Введение в машинное обучение

Вычислительные методы (2)

Лю Шисян (刘世贤)

Ассистент
Кафедра теплофизики (Э6)
МГТУ им. Н.Э. Баумана
lyu@bmstu.ru

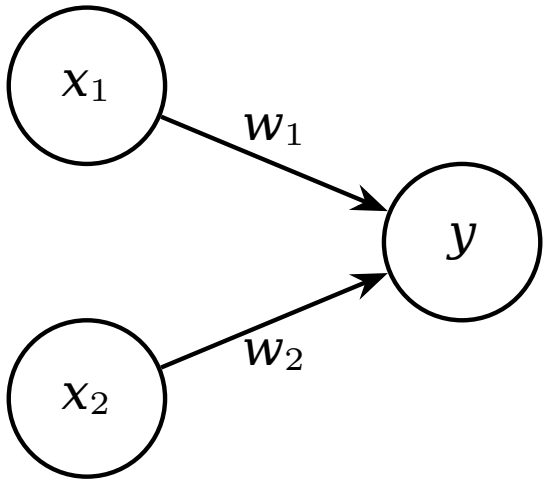
Содержание

Что такое персептрон

Персептрон (perceptron) —базовый элемент нейронных сетей, лежащий в основе современных моделей машинного обучения.

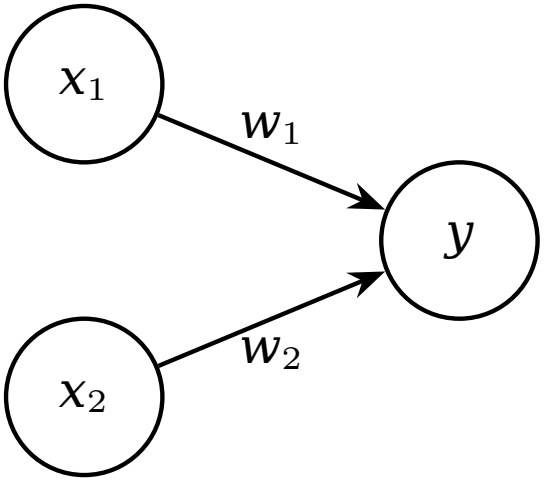
❋ Персептрон принимает несколько входных сигналов и формирует один выходной. В данной модели:

- 0 означает *сигнал не проходит*,
- 1 означает *сигнал проходит*.



★ Каждому входу соответствует вес w, который отражает важность сигнала.
★ Чем больше вес, тем значимее соответствующий сигнал.

Математическое представление персептрона



- x_1, x_2 —входные сигналы;
- w_1, w_2 —веса, отражающие важность сигналов;
- y —выходной сигнал;
- θ —порог активации нейрона.

❋ Нейрон вычисляет взвешенную сумму входов: $w_1x_1 + w_2x_2$ и сравнивает её с пороговым значением θ. Если сумма превышает порог, нейрон активируется ($y = 1$); иначе остаётся в неактивном состоянии ($y = 0$).

$$y = \begin{cases} 0, & \text{если } w_1x_1 + w_2x_2 \leq \theta \\ 1, & \text{если } w_1x_1 + w_2x_2 > \theta \end{cases} \tag{1}$$

➠ Далее мы рассмотрим, как можно использовать персептрон для реализации простых логических схем!

Код реализован в Jupyter Notebook: perceptron.ipynb

Реализация логического элемента И (AND gate)

★ Элемент **И** имеет два входа и один выход.

★ Он выдаёт 1, только если оба входа равны 1.

★ Такое соответствие между входами и выходом представлено в таблице истинности:

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1

❋ **Задача** —подобрать такие параметры w_1, w_2, θ, чтобы персептрон реализовывал поведение элемента И.

Например, при выборе $(w_1, w_2, \theta) = (0.5, 0.5, 0.7)$ персептрон будет выдавать $y = 1$ только при $x_1 = x_2 = 1$.

Также подойдут: $(0.5, 0.5, 0.8)$ или $(1.0, 1.0, 1.0)$ —все эти варианты обеспечивают активацию нейрона только в нужной строке таблицы.

★ Ключевое условие: взвешенная сумма входов должна превышать порог только тогда, когда оба входа активны.

Это и есть реализация логической функции с помощью персептрона.

Реализация логического элемента НЕ–И (NAND gate)

★ Элемент **НЕ–И** инвертирует результат И элемента.

★ Он выдаёт 1 во *всех* случаях, кроме $x_1 = x_2 = 1$.

★ Соответствие входов и выхода отражено в таблице истинности:

x_1	x_2	y
0	0	1
1	0	1
0	1	1
1	1	0

❋ **Задача** —подобрать такие параметры w_1, w_2, θ, чтобы персептрон реализовывал поведение элемента NAND.

Например, параметры $(w_1, w_2, \theta) = (-0.5, -0.5, -0.7)$ создают нужное поведение: выход $y = 0$ получается лишь при $x_1 = x_2 = 1$.

★ На самом деле достаточно изменить знаки параметров, реализующих элемент И, —и получится элемент НЕ-И (NAND).

★ Ключевое условие: взвешенная сумма $w_1 x_1 + w_2 x_2$ не должна превышать порог θ только при обоих активных входах.

Это и обеспечивает реализацию логического элемента НЕ–И на основе персептрона.

Реализация логического элемента ИЛИ (OR gate)

★ Элемент **ИЛИ** (OR) возвращает 1, если *хотя бы один* из входов активен.

★ Он выдаёт 0 только при $x_1 = x_2 = 0$, во всех остальных случаях результат равен 1.

★ Поведение ИЛИ элемента представлено в таблице истинности:

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	1

❋ **Задача** —определить параметры w_1, w_2, θ, при которых персептрон реализует элемент ИЛИ.

★ Например, параметры $(w_1, w_2, \theta) = (0.5, 0.5, 0.4)$ обеспечивают корректную работу: сумма превышает порог, если хотя бы один из входов равен 1.

★ Ключевое условие: взвешенная сумма входов должна превышать θ во всех случаях, кроме $x_1 = x_2 = 0$.

Так реализуется логическая функция OR на базе персептрона.

Настройка параметров персептрона: человек или машина?

★ Параметры персептрона (веса и порог) на предыдущих слайдах подбирали мы сами —вручную, глядя на таблицу истинности. В этом случае человек выбирает параметры на основе знания задачи.

★ Однако в машинном обучении основная задача —научить компьютер находить подходящие параметры самостоятельно.

Обучение означает процесс автоматического подбора параметров модели (персептрона), чтобы она соответствовала данным. Задача человека —задать структуру модели и передать машине данные.

❋ Таким образом: И, ИЛИ и НЕ–И можно реализовать с помощью одного и того же персептрона, меняя только значения весов и порога.

Введение смещения (bias) в персептрон

➠ Для удобства мы вводим **смещение** (bias) $b = -\theta$, и переписываем формулу в более универсальном виде:

$$y = \begin{cases} 0, & \text{если } b + w_1 x_1 + w_2 x_2 \leq 0 \\ 1, & \text{если } b + w_1 x_1 + w_2 x_2 > 0 \end{cases} \tag{2}$$

★ **Смысл:**

- w_1, w_2 – отвечают за важность каждого входа;
- b – определяет, насколько *легко* нейрон активируется в целом.

Если $b = -0.1$, достаточно, чтобы сумма входов чуть превышала 0.1 —и нейрон сработает.

Если же $b = -20.0$, активация произойдёт только при очень сильных сигналах.

❋ Таким образом, b задаёт общий уровень чувствительности нейрона, независимо от отдельных весов.

