

# Методы машинного обучения

Шорохов С.Г.

кафедра математического моделирования и искусственного интеллекта

Лекция 3. Введение в глубокое обучение





# Глубокое обучение

Глубокое обучение представляет собой семейство методов машинного обучения, основывающихся на искусственных нейронных сетях.

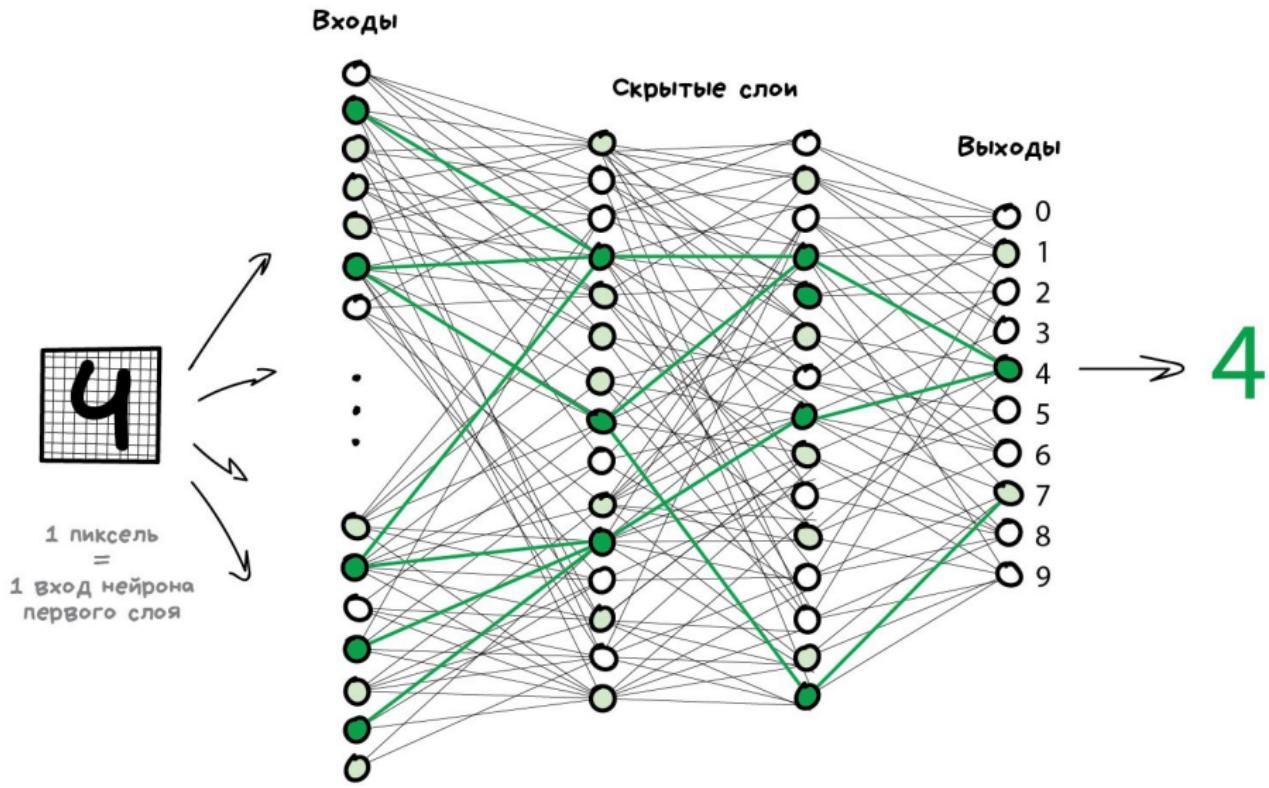
Математически **искусственная нейронная сеть** представляет собой направленный граф с нейронами в качестве вершин и связями между нейронами в виде ребер, причем вход для каждого нейрона является функцией взвешенной суммы выходов всех нейронов, связанных с ним входящими ребрами. Тогда выход нейронной сети равен

$$f(x; \theta) = \psi_d(\dots \psi_2(\psi_1(x))), \quad \psi_i(x) = \sigma_i(w^{(i)}x + b^{(i)})$$

Здесь каждый слой сети представляется **функцией активации**  $\sigma_i$  с аргументом в виде взвешенной суммы входных данных  $x$  с **весами**  $w^{(i)}$  и **смещениями**  $b^{(i)}$ . Число слоев  $d$  называется **глубиной** нейронной сети и количество нейронов в слое представляет собой **ширину** этого слоя.

Целью глубокого обучения является определение набора параметров сети  $\theta = \{w^{(i)}, b^{(i)}\}_{i=1}^d$ , который минимизирует **функцию потерь**  $\mathcal{L}(\theta)$ , определяющую качество модели при заданном наборе параметров  $\theta$ .

# Визуализация нейронной сети





Искусственные нейронные сети (ANN) или просто нейронные сети (NN) устроены по аналогии с биологическими нейронными сетями.

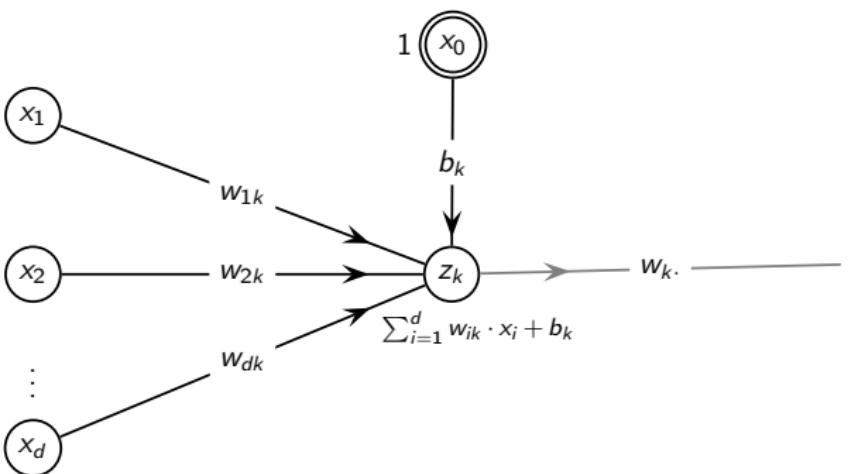
Реальный биологический нейрон (нервная клетка) состоит из дендритов, тела клетки и аксона, который ведет к т.н. синаптическим окончаниям. Нейрон передает информацию посредством электрохимических сигналов.

Когда на дендритах нейрона достаточная концентрация ионов, он генерирует электрический импульс вдоль своего аксона, называемый потенциалом действия, который, в свою очередь, активирует синаптические окончания, высвобождая больше ионов и, таким образом, вызывая поток информации к дендритам других нейронов.

Человеческий мозг имеет порядка 100 миллиардов нейронов, каждый из которых имеет от 1000 до 10000 связей с другими нейронами.

# Устройство искусственной нейронной сети

Искусственные нейронные сети состоят из абстрактных нейронов, которые пытаются имитировать реальные нейроны на очень высоком уровне. Их можно описать с использованием взвешенного ориентированного графа  $G = (V, E)$ , где каждый узел  $v_i \in V$  представляет собой нейрон, а каждое ориентированное ребро  $(v_i, v_j) \in E$  представляет соединение синаптического окончания (выхода) нейрона  $v_i$  с дендритом (входом) нейрона  $v_j$ . Вес ребра  $w_{ij}$  обозначает силу синапса (контакта между двумя нейронами).





# Искусственный нейрон

Искусственный нейрон действует как блок обработки, который сначала агрегирует входящие сигналы через взвешенную сумму, а затем применяет некоторую функцию для генерации выходных данных. Чистый вход нейрона  $z_k$  равен

$$\text{net}_k = b_k + \sum_{i=1}^d w_{ik} x_i = b_k + \mathbf{w}_k^T \mathbf{x}$$

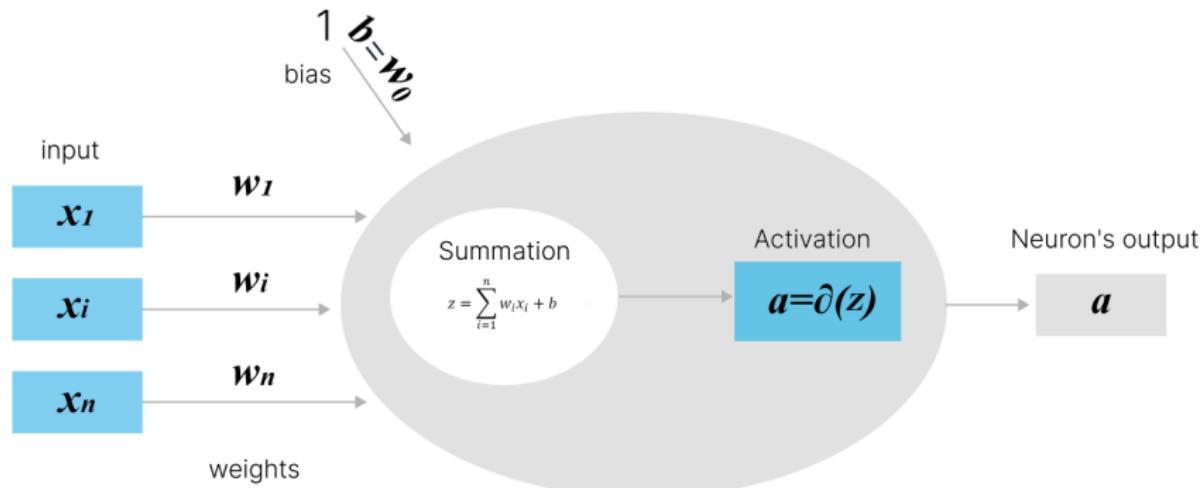
Для удобства можно рассматривать специальный нейрон смещения  $x_0$ , значение которого всегда фиксировано и равно 1, а вес от нейрона  $x_0$  к нейрону  $z_k$  равен  $b_k$ .

