстрых и объемных вычислений.

Но это еще не все. Даже запись компьютерных инструкций по вычислениям для нейросетей, состоящих из 4, 8 или вообще 100 слоев быстро наскучит и не оберегает нас от ошибок в программном коде.

И тут нам на помощь вновь приходит математика, у которой есть специальные объекты, с помощью которых можно считать выход нейросети любого размера очень быстро и с помощью коротких команд.

Эти объекты называют **матрицами**. Ими мы сейчас и займемся.

## 1.8 Умножать матрицы полезно... Серьезно!

Матрицы проходят на первом курсе университета по любой технической специальности. Большинство студентов с отвращением вспоминают их. Помимо прочего, часто своей нудностью и бессмысленностью выделяется именно умножение матриц друг на друга.

В предыдущем разделе мы вручную посчитали выходы нейросети из 2 слоев с 2 нейронами в каждом слое. Работы было немало. А теперь представьте, сколько всего придется считать для нейросети из 5 слоев по 100 нейронов в каждом? Да просто запись самих вычисляемых выражений представляет из себя уже серьезную проблему ... все эти суммы сигналов, помноженных на их веса, которые затем попадают в функцию активации ... все надо считать для каждого нейрона, для каждого слоя ... ужас!

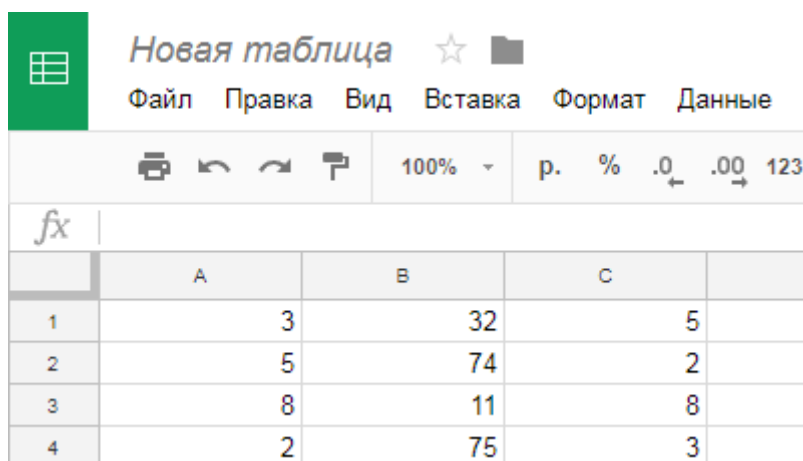
Как нам помогут матрицы? Они помогут нам в двух моментах. Во-первых, они помогают записать все вычисления в очень сжатой и простой форме. Никто не любит записывать и высчитывать сложные выражения. Это быстро наскучивает, а также приводит к вычислительным ошибкам. Во-вторых, много языков программирования могут работать с матрицами и с циклами. Компьютеры выполняют все вычисления быстро и без ошибок.

Короче говоря, матрицы позволяют представить нужные нам расчеты в простом и сжатом виде, а компьютеры проводят эти расчеты быстро и без ошибок.

Надеюсь, я смог понятно объяснить, зачем мы будем использовать матрицы даже несмотря на не самый приятный опыт работы с ними у некоторых читателей. Давайте начнем и снимем покров тайны с матрицы и их умножения.

**Матрица** — просто таблица, заполненная числами. Вот и все.

Наверняка многие из вас работали с Excel, или с другими программами для работы с таблицами. В таком случае вас уже не испугает обыкновенная таблица с числами. В Excel и других программах используют термин таблица. Мы можем называть ее и матрицей. На картинке ниже изображен фрагмент сайта "Google Таблицы".



The screenshot shows a spreadsheet window titled "Новая таблица" (New Table). The menu bar includes "Файл", "Правка", "Вид", "Вставка", "Формат", and "Данные". The toolbar contains icons for print, undo, redo, and a zoom dropdown set to 100%. The formula bar shows "fx". The table has 4 rows and 3 columns labeled A, B, and C. The data is as follows:

	A	B	C
1	3	32	5
2	5	74	2
3	8	11	8
4	2	75	3

Как мы уже выяснили, матрица представляет собой просто таблицу из чисел. Вот пример матрицы с размерностью 2 на 3:

$$\begin{pmatrix} 23 & 43 & 22 \\ 43 & 12 & 54 \end{pmatrix}$$

С размерностью 2 на 3 означает, что в матрице 2 строки и 3 столбца.

Некоторые люди предпочитают использовать квадратные скобки вокруг матрицы:

$$\begin{bmatrix} 23 & 43 & 22 \\ 43 & 12 & 54 \end{bmatrix}$$

Кстати, матрицы не обязательно должны состоять из чисел. Например, они могут состоять из **переменных**, которые скрывают под собой какое-то еще неуказанное число:

$$\begin{pmatrix} \text{длина корабля} & \text{длина самолета} \\ \text{ширина корабля} & \text{ширина самолета} \end{pmatrix}$$

Для нас важное значение имеет операция умножения матриц друг на друга. Возможно вы уже знаете, как это делается. Если нет, то сейчас мы подробно рассмотрим этот процесс.

В примере ниже мы умножаем две простые матрицы друг на друга:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1 \times 5) + (2 \times 7) & (1 \times 6) + (2 \times 8) \\ (3 \times 5) + (4 \times 7) & (3 \times 6) + (4 \times 8) \end{pmatrix} \\ = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Можно заметить, что мы не просто умножаем соответствующие элементы друг на друга, как можно было ожидать исходя из названия операции "умножение матриц". Если бы это было так, что левый верхний элемент итоговой матрицы равнялся бы  $1 \times 5$ , а правый нижний  $4 \times 8$ .

Умножение матриц происходит по-другому. Возможно, вы уже заметили, как оно происходит по приму выше. Вот подсказка:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1 \times 5) + (2 \times 7) & (1 \times 6) + (2 \times 8) \\ (3 \times 5) + (4 \times 7) & (3 \times 6) + (4 \times 8) \end{pmatrix} \\ = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Верхний левый элемент итоговой матрицы мы получаем, беря элементы первой строки первой матрицы и первого столбца второй матрицы. Мы умножаем соответствующие по номеру следования элементы друг на друга и складываем друг с другом. Разберем подробно. Ищем левый верхний элемент итоговой матрицы. Берем 1 из верхней строки первой матрицы и умножаем на 5 из левого столбца второй матрицы. Получаем  $1 \times 5 = 5$ . Запоминаем результат и идем дальше вправо в первой матрице и вниз во второй. В первой матрице у нас следующим вправо элементом идет 2. Во второй матрице следующим вниз элементом идет 7. Умножаем  $2 \times 7 = 14$  и складываем с предыдущим результатом:  $5 + 14 = 19$ . Так мы получили левый верхний элемент итоговой матрицы.

Слов для описания требуется много, но тут главное понять идею и внимательно изучить пример. Вот так мы получаем нижний правый элемент итоговой матрицы:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1 \times 5) + (2 \times 7) & (1 \times 6) + (2 \times 8) \\ (3 \times 5) + (4 \times 7) & (3 \times 6) + (4 \times 8) \end{pmatrix} \\ = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Когда мы хотим найти значение элемента итоговой матрицы, мы смотрим номер строки и столбца искомого элемента (в этом примере — вторая строка и второй столбец), умножаем соответствующие по номеру элементы ( $3 \times 6$  и  $4 \times 8$ ) и складываем результаты друг с другом:  $18 + 32 = 50$ .

А вот умножение матриц, в которых вместо чисел используются переменные:

$$\begin{pmatrix} a & b & \dots \\ c & d & \dots \end{pmatrix} \begin{pmatrix} e & f \\ g & h \\ \dots & \dots \end{pmatrix} = \begin{pmatrix} (a \times e) + (b \times g) + \dots & (a \times f) + (b \times h) + \dots \\ (c \times e) + (d \times g) + \dots & (c \times f) + (d \times h) + \dots \end{pmatrix} \\ = \begin{pmatrix} ae + bg + \dots & af + bh + \dots \\ ce + dg + \dots & cf + dh + \dots \end{pmatrix}$$

Вместо букв в примере выше мы могли бы использовать любые числа. Мы просто дали более общее определение умножения матриц. Более общее оно еще и потому что в этом примере под точками могут скрываться дополнительные столбцы и строки.

Нельзя просто так взять и умножить две матрицы друг на друга. Они должны быть совместимы. Возможно, вы уже заметили это. Две матрицы совместимы, если количество столбцов первой матрицы равно количеству строк второй матрицы. В противном случае при умножении вам просто не хватит элементов. Например, матрицу "2 на 2" не получится умножить на матрицу "5 на 5".

