Нейронные сети

Подробное и исчерпывающее объяснение на n десятков минут с матрицами, функциями, производными и схемами

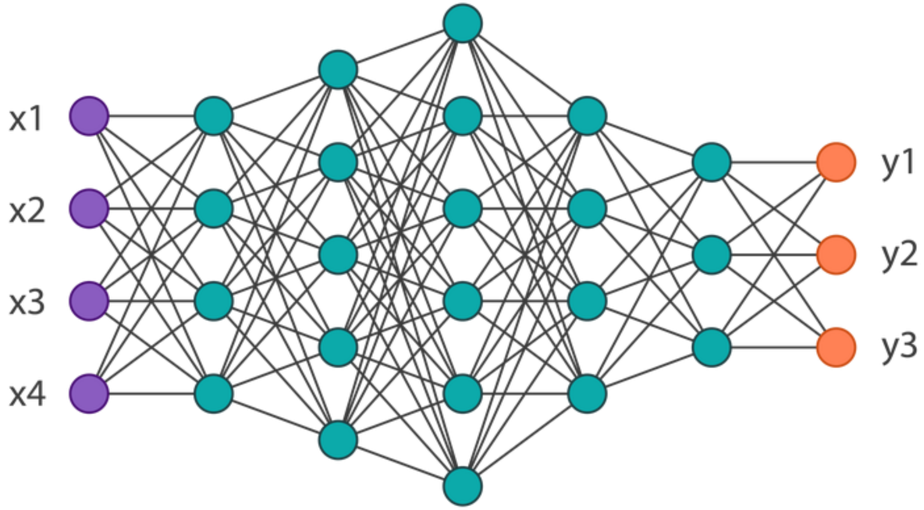
План презентации

1. Объяснение идеи нейросетей
2. Небольшое объяснение линейной алгебры
3. Объяснение математики нейросетей и нотации
4. Объяснение кода

P.S. Этот план предельно общий и суть отражает неглубоко, но достаточно, чтобы дать вам необходимые ожидания.

§ Что такое нейронные сети?

Нейронные сети – это связанные между собой слои нейронов.



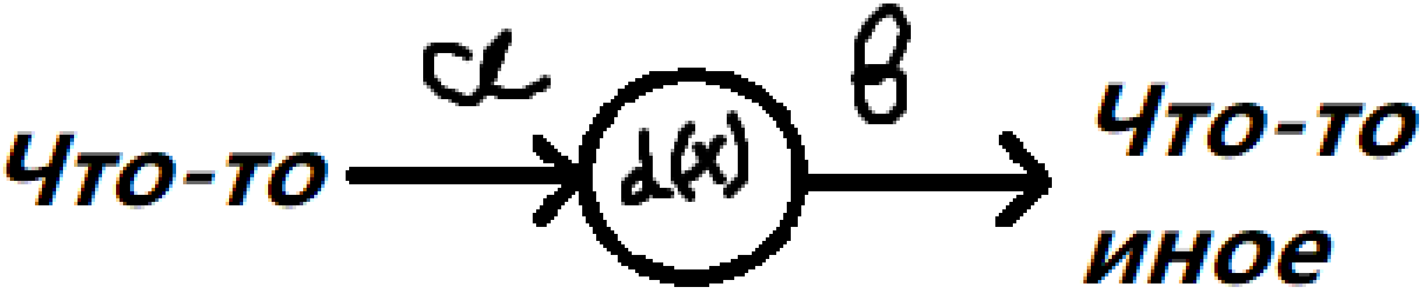
Нейросеть:

1. Принимает на вход набор чисел;
2. Обрабатывает этот набор;
3. Выдаёт на вход другой набор чисел, как результат.

В общем: нейронная сеть – это отображение $F(X)$, которое переводит набор $X = \{x_1, x_2, \dots x_n\}$ в набор $Y = \{y_1, y_2, \dots y_m\}$.
Обратите внимание, что длины наборов могут не совпадать!

Как работает нейрон

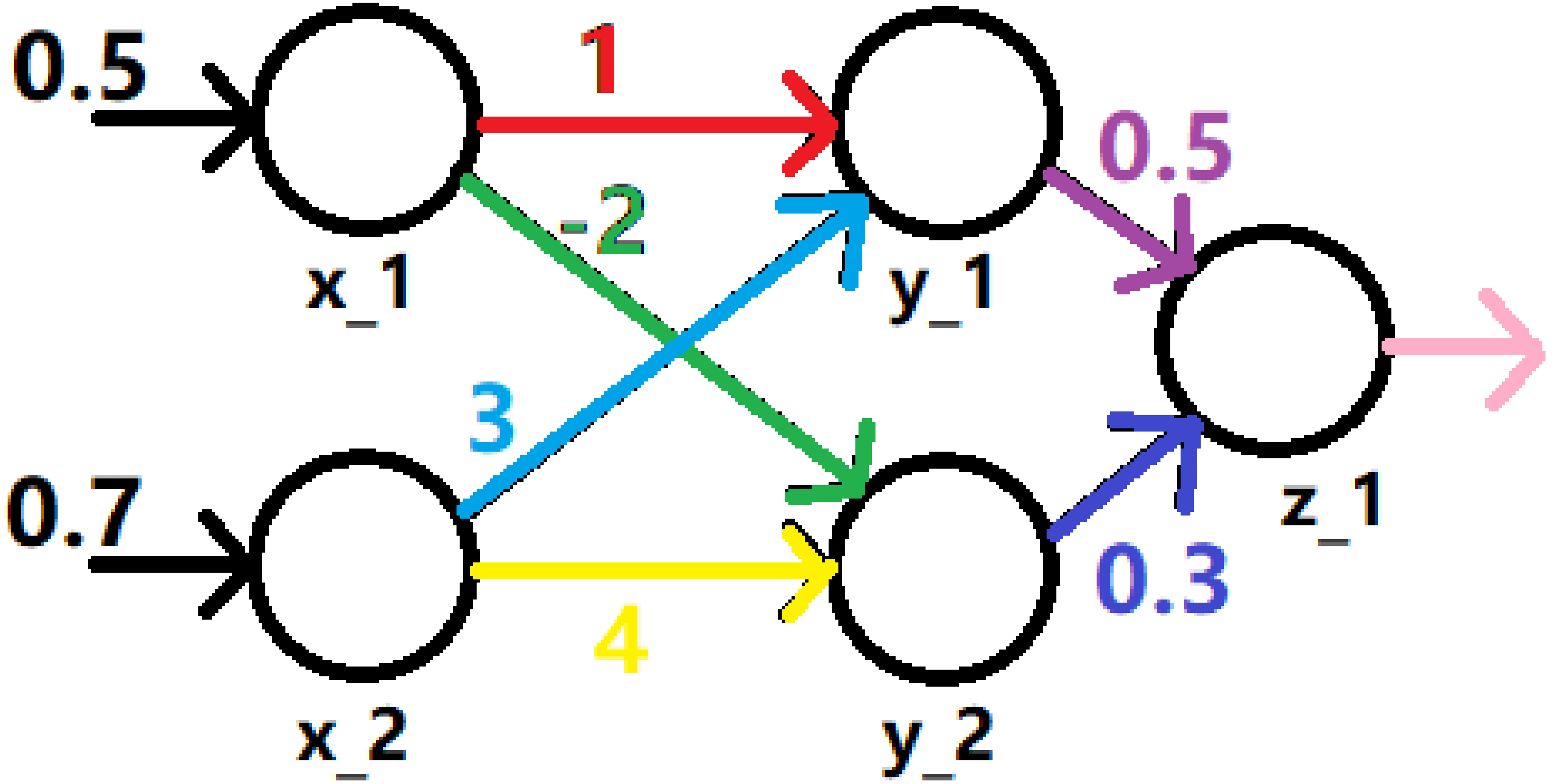
Нейрон суммирует все свои входы, обрабатывает сумму и шлёт результат дальше! Таких нейронов в сетях сотни.



Внутреннее преобразование можно как угодно задавать, а также как-либо управлять силой входного-выходного сигнала, домножая его на какие-нибудь числа (например, Что-то$\times\ a$, преобразовали, преобразовали$\times\ b$ $=$Что-то иное).

Пример нейросети

Рассмотрим простейший перцептрон. Чёрные 0.5 и 0.7 – здесь числа, которые идут на вход, а не веса!



Давайте просчитаем перцептрон!

Пока без преобразований в нейронах!