Наконец, выходное значение нейрона  $z_k$  задается как некоторая функция активации  $f(\cdot)$ , применяемая к чистому входу  $\text{net}_k$  нейрона  $z_k$

$$z_k = f(\text{net}_k)$$

# Схема работы искусственного нейрона

Искусственный нейрон представляет собой систему из двух компонентов – сумматора и функции активации.





# Теорема представления Колмогорова-Арнольда

А.Н.Колмогоров и В.И.Арнольд (в 1956–1958 гг) доказали, что если  $f$  – это многомерная непрерывная функция, то  $f$  можно записать в виде конечной композиции непрерывных функций одной переменной и бинарной операции сложения, а именно,

$$f(x_1, \dots, x_n) = \sum_{j=1}^{2n+1} g_j \left( \sum_{i=1}^n h_{ij}(x_i) \right),$$

где  $g_j$  и  $h_{ij}$  – непрерывные функции одной переменной и функции  $h_{ij}$  не зависят от выбора функции  $f$ . Эта теорема тесно связана с т.н. 13-й проблемой Гильберта.

Теорема Колмогорова-Арнольда может быть интерпретирована, что любая непрерывная функция  $n$  переменных может быть представлена при помощи нейронной сети с двумя слоями.



# Универсальная теорема аппроксимации

Универсальная теорема аппроксимации (Дж. Цыбенко, 1989) утверждает, что искусственная нейронная сеть прямого распространения (feed-forward) с одним скрытым слоем может аппроксимировать любую непрерывную функцию многих переменных с любой точностью.

Более точно, если дана любая непрерывная функция действительных переменных  $f$  на  $[0, 1]^n$  и  $\varepsilon > 0$ , то существуют векторы  $\mathbf{w}_1, \dots, \mathbf{w}_N, \boldsymbol{\alpha}, \boldsymbol{\beta}$  и параметризованная функция  $G(\cdot, \mathbf{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) : [0, 1]^n \rightarrow \mathbb{R}$  вида

$$G(\mathbf{x}, \mathbf{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \sum_{i=1}^N \alpha_i \varphi(\mathbf{w}_i^T \mathbf{x} + \beta_i), \quad \varphi(\xi) = \frac{1}{1 + e^{-\xi}},$$

такая, что для всех  $x \in [0, 1]^n$  выполняется условие

$$|G(\mathbf{x}, \mathbf{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) - f(\mathbf{x})| < \varepsilon,$$

$\mathbf{w} = (\mathbf{w}_1, \dots, \mathbf{w}_N)$ ,  $\mathbf{w}_i \in \mathbb{R}^n$  – веса между входными нейронами и нейронами скрытого слоя,  $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_N)$  – веса между связями от нейронов скрытого слоя и выходным нейроном,  $\boldsymbol{\beta} = (\beta_1, \dots, \beta_N)$  – смещения для нейронов входного слоя.



# Функции активации

Функция активации преобразует результат агрегации входящие сигналов нейрона в определенный диапазон значений, например, интервал  $(0, 1)$ .

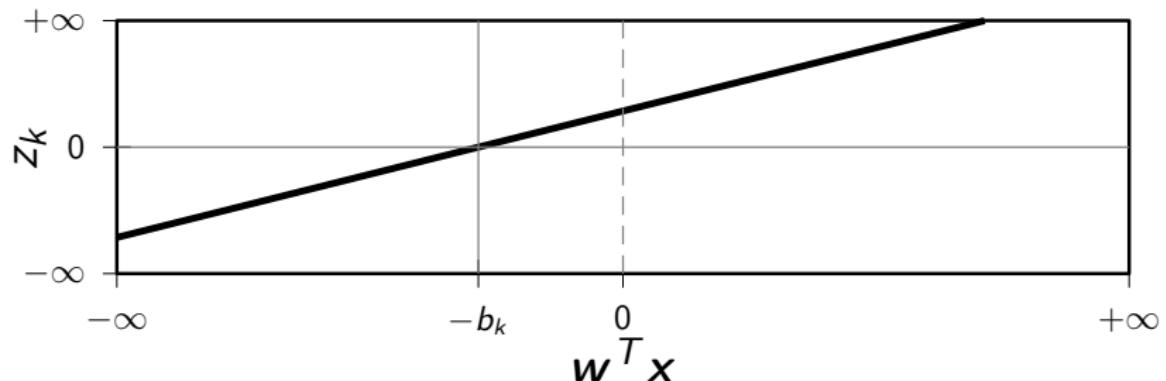
В биологических нейронных сетях функция активации в наиболее простой форме является бинарной – то есть нейрон либо возбуждается, либо нет.

Функции активации должны обладать следующими свойствами:

- **нелинейность:** Функция активации необходима для введения нелинейности в нейронные сети. Если функция активации не применяется, нейронная сеть будет действовать как линейная регрессия с ограниченной способностью к обучению. Только нелинейные функции активации позволяют нейронным сетям решать задачи аппроксимации нелинейных функций.
- **прохождение градиента:** Функции активации должны быть способными пропускать градиент, чтобы было возможно оптимизировать параметры сети градиентными методами, в частности использовать алгоритм обратного распространения ошибки.

# Линейная (тождественная) функция активации

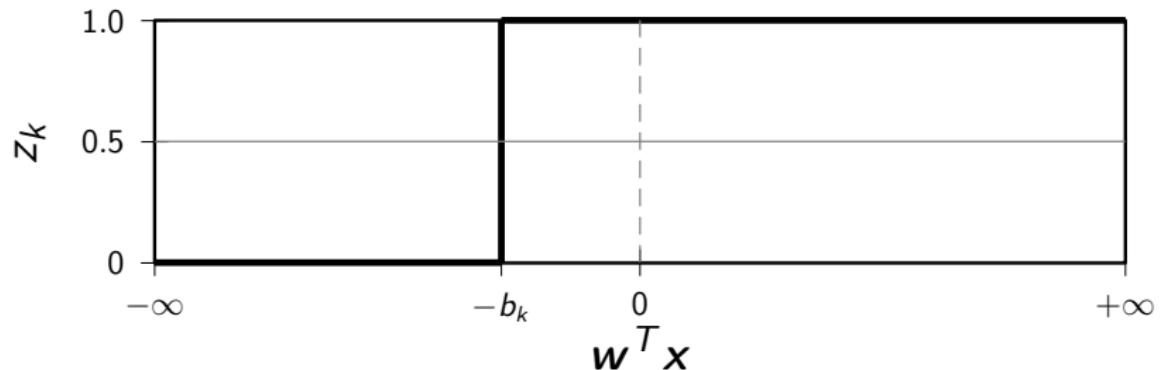
$$f(\text{net}_k) = \text{net}_k \Rightarrow \frac{\partial f(\text{net}_k)}{\partial \text{net}_k} = 1$$



Линейная функция активации применяется во входном слое и выходном слое (для задачи регрессии). В скрытых слоях линейная функция активации, как правило, не применяется, так как в этом случае выходной сигнал нейронной сети становится линейной функцией входных сигналов.

# Ступенчатая (пороговая) функция активации

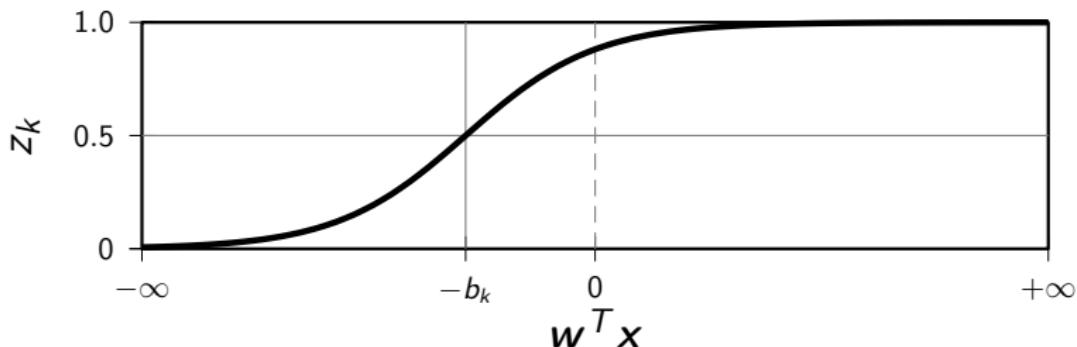
$$f(\text{net}_k) = \begin{cases} 0, & \text{net}_k \leq 0 \\ 1, & \text{net}_k > 0 \end{cases} \Rightarrow \frac{\partial f(\text{net}_k)}{\partial \text{net}_k} = 0$$



Пороговая функция активации использовалась в перцептронах — первых нейронных сетях. В настоящее время пороговая функция активации практически не используется, поскольку не может быть использована для оптимизации параметров нейронной сети методом градиентного спуска.