Как это применить к нейросетям? Мы можем написать следующее:

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} \text{Вход 1} \\ \text{Вход 2} \end{pmatrix} = \begin{pmatrix} (\text{Вход 1} \times w_{1,1}) + (\text{Вход 2} \times w_{2,1}) \\ (\text{Вход 1} \times w_{1,2}) + (\text{Вход 2} \times w_{2,2}) \end{pmatrix}$$

Магия!

Первая матрица содержит веса связей нейросети. Вторая матрица содержит поступившие на первый слой сети сигналы. Результат умножения этих двух матриц — матрица с суммой взвешенных входов. Это действительно так. Помните в предыдущем разделе следующее выражение?

$$x = (\text{выход первого нейрона} \times \text{вес}) + (\text{выход второго нейрона} \times \text{вес})$$

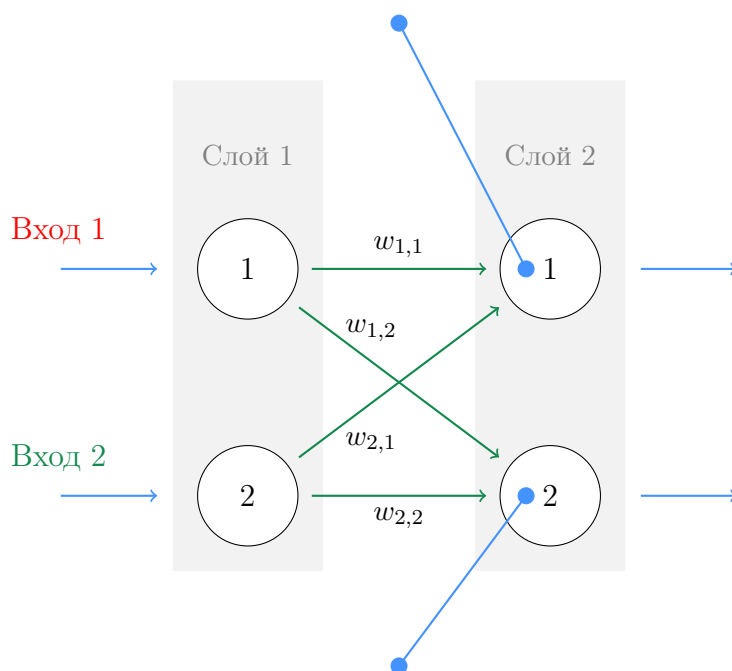
Вот мы и получили сумму взвешенных входов для обоих нейронов сразу:

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} \text{Вход 1} \\ \text{Вход 2} \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Затем эти значения  $x$  мы использовали в качестве аргумента функции активации.

Если у вас еще остались сомнения, вот наглядная демонстрация:

$$x = (\text{Вход 1} \times w_{1,1}) + (\text{Вход 2} \times w_{2,1})$$



$$x = (\text{Вход 1} \times w_{1,2}) + (\text{Вход 2} \times w_{2,2})$$

Это очень удобно!

Почему? Потому что теперь мы можем все суммы всех взвешенных входов нейронов сразу с помощью произведения матриц. Это можно выразить так:

$$\mathbf{X} = \mathbf{W} \times \mathbf{I}$$

В равенстве выше  $\mathbf{W}$  — матрица весов связей, соединяющих первый и второй слои нейросети,  $\mathbf{I}$  — матрица входов сети и  $\mathbf{X}$  — итоговая матрица, содержащая суммы всех взвешенных входов для второго слоя. Буквы, означающие матрицы в математике обычно выделяют **жирным**, чтобы не спутать их с переменными.

Теперь можно особо не париться насчет количества нейронов в каждом слое. Если их будет больше, то вырастут размеры матриц. Но сами наши записи не увеличатся в размерах. Мы все также будем писать  $\mathbf{X} = \mathbf{W} \times \mathbf{I}$  в случае если у нас 2 нейрона или 200!

Если используемый нами язык программирования умеет работать с матрицами, он может самостоятельно найти значение  $\mathbf{X} = \mathbf{W} \times \mathbf{I}$ . Нам больше не придется записывать отдельные выражения для получения суммы взвешенных входов каждого нейрона в каждом слое и вычислять их по отдельности.

Фантастика! Такая несложная вещь, как умножение матриц предстает очень удобным инструментом для работы с нейросетями.

А что насчет функции активации? Тут все просто и не требует произведения матриц. Нужно всего лишь каждый элемент из матрицы  $\mathbf{X}$  использовать в качестве аргумента в функции активации:

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

Это действительно настолько просто. Все суммы взвешенных входов у нас уже имеются в матрице  $\mathbf{X}$ . Как мы видели ранее, функция активации играет роль порога и "сжимателя" выхода нейрона. Примерно так работают биологические нейроны. Итак, матрица, содержащая все выходы нейронов слоя получается так:

$$\mathbf{O} = \text{сигмOID}(\mathbf{X})$$

Важно помнить, что  $\mathbf{X} = \mathbf{W} \times \mathbf{I}$  применяется для вычислений между двумя соседними слоями. Если в нейросети у нас 3 слоя, то нам придется вычислить это выражение для 1 и 2 слоев. Потом, после получения выходов нейронов 2-го слоя мы должны вновь выполнить произведение матриц для 2 и 3 слоев.

Хватит теории. Давайте проверим все эти формулы в деле. Сейчас мы будем использовать нейросеть из 3 слоев по 3 нейрона в каждом.

## Ключевые моменты

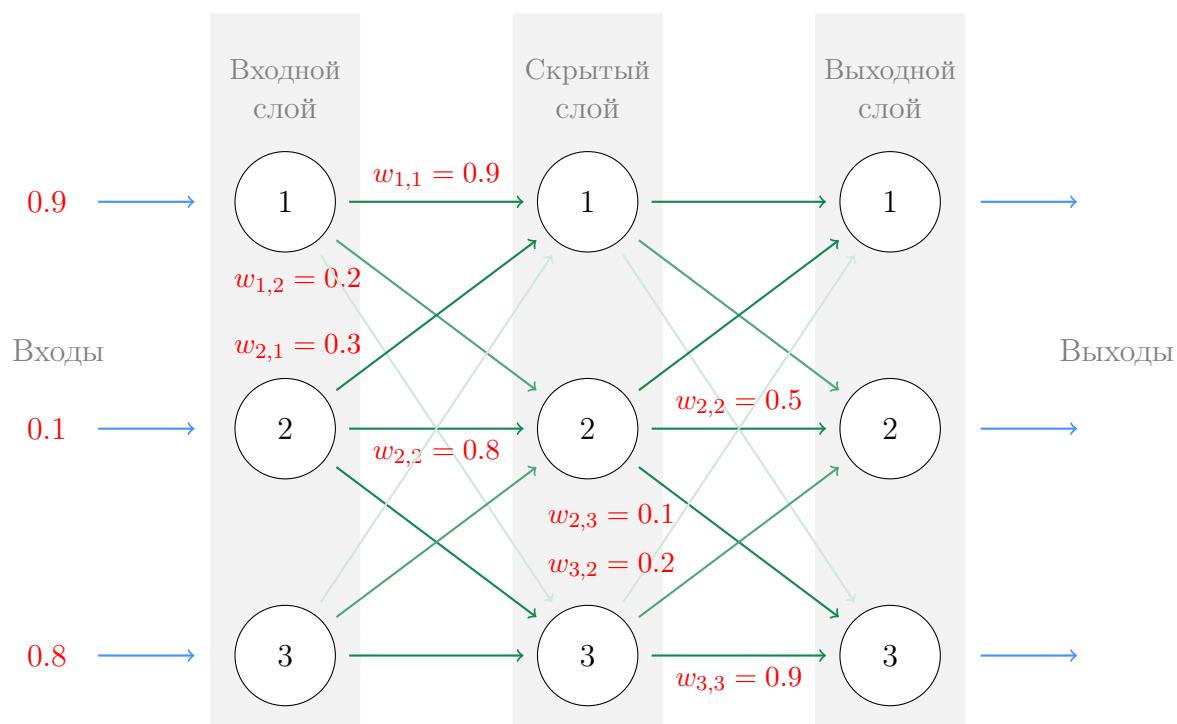
- Для вычисления преобразований сигналов между слоями нейросети мы можем использовать **произведение матриц**.
- Произведение матриц также очень удобно записывать для нас, так как общее выражение не зависит от размера нейросети.
- Что еще более важно, некоторые языки программирования могут работать с матрицами. Они могут эффективно и быстро посчитать произведение матриц вместо нас.



## 1.9 Трехслойная нейросеть и произведение матриц

Давайте применим на практике освоенный в предыдущем разделе материал, а именно — опробуем произведение матриц для вычислений в трехслойной нейросети.

На диаграмме ниже представлена трехслойная нейросеть с 3 нейронами в каждом слое. Некоторые веса связей не отображены для сохранения опрятности рисунка.