Реализация с помощью Python

Функция AND:

```
1  def AND(x1, x2):
2      x = np.array([x1, x2])
3      w = np.array([0.5, 0.5])
4      b = -0.7
5      tmp = np.sum(w * x) + b
6      return int(tmp > 0)
```

Функция OR:

```
1  def OR(x1, x2):
2      x = np.array([x1, x2])
3      w = np.array([0.5, 0.5])
4      b = -0.2
5      tmp = np.sum(w * x) + b
6      return int(tmp > 0)
```

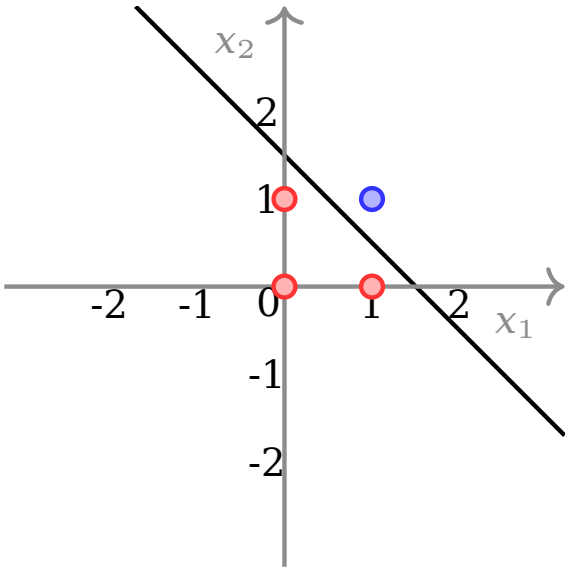
Функция NAND:

```
1  def NAND(x1, x2):
2      x = np.array([x1, x2])
3      w = np.array([-0.5, -0.5])
4      b = 0.7
5      tmp = np.sum(w * x) + b
6      return int(tmp > 0)
```

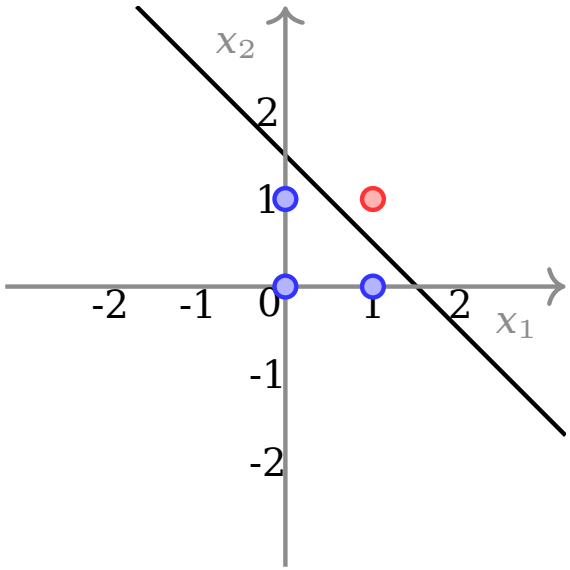
❋ Все три логические функции реализуются одной и той же структурой —различие только в весах и смещении.

Графическое представление логических функций

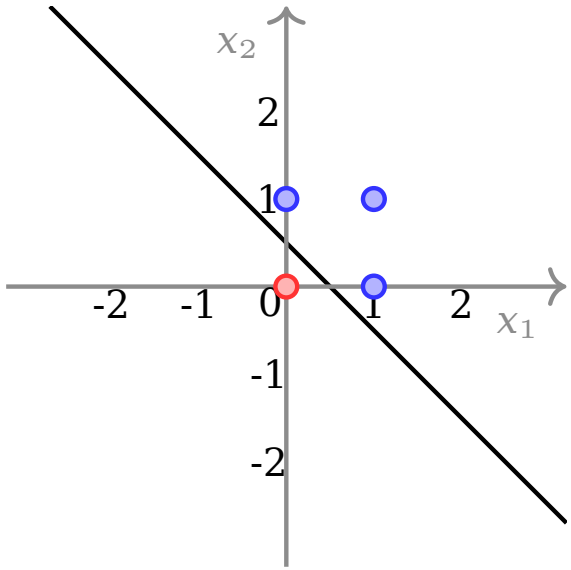
Три основные логические операции, которые может реализовать персептрон, представлены ниже. Каждая из них разделяет пространство прямой $w_1 x_1 + w_2 x_2 + b = 0$ на два класса.



AND (И)



NAND (НЕ–И)



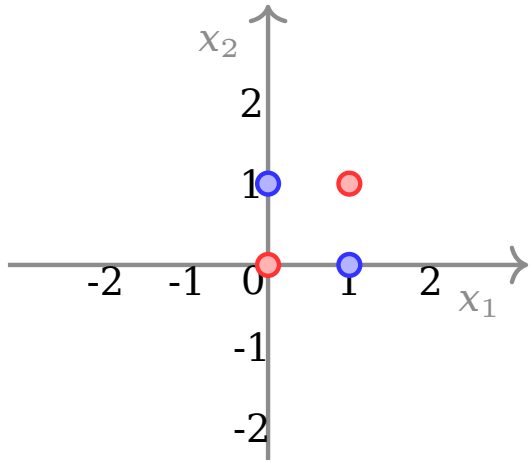
OR (ИЛИ)

Цветом показана область, где персептрон выдаёт 1 (синяя) и 0 (красная).
Каждая операция достигается выбором соответствующих весов и смещения.

Исключающее ИЛИ (XOR) и пределы однослойного персептрона

Таблица истинности XOR:

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	0



- XOR (исключающее «или») выдаёт 1, *только* когда ровно один из входов равен 1.
- Четыре точки данных $(0,0), (1,0), (0,1), (1,1)$ невозможно разделить одной прямой, поэтому пространство не является линейно разделимым.

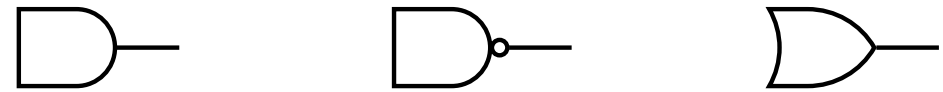
❋ Вывод. Однослойный персептрон способен реализовать лишь линейно разделимые функции (AND, OR, NAND и др.). Для нелинейных задач, таких как XOR, требуется *многослойная нейронная сеть*, способная формировать произвольные —в том числе криволинейные —разделяющие поверхности.

Многослойный персептрон

Именно добавление *скрытых слоёв* позволяет выражать сложные, нелинейные функции, включая XOR.

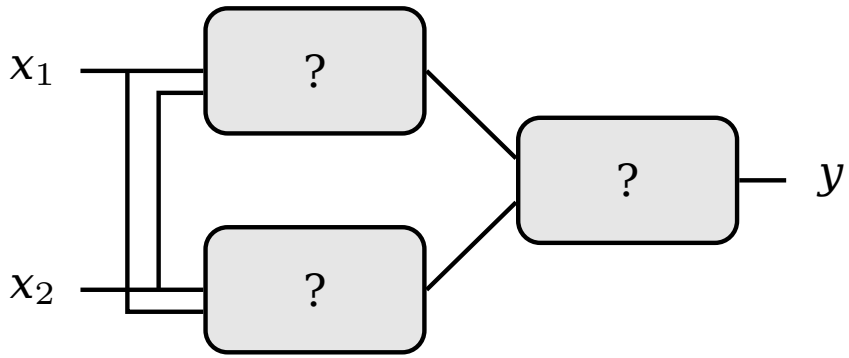
Один из способов построения XOR —это комбинация известных логических элементов: **AND**, **NAND** и **OR**.

