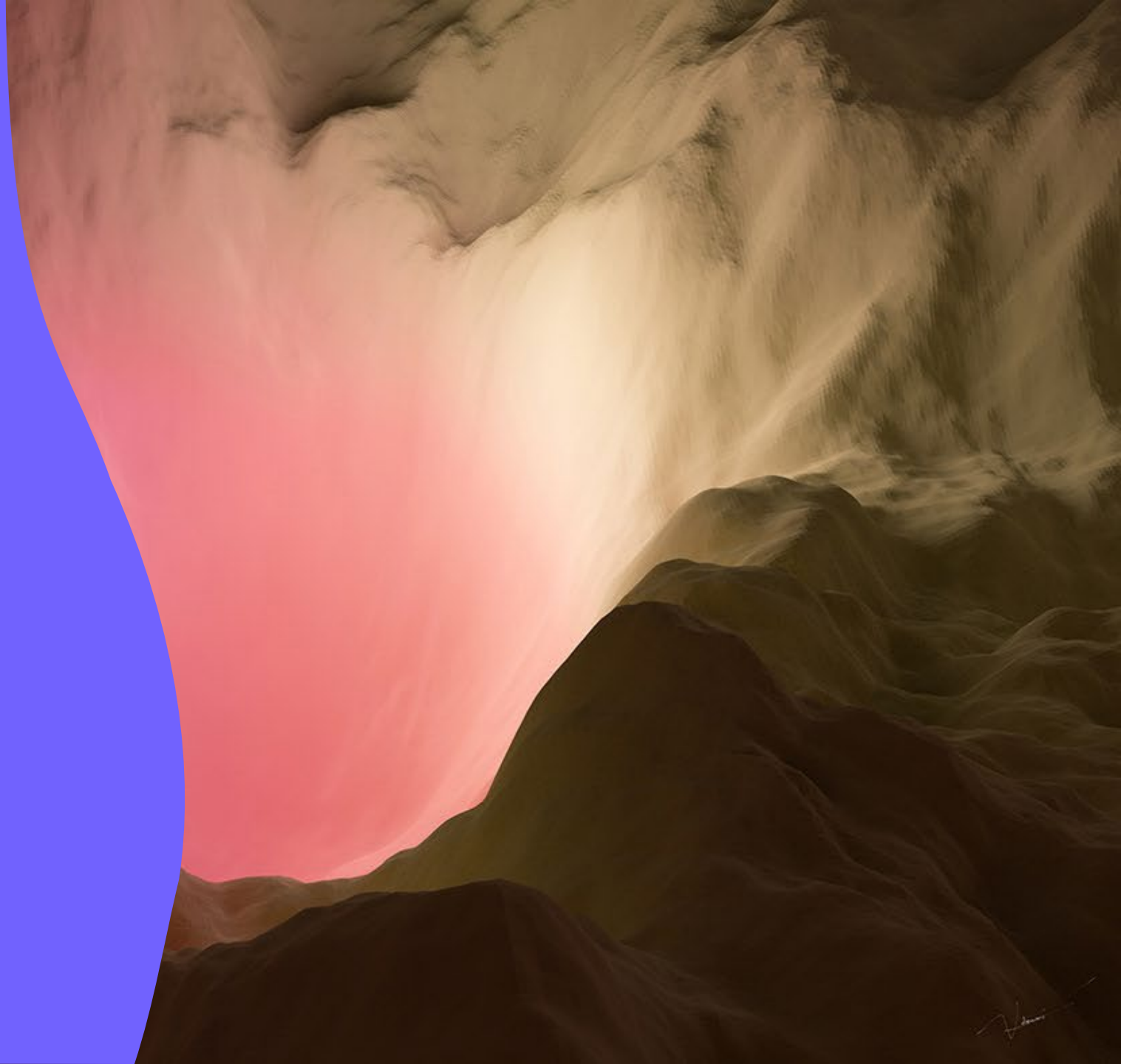


Градиентный спуск.
Функции активации.
Инициализация весов.
Нормализация

Глубинное обучение

Градиентный спуск

Основная идея
Backpropagation
Разновидности



Основная идея

Что из этого – формула шага в градиентном спуске?

- $w^t = w^t + \eta \nabla Q(w^{t-1})$
- $w^t = w^t - \eta \nabla Q(w^{t-1})$
- $w^t = w^{t-1} + \eta \nabla Q(w^{t-1})$
- $w^t = w^{t-1} - \eta \nabla Q(w^{t-1})$

Основная идея

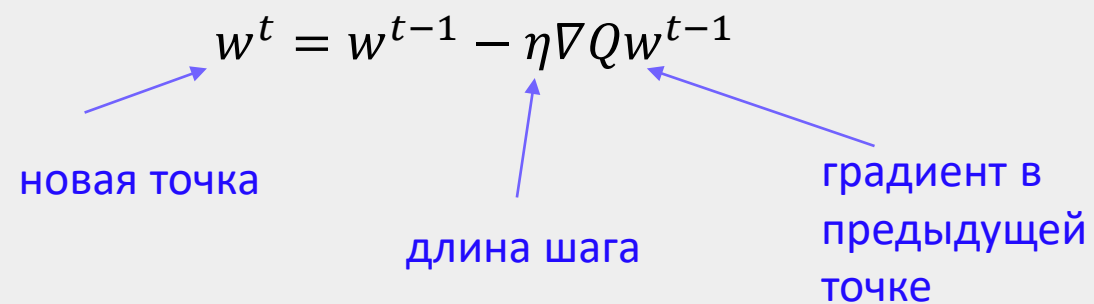
Повторять до сходимости:

$$w^t = w^{t-1} - \eta \nabla Q w^{t-1}$$

новая точка

длина шага

градиент в предыдущей точке

The diagram shows the update formula for the weight vector w in gradient descent. Three blue arrows point from text labels to parts of the formula: one from 'новая точка' to w^t , one from 'длина шага' to η , and one from 'градиент в предыдущей точке' to $\nabla Q w^{t-1}$.

