Введение в машинное обучение

Вычислительные методы (2)

Лю Шисян

sxliu98@gmail.com
Кафедра теплофизики (Э6)
МГТУ им. Н.Э. Баумана

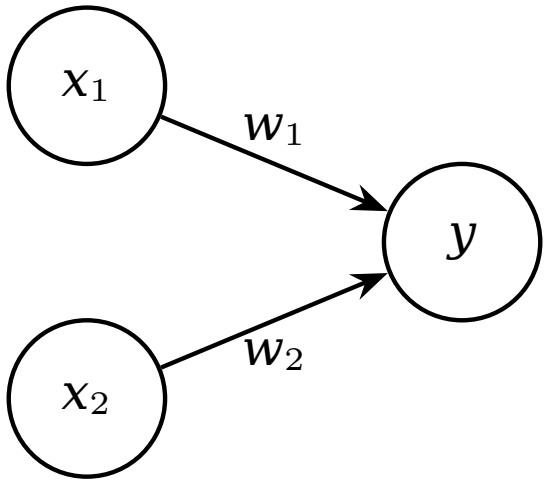
Содержание

Что такое персептрон

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

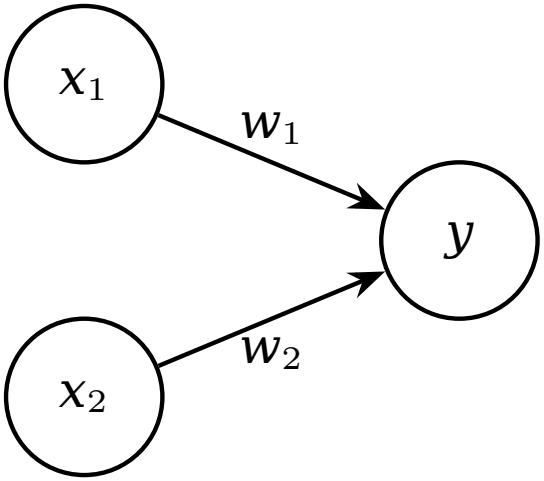
❋ Персептрон принимает несколько входных сигналов и формирует один выходной. В данной модели:

- 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!

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 AND-функции.

★ 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 OR-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)
```

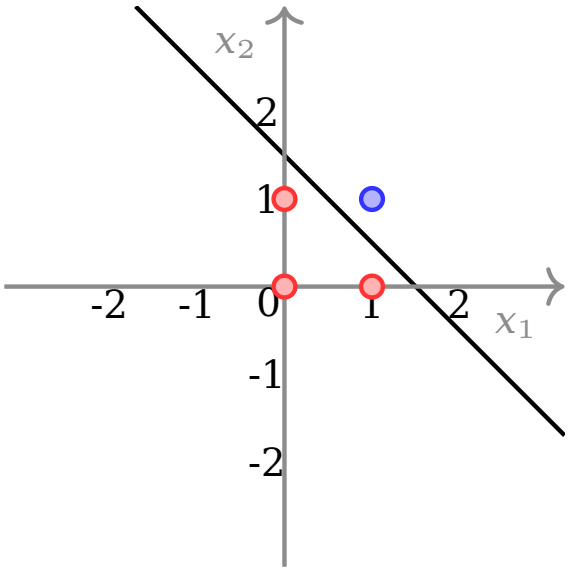
Функция NAND:

```
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)
```

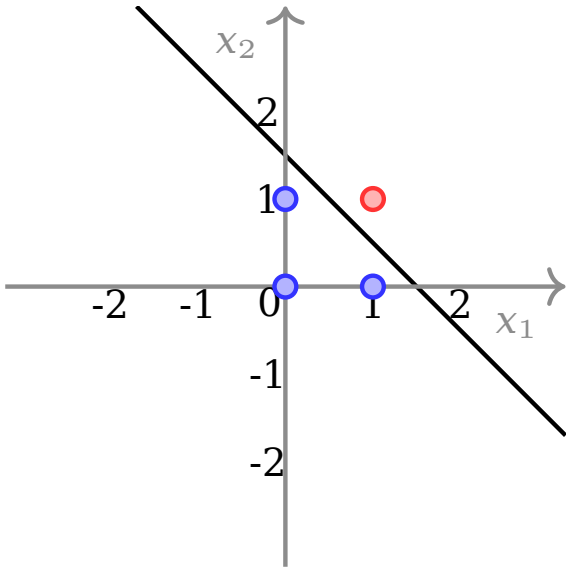
❋ Все три логические функции реализуются одной и той же структурой — различие только в весах и смещении.

Графическое представление логических функций

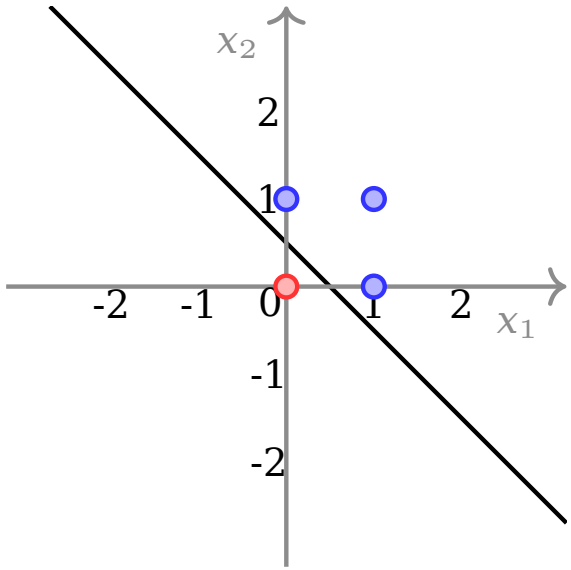
Три основные логические операции, которые может реализовать персептрон, представлены ниже. Каждая из них разделяет пространство прямой $w_1 x_1 + w_2 x_2 + b = 0$ на два класса.