Базовые логические элементы:

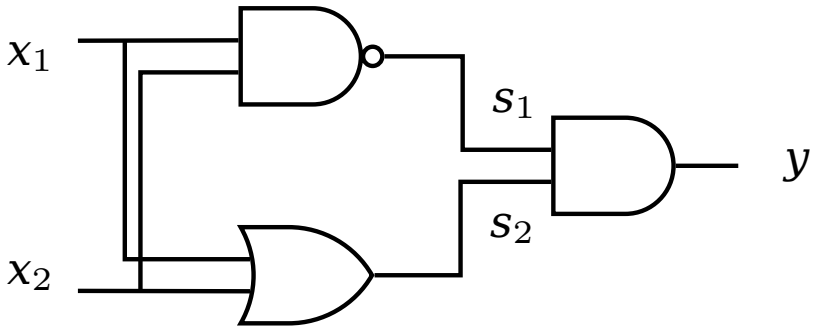


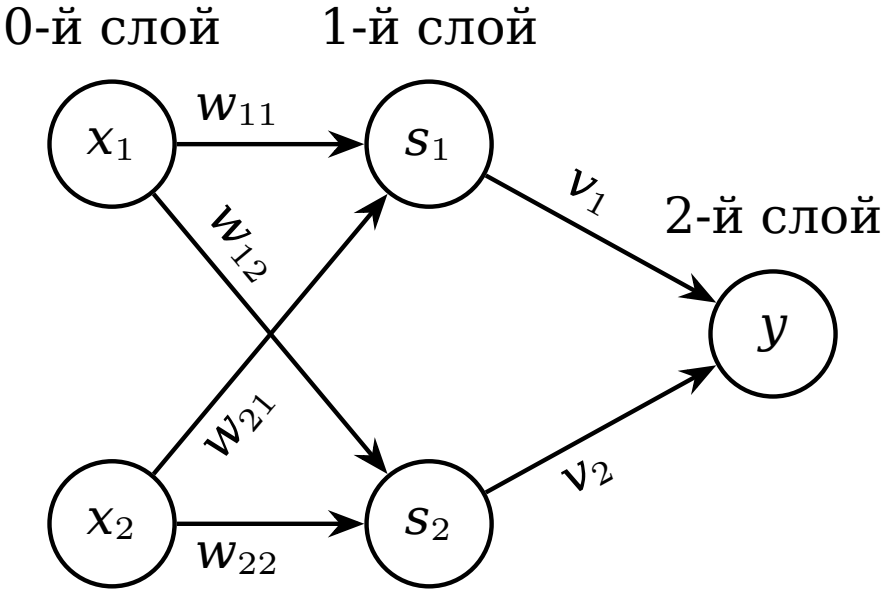
AND NAND OR

Вопрос: как соединить эти элементы, чтобы получить XOR?



Многослойный персептрон





Персептрон отличается от однослойных И и ИЛИ. XOR реализуется с помощью двух слоёв —это **многослойный персептрон**.

Теперь проверим, действительно ли схема реализует XOR. Пусть s_1 — выход NAND, s_2 —выход OR. Подставим их в таблицу истинности, значения x_1, x_2 и y соответствуют выходу XOR.

Таблица истинности XOR:

x_1	x_2	s_1	s_2	y
0	0	1	0	0
1	0	1	1	1
0	1	1	1	1
1	1	0	1	0

Итоги главы

- Персептрон —это алгоритм с входами и выходом. При заданном входе он выдаёт определённое значение.
- Веса и смещения персептрона задаются как параметры.
- С помощью персептрона можно реализовать логические элементы И, НЕ И, ИЛИ и т.д.
- Элемент XOR нельзя реализовать с помощью однослойного персептрона.
- Двухслойный персептрон может реализовать XOR.
- Однослойный персептрон разделяет линейное пространство, а многослойный —нелинейное.
- Теоретически многослойный персептрон способен моделировать вычислительную машину.

Содержание

Переход к нейронным сетям

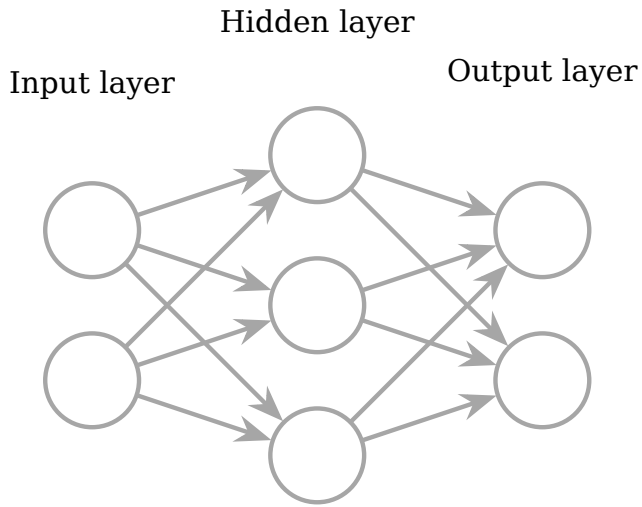
В предыдущей главе мы изучили персептрон. Хорошая новость —даже сложные функции можно выразить с его помощью. Плохая новость —веса приходилось подбирать вручную.

На примере логических элементов И и ИЛИ мы вручную находили подходящие веса. Но такой подход не подходит для сложных задач.

Нейронные сети были придуманы, чтобы решить эту проблему. Их главное преимущество —способность обучаться автоматически.

В этой главе мы рассмотрим, как устроена нейронная сеть и как она работает.

В следующей главе мы изучим, как обучать веса сети.

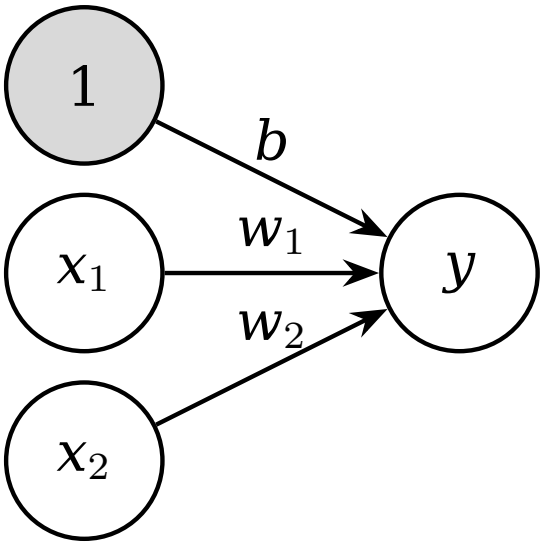


Повторим простейший персептрон

Нейрон получает два входных сигнала x_1 и x_2, и выдаёт выходной сигнал y. Математическая модель выглядит так:

$$y = \begin{cases} 0, & b + w_1 x_1 + w_2 x_2 \leq 0 \\ 1, & b + w_1 x_1 + w_2 x_2 > 0 \end{cases} \tag{3}$$

Здесь b —*смещение* (bias), определяет лёгкость активации нейрона. Параметры w_1 и w_2 —*веса*, отражающие важность каждого входа.



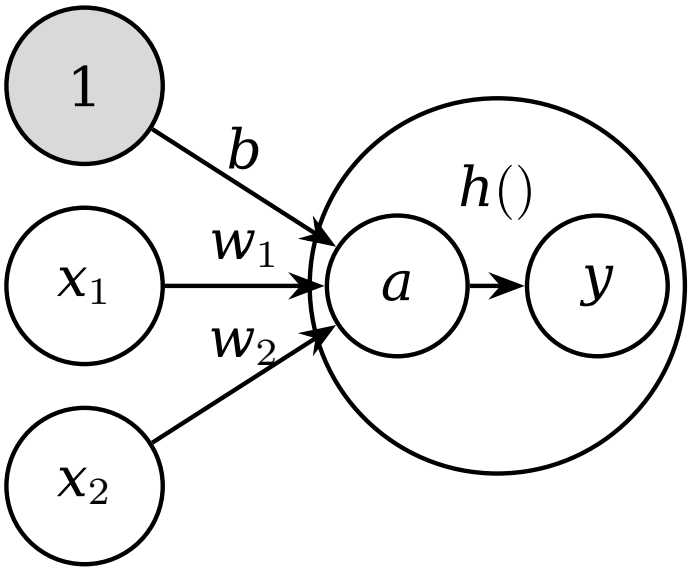
Чтобы явно показать b, можно представить его как входной сигнал со значением 1, соединённый с весом b, как показано на рисунке слева. Мы закрасили этот узел в серый цвет, чтобы отличать от обычных входов.