В теории нейросетей существует несколько понятий, которые надо запомнить. У любой нейросети есть **входной слой**, нейроны которого принимают сигналы. У любой нейросети также есть **выходной слой**, который отвечает за результат работы сети. Наконец, в нейросетях есть **скрытые слои**. Звучит таинственно и злое. К сожалению, нет никаких ужасных и темных причин для такого названия. Оно закрепилось потому что выходы нейронов в слоях между входным и выходным слоями не являются итоговым результатом работы нейросети, а значит они "скрытые". Согласен, что такое название звучит немного по-дурацки, но трудно придумать что-то лучше.

Переходим к вычислениям. На нейроны входного слоя подаются следующие сигналы: 0.9, 0.1 и 0.8. Тогда наша матрица входов **I** выглядит так:

$$\mathbf{I} = \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

Вот и все. С первым слоем покончено, потому что все, что делает входной слой — принимает входящие в нейросеть сигналы.

Дальше идет скрытый слой. В нашей нейросети он всего один. Сейчас мы должны получить сумму взвешенных сигналов для каждого нейрона этого слоя. Как мы помним, каждый нейрон скрытого слоя соединен со всеми нейронами входного слоя и получает от них порцию сигналов, входящих в нейросеть. Сейчас мы не будем углубляться в многочисленные вычисления, как мы делали это ранее. Сейчас мы применим произведение матриц.

Из предыдущего раздела мы имеем формулу для вычисления всех сумм взвешенных входов нейрона:

$$\mathbf{X} = \mathbf{W} \times \mathbf{I}$$

У нас есть  $\mathbf{I}$ , но какие значения мы имеем в матрице  $\mathbf{W}$ ? На диаграмме выше уже приведены некоторые веса связей. В матрице ниже приведены все веса связей между первым и вторым слоями нейросети. Все значения весов случайные:

$$\mathbf{W}_{1-2} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix}$$

Например, вес связи, которая соединяет первый нейрон во входном слое и первый нейрон в скрытом слое равен  $w_{1,1} = 0.9$ , как указано на диаграмме выше. Вес связи, которая соединяет второй нейрон входного слоя со вторым нейроном скрытого слоя равен  $w_{2,2} = 0.8$ , как указано на диаграмме выше. На диаграмме не указан вес связи, которая соединяет третий нейрон входного слоя с первым нейроном скрытого слоя. Поэтому мы взяли значение этого веса из головы:  $w_{3,1} = 0.1$ .

А зачем мы используем индекс  $1 - 2$  рядом с названием нашей матрицы весов  $\mathbf{W}_{1-2}$ ?  $1 - 2$  означает, что наша матрица содержит веса связей, которые соединяют нейроны первого (входного) слоя со вторым (скрытым) слоем. Это сделано потому что далее

нам потребуется другая матрица весов, которые соединяют нейроны скрытого слоя с нейронами выходного слоя. Она будет называться  $\mathbf{W}_{2-3}$ .

Ниже расположена матрица  $\mathbf{W}_{2-3}$  с весами всех связей. Опять можете заметить, что вес связи, которая соединяет третий нейрон скрытого слоя с третьим нейроном выходного слоя равен  $w_{3,3} = 0.9$ .

$$\mathbf{W}_{2-3} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix}$$

Теперь у нас имеются все данные для проведения вычислений.

Давайте сейчас найдем матрицу сумм взвешенных входящих сигналов в нейроны скрытого слоя. Этой матрице тоже надо дать подходящее название, так как далее мы будем считать такую же матрицу, но уже для нейронов выходного слоя. Назовем ее  $\mathbf{X}_2$ .

$$\mathbf{X}_2 = \mathbf{W}_{1-2} \times \mathbf{I}$$

Не будем сейчас считать это произведение, потому что мы как раз и используем матрицы для того, чтобы с ними работал компьютер. Результат приведен ниже:

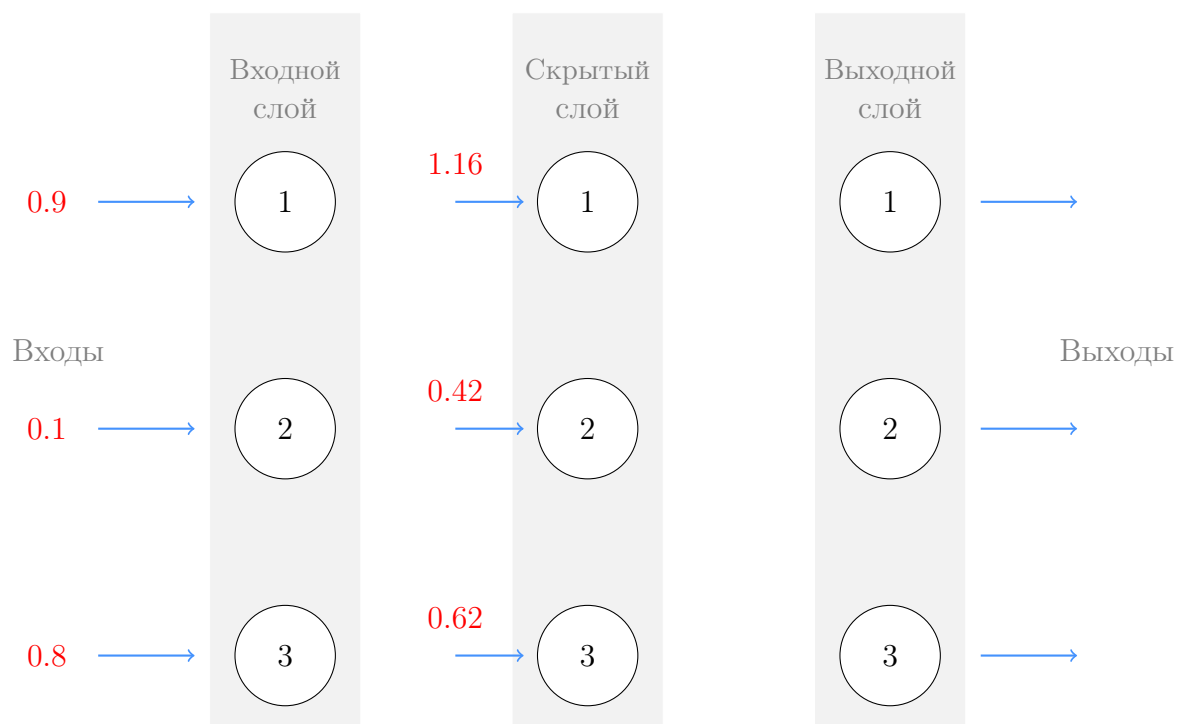
$$\mathbf{X}_2 = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix} \times \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

$$\mathbf{X}_2 = \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

Я просто использовал компьютер для вычисления этого произведения. Во второй части книги вы тоже будете использовать язык программирования Python для расчетов. Сейчас мы этого не делаем чтобы не отвлекаться от текущей задачи.

Итак, у нас есть матрица суммы взвешенных входящих сигналов для каждого нейрона скрытого слоя. Вычислили их за нас компьютер, который умеет находить произведение матриц. Этим достижением можно гордиться!

Давайте отобразим новые данные на диаграмме:



Пока что все отлично. Мы помним, что каждый нейрон модифицирует пришедшие в него сигналы с помощью функции активации. Поэтому нам надо высчитать значение следующего выражения:

$$\mathbf{O}_2 = \text{сигмоид}(\mathbf{X}_2)$$

Выражение выше означает, что каждый элемент из матрицы  $\mathbf{X}_2$  используется как аргумент в сигмоиде. Полученное значение и есть выход данного нейрона.

$$\mathbf{O}_2 = \text{сигмоид} \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

$$\mathbf{O}_2 = \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

На всякий случай перепроверим первый элемент. Уравнение сигмоиды:

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

Первый элемент матрицы  $\mathbf{X}_2$  равен 1.16. Значит  $e^{-1.16} = 0.3135$ . Итого имеем:

$$y = \frac{1}{1 + 0.3135} = 0.761$$

Кстати, вы могли заметить, что после прохода через функцию активации все значения находятся между 0 и 1. Так происходит потому что сигмоида не выдает другие значения. Действительно, знаменатель  $1 + e^{-x}$  всегда больше 1, а в числителе у нас 1. Поэтому мы всегда будем получать какую-то часть 1.

Фух! Давайте сделаем остановку и посмотрим, что у нас получилось. Скрытый слой нашей нейросети полностью завершил свою работу — мы имеем выходы всех нейронов этого слоя. Эти выходы мы получили, применив функцию активации к сумме взвешенных входных сигналов, которые пришли на входы нейронов. Обновим нашу диаграмму.



Если бы мы рассматривали двухслойную нейросеть, то мы бы уже остановились, так как полученных выходы нейронов и были бы результатом работы нейросети. Но мы не останавливаемся, так как у нас есть еще один, третий, слой.