Сходимость

- Останавливаем процесс, если:

$$\|w^t - w^{t-1}\| < \varepsilon$$

- Другой вариант:

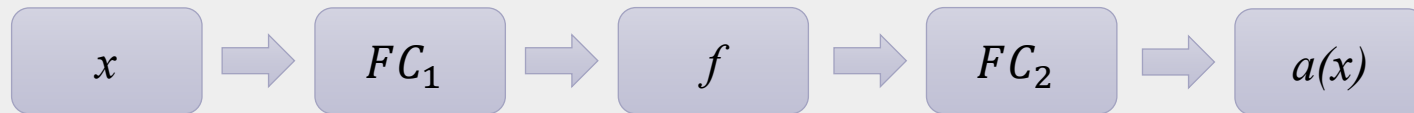
$$\|\nabla Q(w^t)\| < \varepsilon$$

- Обычно в глубинном обучении: останавливаемся, когда ошибка на тестовой выборке перестает убывать

Обучение НС

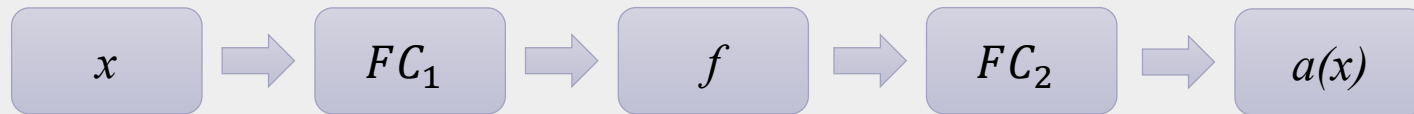
Все слои обычно дифференцируемы, поэтому нужно посчитать производные по всем параметрам.

Так, стоп, а где параметры?



Обучение НС

Все слои обычно дифференцируемы, поэтому нужно посчитать производные по всем параметрам.

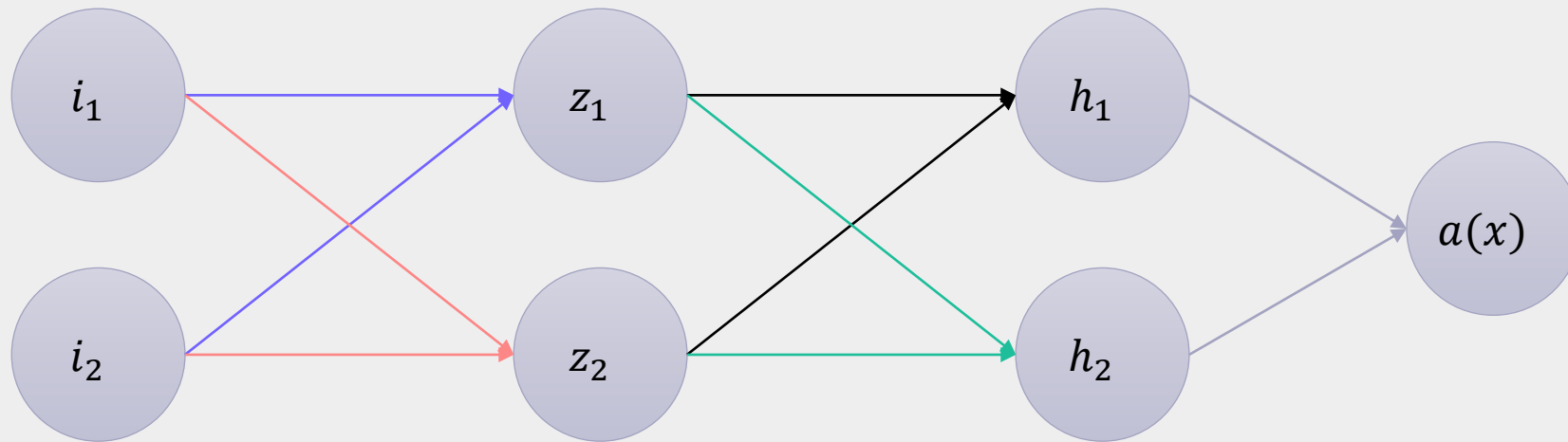


Оптимизируем функционал ошибки:

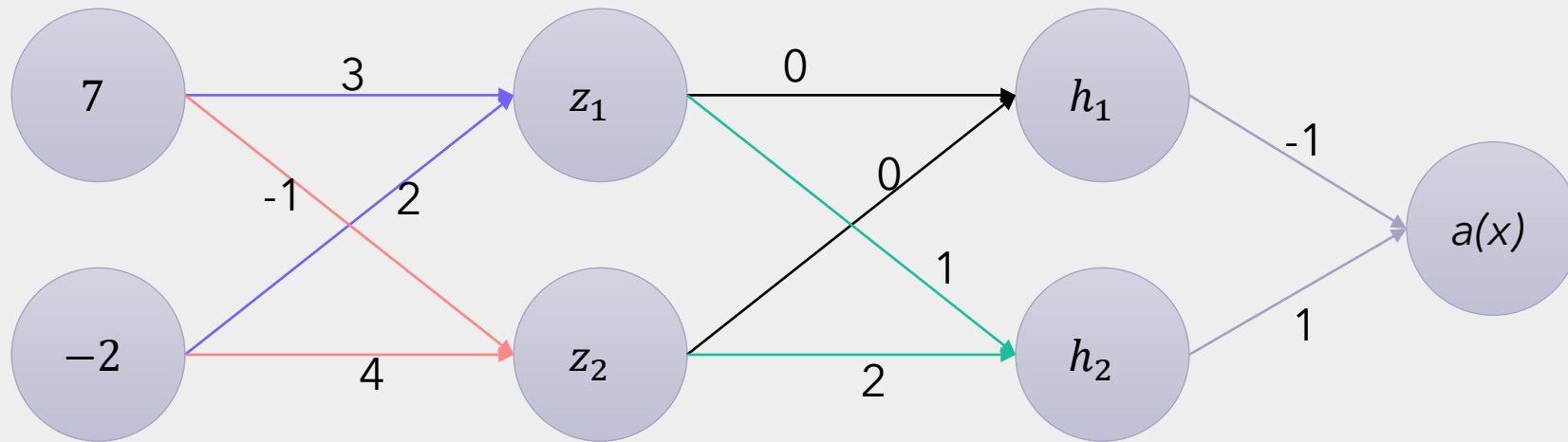
$$\frac{1}{l} \sum_{i=1}^l L(y_i, a(x_i)) \rightarrow \min_a$$

Для каждого параметра считаем частную производную.