Для y_1:

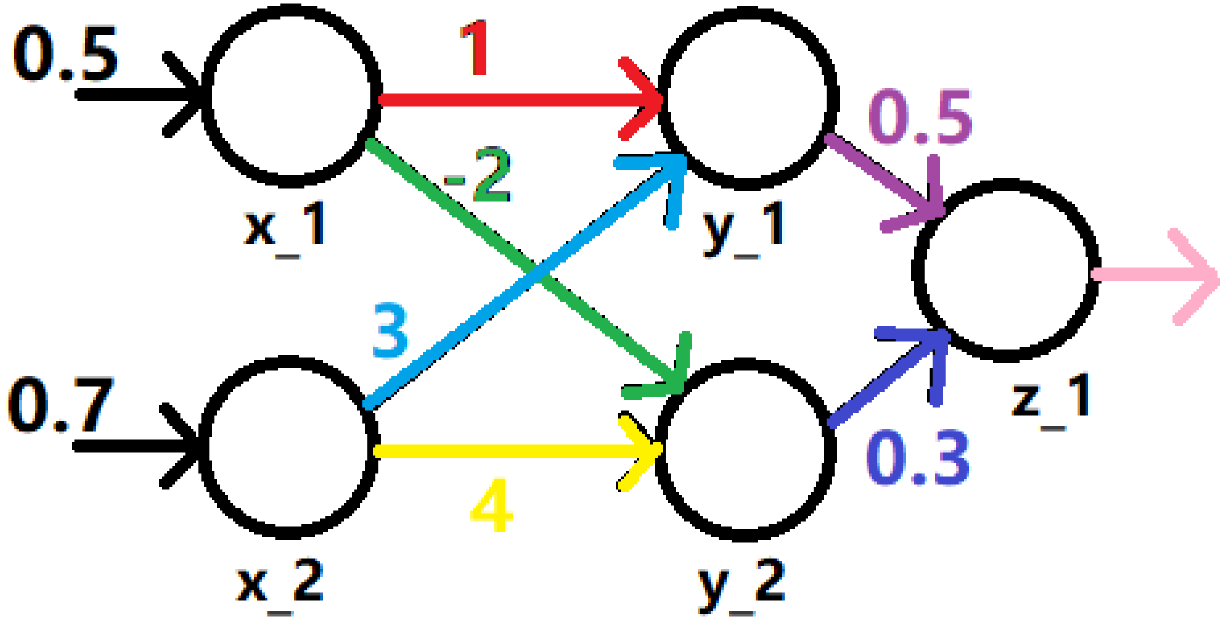
$$y_1 = 0.5 \times 1 + 0.7 \times 3$$

Для y_2:

$$y_2 = 0.5 \times (-2) + 0.7 \times 4$$

Для z_1:

$$z_1 = y_1 \times 0.5 + y_2 \times 0.3$$



Вручную так считать очень просто, но что если нейронов о-о-очень много?

Здесь нам поможет линейная алгебра!

Линейная алгебра: матрицы и векторы!

Векторы – это строки/столбцы с числами, для которых определены алгебраические операции.

Матрицы – это таблицы с числами, то есть стопки векторов, для которых определены алгебраические операции.

Матрицы и векторы позволяют компактно записать то, что происходит в нейросети!

ТАК КАК НА ВХОДЕ/ВЫХОДЕ КАЖДОГО СЛОЯ ИМЕЕМ ВЕКТОР, ТО НАМ НУЖНА ОПЕРАЦИЯ ДОМНОЖЕНИЯ МАТРИЦЫ НА ВЕКТОР!

Домножение матрицы на вектор выглядит так:

$$AB = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \cdots \\ b_{1n} \end{pmatrix} = \begin{pmatrix} a_{11} \times b_1 + & a_{12} \times b_2 + & \cdots + & a_{1n} \times b_n \\ a_{21} \times b_1 + & a_{22} \times b_2 + & \cdots + & a_{2n} \times b_n \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} \times b_1 + & a_{m2} \times b_2 + & \cdots + & a_{mn} \times b_n \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ \cdots \\ c_{1m} \end{pmatrix}$$

А вот вектор на матрицу домножать нельзя! По крайней мере, такая операция не определена и нам, к счастью, не потребуется.

P.S. На самом деле нечто подобное определено и мы с этим столкнёмся.

Сравните вычисления с операцией

Для y_1:

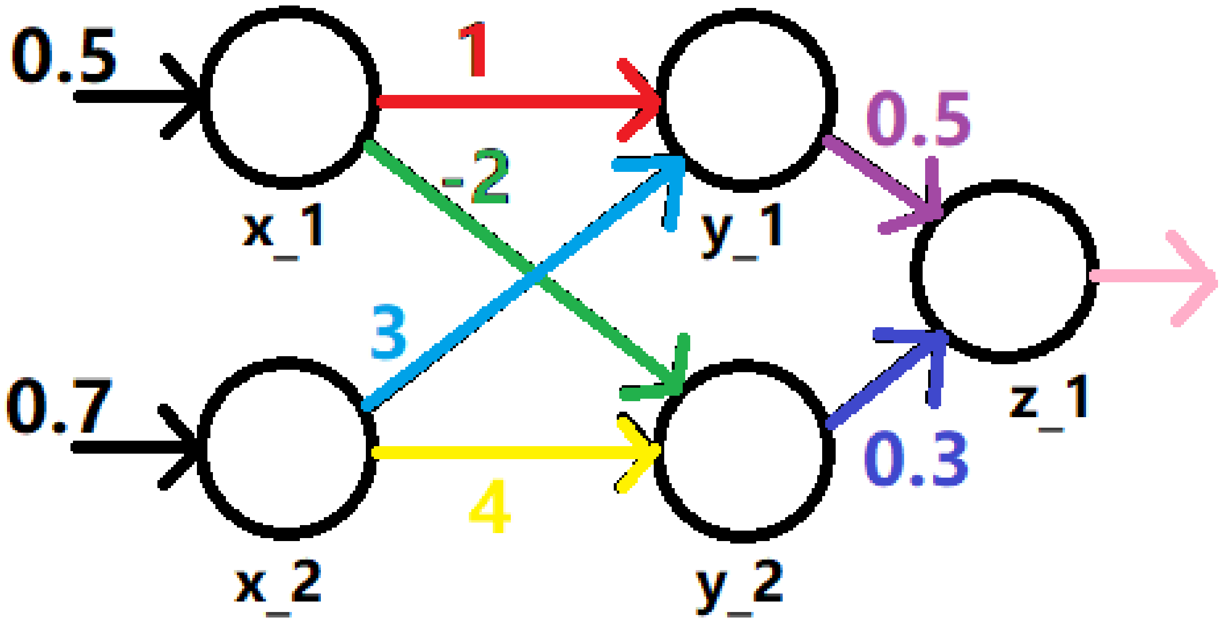
$$y_1 = 0.5 \times 1 + 0.7 \times 3$$

Для y_2:

$$y_2 = 0.5 \times (-2) + 0.7 \times 4$$

Для z_1:

$$z_1 = y_1 \times 0.5 + y_2 \times 0.3$$



$$AB = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \cdots \\ b_{1n} \end{pmatrix} = \begin{pmatrix} a_{11} \times b_1 + & a_{12} \times b_2 + & \cdots + & a_{1n} \times b_n \\ a_{21} \times b_1 + & a_{22} \times b_2 + & \cdots + & a_{2n} \times b_n \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} \times b_1 + & a_{m2} \times b_2 + & \cdots + & a_{mn} \times b_n \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ \cdots \\ c_{1m} \end{pmatrix}$$

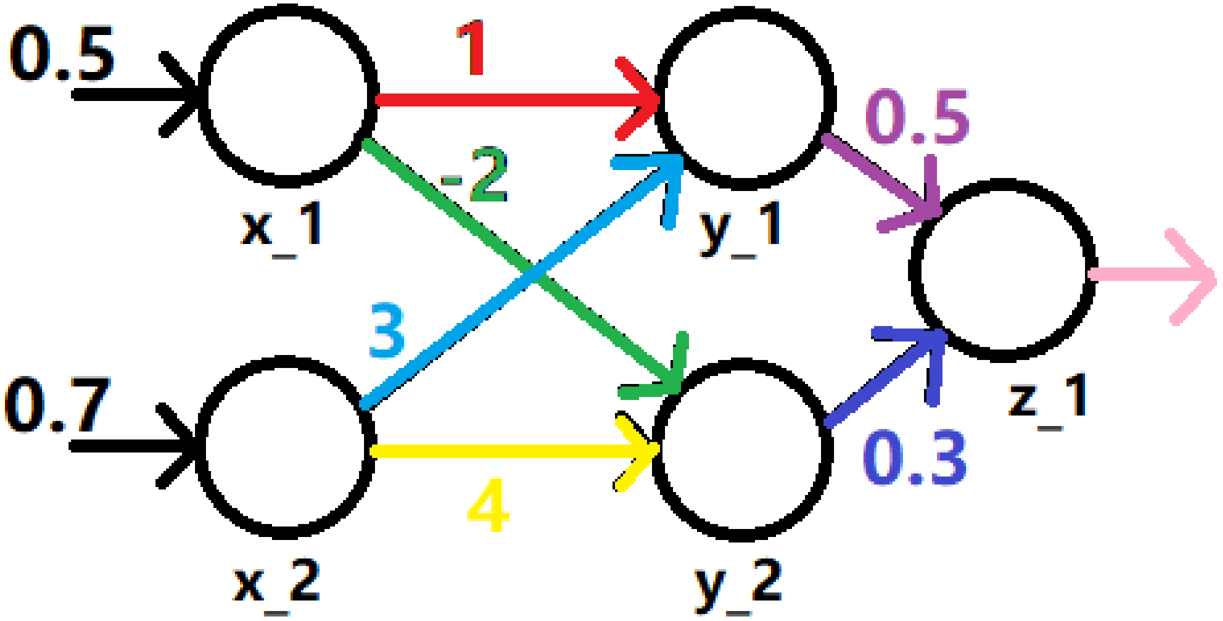
Матрицы в нейронных сетях

Матрица весов первого слоя:

$$W_1 = \begin{pmatrix} 1 & 3 \\ -2 & 4 \end{pmatrix}$$

Обозначение путей к нейронам:

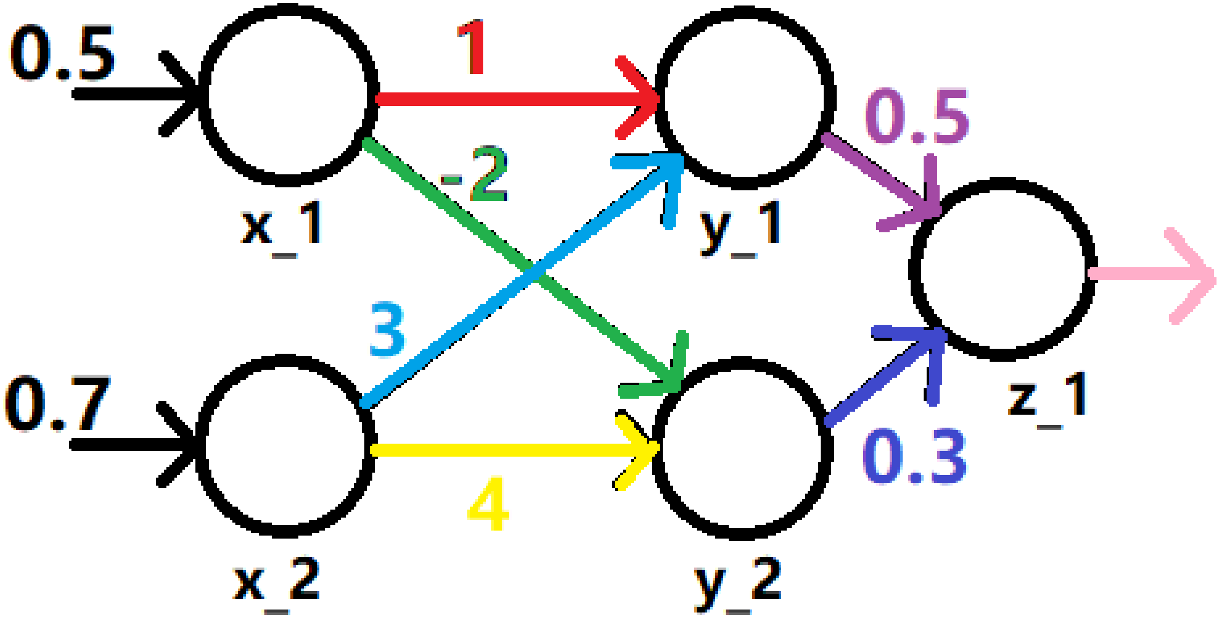
$$W_2 = \begin{pmatrix} x_1 \to y_1 & x_2 \to y_1 \\ x_1 \to y_2 & x_2 \to y_2 \end{pmatrix}$$



Есть закономерность: каждая строка содержит веса путей в нейрон следующего слоя с её номером!
ТО ЕСТЬ ЧИСЛА В ПЕРВОЙ СТРОКЕ – ВЕСА В ПЕРВЫЙ НЕЙРОН, ВО ВТОРОЙ – ВО ВТОРОЙ И ТАК ДАЛЕЕ!
Номера же столбцов указывают на номера исходящих нейронов!

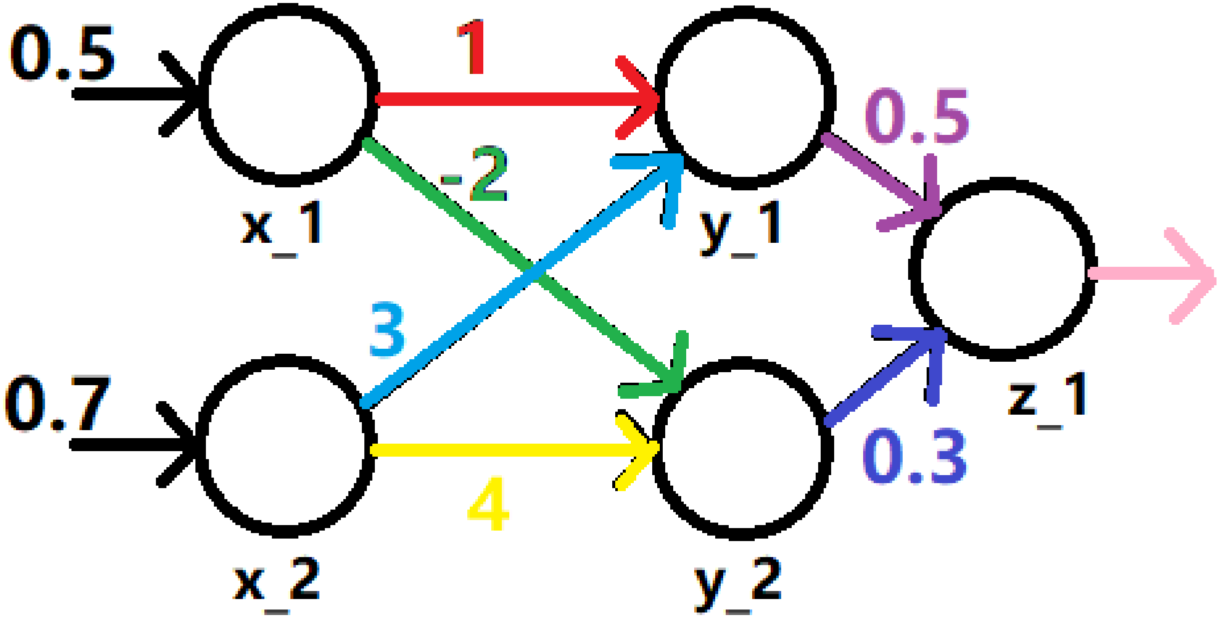
Упражнение на вычисление

Вычислите результат работы уже рассмотренного перцептрона.



Упражнение на вычисление

Вычислите результат работы уже рассмотренного перцептрона.



Вход во второй слой:

$$\mathrm{L}_2 = W_1 \times I = \begin{pmatrix} 1 & 3 \\ -2 & 4 \end{pmatrix} \times \begin{pmatrix} 0.5 \\ 0.7 \end{pmatrix}$$
$$= \begin{pmatrix} 0.5 \times 1 + 3 \times 0.7 \\ -2 \times 0.5 + 4 \times 0.7 \end{pmatrix} = \begin{pmatrix} 0.5 + 2.5 \\ -1 + 2.8 \end{pmatrix}$$
$$= \begin{pmatrix} 2.6 \\ 1.8 \end{pmatrix}$$

Результат нейросети:

$$O = W_2^T \times L_2 = \begin{pmatrix} 0.5 \\ 0.3 \end{pmatrix}^T \times \begin{pmatrix} 2.6 \\ 1.8 \end{pmatrix} = \begin{pmatrix} 0.5 & 0.3 \end{pmatrix} \times \begin{pmatrix} 2.6 \\ 1.8 \end{pmatrix} = 0.5 \times 2.6 + 0.3 \times 1.8$$
$$= 1.3 + 0.54 = 1.84$$

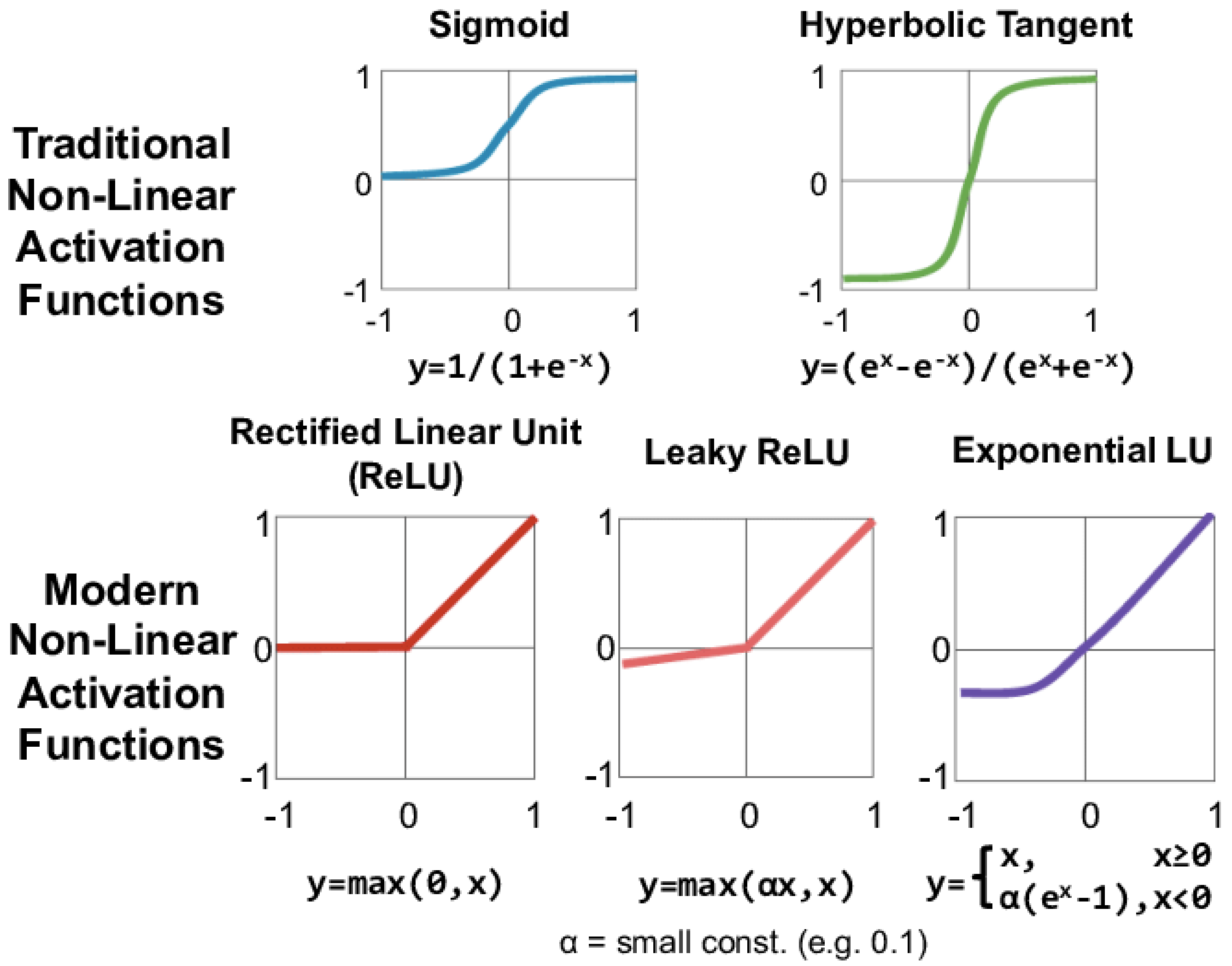
Функции активации в нейронах

Существует много функций активации нейронов. Эти функции преобразуют как-нибудь входное значение.