Введение активационной функции

Чтобы упростить выражение (3), вводится функция $h(a)$, которая возвращает 1, если $a > 0$, иначе 0:

$$a = b + w_1 x_1 + w_2 x_2 \tag{4}$$

$$h(a) = \begin{cases} 0, & a \leq 0 \\ 1, & a > 0 \end{cases} \tag{5}$$



Функция $h(a)$ называется **функцией активации** (activation function), так как она «активирует» или «тормозит» нейрон в зависимости от входной суммы.

На рисунке показано, как схематически изобразить вычисление a и его последующее преобразование через $h()$.

От персептрона к нейронной сети

Ранее в модели персептрона использовалась ступенчатая функция активации —она мгновенно «включает» нейрон, если входное значение превышает порог.

Но что будет, если заменить ступенчатую функцию на другую? Именно так работает нейронная сеть: если заменить жёсткую функцию $h(x)$ на более гладкую, мы получим гибкую модель — нейронную сеть.

Сигмоида (Sigmoid) —одна из самых популярных функций активации:

$$h(x) = \frac{1}{1 + \exp(-x)} \tag{6}$$

Эта функция «сглаживает» переход от 0 к 1. Например:

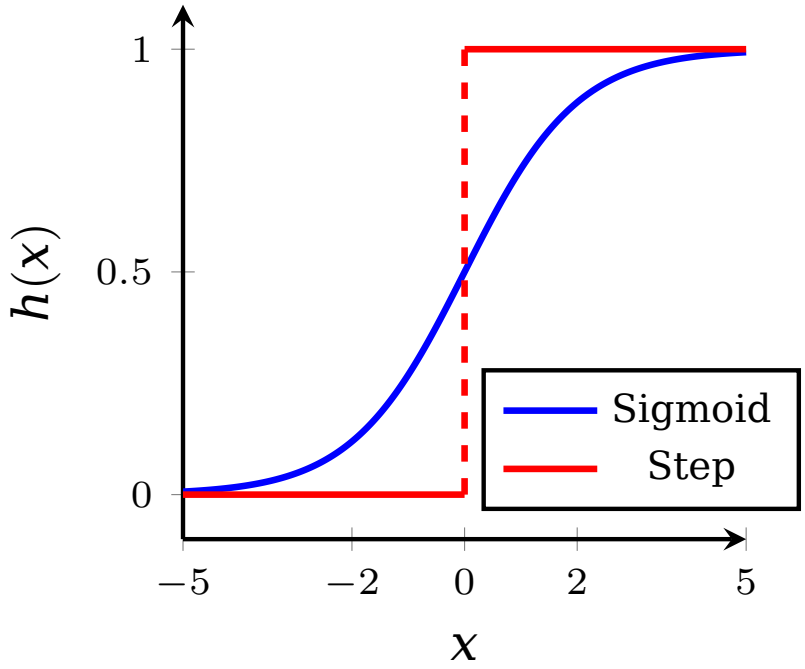
$$h(1.0) \approx 0.731, \quad h(2.0) \approx 0.880$$

Благодаря этой функции сигнал может быть частично активирован и передан дальше по сети. Именно наличие таких активационных функций отличает нейросети от обычных персептронов.

Сравнение: ступенчатая функция и сигмоида

Step и Sigmoid:

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

def step(x):
    return np.array(x > 0, dtype=int)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

x = np.arange(-5, 5, 0.1)
y1 = step(x)
y2 = sigmoid(x)

plt.plot(x, y1, '--r', label='step')
plt.plot(x, y2, 'b', label='sigmoid')
plt.ylim(-0.1, 1.1)
plt.legend()
plt.show()
```



Ступенчатая функция резко меняет значение при $x = 0$, сигмоида обеспечивает плавный переход и подходит для обучения.

- Обе функции возвращают значения в диапазоне от 0 до 1.
- При увеличении входа результат приближается к 1, при уменьшении —к 0.

Активационная функция должна быть нелинейной

Функции *step* и *sigmoid* обладают ещё одним важным свойством: обе являются нелинейными.

В машинном обучении под **нелинейной функцией** понимается функция, *не имеющая форму прямой линии*, в отличие от линейной функции $h(x) = c \cdot x$.

Почему в нейронных сетях активационная функция **обязательно должна быть нелинейной**?

Потому что при использовании линейной функции, какова бы ни была глубина сети, она всегда сводится к одному линейному преобразованию и не даёт преимуществ многослойной архитектуры.

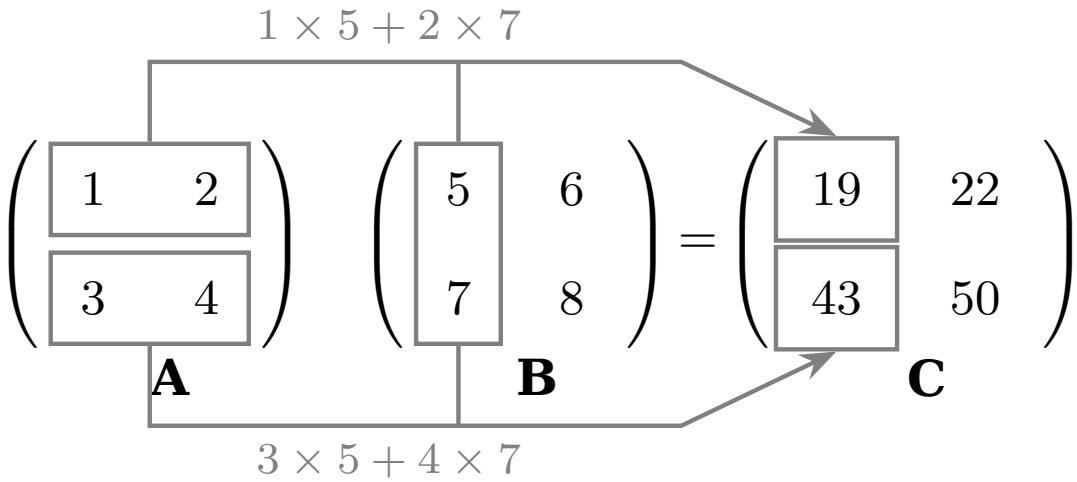
Например, если использовать $h(x) = c \cdot x$, то для трёх слоёв:

$$y(x) = h(h(h(x))) = c^3 \cdot x,$$

что эквивалентно одной линейной трансформации: $y(x) = a \cdot x$.

Вывод: мощь нейронных сетей заключается не в линейных преобразованиях, а в способности строить сложные нелинейные зависимости с помощью активационных функций.

Умножение матриц




```python
import numpy as np

A = np.array([[1, 2],
              [3, 4]])
print("A.shape:", A.shape) # (2, 2)

B = np.array([[5, 6],
              [7, 8]])
print("B.shape:", B.shape) # (2, 2)