AND (И)



NAND (НЕ-И)



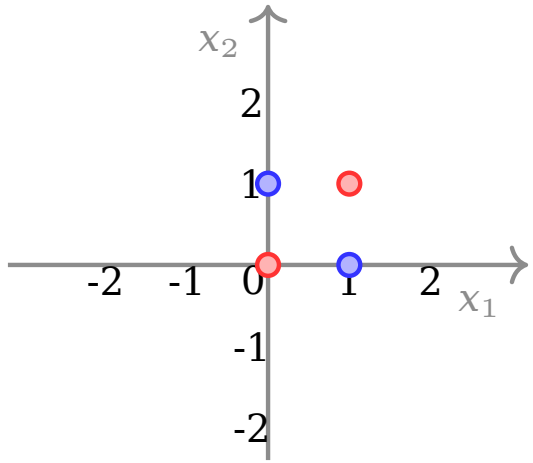
OR (ИЛИ)

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.

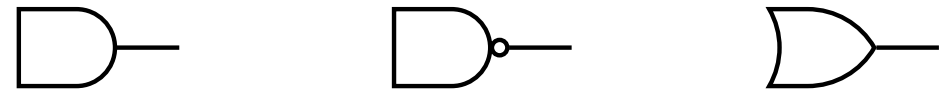
❋ Вывод. Однослойный персептрон способен реализовать лишь линейно разделимые функции (AND, OR, NAND и др.). Для нелинейных задач, таких как XOR, требуется *многослойная нейронная сеть*, способная формировать произвольные — в том числе криволинейные — разделяющие поверхности.

Многослойный персептрон

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

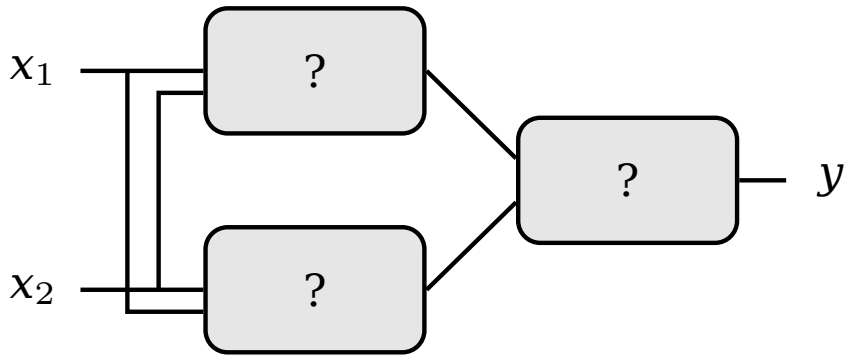
Один из способов построения XOR — это комбинация известных логических элементов: **AND**, **NAND** и **OR**.

Базовые логические элементы:

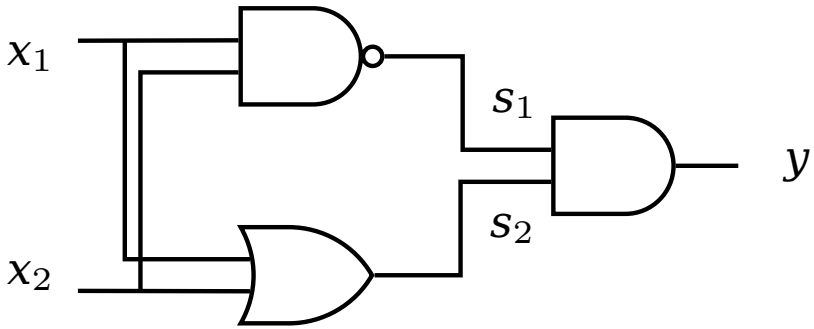


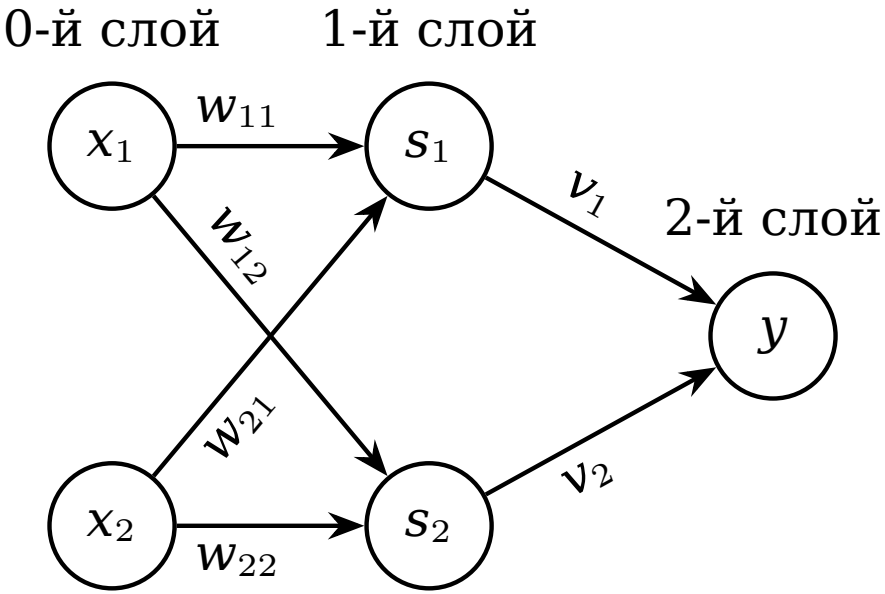
AND NAND OR

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

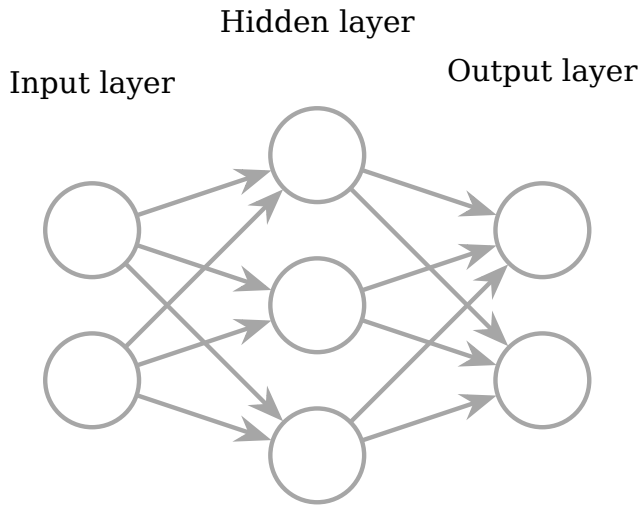
В предыдущей главе мы изучили персептрон. Хорошая новость — даже сложные функции можно выразить с его помощью. Плохая новость — веса приходилось подбирать вручную.