# Функция активации сигмоида

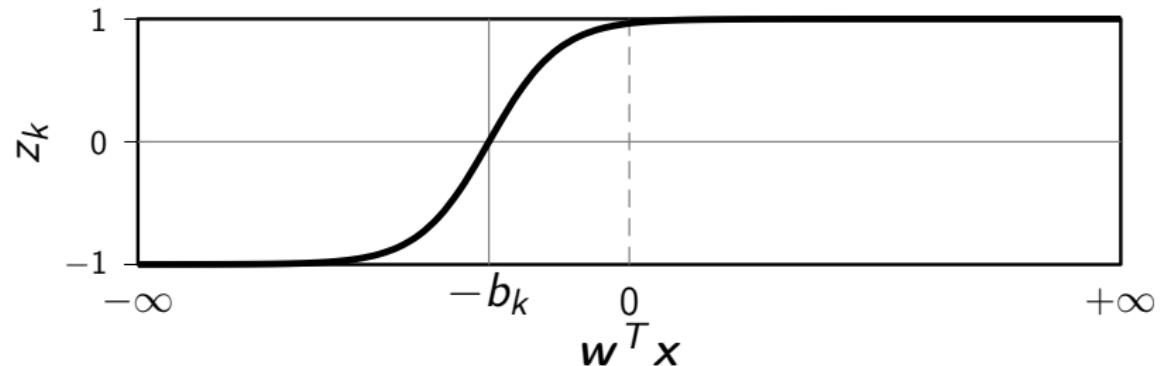
$$f(\text{net}_k) = \frac{1}{1 + \exp(-\text{net}_k)} \Rightarrow \frac{\partial f(\text{net}_k)}{\partial \text{net}_k} = f(\text{net}_k)(1 - f(\text{net}_k))$$



Функция активации сигмоида является гладкой и нелинейной функцией, производная сигмоиды также гладкая функция, отличная от нуля. Сигмоида используется в задачах бинарной классификации в выходном слое. Сигмоида может быть использована и в скрытых слоях, но при больших значениях аргумента производная будет близка к нулю и нейронная сеть может плохо обучаться. Выход сигмоиды не центрирован относительно нуля, что является нежелательным

# Функция активации гиперболический тангенс

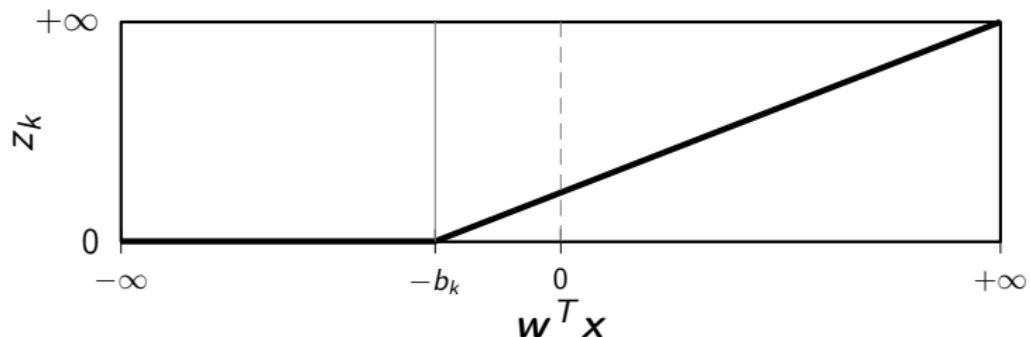
$$f(\text{net}_k) = \tanh(\text{net}_k) = \frac{\exp(\text{net}_k) - \exp(-\text{net}_k)}{\exp(\text{net}_k) + \exp(-\text{net}_k)} \Rightarrow \frac{\partial f(\text{net}_k)}{\partial \text{net}_k} = 1 - f(\text{net}_k)^2$$



Функция активации гиперболический тангенс имеет характеристики, аналогичные сигмоиде. В отличие от сигмоиды значения  $\tanh$  центрированы относительно 0 и градиент  $\tanh$  больше, чем у сигмоиды. Аналогично сигмоиде, гиперболическому тангенсу свойственная проблема исчезновения градиента при больших значениях аргумента.

# Функция активации ReLU

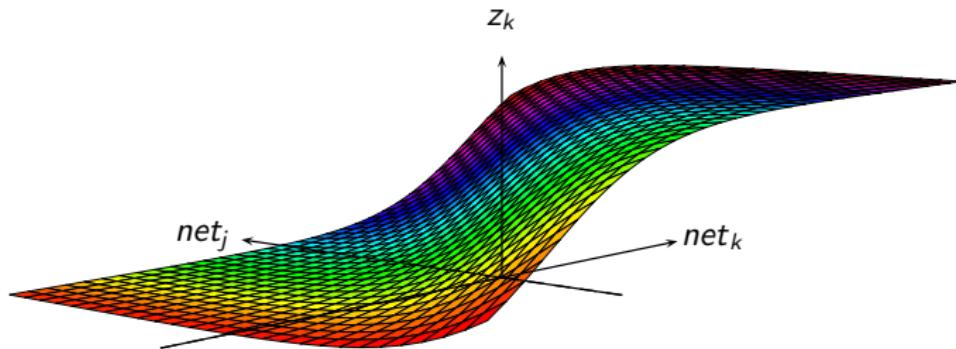
$$f(\text{net}_k) = \max(0, \text{net}_k) = \begin{cases} 0, & \text{net}_k \leq 0 \\ \text{net}_k, & \text{net}_k > 0 \end{cases} \Rightarrow \frac{\partial f(\text{net}_k)}{\partial \text{net}_k} = \begin{cases} 0, & \text{net}_k \leq 0 \\ 1, & \text{net}_k > 0 \end{cases}$$



Использование **функции активации ReLU** (Rectified Linear Unit) не требует выполнения ресурсоемких операций, таких как возведение в степень. Применение ReLU существенно повышает скорость сходимости стохастического градиентного спуска (в некоторых случаях до 6 раз) по сравнению с сигмоидой и гиперболическим тангенсом. Однако, нейроны с ReLU в процессе обучения могут выходить из строя.

# Функция активации софтмакс

$$f(\text{net}_j \mid \mathbf{net}) = \frac{\exp(\text{net}_j)}{\sum_{i=1}^p \exp(\text{net}_i)} \Rightarrow \frac{\partial f(\text{net}_j \mid \mathbf{net})}{\partial \text{net}_k} = \begin{cases} \text{net}_j (1 - \text{net}_j), & k = j \\ -\text{net}_j \text{net}_k, & k \neq j \end{cases}$$



Функция активации софтмакс является обобщением сигмоиды и, в отличие от других функций активации, зависит не только от чистого входа на одном нейроне  $\text{net}_j$ , но зависит от чистых входов всех других нейронов в слое  $\mathbf{net} = (\text{net}_1, \dots, \text{net}_p)^T$ . Функция софтмакс применяется в выходном слое при решении задачи многоклассовой классификации.



# Функции потерь (ошибок)

Функция потерь (loss function) или функция ошибок (error function) или функция издержек (cost function) искусственной нейронной сети позволяет оценить соответствие результата работы нейронной сети ожидаемому результату.

Функцию потерь используют для нахождения оптимальных параметров (весов и смещений) нейронов нейронной сети путем минимизации функции потерь при помощи градиентных или иных методов.

Функция потерь нейронной сети должна удовлетворять следующим условиям:

- функция потерь должна быть средним значением некоторой величины
- функция потерь не должна зависеть от каких-либо иных значений или параметров нейронной сети, кроме значений, выдаваемых нейронами выходного слоя нейронной сети



# Квадратичная функция потерь и ее градиент

Квадратичные потери (ошибки): при заданном входном векторе  $\mathbf{x} \in \mathbb{R}^d$  функция квадратичных потерь измеряет квадрат отклонения между прогнозируемым выходным вектором  $\mathbf{o} \in \mathbb{R}^p$  и истинным откликом  $\mathbf{y} \in \mathbb{R}^p$ , определяемым следующим образом:

$$\mathcal{E}_{\mathbf{x}} = \frac{1}{2} \|\mathbf{y} - \mathbf{o}\|^2 = \frac{1}{2} \sum_{j=1}^p (y_j - o_j)^2,$$

где  $\mathcal{E}_{\mathbf{x}}$  обозначает ошибку на входном векторе  $\mathbf{x}$ . Частная производная квадратичной функции ошибки по конкретному выходному нейрону  $o_j$  равна

$$\frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial o_j} = \frac{1}{2} 2 (y_j - o_j) (-1) = o_j - y_j$$

Для всех выходных нейронов мы можем записать это как

$$\frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial \mathbf{o}} = \mathbf{o} - \mathbf{y}$$

Квадратичная функция потерь часто используется в задачах регрессии.