C = np.dot(A, B)
print(C)
# [[19 22]
#  [43 50]]
```

NumPy vs MATLAB: Сравнение операций

Главное отличие оператора *:

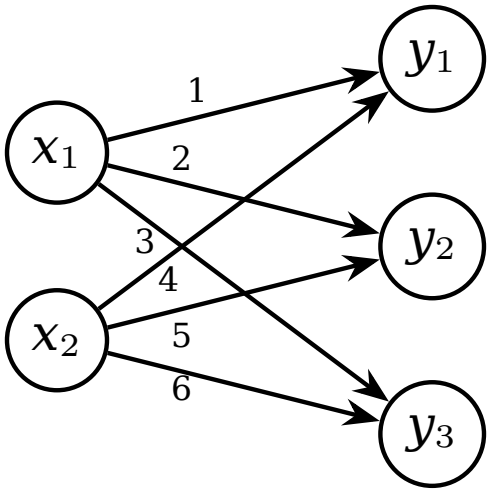
- **MATLAB**: по умолчанию - **матричные** операции.
- **NumPy**: по умолчанию - **поэлементные** операции.

Операция	**NumPy (Python)**	**MATLAB**
Матричное умножение	`A @ B` или `np.dot(A, B)`	`A * B`
Поэлементное умножение	`A * B`	`A .* B`
Скалярное произведение (Inner Product)	`np.dot(u, v)` или `u @ v` (для 1D массивов)	`dot(u, v)` или `u' * v` (для вектор-столбцов)
Внешнее произведение (Outer Product)	`np.outer(u, v)`	`u * v'` (столбец × строка)

Важно: В MATLAB векторы имеют ориентацию (строка/столбец).
В NumPy 1D-массив `(N,)` не является ни строкой, ни столбцом.

Реализация нейросети через умножение матриц

Ниже показана простая нейросеть (без смещений и активаций), где веса представлены матрицей **W**, а входы —**X**. Результат вычисляется как: $\mathbf{Y} = \mathbf{X} \cdot \mathbf{W}$



$$\underset{\mathbf{X}}{\begin{pmatrix} 1 & 2 \end{pmatrix}} \cdot \underset{\mathbf{W}}{\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}} = \underset{\mathbf{Y}}{\phantom{\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}}}$$

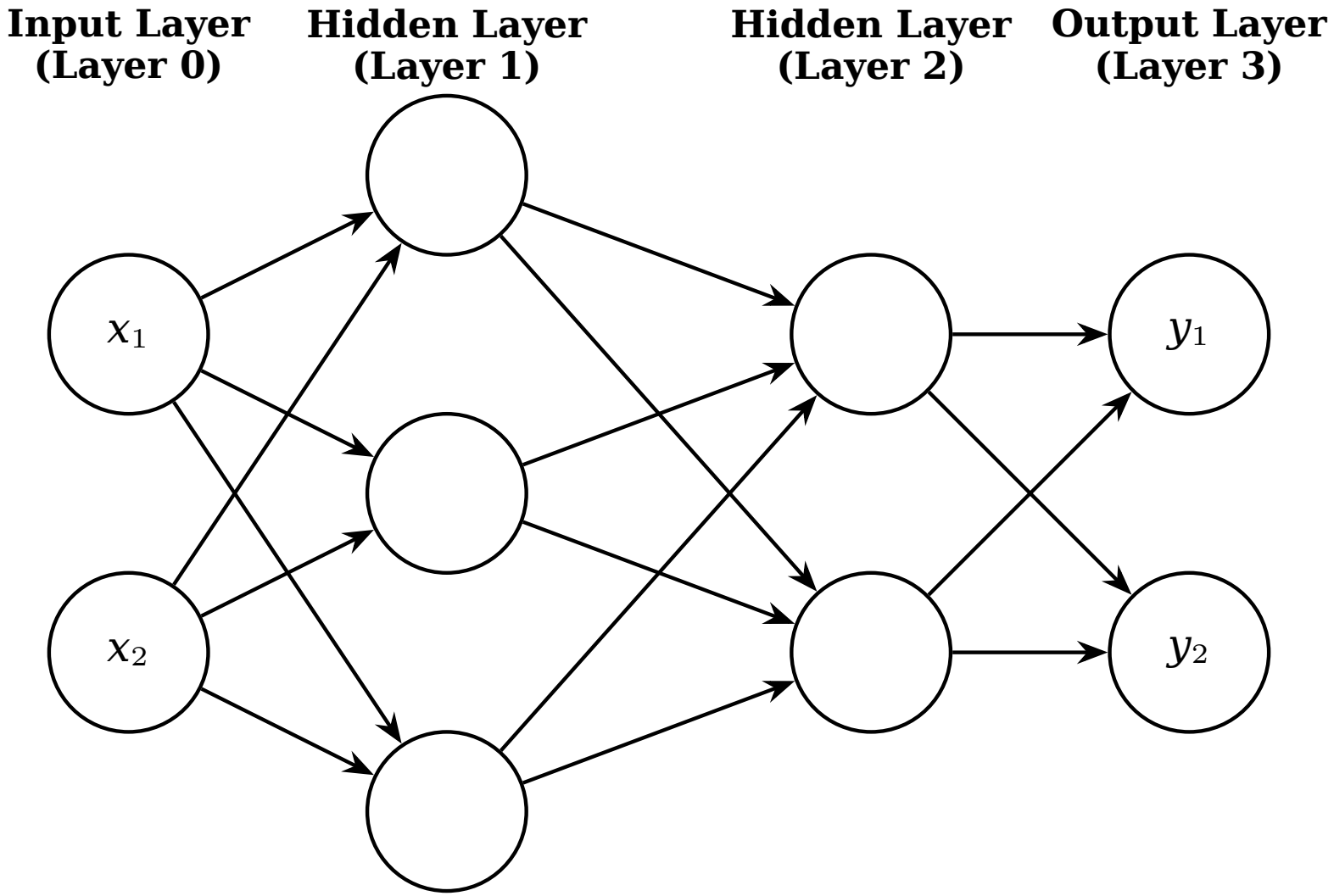
$1 \times 2 \quad 2 \times 3 \quad 3$

same same

Важно, чтобы размеры матриц были согласованы: число столбцов в **X** должно совпадать с числом строк в **W**.

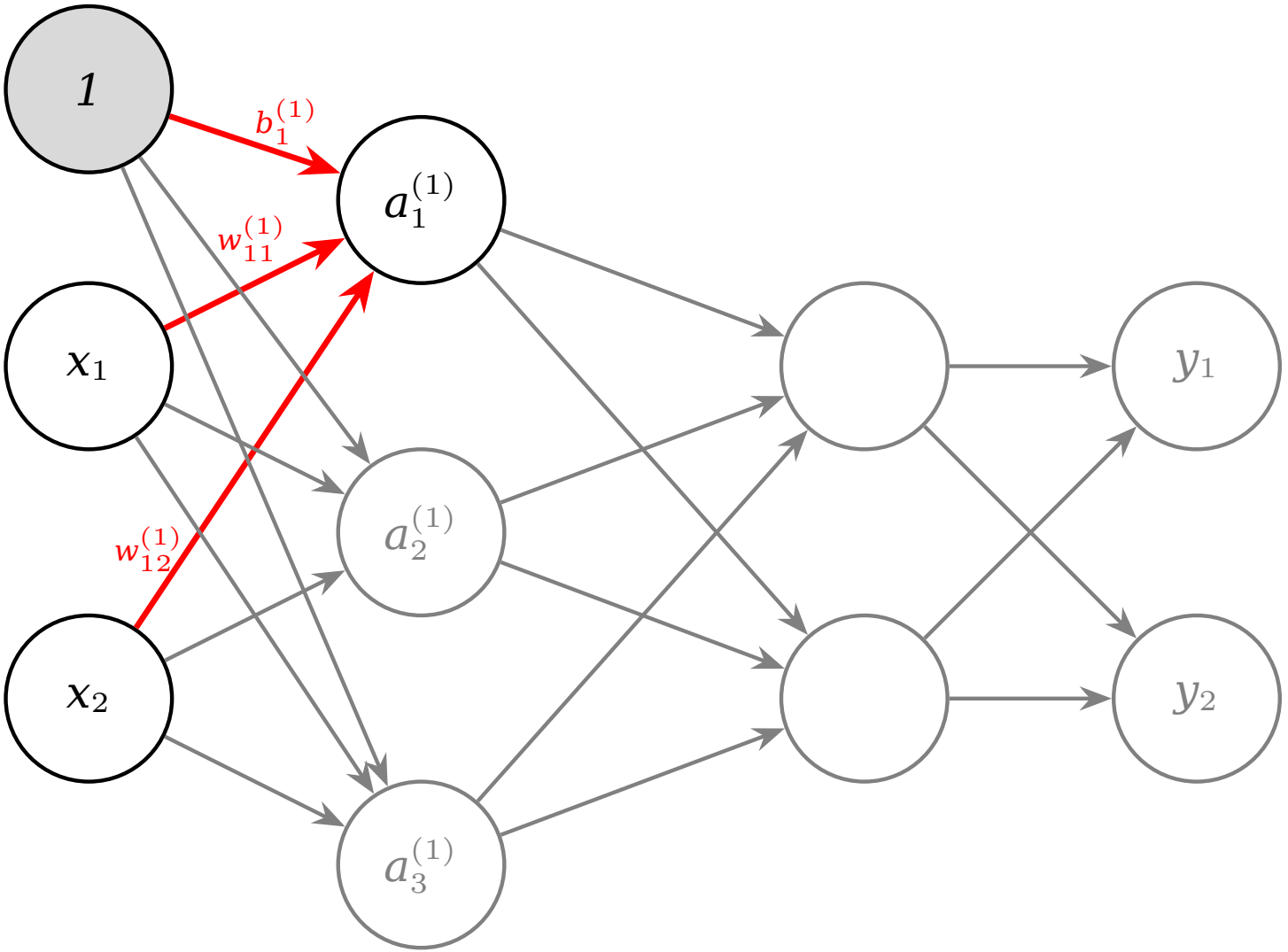
```
X = np.array([1, 2])
print(X.shape)  # (2,)