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.

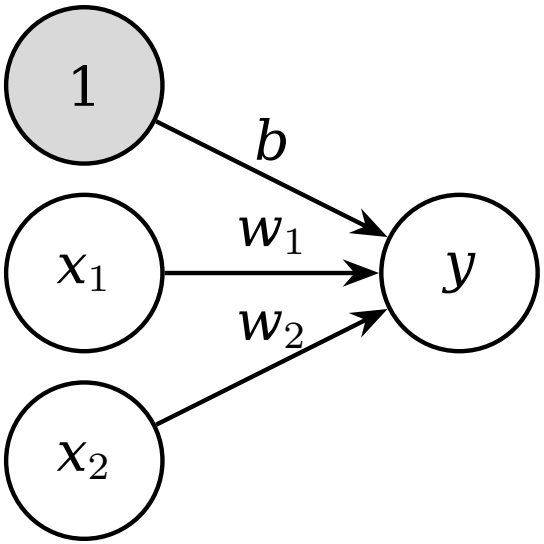


Повторим простейший персептрон

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.



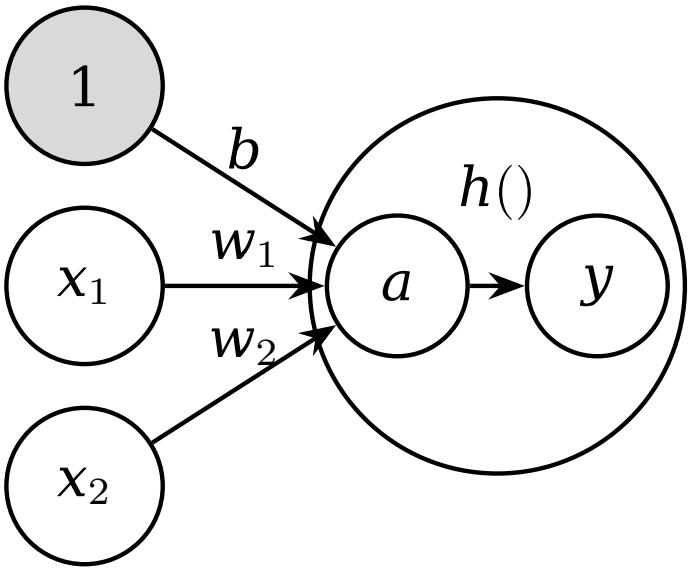
Чтобы явно показать b, можно представить его как входной сигнал со значением 1, соединённый с весом b, как показано на рисунке слева. Мы закрасили этот узел в серый цвет, чтобы отличать от обычных входов.

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 **активационной функцией** (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:

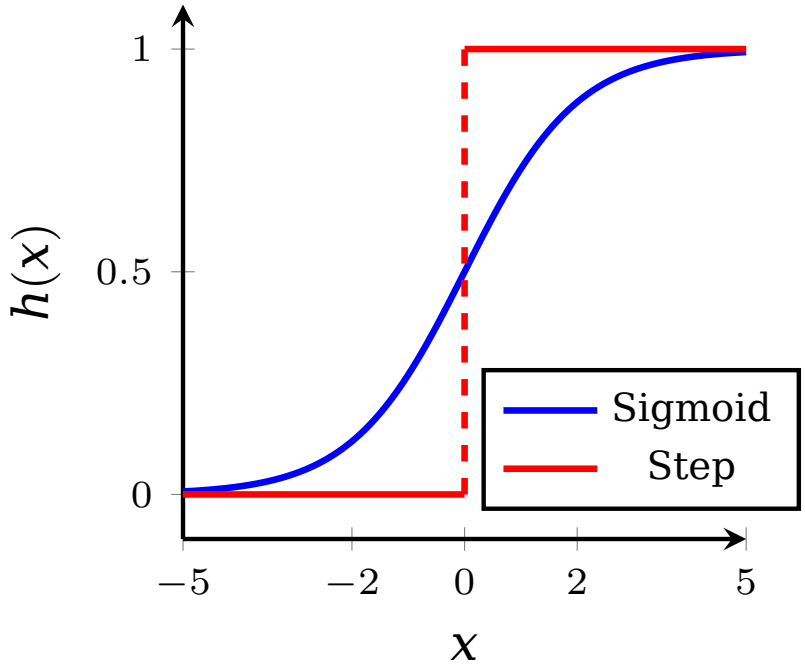
$$h(1.0) \approx 0.731, \quad h(2.0) \approx 0.880$$

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.