# Кросс-энтропия как функция потерь

Кросс-энтропийная функция потерь (логарифмическая функция потерь) применяется для задач классификации с  $K$  классами  $\{c_1, c_2, \dots, c_K\}$  с количеством выходных нейронов  $p = K$  (с одним выходным нейроном на класс).

Каждый из классов кодируется как однокомпонентный (one-hot) вектор, при этом  $i$ -й класс  $c_i$  кодируется как  $i$ -й стандартный базисный вектор  $\mathbf{e}_i = (e_{i1}, e_{i2}, \dots, e_{iK})^T \in \{0, 1\}^K$ , где  $e_{ii} = 1$  и  $e_{ij} = 0$  для всех  $j \neq i$ .

Функция кросс-энтропии определяется как

$$\mathcal{E}_{\mathbf{x}} = - \sum_{i=1}^K y_i \ln o_i = - (y_1 \ln o_1 + \dots + y_K \ln o_K)$$

Обратите внимание, что только один элемент вектора  $\mathbf{y}$  равен 1, а остальные равны 0.



# Градиент кросс-энтропии

Частная производная кросс-энтропийной функции ошибок по отношению к конкретному выходному нейрону  $o_j$  равна

$$\frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial o_j} = -\frac{y_j}{o_j}$$

Таким образом, вектор частных производных функции ошибок по отношению к выходным нейронам задается как

$$\frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial \mathbf{o}} = \left( \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial o_1}, \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial o_2}, \dots, \frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial o_K} \right)^T = \left( -\frac{y_1}{o_1}, -\frac{y_2}{o_2}, \dots, -\frac{y_K}{o_K} \right)^T$$

Кросс-энтропия часто используется при обучении нейронных сетей, предназначенных для многоклассовой классификации.



# Бинарная кросс-энтропия и ее градиент

Пусть рассматривается случай двух классов, причем положительный класс представляется единицей, а отрицательный класс представляется нулем.

Пусть задан входной вектор  $\mathbf{x} \in \mathbb{R}^d$ , при этом отклик  $y \in \{0, 1\}$ , и имеется только один выходной нейрон  $o$ , тогда **бинарная кросс-энтропийная функция потерь** равна

$$\mathcal{E}_{\mathbf{x}} = - (y \ln o + (1 - y) \ln (1 - o))$$

Частная производная функции  $\mathcal{E}_{\mathbf{x}}$  по выходному нейрону  $o$  равна

$$\begin{aligned}\frac{\partial \mathcal{E}_{\mathbf{x}}}{\partial o} &= \frac{\partial}{\partial o} (-y \ln o - (1 - y) \ln (1 - o)) = \\ &= - \left( \frac{y}{o} + \frac{1 - y}{1 - o} (-1) \right) = \frac{o - y}{o (1 - o)}\end{aligned}$$

**Бинарная кросс-энтропия** используется при обучении нейронных сетей, предназначенных для **бинарной классификации**.



# Градиентный спуск при обучении нейронных сетей

Для нейронной сети с параметрами  $\Theta$ , имеющей выход  $\hat{\mathbf{y}} = M(\mathbf{x} \mid \Theta)$  для заданных входных данных  $\mathbf{x}$ , и заданной функции потерь  $L(\mathbf{y}, \hat{\mathbf{y}})$  – некоторая функция потерь для заданных входных данных  $\mathbf{x}$ , цель обучения состоит в том, чтобы найти параметры, минимизирующие потери по всем точкам (экземплярам) обучающего набора данных:

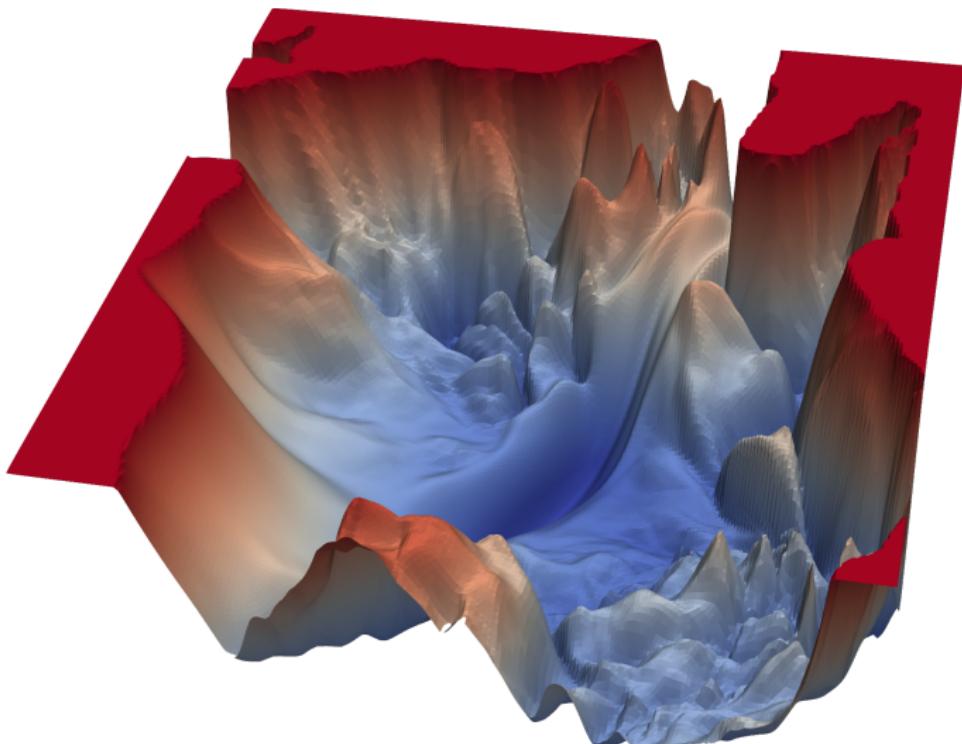
$$\min_{\Theta} J(\Theta) = \min_{\Theta} \sum_{i=1}^n L(\mathbf{y}_i, \hat{\mathbf{y}}_i) = \min_{\Theta} \sum_{i=1}^n L(\mathbf{y}_i, M(\mathbf{x}_i \mid \Theta))$$

При использовании градиентного спуска поиск минимума осуществляется в направлении наибольшего убывания функции (в направлении, противоположном градиенту функции)

- ➊ Выбирается скорость обучения  $\mu$
- ➋ Инициализируются начальные значения параметров  $\Theta_0$
- ➌ В цикле до тех пор, пока  $\Theta_t$  или  $J(\Theta_t)$  не сошлись к некоторым значениям:
  - ➊ Вычисляется градиент  $\nabla_{\Theta} J(\Theta_t) = \nabla_{\Theta} (\sum_{i=1}^n L(\mathbf{y}_i, M(\mathbf{x}_i \mid \Theta)))$
  - ➋ Обновляются значения параметров  $\Theta_{t+1} = \Theta_t - \mu \nabla_{\Theta} J(\Theta_t)$

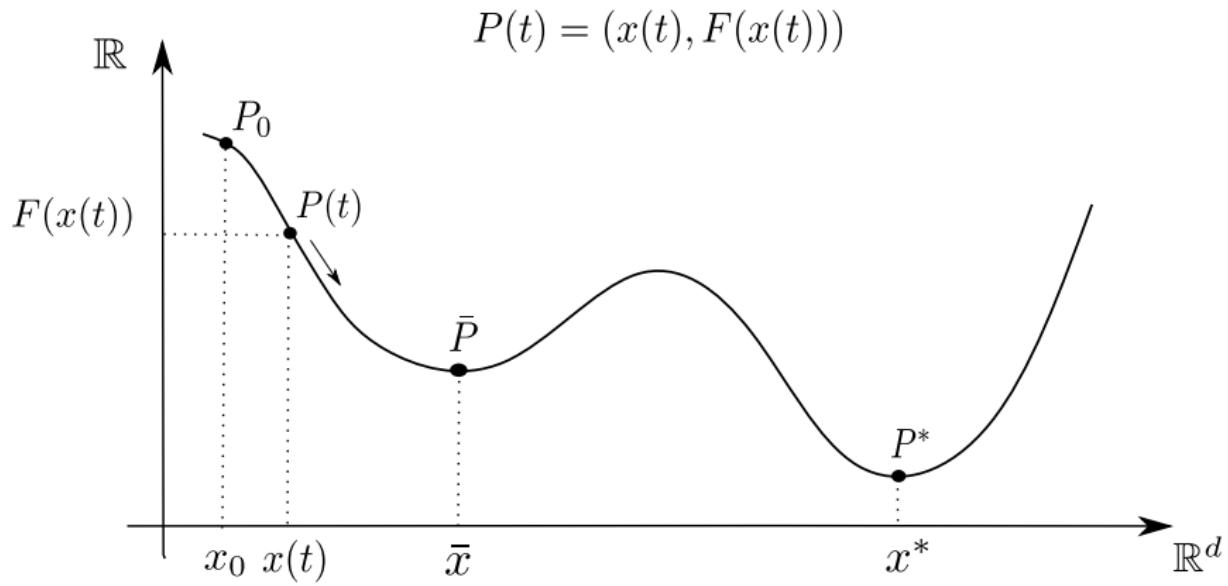
# Поверхность ошибки нейронной сети

Поверхность ошибки нейронной сети может иметь весьма замысловатый вид (нейронная сеть VGG-56 на наборе данных CIFAR-10):





Градиентный спуск по поверхности ошибки нейронной сети может быть смоделирован как движение тяжелого шара с инерцией и трением:





# Метод импульса (Momentum)

Идея: применить к градиенту экспоненциальное скользящее среднее.