W = np.array([[1, 3, 5], [2, 4, 6]])
print(W.shape)  # (2, 3)

Y = np.dot(X, W)
print(Y)        # [ 5 11 17 ]
```

Реализация 3-х слойной нейронной сети



Код реализован в Jupyter Notebook: perceptron.ipynb

От слоя 0 к слою 1



До активации значение нейрона $a_1^{(1)}$ первого скрытого слоя вычисляется по формуле: $a_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$

Вычисление взвешенной суммы в первом слое

Если использовать матричную форму записи, то взвешенную сумму на первом скрытом слое можно выразить как:

$$A^{(1)} = XW^{(1)} + B^{(1)} \tag{7}$$

где

$$A^{(1)} = \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{pmatrix}, \quad X = \begin{pmatrix} x_1 & x_2 \end{pmatrix}, \quad B^{(1)} = \begin{pmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{pmatrix}$$

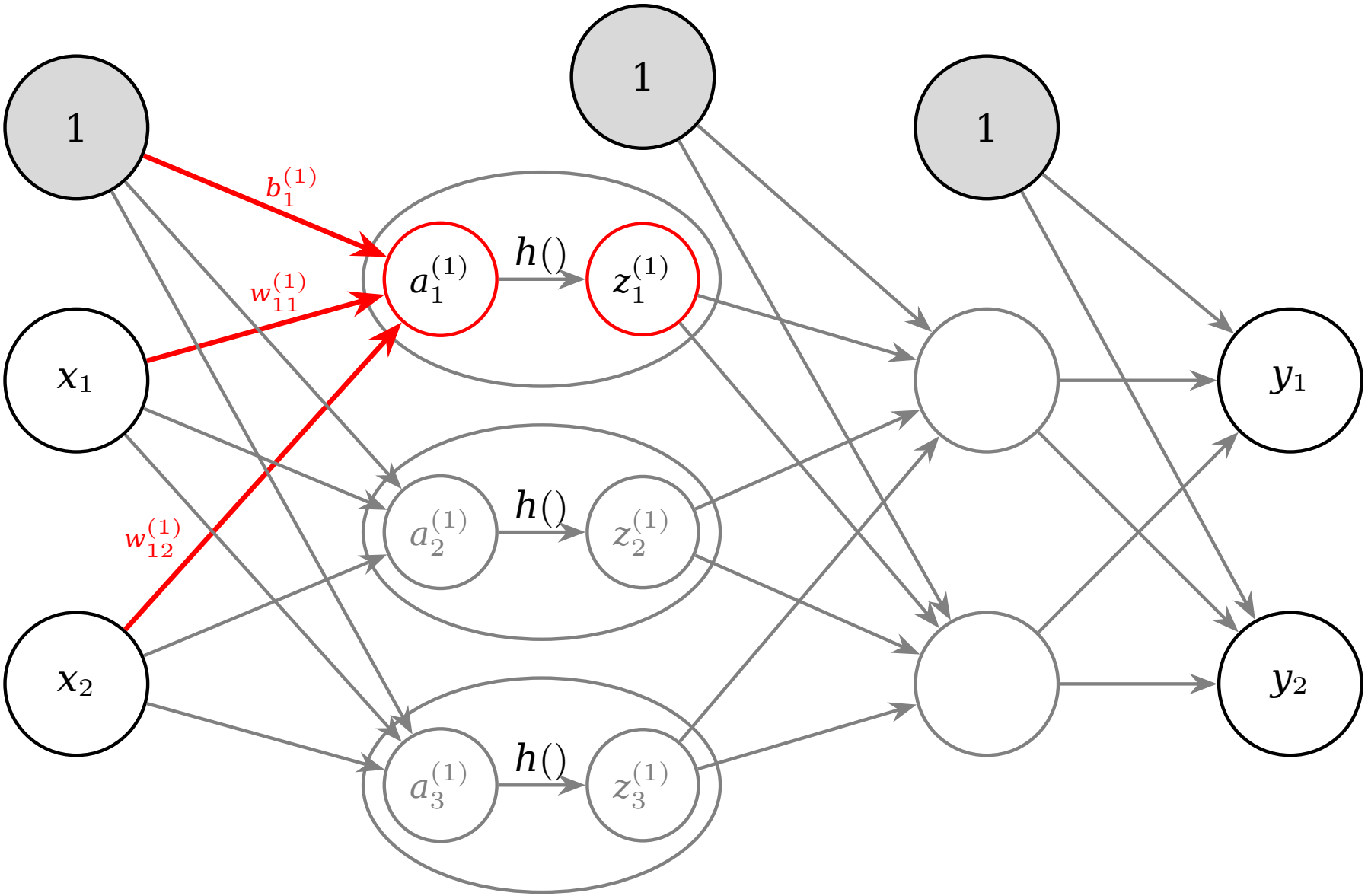
$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

Затем к результату $A^{(1)}$ применяется функция активации, чтобы получить выход $Z^{(1)}$.

$$Z^{(1)} = h(A^{(1)}) \tag{8}$$

В качестве функции активации можно использовать, например, сигмоидную (Sigmoid) функцию.

От слоя 0 к слою 1: взвешенная сумма и активация



Общий принцип вычислений в последующих слоях

Точно так же вычисляются следующие слои (например, для трёхслойной сети):

$$A^{(2)} = Z^{(1)}W^{(2)} + B^{(2)}, \qquad Z^{(2)} = h(A^{(2)})$$

$$A^{(3)} = Z^{(2)}W^{(3)} + B^{(3)}, \qquad Y = h(A^{(3)})$$

В общем случае для любого слоя l формулы имеют вид:

$$A^{(l)} = Z^{(l-1)}W^{(l)} + B^{(l)}, \qquad Z^{(l)} = h(A^{(l)}).$$

Таким образом, каждый последующий слой вычисляется по одной и той же схеме:

- матричное умножение;
- добавление смещения (bias);
- применение функции активации.

Функция активации в выходном слое

В **выходном слое** выбор функции активации **зависит от задачи**.

1. **Регрессия**
   - прогноз вещественного числа;
   - используется **тождественная функция**:

$$y = a$$

2. **Бинарная классификация**
   - два класса $(0/1)$;
   - используется **сигмоида**:

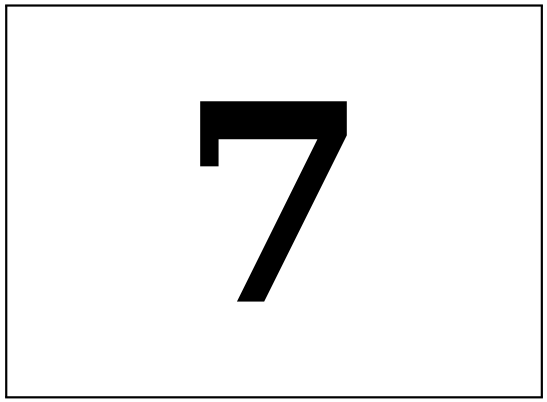
$$y = \frac{1}{1 + e^{-a}}, y \in (0, 1)$$

3. **Многоклассовая классификация**
   - один из n классов;
   - используется **softmax-функция**:

$$y_k = \frac{e^{a_k}}{\sum\limits_{i=1}^{n} e^{a_i}}, \quad \sum_{k=1}^{n} y_k = 1$$

Пример: Распознавание рукописных цифр (MNIST)

Вход (Input)



- Изображение 28×28 пикселей.
- Задача: определить, какая это цифра $(0 \dots 9)$.

Выходной слой (Output Layer)

- **10 нейронов** (по одному на цифру).
- Функция активации: **Softmax**.

Результат (Распределение):

$$\begin{bmatrix} P(y=0) \\ \vdots \\ P(y=7) \\ \vdots \\ P(y=9) \end{bmatrix} \approx \begin{bmatrix} 0.01 \\ \vdots \\ \mathbf{0.92} \\ \vdots \\ 0.03 \end{bmatrix}$$

Softmax превращает выходы нейросети в **вероятности**. Мы выбираем класс с максимальной вероятностью:

$$\text{Prediction} = \arg\max_k (y_k) \quad \Rightarrow \quad \textbf{Цифра 7}$$

Итоги главы

- Нейронная сеть строится как композиция слоёв вида **«линейное преобразование + нелинейная функция активации»**.
- В скрытых слоях используются сглаженные нелинейные функции активации (например, **sigmoid**), что позволяет модели приближать сложные зависимости.
- Благодаря **матричному умножению** и массивам **NumPy** прямой проход по сети можно компактно и эффективно реализовать в коде.
- Функция активации в **выходном слое** выбирается по типу задачи:
  - для **регрессии** —тождественная функция $y = a$;
  - для **бинарной классификации** —**sigmoid**;
  - для **многоклассовой классификации** —**softmax-функция**.

Содержание

Обучение нейронных сетей

❋ В традиционном программировании результат получается по заданным правилам:

правила + данные → ответы

❋ В машинном обучении всё наоборот:

данные + ответы → правила

★ Вместо ручного написания правил мы предоставляем системе множество размеченных примеров.
★ Она самостоятельно находит статистические закономерности, которые позволяют выполнять нужную задачу.

★ При обучении нейронной сети цель состоит в том, чтобы минимизировать функцию потерь —то есть ошибку между предсказанным и истинным значением.