Как действовать дальше? Да точно также, как мы работали со вторым слоем. Никакой разницы нет. Теперь у нас есть сигналы, поступающие на нейроны третьего слоя сети, также как до этого были сигналы, поступающие на нейроны второго слоя сети. Мы опять должны "взвесить" эти сигналы в зависимости от весов связей. Затем суммы этих взвешенных сигналов надо будет использовать в функции активации каждого нейрона. 3-й слой, 53-й или даже 103-й — все слои нейросети работают одинаково, вне зависимости от их номера.

Итак, давайте найдем матрицу сумм взвешенных сигналов, поступивших на входы третьего и последнего в сети слоя. Для этого мы должны умножить матрицу входов на матрицу весов связей. Матрица весов связей между 2 и 3 слоем у нас есть:  $\mathbf{W}_{2-3}$ . А матрица входов у нас равна матрице выходов нейронов второго слоя:  $\mathbf{O}_2$ . Имеем:

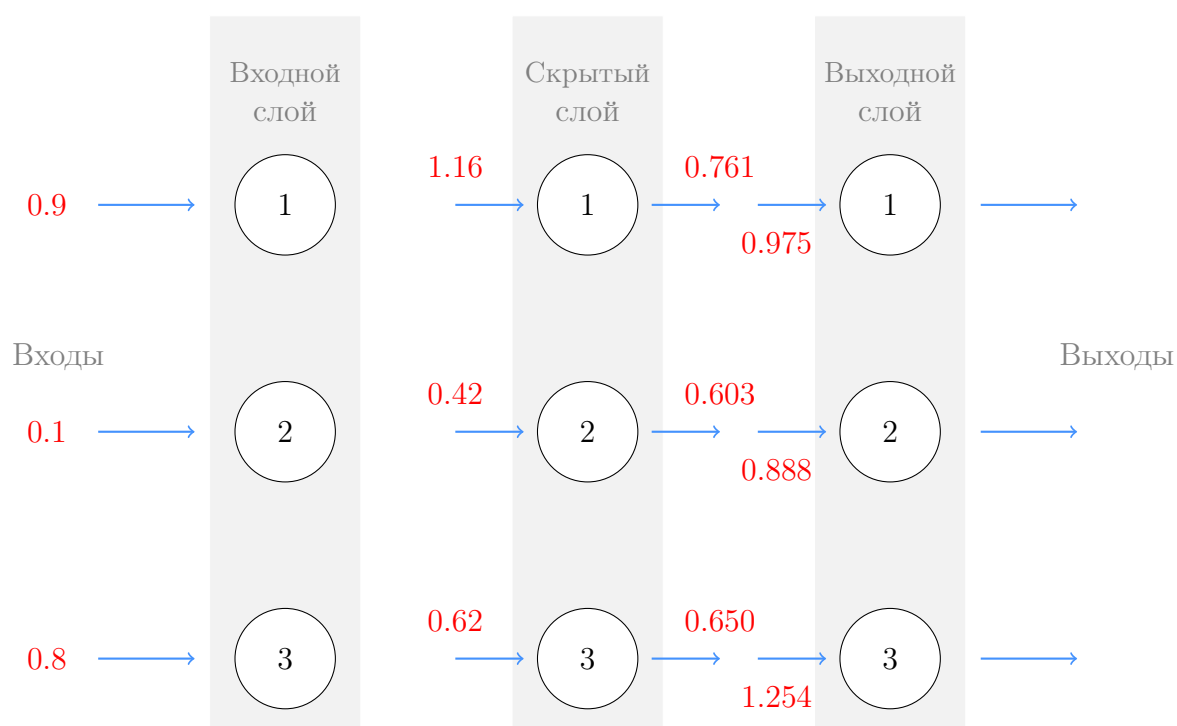
$$\mathbf{X}_3 = \mathbf{W}_{2-3} \times \mathbf{O}_2$$

Перейдем непосредственно к расчетам:

$$\mathbf{X}_3 = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix} \times \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

$$\mathbf{X}_3 = \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

Обновим нашу диаграмму:

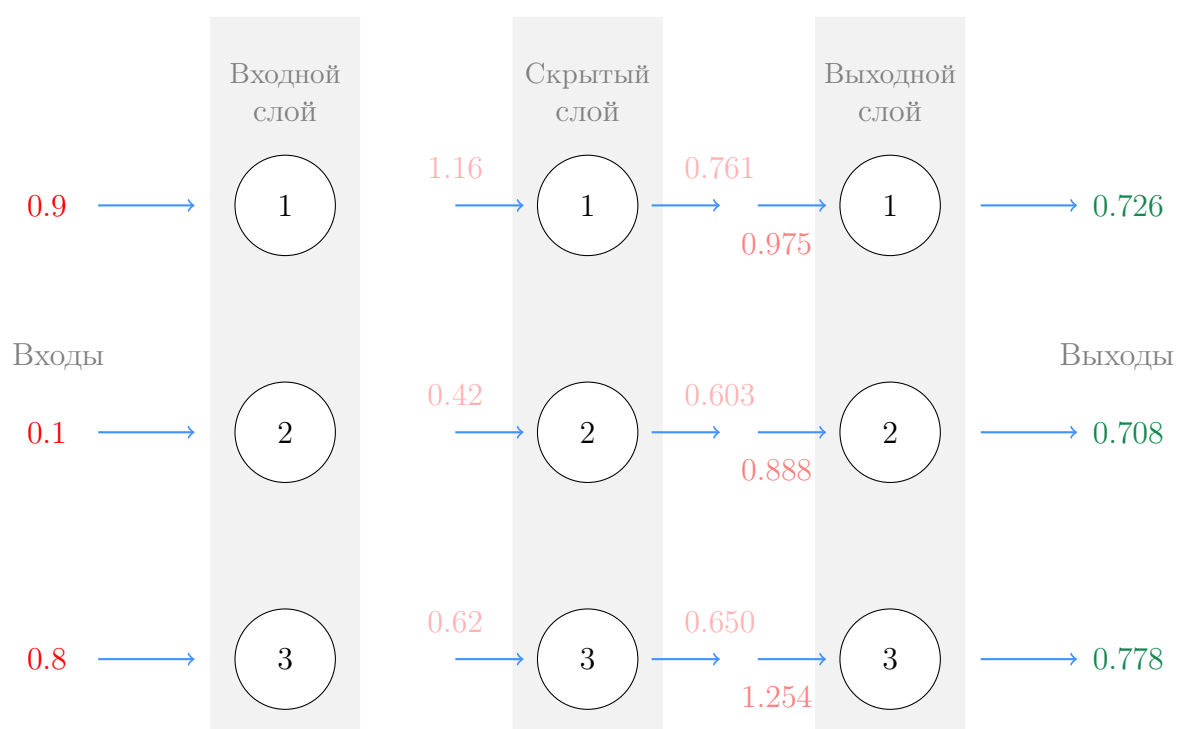


Осталось всего ничего! Надо лишь посчитать выходы всех нейронов выходного слоя нашей сети:

$$\mathbf{O}_3 = \text{сигмоид} \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

$$\mathbf{O}_3 = \begin{pmatrix} 0.726 \\ 0.708 \\ 0.778 \end{pmatrix}$$

Вот и все! У нас есть итоговый результат работы нейросети. Давайте отобразим его на диаграмме:



Итак, выходы сети равны: 0.726, 0.708 и 0.778.

Мы успешно прошли все этапы преобразования входящих в нейросеть сигналов.

А что делать с этим дальше?

А дальше нам надо сравнить выходы сети с примером из заранее готовой обучающей



выборки и получить погрешность. С помощью полученной погрешности надо откалибровать веса связей нейросети для улучшения ее результатов.

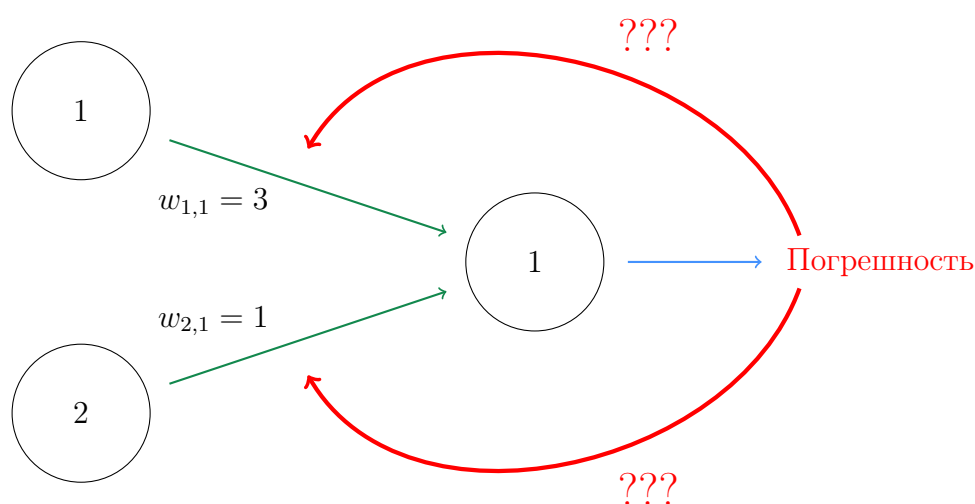
Этот процесс, возможно, окажется самым сложным для понимания моментом в этой книге. Поэтому мы разберем его неспеша и с подробной иллюстрацией всех действий.