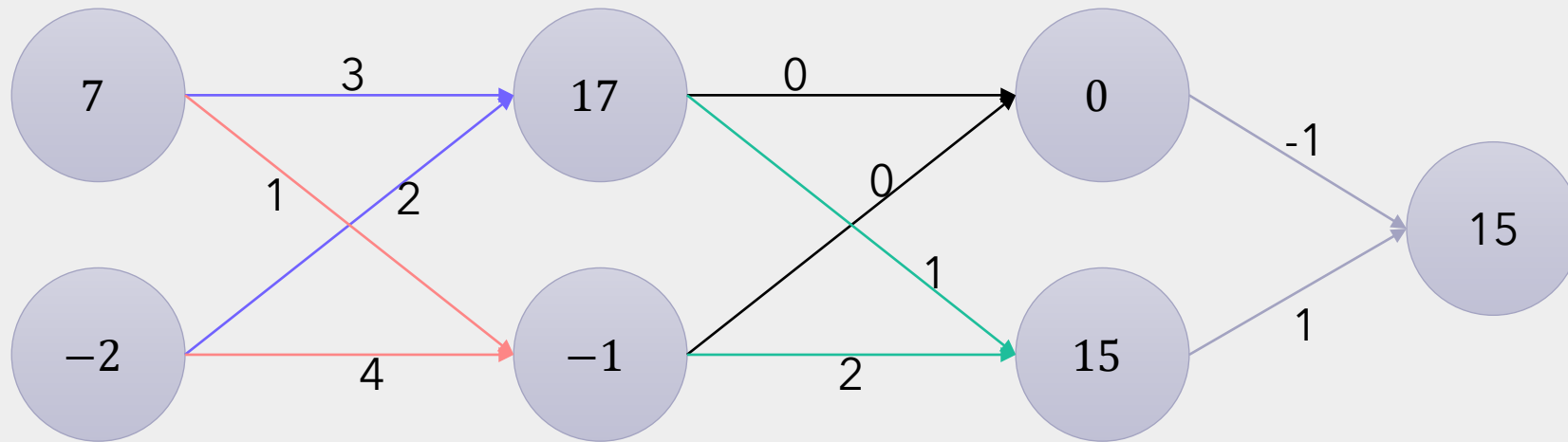
Как считать производные?



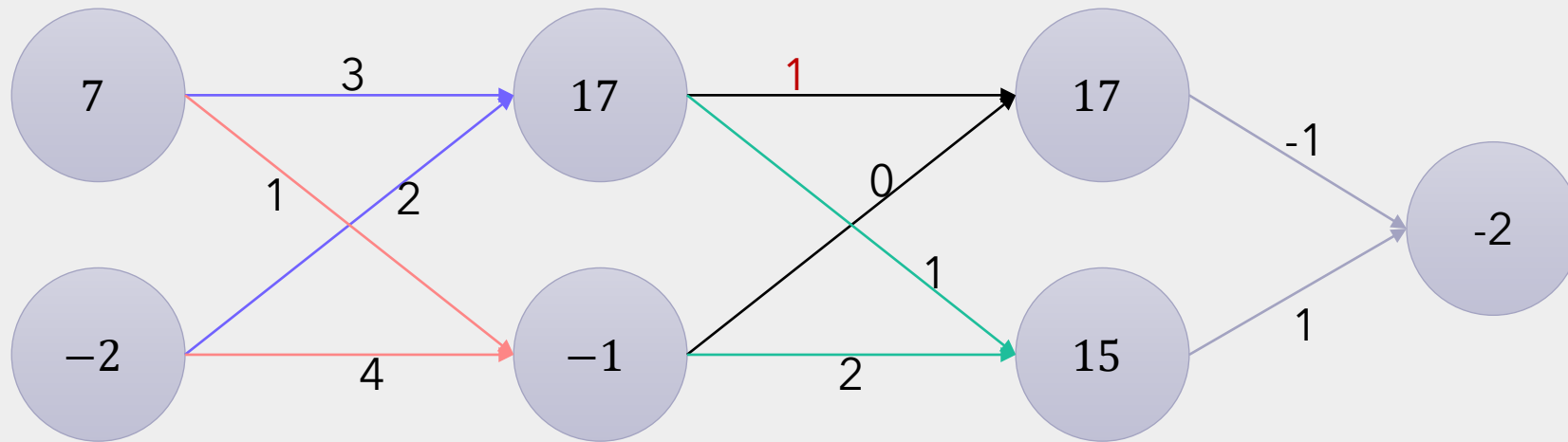
Как считать производные?