В текстах сказано, что *функции активации, обычно, имеют область значений на отрезке от нуля до единицы*, но такое ограничение можно и не соблюдать – математика не разрушится (одну из таких функций мы будем использовать, и уже использовали, просто не говорили об этом).

Хорошая функция – та, от которой легко брать производную! Почему? Потому что дальше мы будем заниматься алгоритмом оптимизации нейронной сети, то есть её обучением, а там, где задача оптимизации, всегда прячется дифференцирование!

Функции активации и их графики



Немного нотации

Перед тем, как перейдём к оптимизации сетей, вспомним нотацию из анализа.

Нам нужны частные производные, чтобы управляться с ростом функции.

Допустим, есть функция двух переменных: $f(x, y) = 2x + y^4$. Тогда частные производные:

$$\frac{\partial f}{\partial x} = 2 \text{ и } \frac{\partial f}{\partial y} = 4y^3$$

Градиентом будет просто вектор: $\mathrm{grad} f = \nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) = (2, 4y^3)$

Градиент $\mathrm{grad} f$, или же ∇f, – это вектор, который показывает направление наискорейшего роста функции в точке, а его длина равна скорости этого роста.

Найдите градиенты функций

$$f(x) = 2x^4$$

$$g(x, y) = e^x + \ln(y)$$

$$h(x, y, z) = e^x - \frac{1}{y^2} + 2^z$$

Найдите градиенты функций

$$f(x) = 2x^4 \to \nabla f = (8x^3)$$

$$g(x, y) = e^x + \ln(y) \to \nabla g = \left(e^x, \frac{1}{y}\right)$$

$$h(x, y, z) = e^x - \frac{1}{y^2} + 2^z \to \nabla h = \left(e^x, \frac{2}{y^3}, 2^z \ln(2)\right)$$

И найдём напоследок производную

Функция: $f(x) = x^{2x\ln(x)}$

И найдём напоследок производную

Функция: $f(x) = x^{2x\ln(x)} = e^{2x\ln(x)\ln(x)} = e^{2x(\ln(x))^2}$(т.к. $a^b = e^{b\ln(a)}$)
$u(x) = 2x(\ln(x))^2 \Longrightarrow f(x) = e^{u(x)}$

$$f'(x) = f(x)\,u'(x) = e^{u(x)}u'(x)$$

$$u'(x) = 2(\ln(x))^2 + 2x2\ln(x)\frac{1}{x}$$

$$u'(x) = 2(\ln(x))^2 + 4\ln(x)$$

$$u'(x) = 2\ln(x)\,(\ln(x) + 2)$$

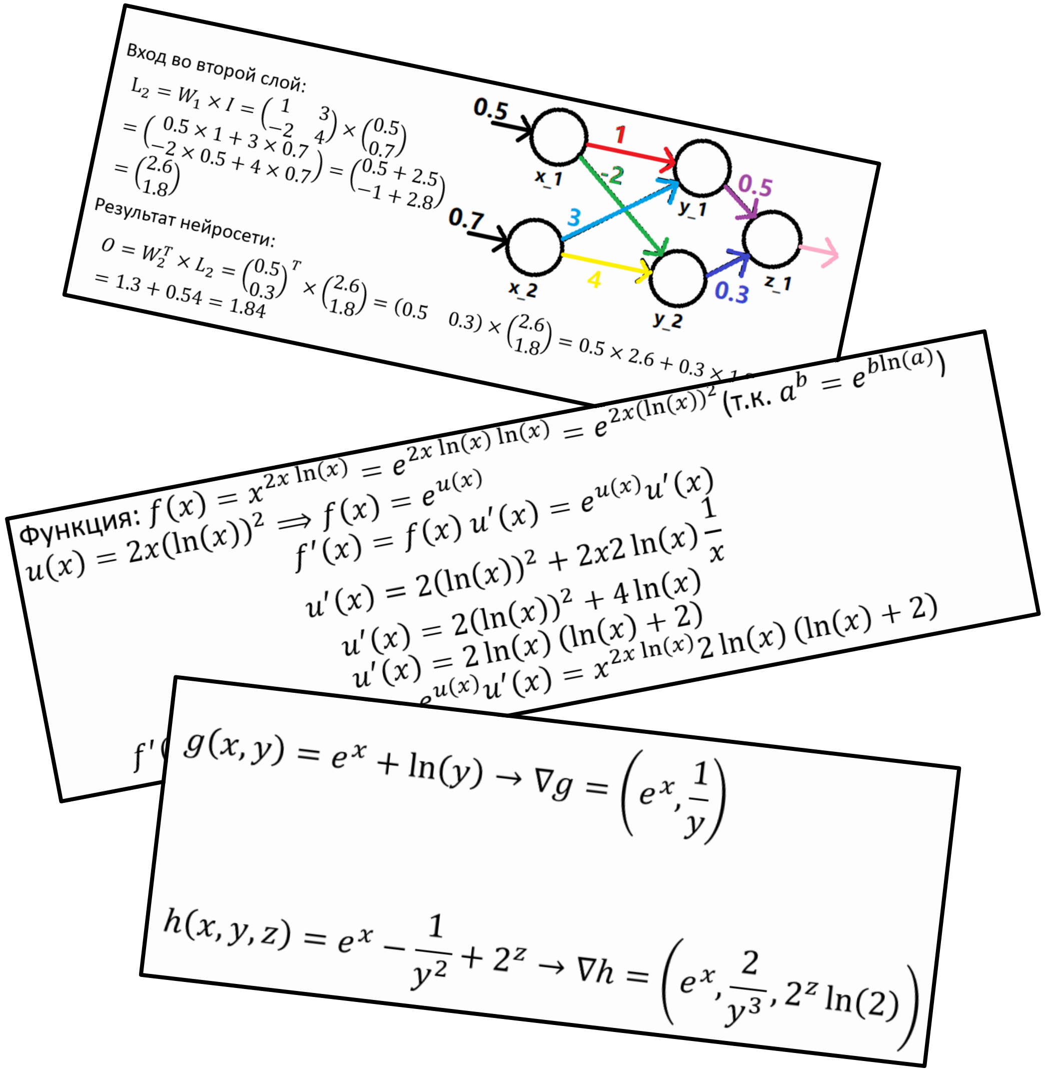
$$f'(x) = f(x)\,u'(x) = e^{u(x)}u'(x) = x^{2x\ln(x)}2\ln(x)\,(\ln(x) + 2)$$

Этот пример учит нас тому, что результатом взятия производной от сколь угодно большого числа применений функций к функциям будет произведение некоторых функций!

Чего мы сейчас достигли?

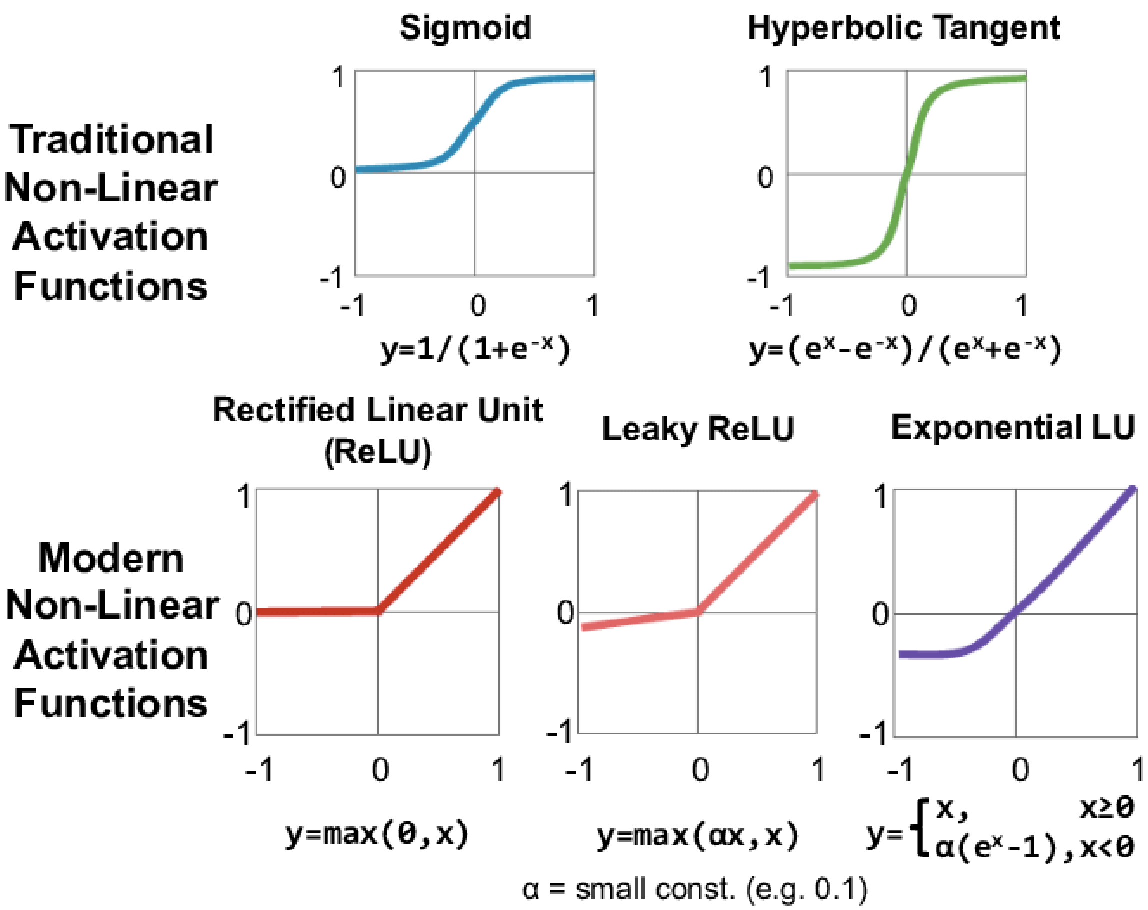
- Мы можем вычислять движения сигналов через матрицы.
- Мы знаем про функции активации и можем брать их производные.
- Мы можем находить градиенты функций.