## 1.10 Калибровка весов нескольких связей

Ранее мы настраивали линейный классификатор с помощью изменения постоянного коэффициента уравнения прямой. Мы использовали погрешность, разность между полученным и желаемым результатами, для настройки классификатора.

Все те операции были достаточно простые, так как сама связь между погрешностью и величины, на которую надо было изменить коэффициент прямой оказалось очень простой.

Но как нам калибровать веса связей, когда на получаемый результат, а значит и на погрешность, влияют сразу несколько нейронов? Рисунок ниже демонстрирует проблему:

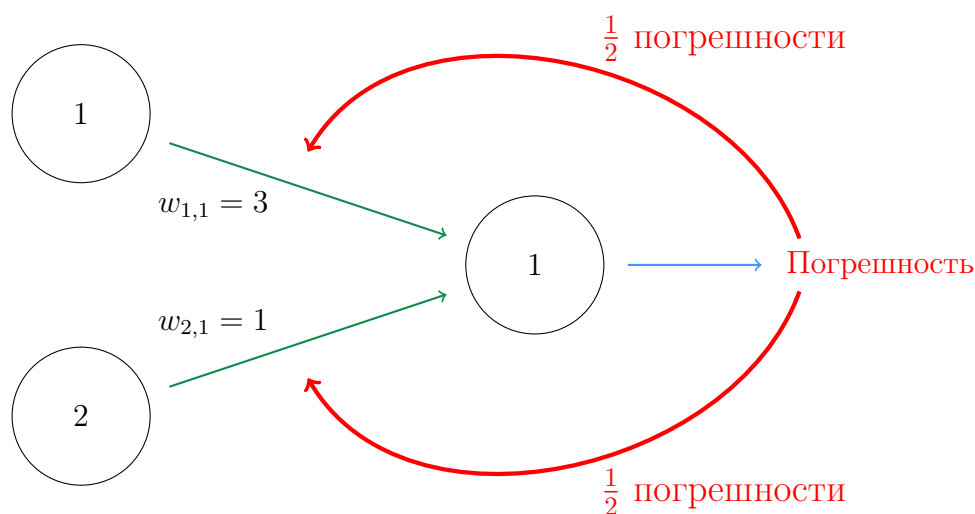


Очень легко работать с погрешностью, когда вход у нейрона всего один. Но сейчас уже два нейрона подают сигналы на два входа рассматриваемого нейрона. Что же делать с погрешностью?

Нет никакого смысла использовать погрешность целиком для корректировки одного веса, потому что в этом случае мы забываем про второй вес. Ведь оба веса задействованы в создании полученного результата, а значит оба веса виновны в итоговой погрешности.

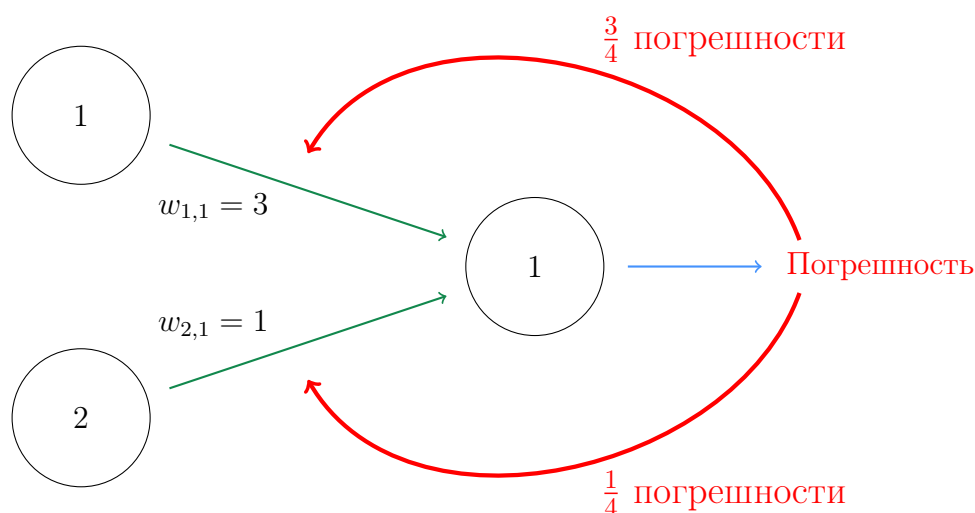
Конечно, существует очень маленькая вероятность того, что только один вес внес погрешность, а второй был идеально откалиброван. Но даже если мы немного поменяем вес, который и так не вносит погрешность, то в процессе дальнейшего обучения сети он все равно придет в норму, так что ничего страшного.

Можно попытаться разделить погрешность одинаково на все нейроны:



Классная идея. Хотя я никогда не пробовал подобный вариант использования погрешности в реальных нейросетях, я уверен, что результаты вышли бы очень достойными.

Другая идея тоже заключается в разделении погрешности, но не поровну между всеми нейронами. Вместо этого мы кладем большую часть ответственности за погрешность на нейроны с большим весом связи. Почему? Потому что за счет своего большего веса они внесли больший вклад в выход нейрона, а значит и в погрешность.



На рисунке изображены два нейрона, которые подают сигналы третьему, выходному

нейрону. Веса связей: 3 и 1. Согласно нашей идее о переносе погрешности на нейроны мы используем  $\frac{3}{4}$  погрешности на корректировку первого (большого) веса и  $\frac{1}{4}$  на корректировку второго (меньшего) веса.

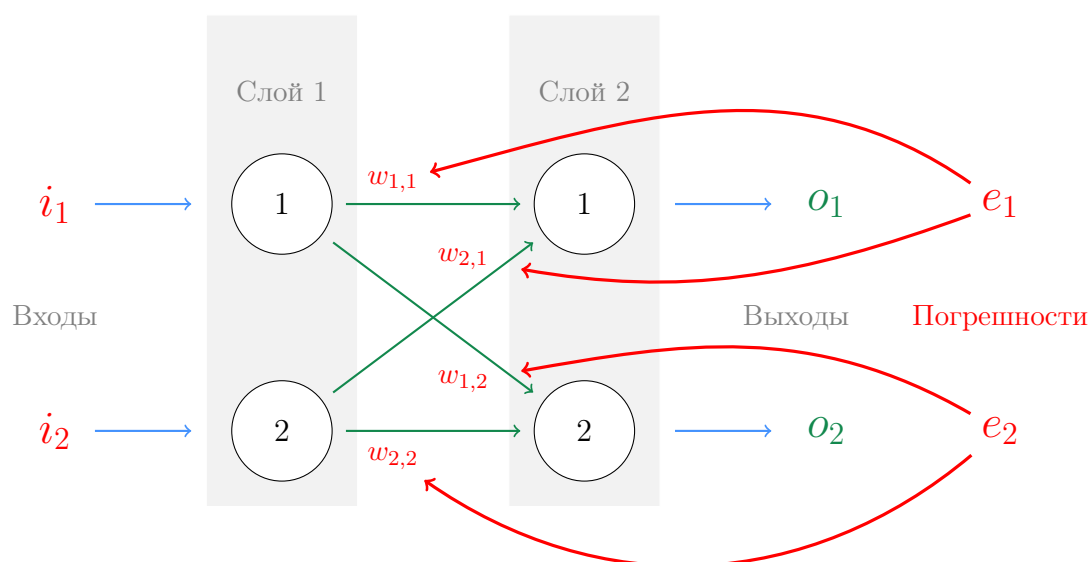
Идею легко развить до любого количества нейронов. Пусть у нас есть 100 нейронов и все они соединены с результирующим нейроном. В таком случае, мы распределяем погрешность на все 100 связей так, чтобы на каждую связь пришлась часть погрешности, соответствующая ее весу.

Как видно, мы используем веса связей для двух задач. Во-первых, мы используем веса в процессе распространения сигнала от входного до выходного слоя. Мы уже разобрались, как это делается. Во-вторых, мы используем веса для распространения ошибки в обратную сторону: от выходного слоя к входному. Из-за этого данный метод использования погрешности называют **методом обратного распространения ошибки**.

Если у нашей сети 2 нейрона выходного слоя, то нам пришлось бы использовать этот метод еще раз, но уже для связей, которые повлияли на выход второго нейрона выходного слоя. Рассмотрим эту ситуацию подробнее.