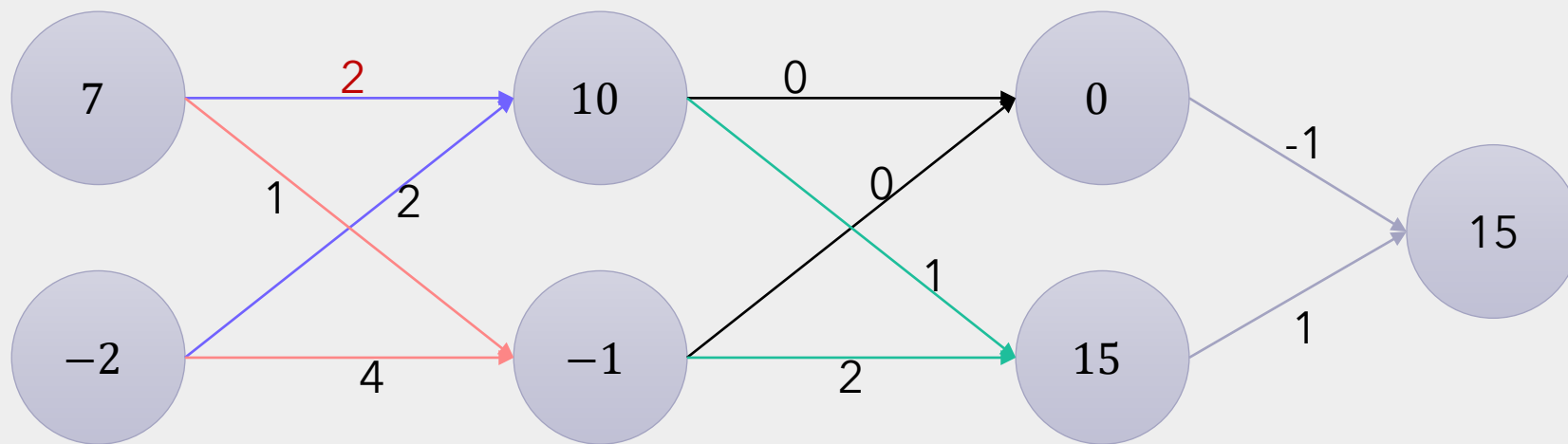
Как считать производные?



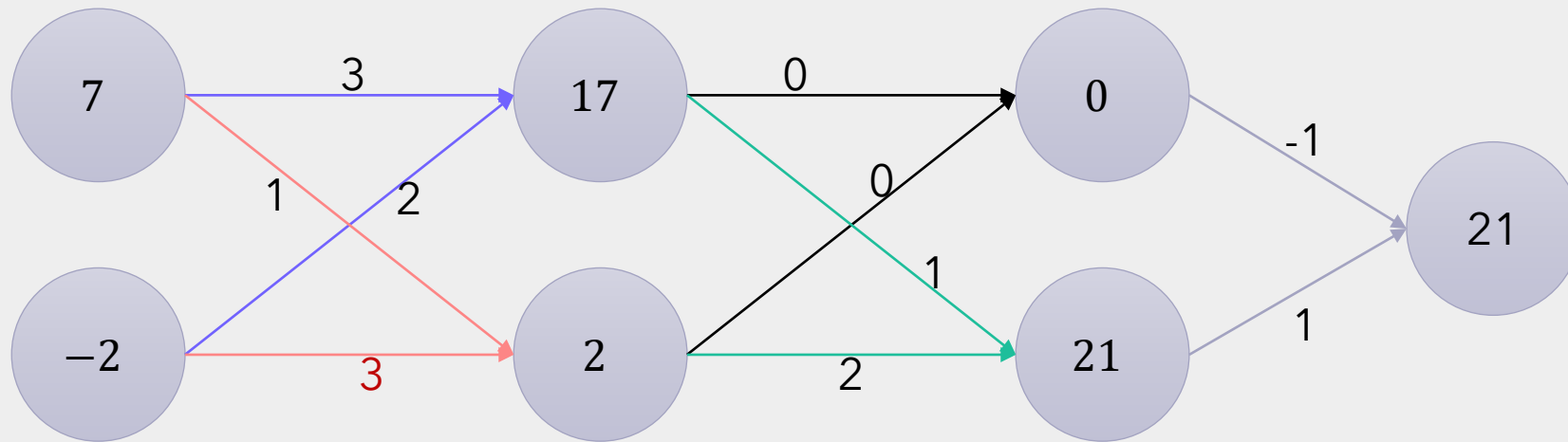
Как считать производные?



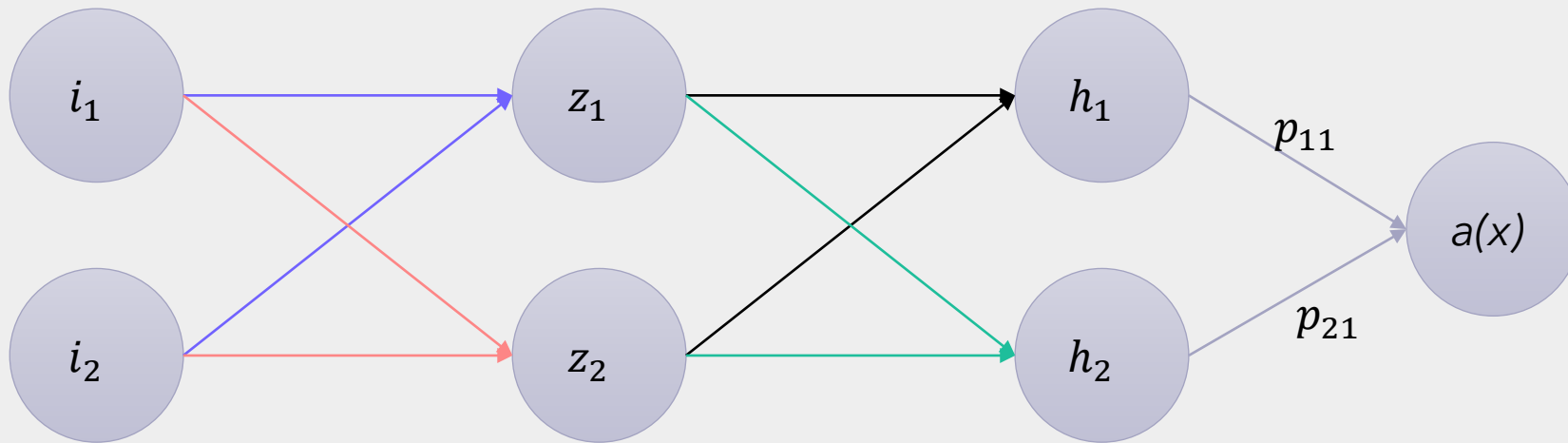
Как считать производные?