Фактически, этого уже достаточно, чтобы работать с нейронными сетями!



Небольшая пауза на отдых и вопросы

Полученные результаты

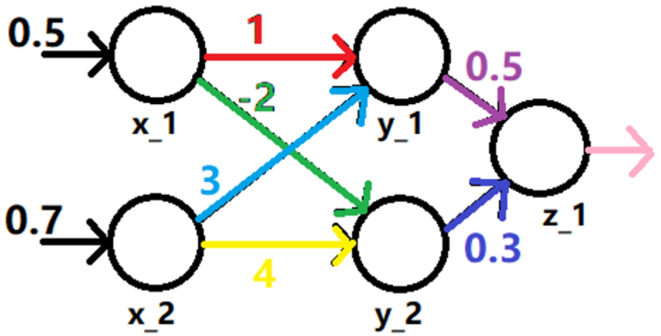


Вход во второй слой:

$$L_2 = W_1 \times I = \begin{pmatrix} 1 & 3 \\ -2 & 4 \end{pmatrix} \times \begin{pmatrix} 0.5 \\ 0.7 \end{pmatrix}$$
$$= \begin{pmatrix} 0.5 \times 1 + 3 \times 0.7 \\ -2 \times 0.5 + 4 \times 0.7 \end{pmatrix} = \begin{pmatrix} 0.5 + 2.5 \\ -1 + 2.8 \end{pmatrix}$$
$$= \begin{pmatrix} 2.6 \\ 1.8 \end{pmatrix}$$

Результат нейросети:

$$O = W_2^T \times L_2 = \begin{pmatrix} 0.5 \\ 0.3 \end{pmatrix}^T \times \begin{pmatrix} 2.6 \\ 1.8 \end{pmatrix} = (0.5 \quad 0.3) \times \begin{pmatrix} 2.6 \\ 1.8 \end{pmatrix} = 0.5 \times 2.6 + 0.3 \times 1.8$$
$$= 1.3 + 0.54 = 1.84$$



$$g(x,y) = e^x + \ln(y) \to \nabla g = \left(e^x, \frac{1}{y}\right)$$

$$h(x,y,z) = e^x - \frac{1}{y^2} + 2^z \to \nabla h = \left(e^x, \frac{2}{y^3}, 2^z \ln(2)\right)$$

§ То, чему пока нет нормальных объяснений

Нейронная сеть может быть оптимизирована различными способами (генетический алгоритм, подбор весов…). Общераспространённый способ – это метод градиентного спуска.

1. Для каждого нейрона высчитывается градиент функции ошибки (каждый нейрон полагается функцией).
2. Этот градиент домножается на скорость обучения (число порядка 0.1 – 0.001).
3. Полученное вычитается из весов этого нейрона.
4. Так делается от конечных нейронов к начальным.

Коэффициент скорости обучения гарантирует нам, что мы не проскочим минимум функции!

Смысл градиентного спуска

Градиентный спуск – это метод, позволяющий так скорректировать нейросеть, чтобы функция ошибки была минимальной для всех входных данных.

Почему это работает?

1. Градиент – вектор, указывающий направление скорейшего роста функции ошибки (чем больше эта функция, тем хуже).
2. Двигаясь в противоположную от градиента сторону, движемся в строну наименьшего роста функции ошибки.
3. Домножение на 0.1 – 0.001 перед вычитанием позволяет не проскочить минимум функции ошибки.

Сложности градиентного спуска

- Иногда не совсем очевидно, как выполнять градиентный спуск.
- Может оказаться так, что выбранная скорость обучения (0.1 – 0.001, на которое домножаем) приводит к тому, что мы перескакиваем минимум;
- При малых скоростях обучения вычисления могут проводиться очень долго;
- Для больших сетей (тысячи и миллионы нейронов) могут потребоваться огромные вычислительные мощности.

Используемая функция активации

Сначала определимся с функцией активации. Это будет ReLU.

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

Её производная равна:

$$ReLU'(x) = 0, x \leq 0$$
$$ReLU'(x) = 1, x > 0$$

Это в разы упрощает процесс вычислений, так как в любой точке, большей нуля, производная будет равна единице!

Мы уже использовали ReLU при вычислении перцептрона, так как просто передавали дальше получившиеся суммы.

Измерение ошибки

Имея функцию активации, матрицы весов и эталонные наборы входных и выходных векторов, по которым будет обучаться сеть, нужно определиться с функцией измерения ошибки.

Пусть она примет следующий вид:

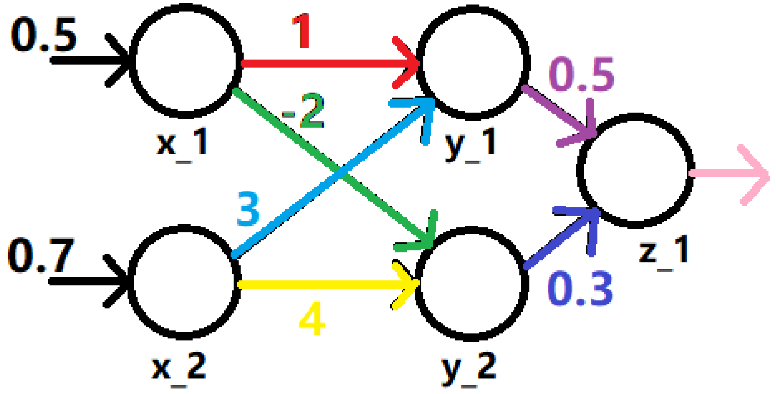
$$E = \frac{1}{2}(O - t)^2$$

где O – результат работы сети, а t – ожидаемое значение.

Вообще функция ошибки может быть любой метрикой, то есть формулой расчёта расстояния между тем, что имеем, и тем, что ожидаем.

Пример на перцептроне

Для удобства будем рассматривать наш перцептрон в качестве примера.



$$I = \begin{pmatrix} 0.5 \\ 0.7 \end{pmatrix}$$

$$W_1 = \begin{pmatrix} 1 & 3 \\ -2 & 4 \end{pmatrix}$$

$$W_2^T = \begin{pmatrix} 0.5 \\ 0.3 \end{pmatrix}^T$$

Активация: $\mathrm{ReLU(x)}$

Ожидаемое: $t = 1$

Ошибка $E = \frac{1}{2}(O - t)^2 = \frac{1}{2}(1.84 - 1)^2 = 0.3528$

Скорость обучения: $\eta = 0.1$

Градиент после выходного слоя

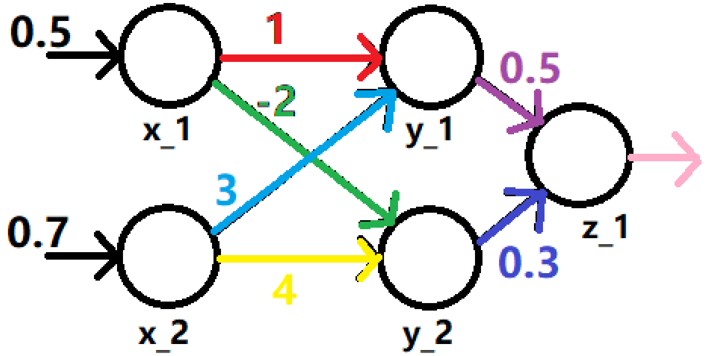
Градиент по выходному:

$$\frac{\partial E}{\partial O} = O - t = 1.84 - 1 = 0.84$$

$$\frac{\partial O}{\partial W_2[i]} = L_2^{(out)}[i]$$

$$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial O} \times \frac{\partial O}{\partial W_2} = 0.84 \times L_2^{(out)}[i]$$

$$\frac{\partial E}{\partial W_2} = 0.84 \times \frac{\partial O}{\partial W_2} = 0.84 \times \begin{pmatrix} 2.6 \\ 1.8 \end{pmatrix} = \begin{pmatrix} 0.84 \times 2.6 \\ 0.84 \times 1.8 \end{pmatrix} = \begin{pmatrix} 2.184 \\ 1.512 \end{pmatrix}$$



$$I = \begin{pmatrix} 0.5 \\ 0.7 \end{pmatrix}$$

$$W_1 = \begin{pmatrix} 1 & 3 \\ -2 & 4 \end{pmatrix}^T$$

$$W_2^T = \begin{pmatrix} 0.5 \\ 0.3 \end{pmatrix}$$

$$\mathrm{ReLU(x)}$$

$$E = \frac{1}{2}(O - t)^2$$

$$t = 1$$

$$\eta = 0.1$$

Градиент после скрытого слоя

Градиент по скрытому:

$$\delta_2 = \frac{\partial O}{\partial L_2^{(out)}} = \frac{\partial E}{\partial O} \times W_2$$