★ Для этого используются специальные алгоритмы — оптимизаторы, которые постепенно корректируют параметры сети, чтобы улучшить результат.

★ Что особенно важно —нейронные сети способны автоматически извлекать параметры из обучающих данных.

Тренировочные и тестовые данные

В машинном обучении данные обычно делятся на две части:

- **Тренировочные данные** используются для обучения модели и настройки параметров.
- **Тестовые данные** применяются для оценки обобщающей способности модели.

Чтобы модель хорошо работала не только на обучающих примерах, но и на новых данных, необходимо проверять её **обобщающую способность**.

Если использовать только один набор данных и для обучения, и для оценки, модель может запомнить особенности этих данных и плохо работать на других —это называется **переобучением** (overfitting).

Цель машинного обучения —не просто распознать обучающие примеры, а научиться решать задачи на любых новых данных. Поэтому важно разделять данные на тренировочные и тестовые, чтобы обеспечить объективную оценку модели.

Код реализован в Jupyter Notebook: learning.ipynb

Функции потерь в нейросетях

В обучении нейронных сетей функция потерь (loss function) служит мерой того, насколько плохо сеть решает задачу —чем больше ошибка, тем хуже параметры.

Наиболее распространённые функции потерь:

- **Среднеквадратичная ошибка (MSE)**:

$$E = \frac{1}{2} \sum_k (y_k - t_k)^2$$

  где y_k —выход сети, t_k —истинная метка.

- **Кросс-энтропия (Cross Entropy)**:

$$E = -\sum_k t_k \log y_k$$

  Здесь t_k принимает значение 1 только для правильного класса (one-hot разметка).

Функция потерь показывает, насколько предсказания сети совпадают с реальными метками. Цель обучения — минимизировать это значение и найти наилучшие веса.

Практический пример: Сравнение моделей

Задача: Классификация изображений (3 класса: Свинья, Собака, Кошка).

У нас есть две модели. Какая из них лучше?

Истина	**Класс**	**Модель 1**	**Модель 2**	**Результат**
$[0, 0, 1]$	Свинья	$[0.3, 0.3, \mathbf{0.4}]$ ✓	$[0.1, 0.2, \mathbf{0.7}]$ ✓	Оба правы
$[0, 1, 0]$	Собака	$[0.3, \mathbf{0.4}, 0.3]$ ✓	$[0.1, \mathbf{0.7}, 0.2]$ ✓	Оба правы
$[1, 0, 0]$	Кошка	$[0.1, 0.2, \mathbf{0.7}]$ ✗	$[0.3, 0.4, \mathbf{0.3}]$ ✗	**Оба ошиблись**

Интуитивный анализ:

- **Модель 1**: Еле угадала правильные ответы (0.4), сильно ошиблась на кошке (0.1).
- **Модель 2**: Уверена в правильных ответах (0.7), ошибка на кошке менее грубая (0.3).

Вывод: Модель 2 лучше. Но как это измерить?

Выбор функции потерь (Loss Comparison)

1. Ошибка классификации (Classification Error)

$$\text{Error} = \frac{\text{Неверные}}{\text{Всего}} \quad \Rightarrow \quad M1 : \frac{1}{3}, \quad M2 : \frac{1}{3}$$

✗ Не видит разницы между моделями.

2. Среднеквадратичная ошибка

$$M1 \approx 0.38, \quad M2 \approx 0.08$$

! Проблема градиентов (Vanishing Gradient):

- Если модель «уверенно ошибается», градиент $\to 0$.
- Обучение останавливается, даже если ошибка большая.

3. Кросс-энтропия —Идеальный выбор для классификации

$$Loss = -\sum_i t_i \log(y_i) \quad \Rightarrow \quad M1 \approx 1.38, \quad M2 \approx 0.64$$

✓ Чем больше ошибка, тем больше градиент (быстрое обучение).

Метод градиентного спуска

Одна из главных задач обучения нейросетей —найти оптимальные параметры (веса и смещения), которые минимизируют функцию потерь.

Метод градиентного спуска —это способ итеративного движения в направлении уменьшения значения функции. Он широко используется для оптимизации в машинном обучении.

Как работает:

- В каждой точке вычисляется градиент —направление наибольшего увеличения функции.
- Чтобы уменьшить значение функции, движемся в обратную сторону —по направлению **градиентного спуска**.
- Повторяем шаги, пока не достигнем минимума или не остановимся на ”плато”.

Важно понимать:

- Градиент не всегда указывает на глобальный минимум.
- Возможны локальные минимумы или точки седла (где градиент равен нулю, но это не минимум).

Скорость обучения

Скорость обучения (learning rate, обозначается η) —это гиперпараметр, определяющий, насколько сильно изменяются параметры модели на каждом шаге градиентного спуска:

$$w_i = w_i - \eta \frac{\partial f}{\partial w_i}$$

- Если скорость обучения слишком большая —значения «разлетаются», и модель не может сойтись.