Аналогия – шарик, катящийся по поверхности с вязким трением.

- ➊ Выбираются значения скоростей обучения  $\mu$  и забывания  $\lambda$
- ➋ Инициализируются начальные значения параметров  $\Theta_0$
- ➌ Начальная оценка функционала  $J_0 = J(\Theta_0)$  и импульса  $\mathbf{P}_0 = 0$
- ➍ В цикле до тех пор, пока  $\Theta_t$  или  $J_t$  не сошлись к некоторым значениям:

- ➊ Выбирается произвольная точка набора данных  $\mathbf{x}_i$
- ➋ Обновляется значение импульса

$$\mathbf{P}_{t+1} = (1 - \mu) \mathbf{P}_t + \mu \nabla_{\Theta} (L(\mathbf{y}_i, M(\mathbf{x}_i | \Theta_t)))$$

- ➌ Обновляются значения параметров

$$\Theta_{t+1} = \Theta_t - \mu \mathbf{P}_t$$

- ➍ Обновляется значение функционала

$$J_{t+1} = (1 - \lambda) J_t + \lambda L(\mathbf{y}_i, M(\mathbf{x}_i | \Theta_t))$$



# Выбор скорости обучения (шага градиента)

- Если скорость обучения  $\mu$  слишком мала, то сходимость будет долгой.
- Если скорость обучения  $\mu$  слишком велика, то алгоритм может не сойтись.
- Можно применять адаптивную скорость обучения  $\mu_t$ , зависящую от номера итерации  $t$ .
  - Исходя из теоретических соображений

$$\mu_t \xrightarrow[t \rightarrow \infty]{} 0, \sum_{t=1}^{\infty} \mu_t = \infty, \sum_{t=1}^{\infty} (\mu_t)^2 < \infty$$

Например, можно выбрать  $\mu_t = t^{-1} = \frac{1}{t}$ .

- Метод наискорейшего градиентного спуска позволяет определить оптимальную скорость обучения:

$$\min_{\mu_t} J(\Theta_t - \mu_t \nabla_{\Theta} J(\Theta_t))$$

- Могут делаться случайные шаги, чтобы не “застревать” в локальных минимумах.



# Методы адаптивных градиентов

## Метод AdaGrad (Adaptive Gradient) 2011

Идея AdaGrad состоит в том, чтобы уменьшать обновления для элементов, которые мы и так часто обновляем.

- Вычисляется градиент  $\mathbf{G}_i = \nabla_{\Theta} (L(\mathbf{y}_i, M(\mathbf{x}_i | \Theta_t)))$
- Накапливается вектор суммы квадратов  $\Gamma_t = \Gamma_{t-1} + \mathbf{G}_i \odot \mathbf{G}_i$
- Обновляются значения параметров  $\Theta_{t+1} = \Theta_t - \mu \mathbf{G}_i \oslash (\sqrt{\Gamma_t} + \epsilon)$

Здесь  $\odot$  и  $\oslash$  – поэлементные умножение и деление соответственно.

Достоинство Adagrad – отсутствие необходимости точно подбирать скорость обучения.

## Метод RMSProp (Running Mean Square) 2012

Идея RMSProp состоит в замене накопления экспоненциальным скользящим средним.

- Вычисляется градиент  $\mathbf{G}_i = \nabla_{\Theta} (L(\mathbf{y}_i, M(\mathbf{x}_i | \Theta_t)))$
- Скользящее экспоненциальное среднее  $\Gamma_t = \alpha \Gamma_{t-1} + (1 - \alpha) \mathbf{G}_i \odot \mathbf{G}_i$
- Обновляются значения параметров  $\Theta_{t+1} = \Theta_t - \mu \mathbf{G}_i \oslash (\sqrt{\Gamma_t} + \epsilon)$



# Метод Adam

Метод **Adam** (Adaptive Momentum) 2014

Идея Adam состоит в объединении методов Momentum и RMSProp.

Входными параметрами являются  $\alpha$  – размер шага,  $\beta_1, \beta_2 \in [0, 1)$  – скорость экспоненциального убывания для оценок моментов.

Инициализируются начальные параметры  $\Theta_0$ , первые и вторые моменты  $\mathbf{m}_0 = 0$  и  $\mathbf{v}_0 = 0$ .

- Вычисляется градиент  $\mathbf{G}_i = \nabla_{\Theta} (L(\mathbf{y}_i, M(\mathbf{x}_i | \Theta_t)))$
- Обновляется 1-й момент  $\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{G}_i$
- Обновляется 2-й момент  $\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) \mathbf{G}_i \odot \mathbf{G}_i$
- Делаются корректировки  $\hat{\mathbf{m}}_t = \mathbf{m}_{t-1} / (1 - \beta_1^t)$ ,  $\hat{\mathbf{v}}_t = \mathbf{v}_{t-1} / (1 - \beta_2^t)$
- Обновляются значения параметров  $\Theta_t = \Theta_{t-1} - \alpha \hat{\mathbf{m}}_t \oslash (\sqrt{\hat{\mathbf{v}}_t} + \epsilon)$

Рекомендуемые значения  $\alpha = 0.001$ ,  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$ ,  $\epsilon = 10^{-8}$



# Градиентный спуск (пакетный градиентный спуск)

Пусть объекты задаются  $d$  числовыми признаками  $X_j$ ,  $j = \overline{1, d}$  и пространство описаний признаков представляет собой  $X = \mathbb{R}^d$ .

Пусть алгоритм имеет параметры  $\mathbf{w}$  и выбрана некоторая функция потерь  $\mathcal{L}$ . Для  $i$ -го объекта выборки и алгоритма с параметрами  $\mathbf{w}$  обозначим функцию потерь через  $\mathcal{L}_i(\mathbf{w})$ . Минимизируем **эмпирический риск**  $Q(\mathbf{w})$ , являющийся оценкой математического ожидания функции потерь

$$\min_{\mathbf{w}} Q(\mathbf{w}) = \min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}_i(\mathbf{w})$$

Если функция потерь принадлежит классу  $C^1(X)$ , то можно применить метод (пакетного) **градиентного спуска**. Выберем начальное приближение вектора весов  $\mathbf{w}^{(0)}$  и каждый следующий вектор весов будем вычислять как

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \sum_{i=1}^n \nabla_{\mathbf{w}} \mathcal{L}_i \left( \mathbf{w}^{(t)} \right),$$

где  $\eta$  – шаг градиента, смысл которого заключается в том, насколько сильно менять вектор весов в направлении, противоположном градиенту. Остановка алгоритма будет определяться сходимостью  $Q(\mathbf{w})$  или  $\mathbf{w}$ .



# Алгоритм пакетного градиентного спуска

Входными данными для алгоритма пакетного градиентного спуска являются матрица входных данных  $\mathbf{D} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ , функция потерь  $\mathcal{L}(\mathbf{x}, \mathbf{w})$ , шаг обучения  $\eta > 0$ , требуемая точность  $\varepsilon > 0$ .

Batch Gradient Descent ( $\mathbf{D}, \mathcal{L}, \eta, \varepsilon$ ):

```
1  $t \leftarrow 0$  // инициализировать счетчик шагов/итераций
2  $\mathbf{w}^{(0)} \leftarrow$  случайный вектор в  $\mathbb{R}^d$  // начальный вектор весов
3 repeat
4    $\nabla_{\mathbf{w}}^{(t)} \leftarrow 0$  // начальное значение градиента для весов  $\mathbf{w}^{(t)}$ 
5   foreach  $i = 1, 2, \dots, n$  do
6     // добавить значение градиента в точке  $\mathbf{x}_i$ 
7      $\nabla_{\mathbf{w}}^{(t)} \leftarrow \nabla_{\mathbf{w}}^{(t)} + \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{x}_i, \mathbf{w}^{(t)})$ 
8   // вычислили значение градиента  $\mathcal{L}$  для  $\mathbf{w}^{(t)}$  по всем точкам  $\mathbf{x}_i$ 
9    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta \cdot \nabla_{\mathbf{w}}^{(t)}$  // обновить оценку для весов
10   $t \leftarrow t + 1$  // увеличить счетчик шагов/итераций
11 until  $\|\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)}\| \leq \varepsilon$ 
```

Algorithm 1: Пакетный градиентный спуск



# Стохастический градиентный спуск

Проблема пакетного градиентного спуска заключается в том, что для определения нового приближения вектора весов необходимо вычислить градиент для каждого элемента данных, что может сильно замедлять вычисления. Идея [стохастического градиентного спуска](#) заключается в ускорении алгоритма за счет использования только одного элемента. То есть теперь новое приближение будет вычисляться как

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^{(t)}),$$

где  $i$  – случайно выбранный индекс точки набора данных. Если для останова алгоритма используется эмпирический риск  $Q(\mathbf{w})$ , то подсчет эмпирического риска  $Q(\mathbf{w})$  на каждом шаге является слишком дорогостоящим, поэтому чтобы ускорить оценку  $Q(\mathbf{w})$ , можно использовать одну из приближенных рекуррентных формул ( $\varepsilon_t = \mathcal{L}_i(\mathbf{w}^{(t)})$ ):

- среднее арифметическое:  $\overline{Q}_t = \frac{1}{t} (\varepsilon_t + \varepsilon_{t-1} + \dots) = \frac{1}{t} \varepsilon_t + \left(1 - \frac{1}{t}\right) \overline{Q}_{t-1}$
- экспоненциальное скользящее среднее:  $\overline{Q}_t = \lambda \varepsilon_t + (1 - \lambda) \overline{Q}_{t-1}$ , где  $\lambda$  – темп забывания предыстории значений  $Q(\mathbf{w})$ .