Как считать производные?



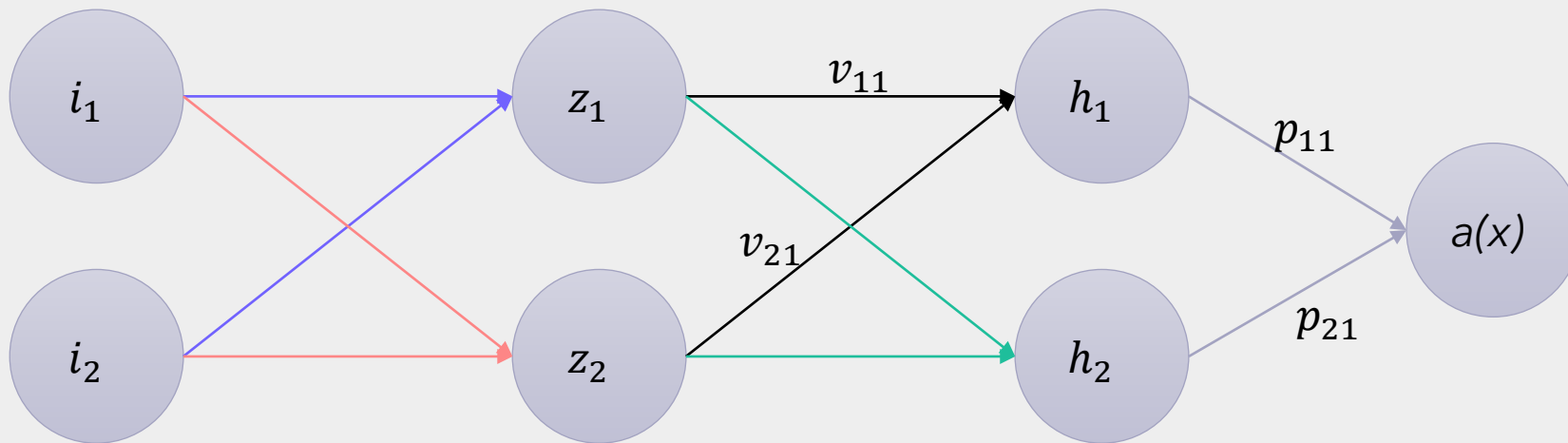
Как считать производные?



$$a(x) = p_{11}h_1(x) + p_{21}h_2(x)$$

$$\frac{\delta a}{\delta p_{11}} = ?$$

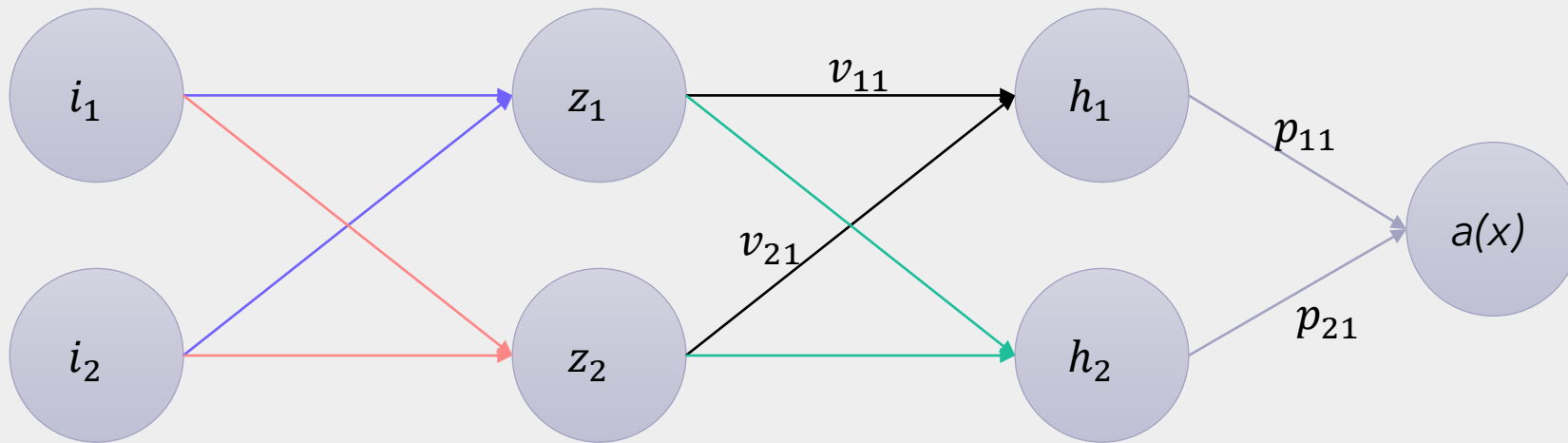
Как считать производные?