## 1.11 Обратное распространение ошибки от выходных нейронов

На диаграмме ниже изображена нейросеть с двумя нейронами во входном слое, как и в предыдущем примере. Но теперь у сети имеется два нейрона и в выходном слое.



Оба выхода сети могут иметь погрешность, особенно в тех случаях, когда сеть еще не натренирована. Мы должны использовать полученные погрешности для настройки весов связей. Можно использовать метод, который мы получили в предыдущем разделе — больше изменяем те нейроны, которые сделали больший вклад в выход сети.

То, что сейчас у нас больше одного выходного нейрона по сути ни на что не влияет. Мы просто используем наш метод дважды: для первого и второго нейронов. Почему так просто? Потому что связи с конкретным выходным нейроном никак не влияют на остальные выходные нейроны. Их изменение повлияет только на конкретный выходной нейрон. На диаграмме выше изменение весов  $w_{1,2}$  и  $w_{2,2}$  не повлияет на результат  $o_1$ .

Погрешность первого нейрона выходного слоя мы обозначили за  $e_1$ . Погрешность равна разнице между желаемым выходом нейрона  $t_1$ , который мы имеем в обучающей выборке и полученным реальным результатом  $o_1$ .

$$e_1 = t_1 - o_1$$

Погрешность второго нейрона выходного слоя равна  $e_2$ .

На диаграмме выше погрешность  $e_1$  разделяется на веса  $w_{1,1}$  и  $w_{2,1}$  соответственно их вкладу в эту погрешность. Аналогично, погрешность  $e_2$  разделяется на веса  $w_{1,2}$  и  $w_{2,2}$ .

Теперь надо определить, какой вес оказал большее влияние на выход нейрона. Например, мы можем определить, какая часть ошибки  $e_1$  пойдет на исправление веса  $w_{1,1}$ :

$$\frac{w_{1,1}}{w_{1,1} + w_{2,1}}$$

А вот так находится часть  $e_1$ , которая пойдет на корректировку веса  $w_{2,1}$ :

$$\frac{w_{2,1}}{w_{1,1} + w_{2,1}}$$

Теперь разберемся, что означают два этих выражения выше. Изначально наша идея заключается в том, что мы хотим сильнее изменить связи с большим весом и слегка изменить связи с меньшим весом.

А как нам понять величину веса относительно всех остальных весов? Для этого мы должны сравнить какой-то конкретный вес (например  $w_{1,1}$ ) с абстрактной "общей" суммой всех весов, повлиявших на выход нейрона. На выход нейрона повлияли два веса:  $w_{1,1}$  и  $w_{2,1}$ . Мы складываем их и смотрим, какая часть от общего вклада приходится на  $w_{1,1}$  с помощью деления этого веса на полученную ранее общую сумму:

$$\frac{w_{1,1}}{w_{1,1} + w_{2,1}}$$

Пусть  $w_{1,1}$  в два раза больше, чем  $w_{2,1}$ :  $w_{1,1} = 6$  и  $w_{2,1} = 3$ . Тогда имеем  $6/(6 + 3) = 6/9 = 2/3$ , а значит  $2/3$  погрешности  $e_1$  пойдет на корректировку  $w_{1,1}$ , а  $1/3$  на корректировку  $w_{2,1}$ .

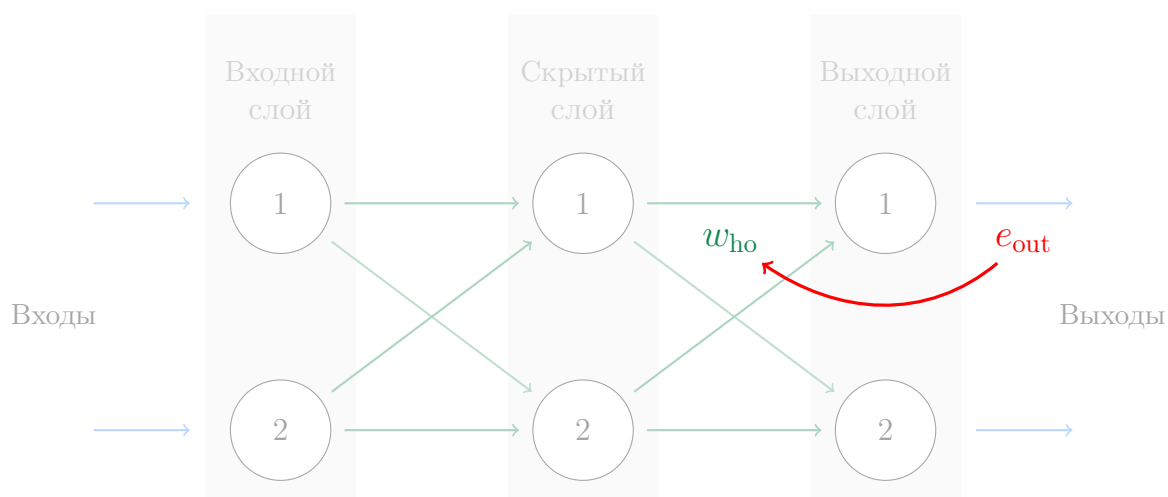
В случае, когда оба веса равны, то каждому достанется по половине погрешности. Пусть  $w_{1,1} = 4$  и  $w_{2,1} = 4$ . Тогда имеем  $4/(4 + 4) = 4/8 = 1/2$ , а значит на каждый вес пойдет  $1/2$  погрешности  $e_1$ .

Прежде чем мы двинемся дальше, давайте на секунду остановимся и посмотрим, чего мы достигли. Нам нужно что-то менять в нейросети для уменьшения получаемой погрешности. Мы решили, что будем менять веса связей между нейронами. Мы также нашли способ, как распределять полученную на выходном слое сети погрешность между весами связей. В этом разделе мы получили формулы для вычисления конкретной части погрешности для каждого веса. Отлично!

Но есть еще одна проблема. Сейчас мы знаем, что делать с весами связей слое, который находится прямо перед выходным слоем сети. А что если наша нейросеть имеет больше 2 слоев? Что делать с весами связей в слоях, которые находятся за предпоследним слоем сети?

## 1.12 Обратное распространение ошибки на множество слоев

На диаграмме ниже изображена простая трехслойная нейросеть с входным, скрытым и выходным слоями.



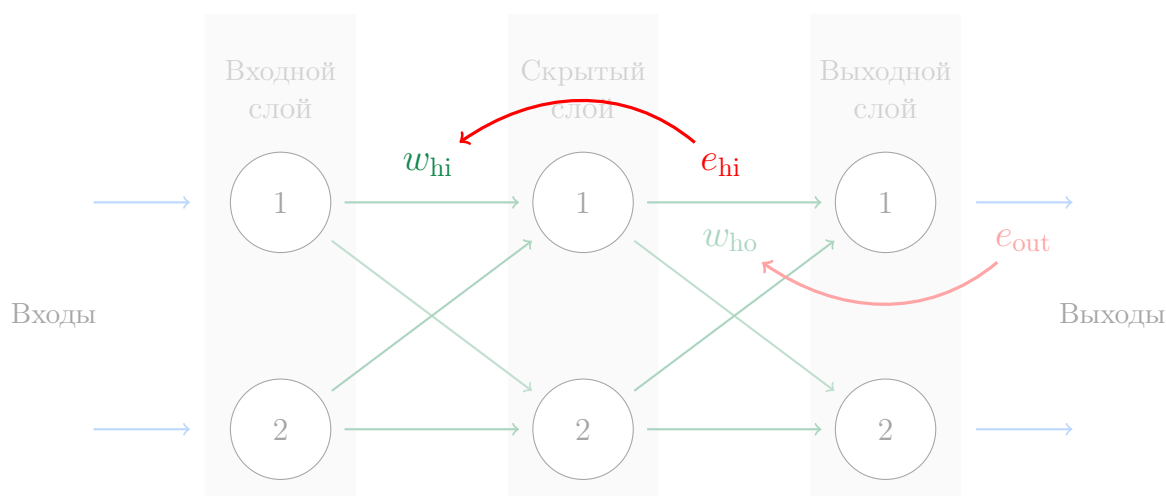
Сейчас мы наблюдаем процесс, который обсуждался в разделе выше. Мы используем погрешность выходного слоя для настройки весов связей, которые соединяют предпоследний слой с выходным слоем.

Для простоты обозначения были обобщены. Погрешности нейронов выходного слоя мы в целом назвали  $e_{out}$ , а все веса связей между скрытым и выходным слоем обозначили за  $w_{ho}$ .

Еще раз повторю, что для корректировки весов  $w_{ho}$  мы распределяем погрешность нейрона выходного слоя по всем весам в зависимости от их вклада в выход нейрона.

