

Нейронные сети (NN) с элементами Deep Learning

Санкт-Петербургский государственный университет
Кафедра статистического моделирования

18 октября 2025

Что такое нейронные сети?

- Нейронные сети — это семейство моделей машинного обучения, которые стали доминирующим подходом во многих областях с 2012 года.
- Идея была предложена еще в 1970-х, но технические возможности для обучения больших сетей появились только в начале 2010-х.
- Совокупность нейросетевых подходов называется **глубинным обучением** (Deep Learning).

End-to-end обучение

- Переход от сложных пайплайнов, где каждая часть решает свою подзадачу, к обучению всей системы как единого целого.
- Например, вся модель от сырых данных до конечного ответа обучается совместно.

Обучение представлений (Representation Learning)

- Автоматическое создание информативных признаков из данных, часто неразмеченных.
- Это позволяет отказаться от ручного конструирования признаков экспертами.

Полносвязные нейросети

- Самая простая архитектура нейронных сетей.
- Состоит из слоев, в каждом из которых есть **нейроны**.
- Нейроны одного слоя соединены со всеми нейронами следующего слоя.

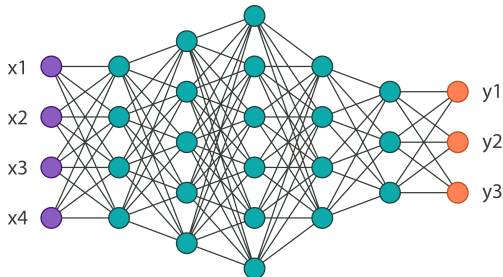


Рис.: Пример структуры полносвязной нейронной сети.

Из чего состоит нейрон?

- **Входы (x_i):** значения, поступающие от предыдущего слоя.
- **Веса (w_i):** параметры, которые показывают важность каждого входа.
- **Смещение (b):** дополнительный обучаемый параметр.
- **Сумматор:** вычисляет взвешенную сумму входов:
$$z = \sum_i w_i x_i + b.$$
- **Функция активации ($f(z)$):** нелинейная функция, которая определяет выходной сигнал нейрона. Примеры: Sigmoid, ReLU.

Что такое нейрон? Математическая модель

Нейрон — это базовая вычислительная единица сети. Он выполняет два последовательных действия:

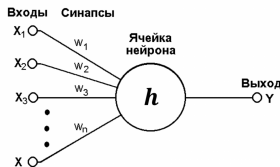
❶ **Аффинное преобразование:**

вычисляет взвешенную сумму своих входов (x_i) с весами (w_i) и добавляет смещение (b). Это называется **логитом**.

❷ **Нелинейное**

преобразование: применяет к результату **функцию активации** $f(z)$.

Формальный нейрон



$$h = \sum_i x_i * w_i \quad y = f(h)$$

Логит: $z = w_1x_1 + w_2x_2 + \dots + w_nx_n + b = \sum_{i=1}^n w_ix_i + b$
Выход нейрона: $a = f(z) = f(\sum_{i=1}^n w_ix_i + b)$

Векторная форма для нейрона

Для удобства и эффективности вычислений операции записываются в векторной форме.

- Входные данные: вектор $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$
- Веса: вектор $\mathbf{w} = [w_1, w_2, \dots, w_n]^T$
- Смещение: скаляр b

Логит: $z = \mathbf{w}^T \mathbf{x} + b$

Выход: $a = f(\mathbf{w}^T \mathbf{x} + b)$

Ключевая идея

Обучение нейрона — это подбор таких векторов весов \mathbf{w} и смещений b , которые минимизируют ошибку на обучающих данных.

Слой из k нейронов, принимающий на вход n значений, можно представить в виде матричных операций.

- **Матрица весов W :** размер $k \times n$. Каждая строка — это вектор весов \mathbf{w}_j^T для j -го нейрона.
- **Вектор входов \mathbf{x} :** размер $n \times 1$.
- **Вектор смещений \mathbf{b} :** размер $k \times 1$.
- **Вектор логитов \mathbf{z} :** размер $k \times 1$.
- **Вектор активаций \mathbf{a} :** размер $k \times 1$.

$$\mathbf{z} = W\mathbf{x} + \mathbf{b}$$
$$\mathbf{a} = f(\mathbf{z})$$

Зачем нужна нелинейность? Без нелинейных функций активации любая глубокая нейронная сеть была бы эквивалентна одному линейному слою.

- Пусть $f(z) = z$ (линейная активация).
- Выход первого слоя: $a_1 = W_1x + b_1$.
- Выход второго слоя: $a_2 = W_2a_1 + b_2 = W_2(W_1x + b_1) + b_2$.
- Раскрыв скобки, получаем: $a_2 = (W_2W_1)x + (W_2b_1 + b_2)$.
- Это эквивалентно одному слою с весами $W' = W_2W_1$ и смещением $b' = W_2b_1 + b_2$.

Вывод

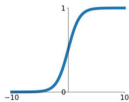
Именно нелинейность позволяет сети аппроксимировать сложные, нелинейные зависимости в данных.

Сигмоида (Sigmoid) Проблема: затухание градиента.