Входными данными для алгоритма стохастического градиентного спуска являются матрица данных  $\mathbf{D} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ , функция потерь  $\mathcal{L}(\mathbf{x}, \mathbf{w})$ , шаг обучения  $\eta > 0$ , требуемая точность  $\varepsilon > 0$ .

Stochastic Gradient Descent ( $\mathbf{D}, \mathcal{L}, \eta, \varepsilon$ ):

```
1  $t \leftarrow 0$  // инициализировать счетчик шагов/итераций
2  $\mathbf{w}^{(0)} \leftarrow$  случайный вектор в  $\mathbb{R}^d$  // начальный вектор весов
3 repeat
4   foreach  $i = 1, 2, \dots, n$  (в случайном порядке) do
5     // обновить оценку для весов по градиенту в точке  $\mathbf{x}_i$ 
6      $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta \cdot \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{x}_i, \mathbf{w}^{(t)})$ 
7    $t \leftarrow t + 1$  // увеличить счетчик шагов/итераций
8 until  $\|\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)}\| \leq \varepsilon$ 
```

Algorithm 2: Стохастический градиентный спуск



## Мини-пакетный градиентный спуск

Мини-пакетный (mini-batch) градиентный спуск – это разновидность алгоритма градиентного спуска, в которой обучающий набор данных разбивается на небольшие партии (мини-пакеты), которые используются для обновления коэффициентов (весов) модели и расчета ошибки модели. То есть каждый следующий вектор весов будет вычисляться как

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \sum_{i \in \mathbf{I}_t} \nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{x}_i, \mathbf{w}^{(t)}),$$

где  $\mathbf{I}_t \subset \{1, \dots, n\}$  – множество индексов точек в мини-пакете для эпохи обучения  $t$ ,  $\eta$  – шаг градиента. Мини-пакетный градиентный спуск стремится найти баланс между надежностью стохастического градиентного спуска и эффективностью пакетного градиентного спуска. Это наиболее распространенная реализация градиентного спуска, используемая в глубоком обучении.

Мини-пакетный градиентный спуск обеспечивает вычислительно более эффективный процесс, чем стохастический градиентный спуск. Пакетирование позволяет эффективно использовать ресурсы и реализовывать параллельные алгоритмы машинного обучения.



# Алгоритм мини-пакетного градиентного спуска

Входными данными для алгоритма мини-пакетного градиентного спуска являются матрица данных  $\mathbf{D} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ , функция потерь  $\mathcal{L}(\mathbf{x}, \mathbf{w})$ , размер мини-пакета  $m$ , шаг обучения  $\eta > 0$ , требуемая точность  $\varepsilon > 0$ .

Mini-Batch Gradient Descent ( $\mathbf{D}, \mathcal{L}, m, \eta, \varepsilon$ ):

```
1  $t \leftarrow 0$  // инициализировать счетчик эпох
2  $\mathbf{w}^{(0)} \leftarrow$  случайный вектор в  $\mathbb{R}^d$  // начальный вектор весов
3 repeat
4    $\nabla_{\mathbf{w}}^{(t)} \leftarrow 0$  // начальное значение градиента для весов  $\mathbf{w}^{(t)}$ 
5    $\mathbf{I}_t \leftarrow$  случайный набор из  $m$  индексов от 1 до  $n$ 
6   foreach  $i \in \mathbf{I}_t$  do
7      $\nabla_{\mathbf{w}}^{(t)} \leftarrow \nabla_{\mathbf{w}}^{(t)} + \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{x}_i, \mathbf{w}^{(t)})$  // градиент в точке  $\mathbf{x}_i$ 
8   // вычислили градиент  $\mathcal{L}$  для  $\mathbf{w}^{(t)}$  по точкам мини-пакета
9    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta \cdot \nabla_{\mathbf{w}}^{(t)}$  // обновить оценку для весов
10   $t \leftarrow t + 1$  // увеличить счетчик эпох
11 until  $\|\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)}\| \leq \varepsilon$ 
```

Algorithm 3: Мини-пакетный градиентный спуск



# Параметры алгоритмов градиентного спуска

Инициализация весов:

- $\mathbf{w}^{(0)} = \mathbf{0}$  (кроме глубокого обучения)
- $w_j^{(0)} = \text{random}\left(-\frac{1}{2n}, \frac{1}{2n}\right)$
- $w_j^{(0)} = \frac{\mathbb{E}[y X_j]}{\mathbb{E}[X_j^2]}$ , где  $y$  – метки классов (для задачи классификации)

Случайный выбор точек набора данных:

- выбор элементов из разных классов
- выбор элементов, на которых ошибка больше

Выбор величины шага градиента:

- $\eta_t = \frac{1}{t}$
- метод скорейшего градиентного спуска:  $\min_{\eta} \mathcal{L}_i(\mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}))$
- при квадратичной функции потерь можно использовать  $\eta = \|\mathbf{x}_i\|^2$
- иногда можно выполнять случайные шаги, чтобы выбивать процесс из локальных минимумов
- метод Левенберга-Марквардта (комбинация градиентного спуска и метода Гаусса — Ньютона)



# Автоматическое дифференцирование

Автоматическое дифференцирование (automatic differentiation, AD) представляет собой набор методов для оценки производной функции, заданной компьютерной программой. AD использует тот факт, что каждая компьютерная программа, какой бы сложной она ни была, выполняет последовательность элементарных арифметических операций (сложение, вычитание, умножение, деление и т. д.) и элементарных функций ( $\exp$ ,  $\log$ ,  $\sin$ ,  $\cos$  и т. д.). При многократном применении цепного правила производные произвольного порядка могут быть вычислены автоматически с приемлемой точностью.

Автоматическое дифференцирование отличается от [символьного дифференцирования](#) и [численного дифференцирования](#). Символьное дифференцирование сталкивается с трудностями преобразования алгоритма в одно математическое выражение и может привести к неэффективному компьютерному коду. Численное дифференцирование (метод конечных разностей) может вносить ошибки округления в процессе дискретизации. Наконец, оба этих классических метода работают медленно при вычислении частных производных функции по множеству входных данных, что необходимо для алгоритмов оптимизации на основе градиента.



# Цепное правило

В основе AD лежит т.н. **цепное правило** (правило дифференцирования сложной функции). Для сложной функции:

$$y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3,$$

где  $w_0 = x$ ,  $w_1 = h(w_0)$ ,  $w_2 = g(w_1)$ ,  $w_3 = f(w_2) = y$ , цепное правило означает

$$\frac{dy}{dx} = \frac{dy}{dw_2} \frac{dw_2}{dw_1} \frac{dw_1}{dx} = \frac{df(w_2)}{dw_2} \frac{dg(w_1)}{dw_1} \frac{dh(w_0)}{dx}$$

Обычно рассматривают два различных режима AD: **прямое дифференцирование** (или прямой режим), при котором цепное правило проходится изнутри наружу (т. е. сначала вычисляется  $dw_1/dx$ , а затем  $dw_2/dw_1$  и, наконец,  $dy/dw_2$ ), и **обратное дифференцирование** (или обратный режим), при котором происходит обход снаружи внутрь (сначала вычисляют  $dy/dw_2$ , затем  $dw_2/dw_1$  и, наконец,  $dw_1/dx$ ). То есть:

- ❶ прямое дифференцирование:  $\frac{dw_i}{dx} = \frac{dw_i}{dw_{i-1}} \frac{dw_{i-1}}{dx}$  при  $w_0 = x$
- ❷ обратное дифференцирование:  $\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}} \frac{dw_{i+1}}{dw_i}$  при  $w_3 = y$



# Прямое дифференцирование

При прямом дифференцировании сначала фиксируется независимая переменная, по которой выполняется дифференцирование, и рекурсивно вычисляется производная каждого подвыражения. В ручном расчете это включает в себя повторную замену производной внутренних функций в цепном правиле:

$$\frac{dy}{dx} = \frac{\partial y}{\partial w_{n-1}} \frac{\partial w_{n-1}}{\partial x} = \frac{\partial y}{\partial w_{n-1}} \left( \frac{\partial w_{n-1}}{\partial w_{n-2}} \frac{\partial w_{n-2}}{\partial x} \right) = \frac{\partial y}{\partial w_{n-1}} \left( \frac{\partial w_{n-1}}{\partial w_{n-2}} \left( \frac{\partial w_{n-2}}{\partial w_{n-3}} \frac{\partial w_{n-3}}{\partial x} \right) \right) = \dots$$

Эту формулу можно обобщить на несколько переменных при помощи матричного произведения якобианов.