Как видно из диаграммы ниже, для корректировки весов связей между входным и скрытым слоем нам надо повторить ту же операцию еще раз. Мы берем погрешности нейронов скрытого слоя  $e_{hi}$  и распределяем их по весам связей между входным и скрытым слоем  $w_{ih}$ :



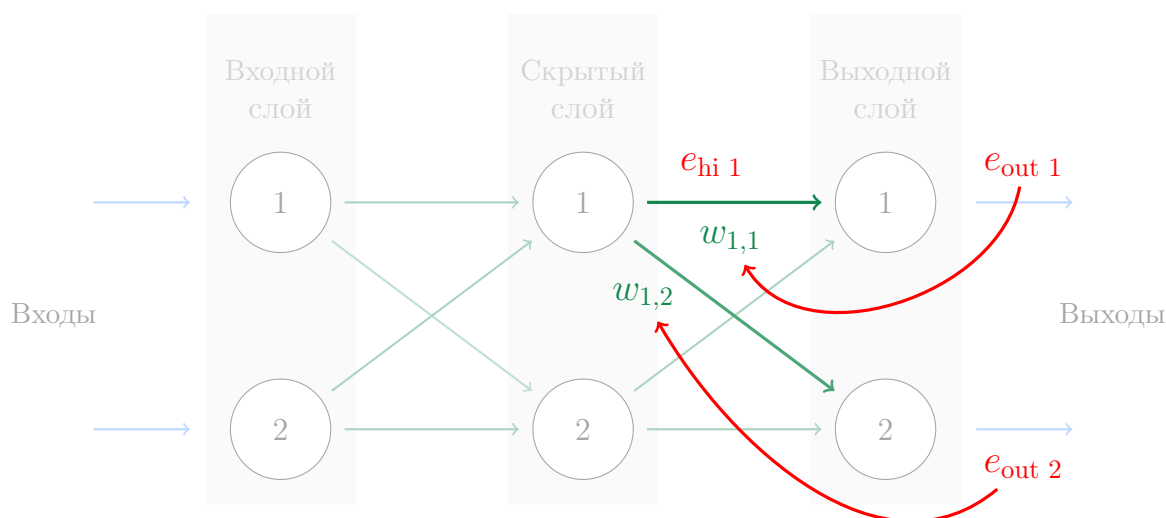


Если бы у нас было бы больше слоев, то мы бы и дальше повторяли этот процесс корректировки, распространяющийся от выходного ко входному слою. И снова вы видите, почему этот способ называется **методом обратного распространения ошибки**.

Для корректировки связей между предпоследним и выходным слоем мы использовали погрешность выходов сети  $e_{out}$ . А чему же равна погрешность выходов нейронов скрытых слоев  $e_{hi}$ ? Это отличный вопрос потому что сходу на этот вопрос ответить трудно. Когда сигнал распространяется по сети от входного к выходному слою мы точно знаем значения выходных нейронов скрытых слоев. Мы получали эти значения с помощью функции активации, у которой в качестве аргумента использовалась сумма взвешенных сигналов, поступивших на вход нейрона. Но как из выходного значения нейрона скрытого слоя получить его погрешность?

У нас нет никаких ожидаемых или заранее подготовленных правильных ответов для выходов нейронов скрытого слоя. У нас есть готовые правильные ответы только для выходов нейронов выходного слоя. Эти выходы мы сравниваем с заранее правильными ответами из обучающей выборки и получаем погрешность. Давайте вновь проанализируем диаграмму выше.

Мы видим, что из первого нейрона скрытого слоя выходят две связи с двумя нейронами выходного слоя. В предыдущем разделе мы научились распределять погрешность на веса связей. Поэтому мы можем получить две погрешности для обоих весов этих связей, сложить их и получить общую погрешность данного нейрона скрытого слоя. Наглядная демонстрация:



Уже из рисунка можно понять, что делать дальше. Но давайте все-таки еще раз пройдемся по всему алгоритму. Нам нужно получить погрешность выхода нейрона скрытого слоя для того, чтобы скорректировать веса связей между текущим и предыдущим слоями. Назовем эту погрешность  $e_{hi}$ . Но мы не можем получить значение погрешности напрямую. Погрешность равна разности между ожидаемым и полученным значениями, но проблема заключается в том, что у нас есть ожидаемые значения только для нейронов выходного слоя нейросети.

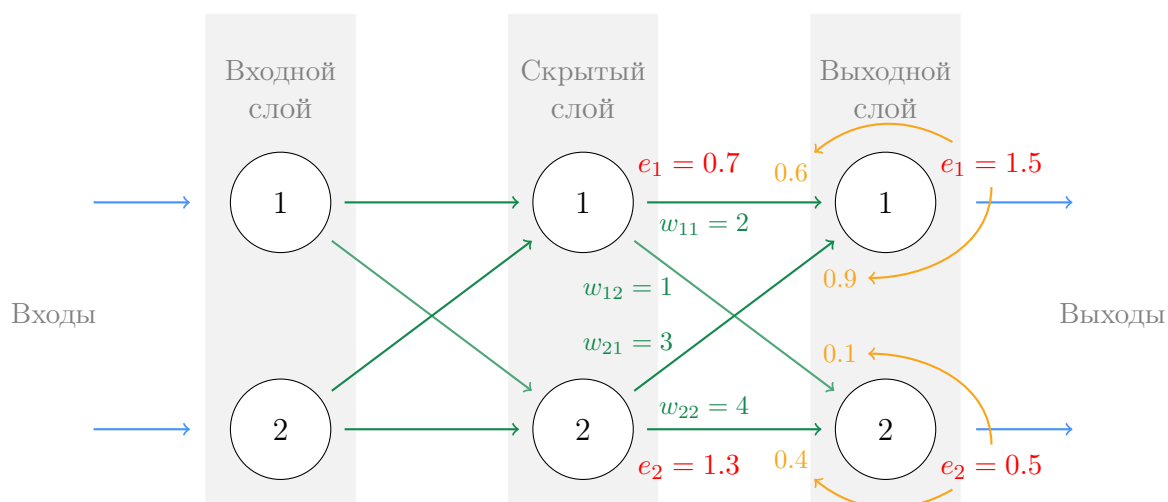
В невозможности прямого нахождения погрешности нейронов скрытого слоя и заключается основная сложность.

Но выход есть. Мы умеем распределять погрешность нейронов выходного слоя по весам связей. Значит на каждый вес связи идет часть погрешности. Поэтому мы складываем части погрешностей, которые относятся к весам связей, исходящих из данного скрытого нейрона. Полученная сумма и будем считать за погрешность выхода данного нейрона. На диаграмме выше часть погрешности  $e_{out\ 1}$  идет на вес  $w_{1,1}$ , а часть погрешности  $e_{out\ 2}$  идет на вес  $w_{1,2}$ . Оба этих веса относятся к связям, исходящим из первого нейрона скрытого слоя. А значит мы можем найти его погрешность:

$e_{hi\ 1}$  = сумма частей погрешностей для весов  $w_{1,1}$  и  $w_{1,2}$

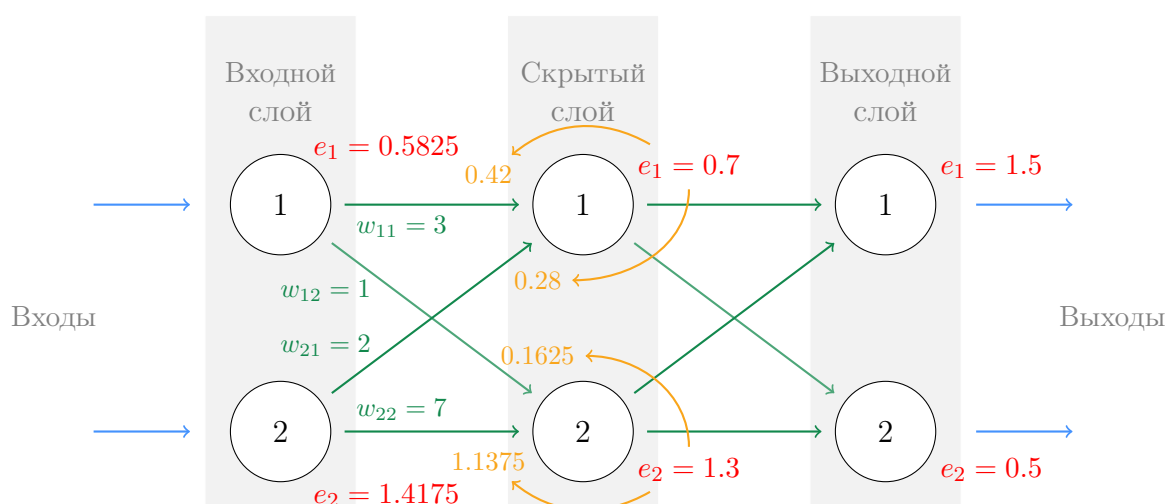
$$e_{hi\ 1} = \left( e_{out\ 1} \cdot \frac{w_{1,1}}{w_{1,1} + w_{2,1}} \right) + \left( e_{out\ 2} \cdot \frac{w_{1,2}}{w_{1,2} + w_{2,2}} \right)$$

Рассмотрим алгоритм на реальной трехслойной нейросети с двумя нейронами в каждом слое:



Давайте отследим обратное распространение одной ошибки/погрешности. Погрешность второго выходного нейрона равна 0.5 и она распределяется на два веса. На вес  $w_{12}$  идет погрешность 0.1, а на вес  $w_{22}$  идет погрешность 0.4. Далее у нас идет второй нейрон скрытого слоя. От него отходят две связи с весами  $w_{21}$  и  $w_{22}$ . На эти веса связей также распределяется погрешность как от  $e_1$ , так и от  $e_2$ . На вес  $w_{21}$  идет погрешность 0.9, а на вес  $w_{22}$  идет погрешность 0.4. Сумма этих погрешностей и дает нам погрешность выхода второго нейрона скрытого слоя:  $0.4 + 0.9 = 1.3$ .