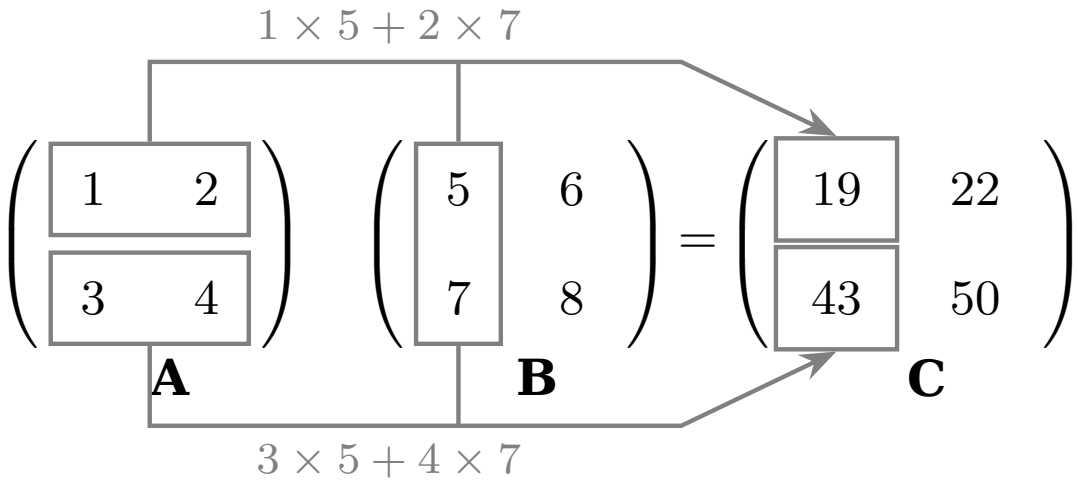
Например, если использовать $h(x) = c \cdot x$, то для трёх слоёв:

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

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.

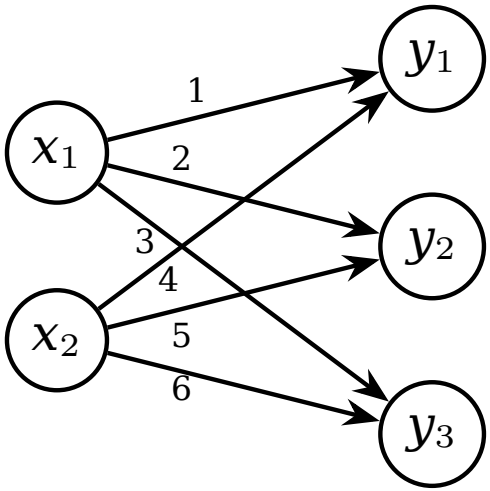
Умножение матриц




```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]]
```

Реализация нейросети через умножение матриц

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



$$\mathbf{X} \cdot \mathbf{W} = \mathbf{Y}$$

$$\mathbf{W} = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

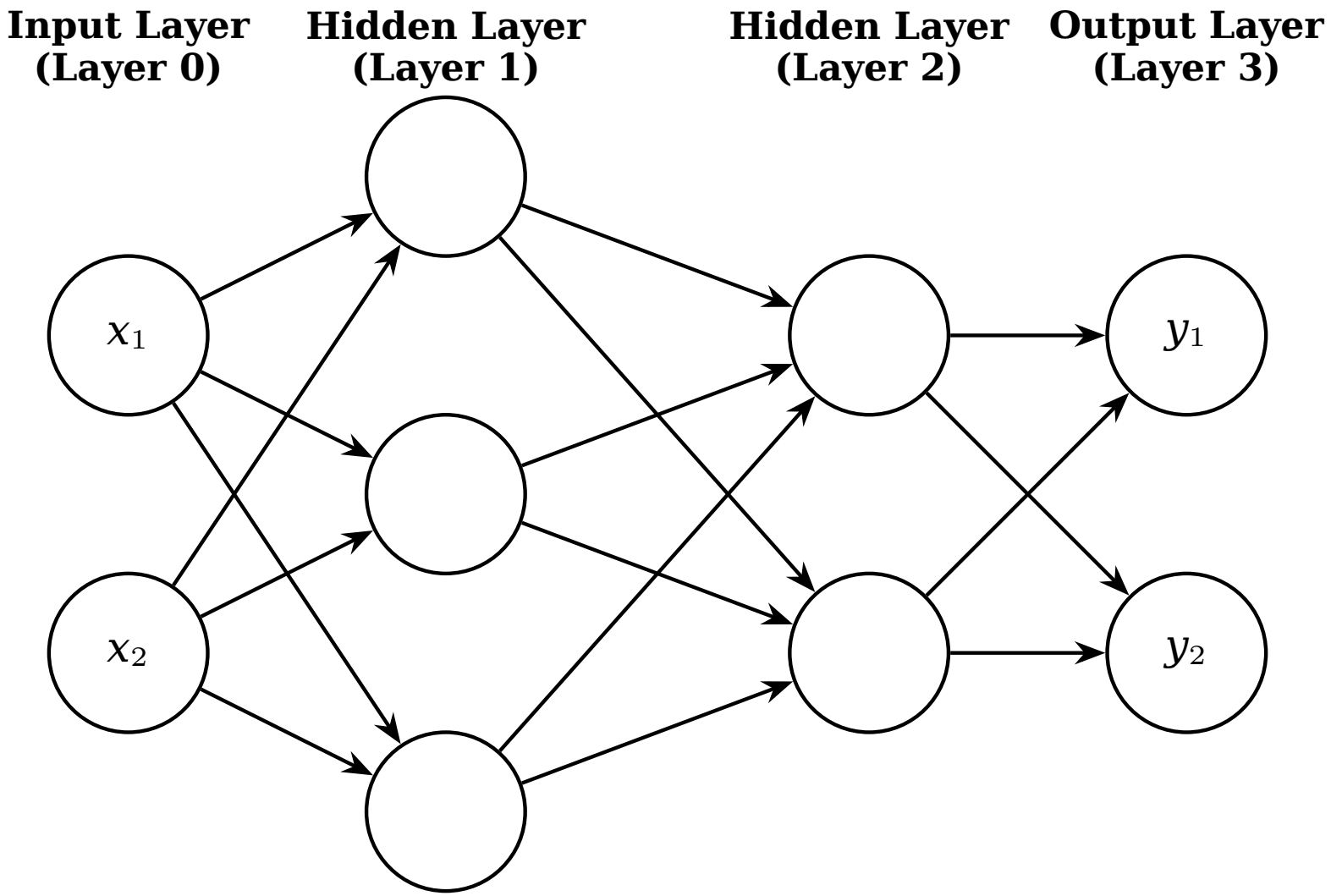
$$2 \quad 2 \times 3 \quad 3$$

same same

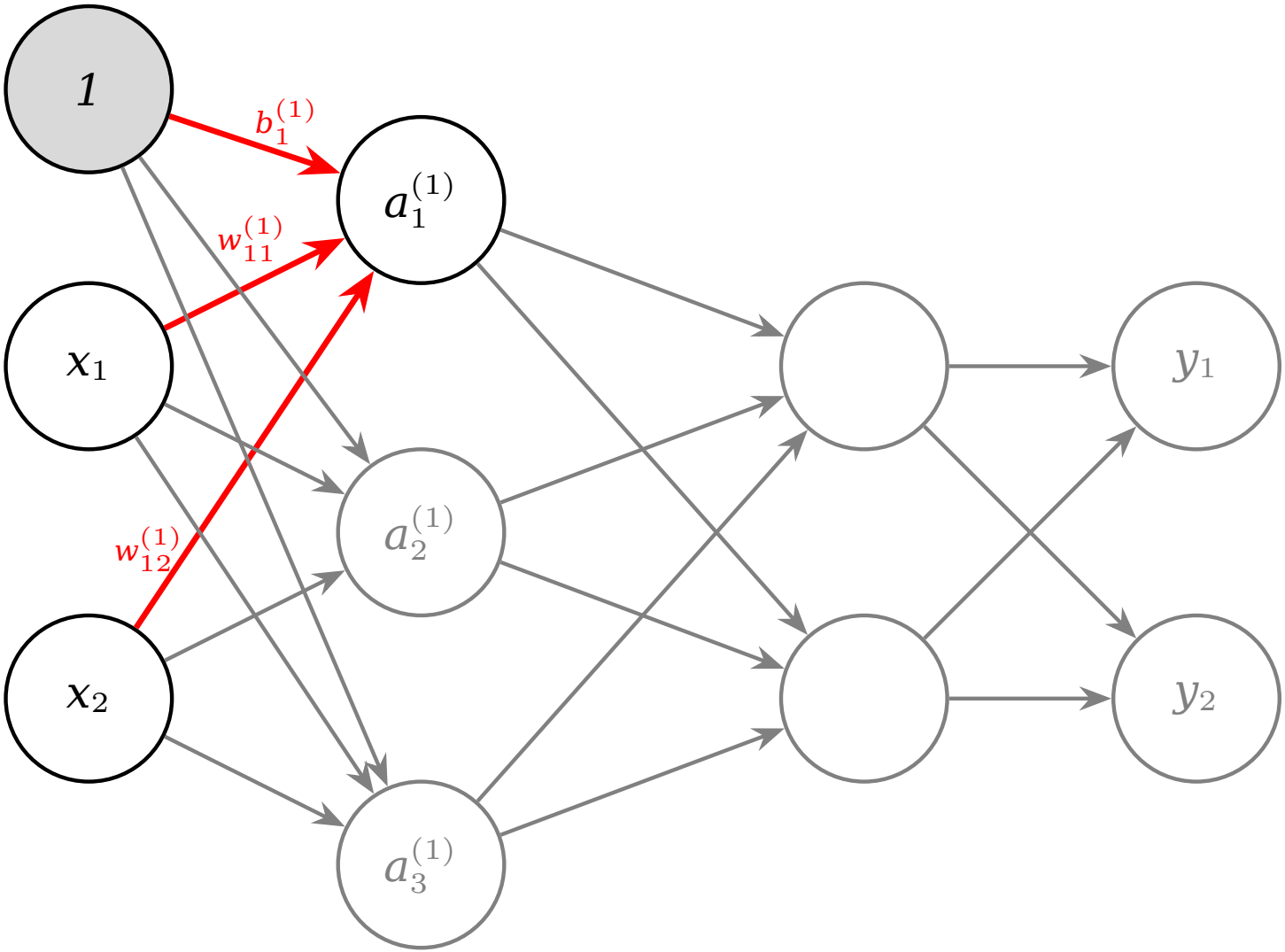
Важно, чтобы размеры матриц были согласованы: число столбцов в $\mathbf{X}$ должно совпадать с числом строк в $\mathbf{W}$.

```
1  X = np.array([1, 2])
2  print(X.shape)  # (2,)
3
4  W = np.array([[1, 3, 5], [2, 4, 6]])
5  print(W.shape)  # (2, 3)
6
7  Y = np.dot(X, W)
8  print(Y)        # [ 5 11 17 ]
```

Реализация 3-х слойной нейронной сети



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

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

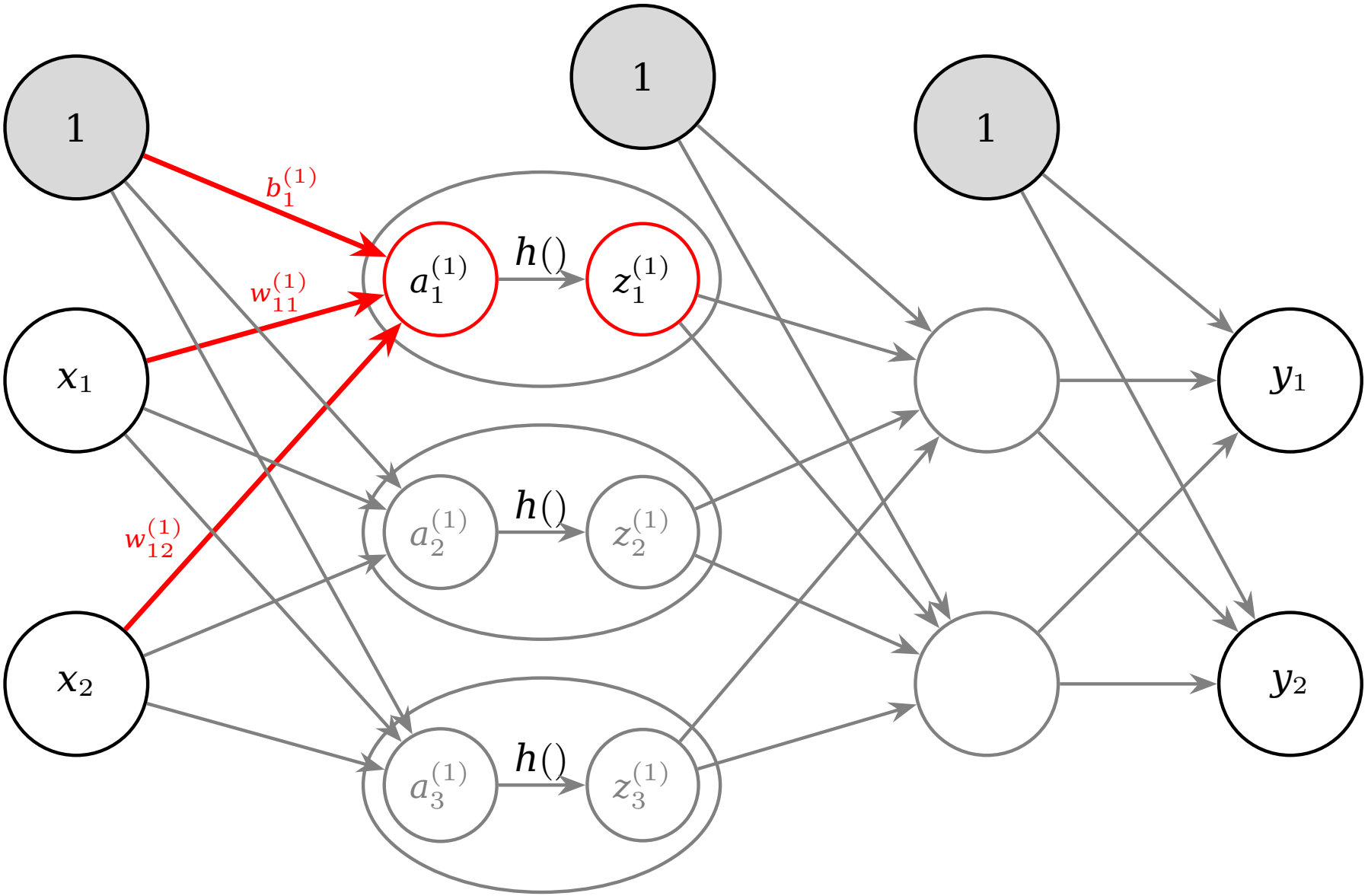
$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

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



Два типа выходного слоя

Слой выхода в нейронной сети может использовать разные функции активации, в зависимости от решаемой задачи. Основные случаи — задачи **регрессии** и **классификации**:

1. **Регрессия:** используется **тождественная функция** (identity function). Она передаёт входной сигнал напрямую на выход без изменений:

$$y = \sigma(a) = a \tag{9}$$

Подходит для прогнозирования непрерывных значений (например, веса, температуры и др.).

2. **Классификация:** используется **softmax-функция**, которая преобразует входной вектор a_k в вероятностное распределение:

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} \tag{10}$$

где k — номер класса, n — общее количество классов. Выходные значения y_k интерпретируются как вероятности принадлежности к каждому классу.

Итоги главы

- В нейронных сетях для скрытых слоёв применяются сглаженные нелинейные функции активации, такие как **sigmoid**.
- С помощью **массивов NumPy** можно эффективно реализовать архитектуру нейронной сети.
- Задачи машинного обучения делятся на два типа:
  - **регрессионные задачи** — предсказание числовых значений;
  - **классификационные задачи** — определение принадлежности к классу.
- Функции активации для выходного слоя:
  - в регрессионных задачах — **тождественная функция**;
  - в классификационных задачах — **softmax-функция**.

Содержание

Обучение нейронных сетей

❋ В традиционном программировании результат получается по заданным правилам:

правила + данные $\rightarrow$ **ответы**

❋ В машинном обучении всё наоборот:

данные + ответы $\rightarrow$ **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.

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.

Метод градиентного спуска

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

Метод градиента — 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:

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

- 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.

Реализация обучения нейронной сети

Обучение нейросети включает следующие ключевые шаги:

1. **Выбор mini-batch** Случайным образом выбирается подмножество обучающих данных (mini-batch). Цель — минимизировать ошибку на этом наборе.
2. **Вычисление градиента** Рассчитывается градиент функции потерь по всем параметрам сети — он указывает направление наибольшего уменьшения ошибки.
3. **Обновление параметров** Параметры (веса) обновляются в направлении, противоположном градиенту.
4. **Повторение** Шаги 1–3 повторяются до достижения сходимости.

В процессе обучения функция потерь постепенно уменьшается — это значит, что нейросеть действительно учится и приближается к оптимальному решению.

Итоги главы

- В машинном обучении данные делятся на обучающую и тестовую выборки.
- Нейронные сети обучаются на обучающей выборке, а обобщающая способность модели оценивается по тестовой.
- Цель обучения — минимизировать функцию потерь путём обновления весов.
- Численное дифференцирование — это способ приближённого вычисления градиента через конечные разности.
- С помощью численного дифференцирования можно вычислить градиенты весов.

Содержание

Стохастический градиентный спуск (SGD)

Стохастический градиентный спуск — это базовый и самый простой метод оптимизации в обучении нейросетей. Его формула:

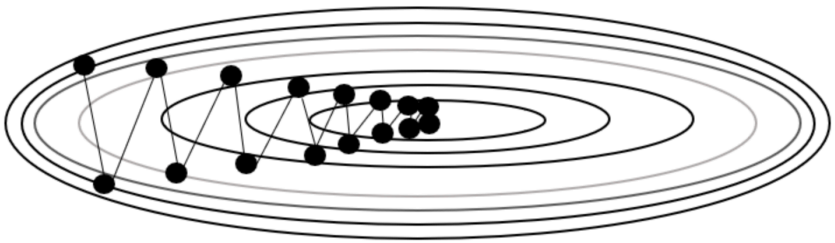
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial L}{\partial \boldsymbol{W}}$$

Обозначения:

- $\boldsymbol{W}$ — вектор параметров (весов);
- η — скорость обучения (обычно 0.01 или 0.001);
- $\frac{\partial L}{\partial \boldsymbol{W}}$ — градиент функции потерь по $\boldsymbol{W}$.