$$a(x) = p_{11}f(v_{11}z_1(x) + v_{21}z_2(x)) + p_{21}h_2(x)$$

$$\frac{\delta a}{\delta v_{11}} = ?$$

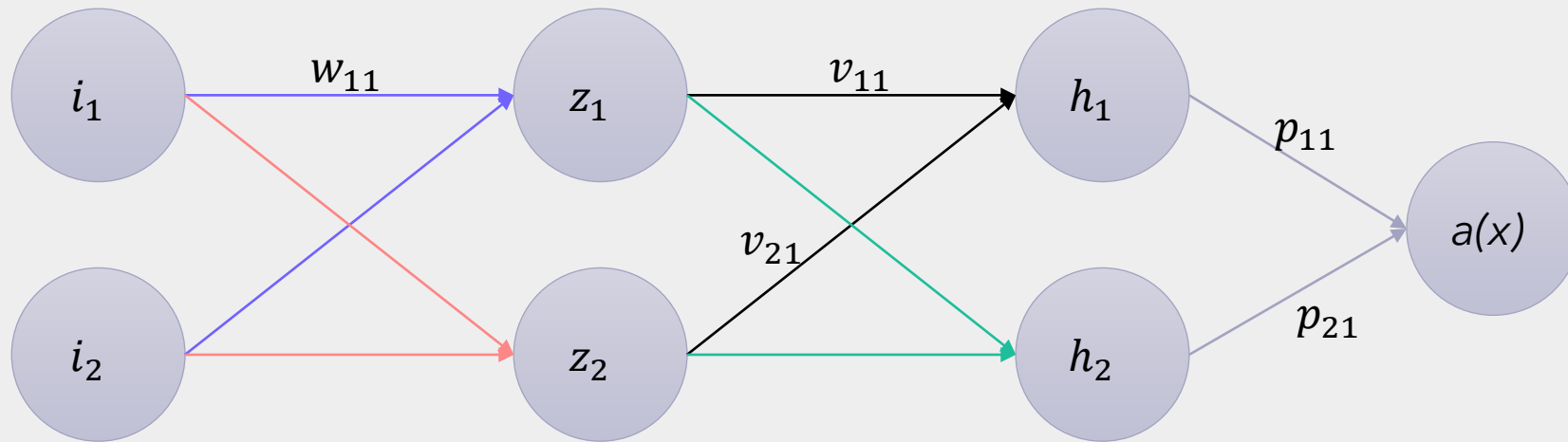
Как считать производные?



$$a(x) = p_{11}f(v_{11}z_1(x) + v_{21}z_2(x)) + p_{21}h_2(x)$$

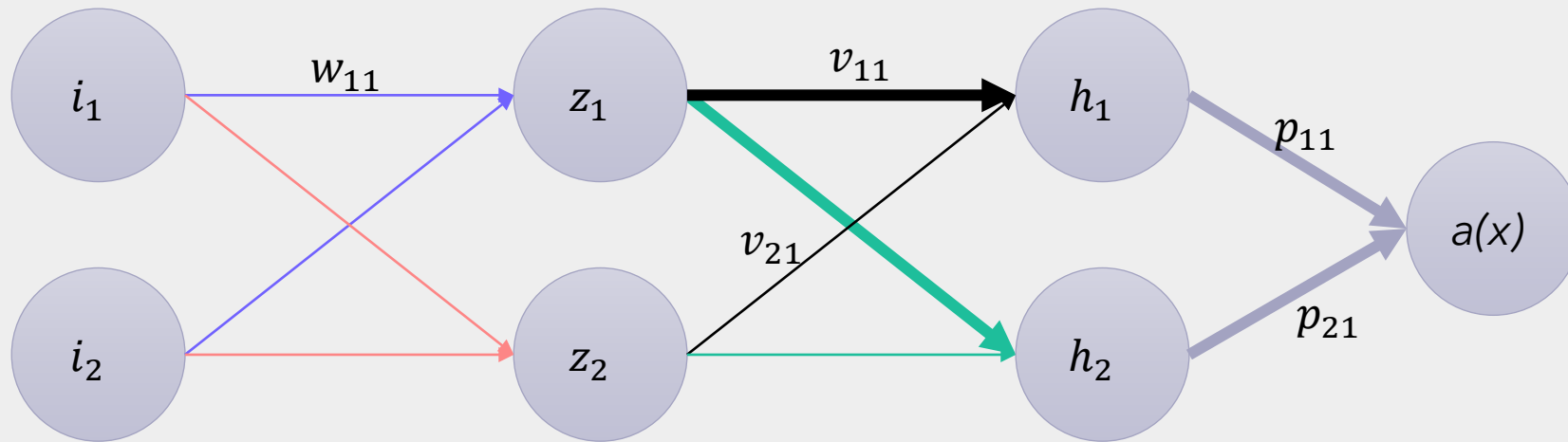
Дифференцируем сложную функцию: $\frac{\delta a}{\delta v_{11}} = \frac{\delta a}{\delta h_1} \cdot \frac{\delta h_1}{\delta v_{11}}$

Как считать производные?



$$\frac{\delta a}{\delta w_{11}} = ?$$

Как считать производные?



$$\frac{\delta a}{\delta w_{11}} = \frac{\delta a}{\delta h_1} \cdot \frac{\delta h_1}{\delta z_1} \cdot \frac{\delta z_1}{\delta w_{11}} + \frac{\delta a}{\delta h_2} \cdot \frac{\delta h_2}{\delta z_1} \cdot \frac{\delta z_1}{\delta w_{11}}$$

Мы идем в обратную сторону по графу и считаем производные

Backpropagation

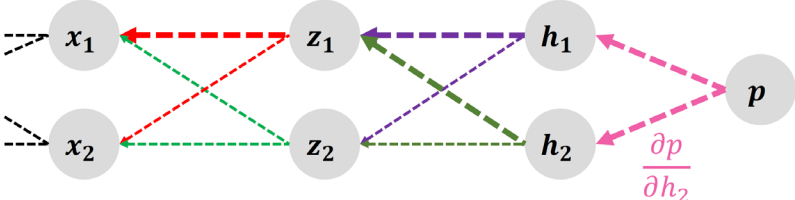
- Мы идем в обратную сторону по графу и считаем производные
- Такой метод и называется Backpropagation (backprop), метод обратного распространения ошибки (производные-то по функционалу ошибки считаем?)