ReLU (Rectified Linear Unit) Стандарт де-факто для скрытых слоев.

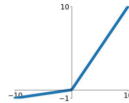
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



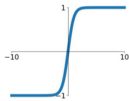
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$

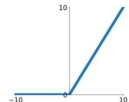


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ReLU

$$\max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

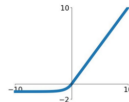


Рис.: Различные функции активации

Функция потерь (Loss Function)

Чтобы обучать сеть, нам нужна мера того, насколько она ошибается в предсказании. Эту меру называют функцией потерь $L(\hat{y}, y)$, где:

- \hat{y} — предсказание сети.
- y — истинное значение.

Примеры:

- **Для регрессии:** Среднеквадратичная ошибка (MSE)

$$L_{MSE} = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2$$

- **Для бинарной классификации:** Бинарная кросс-энтропия

$$L_{BCE} = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$$

Наша цель — найти такие параметры сети (веса W и смещения b), которые минимизируют функцию потерь L .

- **Идея:** двигаться в направлении, противоположном градиенту функции потерь (т.е. в сторону наискорейшего убывания).
- **Градиент** ∇L — это вектор частных производных L по всем параметрам сети.

Правило обновления весов для одного параметра w :

$$w := w - \eta \frac{\partial L}{\partial w}$$

где η — **скорость обучения** (learning rate).

Как эффективно посчитать производную $\frac{\partial L}{\partial w}$ для веса, который находится в глубоком слое сети?

Метод обратного распространения ошибки (Backpropagation) — это, по сути, рекурсивное применение **цепного правила** (chain rule) из матанализа для вычисления градиентов.

Идея

- 1 Сначала вычисляется производная ошибки по выходу последнего слоя.
- 2 Затем эта ошибка движется назад, от слоя к слою.
- 3 На каждом слое вычисляется градиент по его параметрам и градиент по его входу, который передается дальше назад.

Пусть $L = \frac{1}{2}(\hat{y} - y)^2$, $\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x} + b)$.

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial \hat{y}} \frac{d\hat{y}}{dz} \frac{\partial z}{\partial \mathbf{w}} = (\hat{y} - y) \sigma'(z) \mathbf{x}.$$

Аналогично:

$$\frac{\partial L}{\partial b} = (\hat{y} - y) \sigma'(z).$$

Рассмотрим производную потерь L по весу $w_{ij}^{[l]}$ в слое l .

$$\frac{\partial L}{\partial w_{ij}^{[l]}} = \frac{\partial L}{\partial a^{[L]}} \frac{\partial a^{[L]}}{\partial z^{[L]}} \cdots \frac{\partial z^{[l+1]}}{\partial a^{[l]}} \frac{\partial a^{[l]}}{\partial z^{[l]}} \frac{\partial z^{[l]}}{\partial w_{ij}^{[l]}}$$

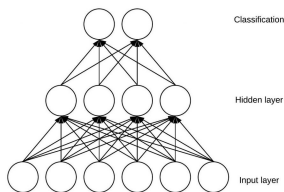
Это выглядит сложно, но на практике вычисляется последовательно:

- ❶ **Forward Pass:** Вычисляем все активации $a^{[l]}$ до самого конца.
- ❷ **Backward Pass:**
 - Вычисляем градиент для последнего слоя $\frac{\partial L}{\partial z^{[L]}}$.
 - Рекурсивно вычисляем $\frac{\partial L}{\partial z^{[l]}}$ через $\frac{\partial L}{\partial z^{[l+1]}}$.
 - Используя $\frac{\partial L}{\partial z^{[l]}}$, находим градиенты по параметрам этого слоя: $\frac{\partial L}{\partial W^{[l]}}$ и $\frac{\partial L}{\partial b^{[l]}}$.

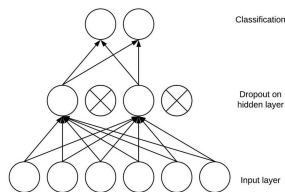
Значимую роль играет начальная инициализация весов сети.

- Плохая инициализация \rightarrow затухание/взрыв градиентов.
- Эвристический подход: случайные числа.
- Классические схемы: **Xavier/Glorot** (для \tanh):
распределение с дисперсией $\sim \frac{2}{n_{in}+n_{out}}$.
- **Kaiming He** (для ReLU): дисперсия $\sim \frac{2}{n_{in}}$.

- L_2 (или L_1) весовая регуляризация: добавить $\frac{\lambda}{2} \sum_{\ell} \|W^{(\ell)}\|_F^2$ в цель \rightarrow дополнительный член $\lambda W^{(\ell)}$ в градиенте.
- Dropout (случайное зануление нейронов на обучении) и Batch normalization (контроль над средним и дисперсией).
- Ранняя остановка (early stopping) по валидационной выборке.



Without Dropout



With Dropout

- SGD — простой стохастический градиентный спуск.
- Моменты: $v \leftarrow \beta v + (1 - \beta) \nabla_{\theta} J$, $\theta \leftarrow \theta - \eta v$.
- Адаптивные методы: AdaGrad, RMSProp, Adam.