Для i-того нейрона $L_2^{(out)}[i]$ (после ReLU):

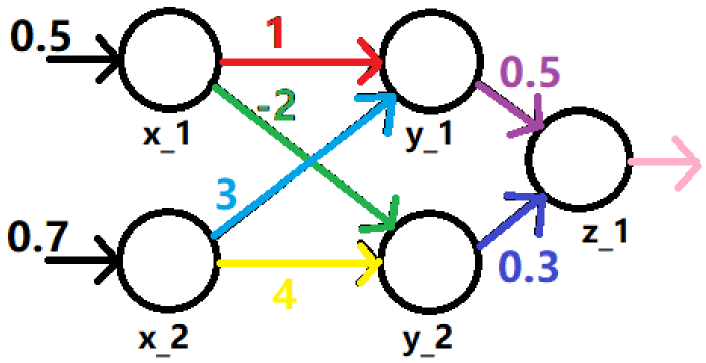
$$\delta_2[i] = (O - t) \times W_2[i]$$

$$\delta_2 = 0.84 \times \begin{pmatrix} 0.5 \\ 0.3 \end{pmatrix} = \begin{pmatrix} 0.42 \\ 0.252 \end{pmatrix}$$

Для i-того нейрона $L_2^{(in)}[i]$ (до ReLU):

$$\frac{\partial L_2^{(out)}[i]}{\partial L_2^{(in)}[i]} = 1, (L_2^{(in)}[i] > 0)$$

$$\delta_2^{(in)} = \delta_2 \odot \frac{\partial L_2^{(out)}[i]}{\partial L_2^{(in)}[i]} = \delta_2 \odot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.42 \\ 0.252 \end{pmatrix} \ (\forall i : \delta_2[i] > 0)$$



$$I = \begin{pmatrix} 0.5 \\ 0.7 \end{pmatrix}$$

$$W_1 = \begin{pmatrix} 1 & 3 \\ -2 & 4 \end{pmatrix}^T$$

$$W_2^T = \begin{pmatrix} 0.5 \\ 0.3 \end{pmatrix}$$

$$\mathrm{ReLU(x)}$$

$$E = \frac{1}{2}(O - t)^2$$

$$t = 1$$

$$\eta = 0.1$$

Градиент после входного слоя

Градиент по W_1:

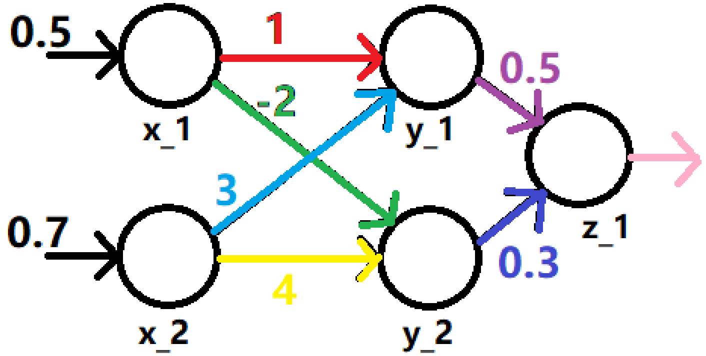
$$\frac{\partial L_2^{(in)}}{\partial W_1} = I^T$$

$$\frac{\partial E}{\partial W_1} = \delta_2^{(in)} \times I^T$$

Для i-того нейрона $L_2^{(out)}[i]$ (после ReLU):

$$\frac{\partial E}{\partial W_1} = \begin{pmatrix} 0.42 \\ 0.252 \end{pmatrix} \times \begin{pmatrix} 0.5 & 0.7 \end{pmatrix} = \begin{pmatrix} 0.42 \times 0.5 & 0.42 \times 0.7 \\ 0.252 \times 0.5 & 0.252 \times 0.7 \end{pmatrix}$$

$$= \begin{pmatrix} 0.21 & 0.294 \\ 0.126 & 0.1764 \end{pmatrix}$$



$$I = \begin{pmatrix} 0.5 \\ 0.7 \end{pmatrix}$$

$$W_1 = \begin{pmatrix} 1 & 3 \\ -2 & 4 \end{pmatrix}$$

$$W_2^T = \begin{pmatrix} 0.5 \\ 0.3 \end{pmatrix}^T$$

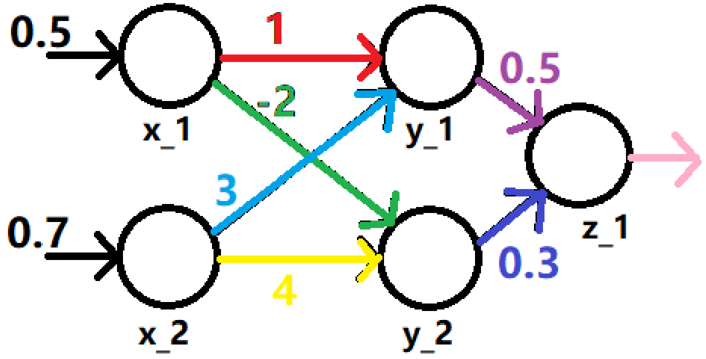
$$\mathrm{ReLU(x)}$$

$$E = \frac{1}{2}(O - t)^2$$

$$t = 1$$

$$\eta = 0.1$$

Обновление весов



Обновление W_2:

$$W_2^{(new)} = W_2 - \eta \times \frac{\partial E}{\partial W_2} = \begin{pmatrix}0.5\\0.3\end{pmatrix} - 0.1 \times \begin{pmatrix}2.184\\1.512\end{pmatrix}$$

$$= \begin{pmatrix}0.5 - 0.2184\\0.3 - 0.1512\end{pmatrix} = \begin{pmatrix}0.2816\\0.1488\end{pmatrix}$$

Обновление W_1:

$$W_1^{(new)} = W_1 - \eta \times \frac{\partial E}{\partial W_1} = \begin{pmatrix}1 & 3\\-2 & 4\end{pmatrix} - 0.1 \times \begin{pmatrix}0.21 & 0.294\\0.126 & 0.1764\end{pmatrix} =$$

$$\begin{pmatrix}1 - 0.021 & 3 - 0.0294\\-2 - 0.0126 & 4 - 0.01764\end{pmatrix} = \begin{pmatrix}0.979 & 2.9706\\-2.0126 & 3.98236\end{pmatrix}$$

В итоге имеем новые веса:

$$W_2^{(new)} = \begin{pmatrix}0.2816\\0.1488\end{pmatrix}$$

$$W_1^{(new)} = \begin{pmatrix}0.979 & 2.9706\\-2.0126 & 3.98236\end{pmatrix}$$

$$I = \begin{pmatrix}0.5\\0.7\end{pmatrix}$$

$$W_1 = \begin{pmatrix}1 & 3\\-2 & 4\end{pmatrix}$$

$$W_2^T = \begin{pmatrix}0.5\\0.3\end{pmatrix}^T$$

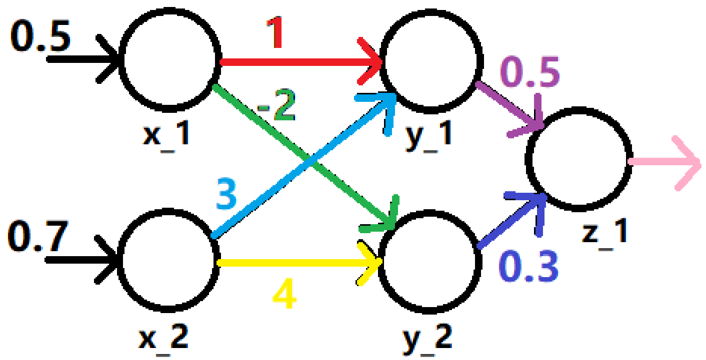
$$\mathrm{ReLU(x)}$$

$$E = \frac{1}{2}(O - t)^2$$

$$t = 1$$

$$\eta = 0.1$$

Пересчитаем сеть



$$I = \begin{pmatrix} 0.5 \\ 0.7 \end{pmatrix}$$

$$W_1^{(new)} = \begin{pmatrix} 0.979 & 2.9706 \\ -2.0126 & 3.98236 \end{pmatrix}$$

$$W_2^{(new)} = \begin{pmatrix} 0.2816 \\ 0.1488 \end{pmatrix}$$

$$\mathrm{ReLU(x)}$$

$$E = \frac{1}{2}(O - t)^2$$

$$t = 1$$

$$\eta = 0.1$$

Пересчитаем сеть

Вход во второй слой:

$$L_2 = W_1^{new} \times I = \begin{pmatrix} 0.979 & 2.9706 \\ -2.0126 & 3.98236 \end{pmatrix} \times \begin{pmatrix} 0.5 \\ 0.7 \end{pmatrix} = \begin{pmatrix} 2.5689 \\ 1.7813 \end{pmatrix}$$