Суть метода: каждый шаг обновления происходит в направлении уменьшения ошибки. Однако:

- SGD может застревать в локальных минимумах;
- Имеет нестабильную траекторию в ”узких долинах”.



Метод Momentum

Метод Momentum использует идею «инерции» — как шарик, катящийся по чаше:

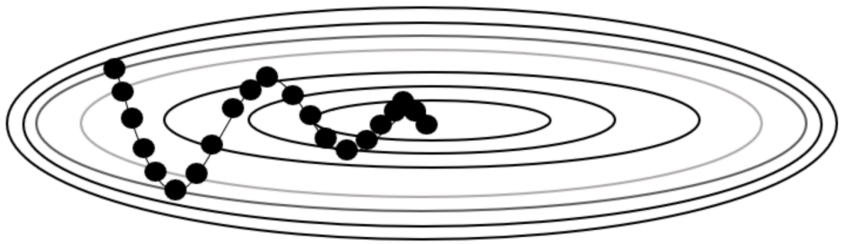
$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \eta \frac{\partial L}{\partial \boldsymbol{W}}, \quad \boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{v}$$

Пояснение:

- $\boldsymbol{v}$ — скорость (накопленное направление);
- α — коэффициент затухания, контролирует ”память” (обычно 0.9);
- Обновление зависит не только от текущего градиента, но и от истории движения.

Преимущества:

- Сглаживает обновления, избегает зигзагообразных траекторий;
- Позволяет быстрее двигаться по ”плоским” осям и замедляться в ”крутых”.



Метод AdaGrad

AdaGrad адаптирует скорость обучения для каждого параметра отдельно, учитывая их историю:

$$\boldsymbol{h} \leftarrow \boldsymbol{h} + \frac{\partial L}{\partial \boldsymbol{W}} \odot \frac{\partial L}{\partial \boldsymbol{W}}, \quad \boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\eta}{\sqrt{\boldsymbol{h}}} \frac{\partial L}{\partial \boldsymbol{W}}$$

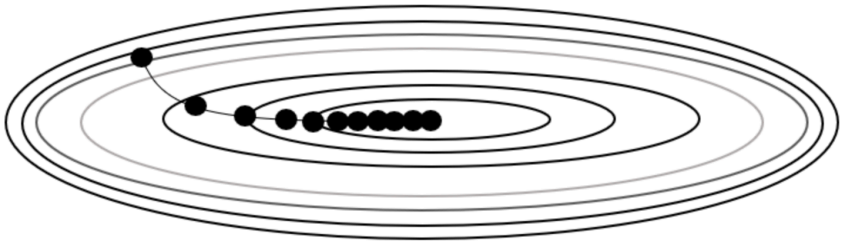
Обозначения:

- $\boldsymbol{h}$ — накопленные квадраты градиентов (поэлементно);
- ε — малая константа для избежания деления на ноль.

Ключевая идея:

- Параметры, часто обновляемые, получают меньшую скорость обучения;
- Это особенно полезно при разреженных признаках (sparse features).

Недостаток: скорость обучения может стать слишком маленькой со временем, и обучение ”замирает”.



Метод Adam (Adaptive Moment Estimation)

Adam объединяет преимущества Momentum и AdaGrad, вычисляя скользящее среднее градиента и его квадрата:

$$\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\frac{\partial L}{\partial \boldsymbol{W}}$$

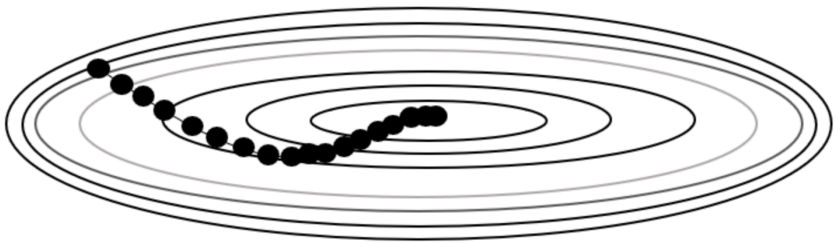
$$\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\left(\frac{\partial L}{\partial \boldsymbol{W}}\right)^2$$

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1-\beta_1^t}, \quad \hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1-\beta_2^t}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \varepsilon}$$

Преимущества:

- Быстрая сходимость, особенно при шумных градиентах;
- Подходит для сложных моделей и нестабильных обучающих данных.



Опыт настройки начальных весов (инициализация)

Корректная инициализация весов играет ключевую роль в стабильном обучении нейросети.

1. **Xavier (Glorot) инициализация:**
   - Подходит для функций активации: `sigmoid`, `tanh` и других S-образных;
   - Основана на симметрии функции вокруг 0;
   - Если число нейронов на входе n, стандартное отклонение:

$$\sigma = \sqrt{\frac{1}{n}}$$

   - Обеспечивает одинаковую дисперсию активаций между слоями.

2. **He (Kaiming) инициализация:**
   - Предназначена для ReLU и её производных;
   - Если входной слой имеет n нейронов, используется:

$$\sigma = \sqrt{\frac{2}{n}}$$

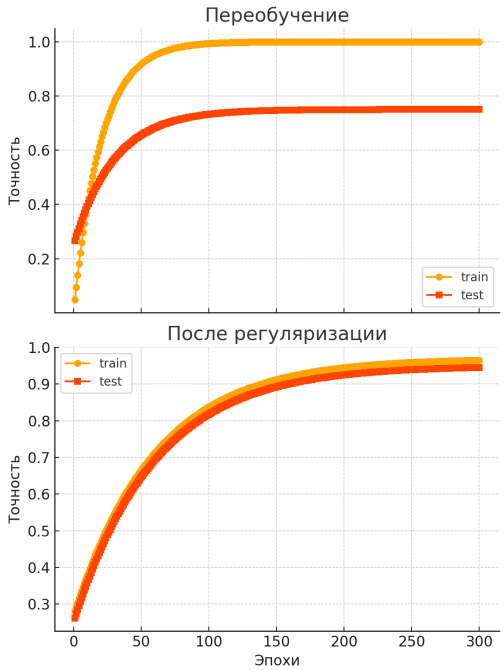
   - Обеспечивает более широкое распределение значений и лучшее распространение градиента.