По сравнению с обратным дифференцированием прямое дифференцировании является естественным и простым в реализации, поскольку поток данных с производными совпадает с порядком вычислений. Каждая переменная  $w$  дополняется своей производной  $\dot{w}$  (хранится как числовое значение, а не как символьное выражение):

$$\dot{w} = \frac{\partial w}{\partial x}$$

Затем производные вычисляются синхронно согласно шагам, указанным выше, и объединяются с другими производными с помощью цепного правила.



# Прямое дифференцирование – пример

В качестве примера рассмотрим функцию:

$$y = f(x_1, x_2) = x_1 x_2 + \sin x_1 = w_1 w_2 + \sin w_1 = w_3 + w_4 = w_5,$$

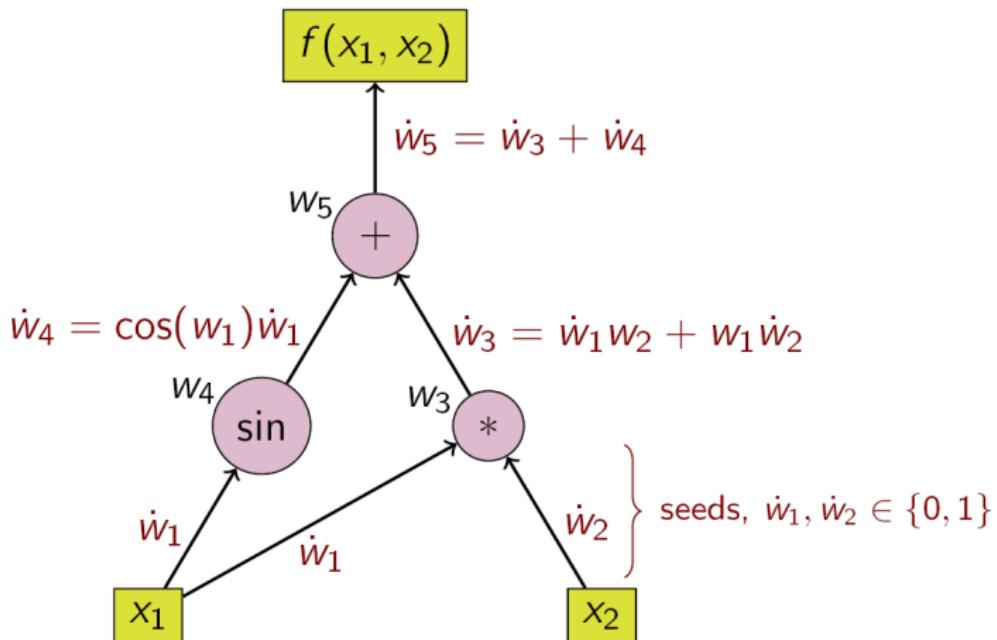
для которой  $\frac{\partial y}{\partial x_1} = x_2 + \cos x_1$ . Отдельные подвыражения в ходе вычисления функции помечены переменными  $w_i$ .

Выбор независимой переменной, по которой выполняется дифференцирование, влияет на начальные значения  $\dot{w}_1$  и  $\dot{w}_2$ . Если требуется вычислить производную этой функции по  $x_1$ , то начальные значения должны быть следующими:

$$\dot{w}_1 = \frac{\partial x_1}{\partial x_1} = 1, \quad \dot{w}_2 = \frac{\partial x_2}{\partial x_1} = 0$$

При заданных выше начальных значениях значения производных распространяются по вычислительному графу с использованием цепного правила. На следующем слайде показано графическое изображение этого процесса в виде вычислительного графа.

Forward propagation  
of derivative values





# Прямое дифференцирование – вычисления

Операции для вычисления значения функции	Операции для вычисления производной функции
$w_1 = x_1$	$\dot{w}_1 = 1$
$w_2 = x_2$	$\dot{w}_2 = 0$
$w_3 = w_1 w_2$	$\dot{w}_3 = w_2 \dot{w}_1 + w_1 \dot{w}_2$
$w_4 = \sin w_1$	$\dot{w}_4 = \cos w_1 \dot{w}_1$
$w_5 = w_3 + w_4$	$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$

$$\frac{\partial y}{\partial x_1} = \dot{w}_5 = \dot{w}_3 + \dot{w}_4 = w_2 \dot{w}_1 + w_1 \dot{w}_2 + \cos w_1 \dot{w}_1 = w_2 + \cos w_1 = x_2 + \cos x_1$$

Чтобы вычислить градиент этой функции, требуется производные от  $f$  не только по  $x_1$ , но и по  $x_2$ , на вычислительном графе выполняется дополнительный цикл прямого дифференцирования с использованием начальных значений  $\dot{w}_1 = 0$ ,  $\dot{w}_2 = 1$ .

Прямое дифференцирование более эффективно, чем обратное дифференцирование, для функций  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  для  $m \gg n$ , поскольку необходимо только  $n$  проходов по графу по сравнению с  $m$  проходами для обратного дифференцирования.



# Обратное дифференцирование

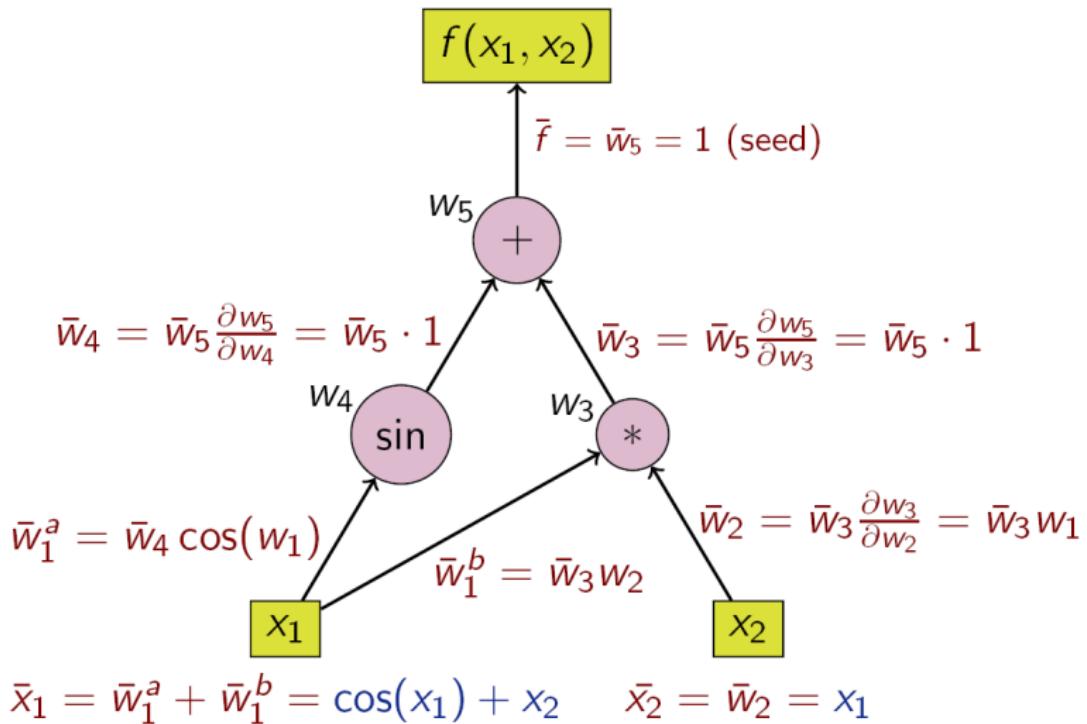
При обратном дифференцировании AD фиксируется зависимая переменная  $y$ , которую нужно дифференцировать, и рекурсивно вычисляется производная по каждому подвыражению. В ручном расчете производные внешних функций повторно подставляются в цепное правило:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1} \frac{\partial w_1}{\partial x} = \left( \frac{\partial y}{\partial w_2} \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = \left( \left( \frac{\partial y}{\partial w_3} \frac{\partial w_3}{\partial w_2} \right) \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = \dots$$

При обратном дифференцировании интерес представляет т.н. присоединенная (adjoint) функция, обозначаемая  $\bar{w}$ . Это производная выбранной зависимой переменной  $y$  по подвыражению  $w$ :

$$\bar{w} = \frac{\partial y}{\partial w}$$

Обратное дифференцирование происходит по цепному правилу снаружи внутрь или, в случае вычислительного графа на следующем слайде, сверху вниз.