Результат нейросети:

$$O = W_2^T \times L_2 = \begin{pmatrix} 0.5 \\ 0.3 \end{pmatrix}^T \times \begin{pmatrix} 2.5689 \\ 1.7813 \end{pmatrix} = \begin{pmatrix} 0.5 & 0.3 \end{pmatrix} \times \begin{pmatrix} 2.5689 \\ 1.7813 \end{pmatrix}$$

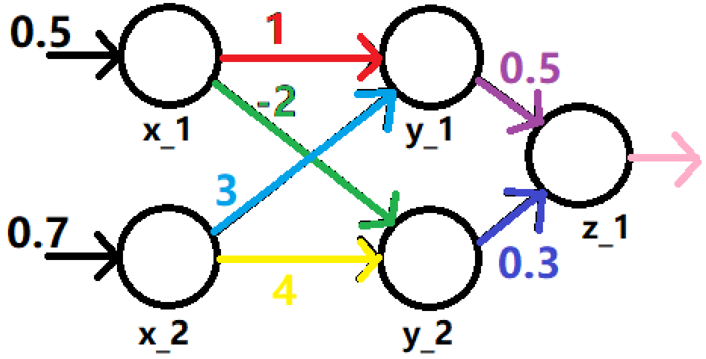
$= 1.8188$

Старая ошибка: $E_{old} = \frac{1}{2}(1.84 - 1)^2 = 0.3528$

Новая ошибка: $E_{new} = \frac{1}{2}(1.8188 - 1)^2 \approx 0.33521672$

Ошибка сократилась уже во втором знаке после запятой!

Если каждую эпоху обучения будет убывать 0.02… ошибки, то мы придём к оптимальному результату, примерно, через 17 с половиной эпох обучения (но это не точно!).



$$I = \begin{pmatrix} 0.5 \\ 0.7 \end{pmatrix}$$

$$W_1^{(new)} = \begin{pmatrix} 0.979 & 2.9706 \\ -2.0126 & 3.98236 \end{pmatrix}$$

$$W_2^{(new)} = \begin{pmatrix} 0.2816 \\ 0.1488 \end{pmatrix}$$

$$\mathrm{ReLU(x)}$$

$$E = \frac{1}{2}(O - t)^2$$

$$t = 1$$

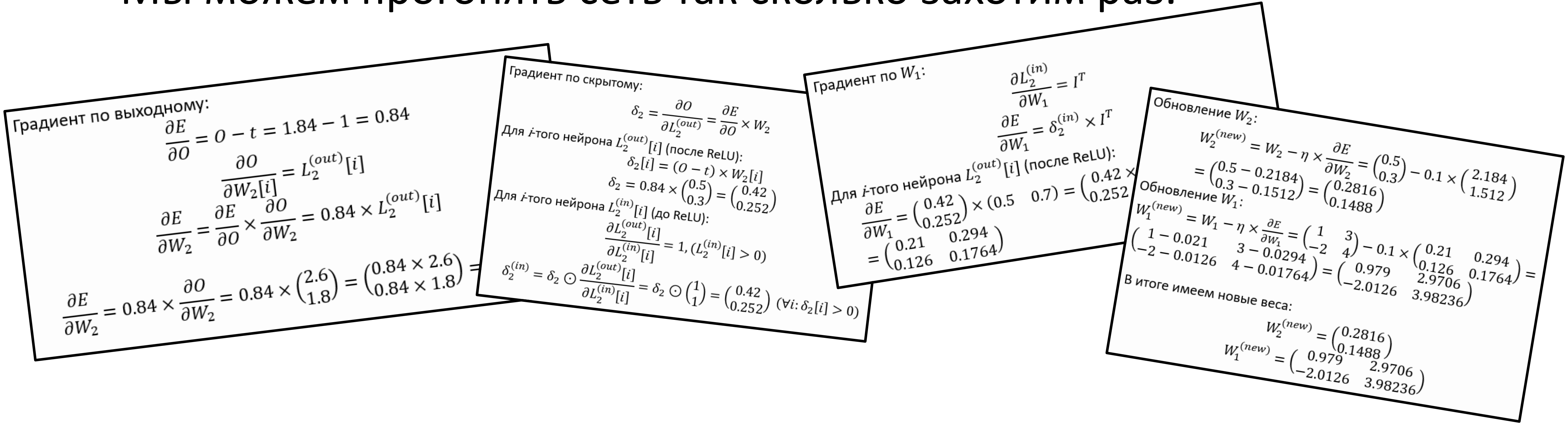
$$\eta = 0.1$$

Некоторые замечания

- На слоях нейронов могут быть разные функции активации – тогда брать производные станет сложнее, однако такое вполне возможно.
- Можно применять разные метрики для вычисления ошибок, что приведёт к разным производным и, как следствие, разным градиентам и другим скоростям обучения.
- Можно прогонять сеть сразу через несколько обучающих примеров (у нас был только один), после чего вычислять среднюю ошибку и использовать её для коррекции весов.

Что теперь у нас есть?

- Мы можем проводить оценку точности работы сети;
- Мы можем вычислять градиент функции ошибки для нейронов;
- Мы можем обновлять веса;
- Мы можем прогонять сеть так сколько захотим раз.



Небольшая пауза на отдых и вопросы

Полученные результаты

Градиент по выходному:

$$\frac{\partial E}{\partial O} = O - t = 1.84 - 1 = 0.84$$

$$\frac{\partial O}{\partial W_2[i]} = L_2^{(out)}[i]$$

$$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial O} \times \frac{\partial O}{\partial W_2} = 0.84 \times L_2^{(out)}[i]$$

$$\frac{\partial E}{\partial W_2} = 0.84 \times \frac{\partial O}{\partial W_2} = 0.84 \times \begin{pmatrix} 2.6 \\ 1.8 \end{pmatrix} = \begin{pmatrix} 0.84 \times 2.6 \\ 0.84 \times 1.8 \end{pmatrix} = \begin{pmatrix} 2.184 \\ 1.512 \end{pmatrix}$$

Градиент по скрытому:

$$\delta_2 = \frac{\partial O}{\partial L_2^{(out)}} = \frac{\partial E}{\partial O} \times W_2$$

Для i-того нейрона $L_2^{(out)}[i]$ (после ReLU):

$$\delta_2[i] = (O - t) \times W_2[i]$$

$$\delta_2 = 0.84 \times \begin{pmatrix} 0.5 \\ 0.3 \end{pmatrix} = \begin{pmatrix} 0.42 \\ 0.252 \end{pmatrix}$$

Для i-того нейрона $L_2^{(in)}[i]$ (до ReLU):

$$\frac{\partial L_2^{(out)}[i]}{\partial L_2^{(in)}[i]} = 1, (L_2^{(in)}[i] > 0)$$

$$\delta_2^{(in)} = \delta_2 \odot \frac{\partial L_2^{(out)}[i]}{\partial L_2^{(in)}[i]} = \delta_2 \odot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.42 \\ 0.252 \end{pmatrix} \ (\forall i: \delta_2[i] > 0)$$

Градиент по W_1:

$$\frac{\partial L_2^{(in)}}{\partial W_1} = I^T$$

$$\frac{\partial E}{\partial W_1} = \delta_2^{(in)} \times I^T$$

Для i-того нейрона $L_2^{(out)}[i]$ (после ReLU):

$$\frac{\partial E}{\partial W_1} = \begin{pmatrix} 0.42 \\ 0.252 \end{pmatrix} \times (0.5 \quad 0.7) = \begin{pmatrix} 0.42 \times 0.5 & 0.42 \times 0.7 \\ 0.252 \times 0.5 & 0.252 \times 0.7 \end{pmatrix}$$

$$= \begin{pmatrix} 0.21 & 0.294 \\ 0.126 & 0.1764 \end{pmatrix}$$

Обновление W_2:

$$W_2^{(new)} = W_2 - \eta \times \frac{\partial E}{\partial W_2} = \begin{pmatrix} 0.5 \\ 0.3 \end{pmatrix} - 0.1 \times \begin{pmatrix} 2.184 \\ 1.512 \end{pmatrix}$$

$$= \begin{pmatrix} 0.5 - 0.2184 \\ 0.3 - 0.1512 \end{pmatrix} = \begin{pmatrix} 0.2816 \\ 0.1488 \end{pmatrix}$$

Обновление W_1:

$$W_1^{(new)} = W_1 - \eta \times \frac{\partial E}{\partial W_1} = \begin{pmatrix} 1 & 3 \\ -2 & 4 \end{pmatrix} - 0.1 \times \begin{pmatrix} 0.21 & 0.294 \\ 0.126 & 0.1764 \end{pmatrix} =$$

$$\begin{pmatrix} 1 - 0.021 & 3 - 0.0294 \\ -2 - 0.0126 & 4 - 0.01764 \end{pmatrix} = \begin{pmatrix} 0.979 & 2.9706 \\ -2.0126 & 3.98236 \end{pmatrix}$$

В итоге имеем новые веса:

$$W_2^{(new)} = \begin{pmatrix} 0.2816 \\ 0.1488 \end{pmatrix}$$

$$W_1^{(new)} = \begin{pmatrix} 0.979 & 2.9706 \\ -2.0126 & 3.98236 \end{pmatrix}$$

- Производная сложной функции, как в случае со слоями, – это произведение производных вложенных функций и её самой;
- Можно по-разному мерять ошибку;
- Можно брать среднюю ошибку после прогона по нескольким примерам.