- Если слишком маленькая —обучение идёт очень медленно или останавливается вовсе.

Скорость обучения —это гиперпараметр, в отличие от весов и смещений, которые настраиваются автоматически. Гиперпараметры задаются вручную и часто подбираются перебором.

Типичные значения: `0.01`, `0.001` и т.д.
В обучении нейросетей часто нужно экспериментировать с этим параметром, чтобы достичь хорошей сходимости.

Mini-batch в обучении нейросетей

При обучении нейронных сетей данные редко подаются в модель целиком. Вместо этого используются три режима обучения:

- **Полный градиентный спуск (Full batch)** —используется весь набор данных. Медленно и требует много памяти.
- **Стохастический градиентный спуск (SGD)** — обновление по одному примеру. Быстро, но обладает высокой шумностью.
- **Мини-пакетный градиентный спуск (Mini-batch)** — обновление по небольшим порциям данных (например, 16, 32 или 64 примера).

Почему используют mini-batch?

- ★ ускоряет обучение по сравнению с full batch;
- ★ снижает шум по сравнению с SGD;
- ★ хорошо работает с векторизацией и ускорителями (GPU);
- ★ делает процесс обучения более стабильным.

Mini-batch —наиболее популярный и практически полезный режим обучения нейросетей.

Численное дифференцирование градиента

Чтобы понять, как работает градиентный спуск, можно сначала приблизить градиент **численным дифференцированием**.

Пусть есть функция потерь $L(w)$ от одного параметра w. Тогда производную можно приближённо вычислить по формуле конечных разностей:

$$\frac{\partial L}{\partial w} \approx \frac{L(w+\delta) - L(w-\delta)}{2\delta}, \qquad \delta \text{ —маленькое число.}$$

Плюсы:

- ★ простая и универсальная идея;
- ★ подходит для **проверки** правильности аналитического градиента.

Минусы:

- ★ очень медленно —для каждого параметра нужно пересчитывать функцию потерь;
- ★ численные ошибки при слишком большом или слишком маленьком δ.

Обратное распространение ошибки (backpropagation)

В реальных нейросетях параметры считаются **анализически**. Для этого используется алгоритм **обратного распространения ошибки** (backpropagation).

Идея backpropagation:

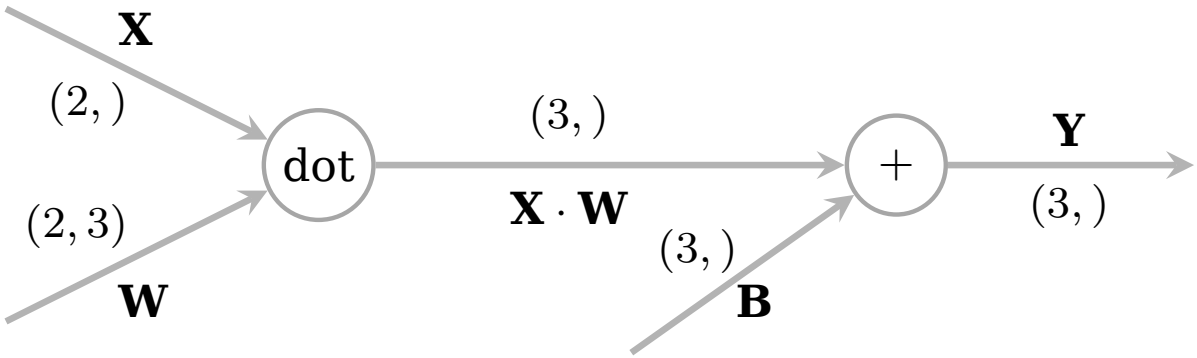
- ★ прямой проход: считаем выход сети и значение функции потерь L;
- ★ затем по правилу цепочки (chain rule) шаг за шагом вычисляем производные $\frac{\partial L}{\partial W^{(l)}}$ для всех слоёв l;
- ★ используя эти производные, обновляем веса методом градиентного спуска.

Преимущества backpropagation:

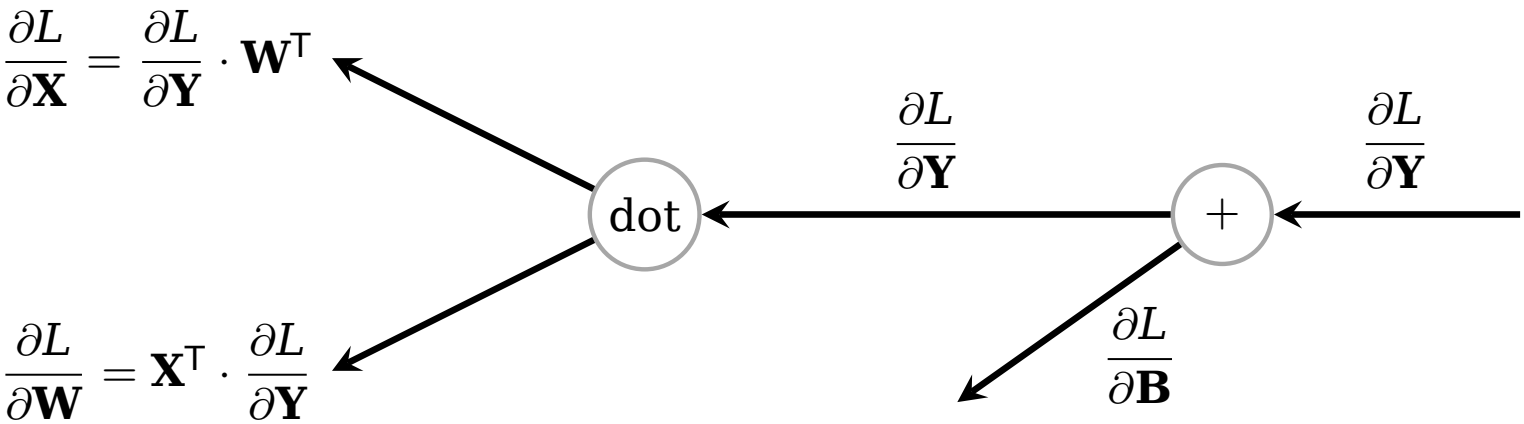
- ★ вычисляет градиент **одним проходом** по сети;
- ★ хорошо работает для сетей с большим числом параметров;
- ★ именно этот метод реализован во всех фреймворках (`PyTorch`, `TensorFlow` и др.).

Обратное распространение ошибки (backpropagation)

Прямой проход (forward)



Обратное распространение (backward)



Обратное распространение: Правила (Backprop Rules)

Как ошибка ($\frac{\partial L}{\partial Y}$) проходит через узлы?

1. Сложение (+)

«Распределитель»

Градиент передаётся на все входы **без изменений**.

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial Y}$$

2. Умножение (dot)

«Переключатель»

Градиент умножается на **транспонированный** вход соседа.

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot W^\top$$

$$\frac{\partial L}{\partial W} = X^\top \cdot \frac{\partial L}{\partial Y}$$

Итоги: как обучается нейронная сеть

- Нейронная сеть обучается, корректируя параметры так, чтобы минимизировать **функцию потерь**.
- Обучение происходит итеративно:
  1. прямой проход —вычисляем выходы сети и значение L;
  2. обратное распространение —находим градиенты $\frac{\partial L}{\partial W}$ и $\frac{\partial L}{\partial B}$;
  3. обновляем параметры с помощью градиентного спуска.
- Для ускорения и стабилизации обучения данные подаются небольшими порциями —**mini-batch**.
- Численное дифференцирование может использоваться для проверки корректности градиентов, но основная работа выполняется алгоритмом **backpropagation**.
- Если градиенты вычислены правильно и скорость обучения выбрана успешно, функция потерь постепенно уменьшается —сеть действительно учится.