 Backward propagation  
of derivative values




# Обратное дифференцирование – вычисления

В графе на предыдущем слайде происходило два прохода по вычислительному графу:

- прямой проход при вычислении значения функции с запоминанием промежуточных результатов
- обратный проход при вычислении значений производных функции

Функция в примере является скалярной, поэтому для вычисления производных используется только одно начальное значение, а для вычисления (двухкомпонентного) градиента требуется только один цикл прохода по вычислительному графу. Это только половина работы по сравнению с прямым дифференцированием, но обратное дифференцирование требует хранения промежуточных переменных  $w_i$ , а также инструкций для их вычисления в структуре данных, известной как «лента» (tape), которая может потреблять значительный объем памяти, если вычислительный граф большой. Это можно до некоторой степени смягчить, сохраняя только подмножество промежуточных переменных, а затем реконструируя необходимые рабочие переменные путем повторения вычислений. Для сохранения промежуточных состояний также можно использовать контрольные точки.



# Обратное дифференцирование – таблица

Операции для вычисления производной с использованием обратного дифференцирования показаны в таблице ниже (обратите внимание на обратный порядок при вычислении производной функции):

Операции для вычисления значения функции	Операции для вычисления производной функции
$w_1 = x_1$	$\bar{w}_5 = \frac{\partial y}{\partial w_5} = \frac{\partial y}{\partial y} = 1$
$w_2 = x_2$	$\bar{w}_4 = \frac{\partial y}{\partial w_4} = \frac{\partial y}{\partial w_5} \frac{\partial w_5}{\partial w_4} = \frac{\partial y}{\partial w_5} = \bar{w}_5$
$w_3 = w_1 w_2$	$\bar{w}_3 = \frac{\partial y}{\partial w_3} = \frac{\partial y}{\partial w_5} \frac{\partial w_5}{\partial w_3} = \frac{\partial y}{\partial w_5} = \bar{w}_5$
$w_4 = \sin w_1$	$\bar{w}_2 = \frac{\partial y}{\partial w_2} = \frac{\partial y}{\partial w_3} \frac{\partial w_3}{\partial w_2} = \frac{\partial y}{\partial w_3} w_1 = \bar{w}_3 w_1$
$w_5 = w_3 + w_4$	$\bar{w}_1 = \frac{\partial y}{\partial w_1} = \frac{\partial y}{\partial w_3} \frac{\partial w_3}{\partial w_1} + \frac{\partial y}{\partial w_4} \frac{\partial w_4}{\partial w_1} = \bar{w}_3 w_2 + \bar{w}_4 \cos w_1$

$$\frac{\partial y}{\partial x_1} = \bar{w}_3 w_2 + \bar{w}_4 \cos w_1 = \bar{w}_5 w_2 + \bar{w}_5 \cos w_1 = w_2 + \cos w_1 = x_2 + \cos x_1$$



# Обратное дифференцирование – вычисления

Графом потока данных вычисления функции можно манипулировать, чтобы вычислить градиент этой функции. Это делается путем добавления присоединенного узла для каждого основного узла, соединенного присоединенными ребрами, которые параллельны основным ребрам, но проходятся в противоположном направлении. Узлы в присоединенном графе представляют умножение на производные функций, вычисляемых узлами в основном графе.

Обратное дифференцирование более эффективно, чем прямое дифференцирование, для функций  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  для  $m \ll n$ , поскольку необходимо только  $m$  проходов по сравнению с  $n$  проходами для прямого дифференцирования.

Обратный режим AD был впервые опубликован в 1976 году Сеппо Линнаинмаа (Seppo Linnainmaa).

Обратное распространение ошибок в нейронных сетях (многослойных персептронах) является частным случаем обратного режима AD (обратного дифференцирования).



# Пример обратного дифференцирования (1)

Вычислить градиент функции  $f(x_1, x_2) = e^{x_1 x_2} + x_2^3$  по переменным  $x_1, x_2$  при помощи алгоритма обратного дифференцирования.

**Решение:**

Функция  $f(x_1, x_2)$  может быть вычислена при помощи следующей последовательности элементарных операций и функций:

$$w_1 = x_1, w_2 = x_2 \Rightarrow w_3 = w_1 w_2, w_4 = w_2^3 \Rightarrow w_5 = e^{w_3} \Rightarrow w_6 = w_4 + w_5$$

или

$$y = f(x_1, x_2) = e^{x_1 x_2} + x_2^3 = e^{w_1 w_2} + w_2^3 = e^{w_3} + w_4 = w_5 + w_4 = w_6$$

Таким образом, имеют место следующие прямые зависимости между переменными  $w_i$ :

- $w_3$  непосредственно зависит от  $w_1$  и  $w_2$
- $w_4$  непосредственно зависит от  $w_2$
- $w_5$  непосредственно зависит от  $w_3$
- $w_6$  непосредственно зависит от  $w_4$  и  $w_5$



## Пример обратного дифференцирования (2)

$$y = f(x_1, x_2) = e^{x_1 x_2} + x_2^3 = e^{w_1 w_2} + w_2^3 = e^{w_3} + w_4 = w_5 + w_4 = w_6$$

Иначе, одни переменные  $w_i$  непосредственно влияют на другие переменные  $w_j$  следующим образом:

- переменная  $w_5$  непосредственно влияет на  $w_6$
- переменная  $w_4$  непосредственно влияет на  $w_6$
- переменная  $w_3$  непосредственно влияет на  $w_5$
- переменная  $w_2$  непосредственно влияет на  $w_3$  и  $w_4$
- переменная  $w_1$  непосредственно влияет на  $w_3$

При обратном дифференцировании в цепном правиле

$$\frac{\partial y}{\partial w_j} = \sum_i \frac{\partial w_i}{\partial w_j} \frac{\partial y}{\partial w_i} = \frac{\partial w_1}{\partial w_j} \frac{\partial y}{\partial w_1} + \frac{\partial w_2}{\partial w_j} \frac{\partial y}{\partial w_2} + \dots$$

остаются слагаемые только для тех переменных  $w_i$ , которые непосредственно зависят от переменной  $w_j$ , т.е. когда  $\frac{\partial w_i}{\partial w_j} \neq 0$ .



## Пример обратного дифференцирования (3)

$$y = f(x_1, x_2) = e^{x_1 x_2} + x_2^3 = e^{w_1 w_2} + w_2^3 = e^{w_3} + w_4 = w_5 + w_4 = w_6$$

Поэтому получаем

$$\bar{w}_6 = \frac{\partial y}{\partial w_6} = [y \equiv w_6] = \frac{\partial y}{\partial y} = 1$$

$$\bar{w}_5 = \frac{\partial y}{\partial w_5} = \frac{\partial w_6}{\partial w_5} \frac{\partial y}{\partial w_6} = 1 \cdot \frac{\partial y}{\partial w_6} = \bar{w}_6$$

$$\bar{w}_4 = \frac{\partial y}{\partial w_4} = \frac{\partial w_6}{\partial w_4} \frac{\partial y}{\partial w_6} = 1 \cdot \frac{\partial y}{\partial w_6} = \bar{w}_6$$

$$\bar{w}_3 = \frac{\partial y}{\partial w_3} = \frac{\partial w_5}{\partial w_3} \frac{\partial y}{\partial w_5} = e^{w_3} \cdot \frac{\partial y}{\partial w_5} = e^{w_3} \bar{w}_5$$

$$\bar{w}_2 = \frac{\partial y}{\partial w_2} = \frac{\partial w_3}{\partial w_2} \frac{\partial y}{\partial w_3} + \frac{\partial w_4}{\partial w_2} \frac{\partial y}{\partial w_4} = w_1 \frac{\partial y}{\partial w_3} + 3w_2^2 \frac{\partial y}{\partial w_4} = w_1 \bar{w}_3 + 3w_2^2 \bar{w}_4$$

$$\bar{w}_1 = \frac{\partial y}{\partial w_1} = \frac{\partial w_3}{\partial w_1} \frac{\partial y}{\partial w_3} = w_2 \frac{\partial y}{\partial w_3} = w_2 \bar{w}_3$$

Итак, градиент функции  $f(x_1, x_2)$ , вычисленный при помощи обратного диф-



## Пример обратного дифференцирования (4)

$$y = f(x_1, x_2) = e^{x_1 x_2} + x_2^3 = e^{w_1 w_2} + w_2^3 = e^{w_3} + w_4 = w_5 + w_4 = w_6$$

$$\bar{w}_6 = 1$$

$$\bar{w}_5 = \bar{w}_6$$

$$\bar{w}_4 = \bar{w}_6$$

$$\bar{w}_3 = e^{w_3} \bar{w}_5$$

$$\bar{w}_2 = w_1 \bar{w}_3 + 3w_2^2 \bar{w}_4$$

$$\bar{w}_1 = w_2 \bar{w}_3$$

Итак, градиент функции  $f(x_1, x_2)$ , вычисленный при помощи обратного дифференцирования, равен

$$\frac{\partial y}{\partial x_1} = \bar{w}_1 = w_2 \bar{w}_3 = x_2 e^{x_1 x_2}, \quad \frac{\partial y}{\partial x_2} = \bar{w}_2 = x_1 e^{x_1 x_2} + 3x_2^2$$