§ Начало объяснения кода

Код написан на Python с применением библиотек. Нас интересует только математика происходящего, а не техника.

У нас уже достаточно знаний для того, чтобы прояснить всё происходящее пройдём по фрагментам программы и объясним их математику.

ПРЕДУПРЕЖДЕНИЕ: ПОНЯТНОСТЬ НЕ ГАРАНТИРОВАНА!!!

Если верите, что вам это поможет, то прочтите все мантры, что знаете, перед тем, как мы продолжим.

Импорт всего и инициализация rnd

```
import pandas
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

np.random.seed(42)
tf.random.set_seed(42)

from tensorflow.keras.datasets import boston_housing

(train_data, train_targets), (test_data, test_targets) = boston_housing.load_data()
```

Здесь просто импортируются все библиотеки и набор данных для тренировки и валидации нейронной сети, а также инициализируется генератор случайных чисел. Цель нейросети – предсказывать стоимость дома по входным параметрам.

Нормализация данных

```
mean = train_data.mean(axis=0)
print(f"{mean}")

train_data -= mean
print(train_data)

std = train_data.std(axis=0)
print(std)

train_data /= std

print(train_data)

test_data -= mean
test_data /=std
```

Обучающие данные нормализуются для того, чтобы ограничить масштаб значений. Тестовые нормализуются к параметрам обучающих.

Создание нейросети

```
def build_model():
  model = keras.Sequential([
      layers.Dense(128, activation='relu', input_shape=(train_data.shape[1],)),
      layers.Dense(128, activation='relu'),
      layers.Dense(64, activation='relu'),

      layers.Dense(1)
  ])

  model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
  return model
```

Первый входной слой имеет 13 входов (по числу признаков) и 128 нейронов, используется ReLU. Второй слой имеет 128 с ReLU. Третий – 64 нейрона с ReLU, которые суммируются в выходной нейрон. Сеть полносвязная. Изначально, веса – случайные числа.

Проблемы обычного градиентного спуска

Если использовать «наш метод оптимизации», то некоторые значения весов будут меняться слишком быстро, а некоторые иные – слишком медленно.

Также возможна ситуация, когда по разным направлениям мы имеем разную крутизну функции ошибки, что при общей для всех скорости обучения может привести к блужданию вокруг интересующей нас точки минимума.

Хотелось бы, в зависимости от предыдущих состояний, регулировать скорость движения вдоль градиента.

Оптимизатор RMSProp

Все эти проблемы решаются нормировкой градиента по следующим формулам:

$$G = \alpha G + (1-\alpha)\nabla L_i\left(\omega^{(t)}\right) \odot \nabla L_i\left(\omega^{(t)}\right)$$

$$\omega^{(t+1)} = \omega^{(t)} - \eta \cdot \frac{\nabla L_i\left(\omega^{(t)}\right)}{\sqrt{G} + \varepsilon}$$

Эпсилон – это малая константа для избежания делений на ноль.

Альфа – коэффициент затухания.

Чем хорош RMSProp?

- Позволяет учитывать предыдущие состояния;
- Позволяет индивидуально для каждого направления подбирать скорость движения к минимуму.

Вследствие этих двух достоинств быстрее сходится к минимуму.

$$G = \alpha G + (1-\alpha)\nabla L_i\left(\omega^{(t)}\right) \odot \nabla L_i\left(\omega^{(t)}\right)$$

$$\omega^{(t+1)} = \omega^{(t)} - \eta \cdot \frac{\nabla L_i\left(\omega^{(t)}\right)}{\sqrt{G+\varepsilon}}$$

Метрики

Среднеквадратичная ошибка для обучения считается по формуле:

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(y_i - \widehat{y_i})^2$$

где n – число примеров, y_i – настоящее значение, $\widehat{y_i}$ - предсказанное.

Средняя абсолютная:

$$MAE = \frac{1}{n}\sum_{i=0}^{n}|y_i - \widehat{y_i}|$$

Средняя абсолютная используется для удобства отслеживания изменений в точности работы сети.

ЭТИ МЕТРИКИ ПОЗВОЛЯЮТ МЕНЯТЬ ВЕСА РАЗ В НЕСКОЛЬКО ЭПОХ!!!

Обучение модели

```
history = model.fit(train_data, train_targets,
                    epochs=200,
                    batch_size=16,
                    validation_split=0.2
                    )

mae_history = history.history['val_mae']
print(f"{mae_history}")

test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)

print(test_mse_score, test_mae_score)

predictions = model.predict(test_data[:20])
```

Данные проходят 200 раз, веса обновляются через 16 эпох (помним MSE и MSA), 20% данных уходит на валидацию. Затем проводится оценка и делаются 20 предсказаний на основе входных данных.

Значения по умолчанию RMSProp: альфа 0.9, эпсилон 10^(-7), эта 0.001.

Формулы нейронной сети из кода

Далее в коде идёт простая отрисовка графиков, потому перейдём к формулам.

```
predictions = model.predict(test_data[:20])

train_mean_target = train_targets.mean()
train_std_target = train_targets.std()

predictions_denorm = predictions * train_std_target + train_mean_target
```

Кстати, результаты работы сети нужно денормировать, чтобы получить не в средних отклонениях, а в настоящих значениях.

Полносвязный слой

Для одного нейрона в слое l:

$$z_j^{(l)} = \sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} \, a_i^{(l-1)} + b_j^{(l)}$$

где:

$z_j^{(l)}$ — взвешенная сумма для нейрона j в слое l

$w_{ji}^{(l)}$ — вес связи от нейрона i предыдущего слоя к нейрону j текущего слоя

$a_i^{(l-1)}$ — активация нейрона i из предыдущего слоя $l-1$

$b_j^{(l)}$ — смещение нейрона j в слое l

n_{l-1} — количество нейронов в слое $l-1$

Матричная форма

$$Z^{(l)} = W^{(l)}A^{(l-1)} + b^{(l)}$$

где:

- $Z^{(l)} \in \mathbb{R}^{n_l}$ — вектор взвешенных сумм слоя l
- $W^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ — матрица весов слоя l
- $A^{(l-1)} \in \mathbb{R}^{n_{l-1}}$ — вектор активаций предыдущего слоя
- $b^{(l)} \in \mathbb{R}^{n_l}$ — вектор смещений слоя l

Матричная форма

Слой 1 (128 нейронов):

$$\mathrm{Z}^{(1)} = \mathrm{W}^{(1)}\mathrm{X} + \mathrm{b}^{(1)}, \mathrm{W}^{(1)} \in \mathbb{R}^{128\times 13},\ \mathrm{b}^{(1)} \in \mathbb{R}^{128}$$

$$\mathrm{H}^{(1)} = ReLU\left(\mathrm{Z}^{(1)}\right)$$

Слой 2 (128 нейронов):

$$\mathrm{Z}^{(2)} = \mathrm{W}^{(2)}\mathrm{H}^{(1)} + \mathrm{b}^{(2)}, \mathrm{W}^{(2)} \in \mathbb{R}^{128\times 128},\ \mathrm{b}^{(2)} \in \mathbb{R}^{128}$$

$$\mathrm{H}^{(2)} = ReLU\left(\mathrm{Z}^{(2)}\right)$$

Слой 3 (64 нейрона):

$$\mathrm{Z}^{(3)} = \mathrm{W}^{(3)}\mathrm{H}^{(2)} + \mathrm{b}^{(3)}, \mathrm{W}^{(3)} \in \mathbb{R}^{64\times 128},\ \mathrm{b}^{(3)} \in \mathbb{R}^{64}$$

$$\mathrm{H}^{(3)} = ReLU\left(\mathrm{Z}^{(3)}\right)$$

Выходной слой (1 нейрон):

$$y_{pred} = \mathrm{W}^{(4)}\mathrm{H}^{(3)} + b^{(4)}, \mathrm{W}^{(4)} \in \mathbb{R}^{1\times 64},\ b^{(4)} \in \mathbb{R}$$

Оставшиеся формулы

Производная MSE по весам выходного слоя:

$$\frac{\partial MSE}{\partial w_{ij}^{(4)}} = \frac{2}{n}\sum_{i=0}^{n}\left(\mathrm{y}_{pred}^{(i)} - \mathrm{y}_{true}^{(i)}\right)^2 \times h_j^{(3)}$$

Прямой проход:

$$\mathbf{H}^{(1)} = \mathrm{ReLU}(\mathbf{W}^{(1)}\mathbf{X} + \mathbf{b}^{(1)})$$
$$\mathbf{H}^{(2)} = \mathrm{ReLU}(\mathbf{W}^{(2)}\mathbf{H}^{(1)} + \mathbf{b}^{(2)})$$
$$\mathbf{H}^{(3)} = \mathrm{ReLU}(\mathbf{W}^{(3)}\mathbf{H}^{(2)} + \mathbf{b}^{(3)})$$
$$y_{\mathrm{pred}} = \mathbf{W}^{(4)}\mathbf{H}^{(3)} + b^{(4)}$$

Конец и итоги

К сожалению, вычисление сети с методом оптимизации RMSProp сложнее, чем кажется на первый взгляд. Этим можно заняться сейчас по формулам, рекуррентно меняя градиент.

Главное – получено общее представление о том, как работают нейронные сети.

Рассмотренный в коде перцептрон, фактически, ничем по смыслу не отличается от перцептрона-примера – различия только в числе входов и нейронов, а также том, что мы каждый раз делим градиент на коэффициент, получаемый из замурыжной формулы.

Спасибо за внимание!