Подавление переобучения: 1. L2-регуляризация

Переобучение — частая проблема в машинном обучении. Модель хорошо запоминает обучающие данные, но плохо обобщает на новые примеры.

Причины переобучения:

- Слишком много параметров;
- Недостаток обучающих данных.



❋ **L2-регуляризация** — добавление штрафа за слишком большие веса:

$$L' = L + \frac{1}{2}\lambda \boldsymbol{W}^2$$

- L — исходная функция потерь;
- $\boldsymbol{W}$ — вектор весов;
- λ — гиперпараметр регуляризации.

Механизм:

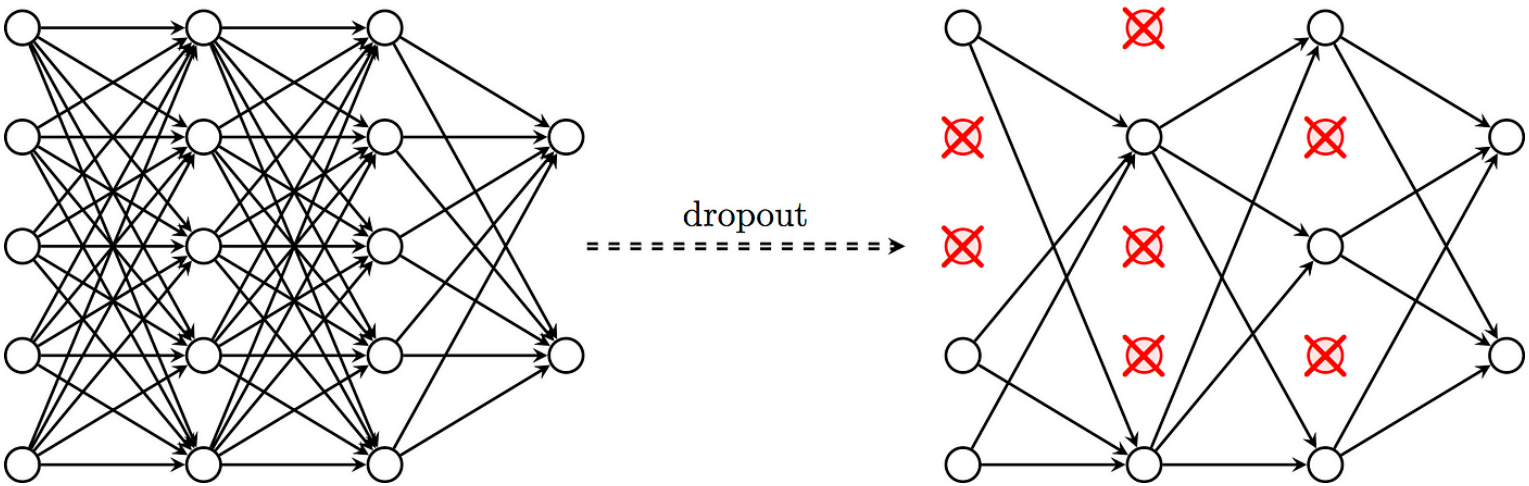
- Улучшает обобщающую способность модели;
- В градиент добавляется слагаемое: $\lambda \boldsymbol{W}$.

Подавление переобучения: 2. Dropout

❋ **Dropout** — это эффективный метод , особенно для сложных моделей. В отличие от L2-регуляризации, Dropout может предотвратить переобучение даже в мощных нейросетях.

Принцип работы:

- В процессе обучения случайно удаляются нейроны скрытого слоя;
- Каждый раз выбирается новый набор нейронов для удаления.



Оптимизация гиперпараметров

Гиперпараметры — параметры, не обучаемые градиентом, но сильно влияющие на итоговую точность модели. Типичные примеры: *скорость обучения, размер батча, регуляризация, число нейронов*.

Цель: подобрать наилучшие значения гиперпараметров.

Поэтапная стратегия:

Шаг 0: Определить диапазон значений;

Шаг 1: Случайно выбрать один набор параметров;

Шаг 2: Обучить модель на **train**, оценить точность на **val**;

Шаг 3: Повторить 1–2 (например, 100 раз), затем сузить диапазон и повторить.

Разделение данных:

- **Train** — для обучения параметров модели;
- **Validation** — для выбора гиперпараметров;
- **Test** — используется один раз, только для финальной оценки.

Итоги главы

- Помимо SGD, мы рассмотрели методы обновления параметров: Momentum, AdaGrad, Adam и др.
- Правильная инициализация весов играет ключевую роль в эффективном обучении нейросетей.
- Среди популярных стратегий инициализации — Xavier и He инициализация.
- Для подавления переобучения применяются методы регуляризации: L2-регуляризация и Dropout.
- Эффективный поиск гиперпараметров возможен с помощью постепенного сужения диапазона значений.

Перспектива

Что дальше? В дальнейшем мы изучим, как машинное обучение может применяться для решения задач в области **теплофизики и теплопереноса**.

Возможные направления:

- Предсказание теплопроводности материалов с использованием нейросетей;
- Построение surrogate-моделей для ускорения численного моделирования;
- Инверсия параметров: извлечение теплофизических свойств по экспериментальным данным;
- Применение графовых нейросетей для моделирования теплопереноса в наноструктурах;
- Оптимизация структур (например, метаповерхностей) с помощью ML-алгоритмов.

Интеграция физики и машинного обучения открывает новые горизонты в численном моделировании и интеллектуальном прогнозировании!