Но это еще не конец. Теперь надо распределить погрешность нейронов выходного слоя на веса связей между входным и скрытым слоями. Проиллюстрируем этот процесс дальнейшего распространения ошибки:



### Ключевые моменты

- Обучение нейросетей заключается в корректировки весов связей. Корректировка зависит от **погрешности** — разности между ожидаемым ответом из обучающей выборки и реально полученным результатами.
- Погрешность для нейронов выходного слоя рассчитывается как разница между желаемым и полученным результатами.
- Однако, погрешность скрытых нейронов определить напрямую нельзя. Одно из популярных решений — сначала необходимо распределить известную погрешность на все веса связей, а затем сложить те части погрешностей, которые относятся к связям, исходящим из одного нейрона. Сумма этих частей погрешностей и будет являться общей погрешностью этого нейрона.

## 1.13 Обратное распространение ошибки и произведение матриц

А можно ли использовать матрицы для упрощения всех этих трудных вычислений? Они ведь помогли нам ранее, когда мы рассчитывали проход сигнала по сети от входного к выходному слою.

Если бы мы могли выразить обратное распространение ошибки через произведение матриц, то все наши вычисления разом бы уменьшились, а компьютер бы сделал всю грязную и повторяющуюся работу за нас.

Начинаем мы с самого конца нейросети — с матрицы погрешностей ее выходов. В примере выше у нас имеется две погрешности сети:  $e_1$  и  $e_2$ .

$$\mathbf{E}_{\text{out}} = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Теперь нам надо получить матрицу погрешностей выходов нейронов скрытого слоя. Звучит довольно жутко, поэтому давайте действовать по шагам. Из предыдущего раздела вы помните, что погрешность нейрона скрытого слоя высчитывается как сумма частей погрешностей весов связей, исходящих из этого нейрона.

Сначала рассматриваем первый нейрон скрытого слоя. Как было показано в предыдущем разделе, погрешность этого нейрона высчитывается так:

$$e_{\text{hid } 1} = \left( e_1 \cdot \frac{w_{11}}{w_{11} + w_{21}} \right) + \left( e_2 \cdot \frac{w_{12}}{w_{12} + w_{22}} \right)$$

Погрешность второго нейрона скрытого слоя высчитывается так:

$$e_{\text{hid } 2} = \left( e_1 \cdot \frac{w_{21}}{w_{11} + w_{21}} \right) + \left( e_2 \cdot \frac{w_{22}}{w_{12} + w_{22}} \right)$$

Получаем матрицу погрешностей скрытого слоя:

$$\mathbf{E}_{\text{hid}} = \begin{pmatrix} e_{\text{hid } 1} \\ e_{\text{hid } 2} \end{pmatrix} = \begin{pmatrix} e_1 \cdot \frac{w_{11}}{w_{11} + w_{21}} + e_2 \cdot \frac{w_{12}}{w_{12} + w_{22}} \\ e_1 \cdot \frac{w_{21}}{w_{11} + w_{21}} + e_2 \cdot \frac{w_{22}}{w_{12} + w_{22}} \end{pmatrix}$$

Многие из вас уже заметили произведение матриц:

$$\mathbf{E}_{\text{hid}} = \begin{pmatrix} \frac{w_{11}}{w_{11} + w_{21}} & \frac{w_{12}}{w_{12} + w_{22}} \\ \frac{w_{21}}{w_{11} + w_{21}} & \frac{w_{22}}{w_{12} + w_{22}} \end{pmatrix} \times \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Таким образом, мы смогли выразить вычисление матрицы погрешностей нейронов выходного слоя через произведение матриц. Однако выражение выше получилось немного сложнее, чем мне хотелось бы.

В формуле выше мы используем большую и неудобную матрицу, в которой делим каждый вес связи на сумму весов, приходящих в данный нейрон. Мы самостоятельно сконструировали эту матрицу.

Было бы очень удобно записать произведение из уже имеющихся матриц. А имеются у нас только матрицы весов связей, входных сигналов и погрешностей выходного слоя.

К сожалению, формулу выше никак нельзя записать в мега-простом виде, который мы получили для прохода сигнала в предыдущих разделах. Вся проблема заключается в жутких дробях, с которыми трудно что-то поделать.

Но нам очень нужно получить простую формулу для удобного и быстрого расчета матрицы погрешностей.

Пора немного пошалить!

Вновь обратим взор на формулу выше. Можно заметить, что самым главным является умножение погрешности выходного нейрона  $e_n$  на вес связи  $w_{ij}$ , которая к этому выходному нейрону подсоединена. Чем больше вес, тем большую погрешность получит нейрон скрытого слоя. Эту важную деталь мы сохраняем. А вот знаменатели дробей служат лишь нормализующим фактором. Если их убрать, то мы лишимся масштабирования ошибки на предыдущие слои, что не так уж и страшно. Таким образом, мы можем избавиться от знаменателей:

$$e_1 \cdot \frac{w_{11}}{w_{11} + w_{21}} \longrightarrow e_1 \cdot w_{11}$$

Запишем теперь формулу для получения матрицы погрешностей, но без знаменателей в левой матрице:

$$\mathbf{E}_{\text{hid}} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \times \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Так гораздо лучше!

В разделе по использованию матриц при расчетах прохода сигнала по сети мы использовали следующую матрицу весов:

$$\begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{pmatrix}$$

Сейчас, для расчета матрицы погрешностей мы используем такую матрицу:

$$\begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix}$$

Можно заметить, что во второй матрице элементы как бы отражены относительно диагонали матрицы, идущей от левого верхнего до правого нижнего края матрицы:  $w_{21}$  и  $w_{12}$  поменялись местами. Такая операция над матрицами существует и называется она **транспонированием** матрицы. Ранее мы использовали матрицу весов  $\mathbf{W}$ . Транспонированные матрицы имеют специальный значок справа сверху:  $^T$ . В расчете матрицы погрешностей мы используем транспонированную матрицу весов:  $\mathbf{W}^T$ .

Вот два примера для иллюстрации транспонирования матриц. Заметьте, что данную операцию можно выполнять даже для матриц, в которых число столбцов и строк различно.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}^T = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Мы получили то, что хотели — простую формулу для расчета матрицы погрешностей нейронов скрытого слоя:

$$\mathbf{E}_{\text{hid}} = \mathbf{W}^T \times \mathbf{E}_{\text{out}}$$

Это все конечно отлично, но правильно ли мы поступили, просто проигнорировав знаменатели? Да.

Дело в том, что сеть обучается не мгновенно, а проходит через множество шагов обучения. Таким образом она постепенно калибрует и корректирует собственные веса до приемлемого значения. Поэтому способ, с помощью которого мы находим погрешность не так важен. Я уже упоминал ранее, что мы могли бы разделить всю погрешность просто пополам между всеми весами, вносящими вклад в выход нейрона и даже тогда, по окончании обучения, сеть выдавала бы неплохие результаты.

Безусловно, игнорирование знаменателей повлияет на процесс обучения сети, но рано или поздно, через сотни и тысячи шагов обучения, она дойдет до правильных результатов что со знаменателями, что без них.

Убрав знаменатели, мы сохранили общую суть обратного распространения ошибки — чем больше вес, тем больше его надо скорректировать. Мы просто убрали смягчающий фактор.

### Ключевые моменты

- Обратное распространение ошибки может быть выражено через произведение матриц.
- Это позволяет нам удобно и эффективно производить расчеты вне зависимости от размеров нейросети.
- Получается что и прямой проход сигнала по нейросети и обратно распространение ошибки можно выразить через матрицы с помощью очень похожих формул.

Сделайте перерыв. Вы его заслужили. Следующие несколько разделов будут финальными и очень крутыми. Но их надо проходить на свежую голову.



## Глава 2

## The Second Chapter