3: $\frac{\partial p}{\partial h_1}$ $\frac{\partial p}{\partial h_2}$

2: $\frac{\partial p}{\partial z_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1}$ $\frac{\partial p}{\partial z_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2}$

1: $\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$

$\frac{\partial p}{\partial x_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_2}$



Многие производные попадаются несколько раз: если мы будем использовать императивный подход к вычислениям, нам придется их каждый раз пересчитывать, и сложность вычислений будет расти по экспоненте

Поэтому и используется символьный подход: мы сперва строим **граф вычислений**, а потом уже только подставляем в матрицу готовые производные

Разновидности градиентного спуска

- *Full GD*
- *Stochastic GD*
- *Mini Batch GD (MBGD)*
- *SGD with Momentum*
- *Nesterov SGD*
- *Adagrad*
- *RMSProp*
- *Adam*
- *NAdam*
- *AdamW*

Разновидности градиентного спуска

Full GD:

$$w^t = w^{t-1} - \eta \nabla Q(w^{t-1})$$

считаем по полной выборке: посчитали частные производные для каждого объекта и усреднили

Stochastic GD:

$$w^t = w^{t-1} - \eta \nabla Q(w^{t-1})$$

формула та же, но считаем на каждом шаге для одного случайного объекта

Mini Batch GD:

считаем на каждом шаге для случайного батча

[поиграться](#)

Разновидности градиентного спуска

Momentum:

$$g_t = \alpha g_{t-1} + \eta \frac{1}{m} \nabla_w L(f(x_i, w), y_i)$$
$$w = w - \eta \times g$$

Накапливаем момент инерции: каждый раз к градиенту добавляем какую-то часть от прошлого градиента (с весом α).

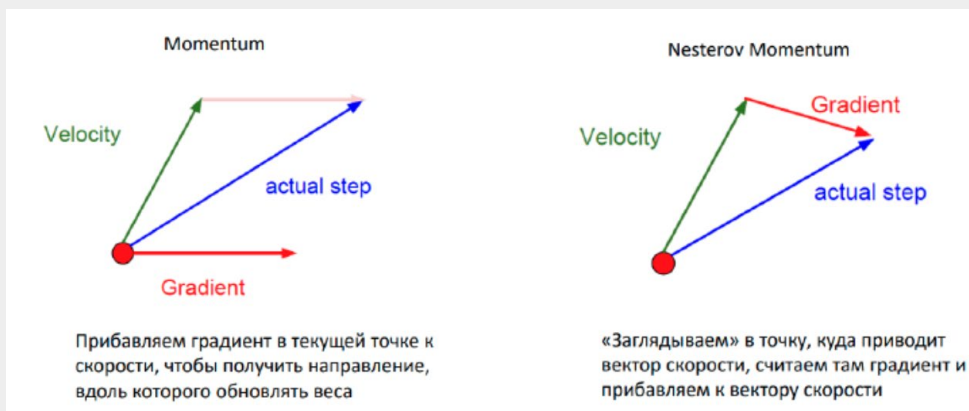


Разновидности градиентного спуска

Nesterov:

$$\begin{cases} m_{i+1} = \beta m_i - \gamma \Delta f(w_i + \beta m_i) \\ w_{i+1} = w_i + m_{i+1} \end{cases}$$

Добавляем момент по-другому, пытаясь предсказать, в какую сторону будем двигаться.



Адаптивные варианты градиентного спуска

Адаптивные варианты подстраивают learning rate в зависимости от обучения.

Adagrad:

$$G_j^t = G_j^{t-1} + (\nabla Q(w^{t-1}))_j^2$$
$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{G_j^t + \varepsilon}} (\nabla Q(w^{t-1}))_j$$

Чем больше сделано шагов по этому весу w , тем больше будет знаменатель и ниже learning rate.

Адаптивные варианты градиентного спуска

RMSProp:

его предложил Дж. [Хинтон](#), отец глубинного обучения.

Делаем все то же, что в Adagrad, только добавляем коэффициент затухания, чтобы накопленный градиент рос помедленнее.

$$G_j^t = \alpha G_j^{t-1} + (1 - \alpha) (\nabla Q(w^{t-1}))_j^2$$
$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{G_j^t + \varepsilon}} (\nabla Q(w^{t-1}))_j$$

Адаптивные варианты градиентного спуска

Adam:

Будем сочетать RMSProp и Momentum. Обычно работает лучше всего

NAdam:

Будем вычислять Momentum по Нестерову.

AdamW:

Это Адам с затуханием весов. В библиотеке torch есть параметр у обычного Adam `weight_decay`, но он реализован немножечко по-другому:

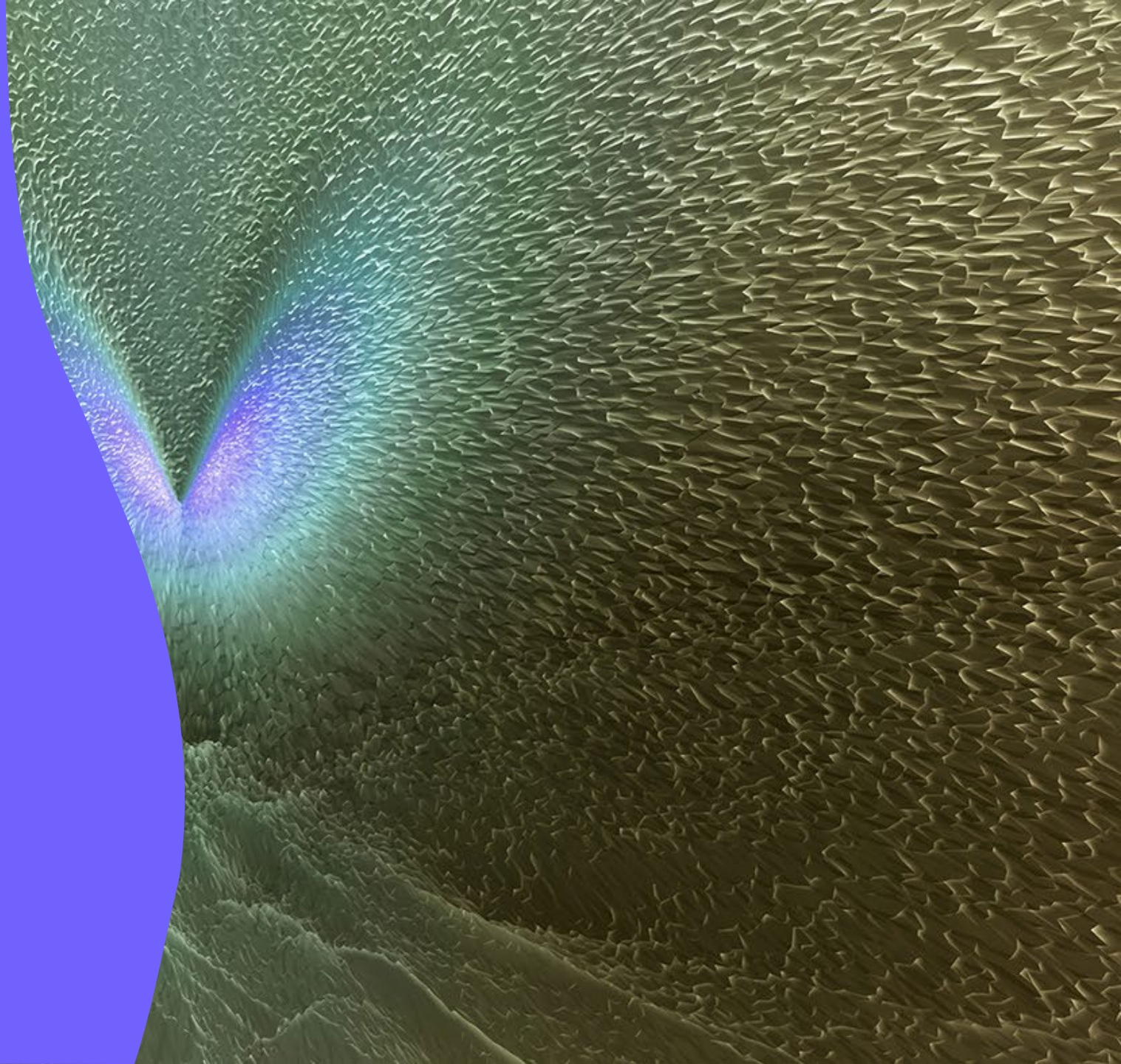
In Adam

`weight_decay` (float, optional) – weight decay (L2 penalty) (default: 0)

In AdamW

`weight_decay` (float, optional) – weight decay coefficient (default: 1e-2)

Функции активации

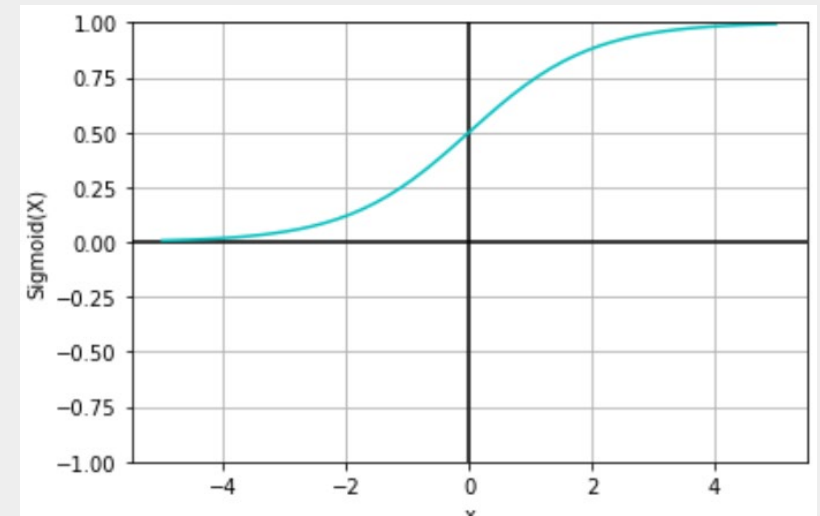


Сигмоида

Хорошая нелинейная функция, ее основной недостаток – на больших по модулю значениях, типа (-100) , ее касательная будет практически лежать, а значит, градиент будет очень маленьким числом. Этот артефакт называется *затухание градиентов* и для больших архитектур делает сигмоиду непригодной (может наступить *паралич сети*).

Соответственно, недостатки сигмоиды:

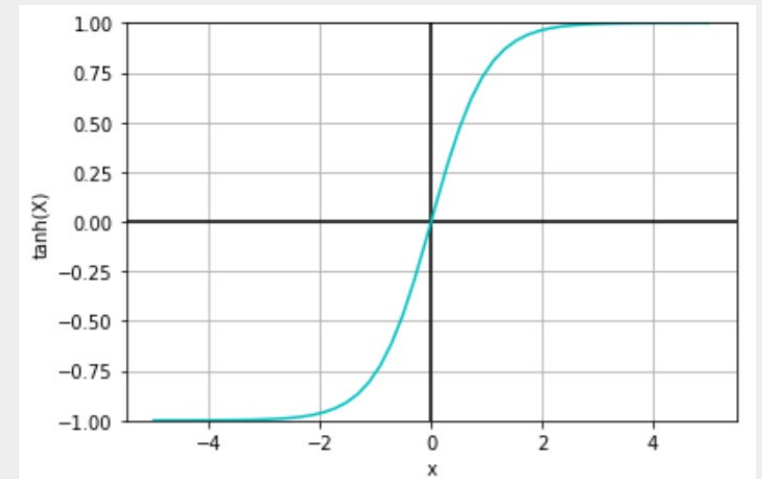
- Затухание градиента
- Не центрирована относительно нуля
- Вычислять градиент дорого



Тангенс

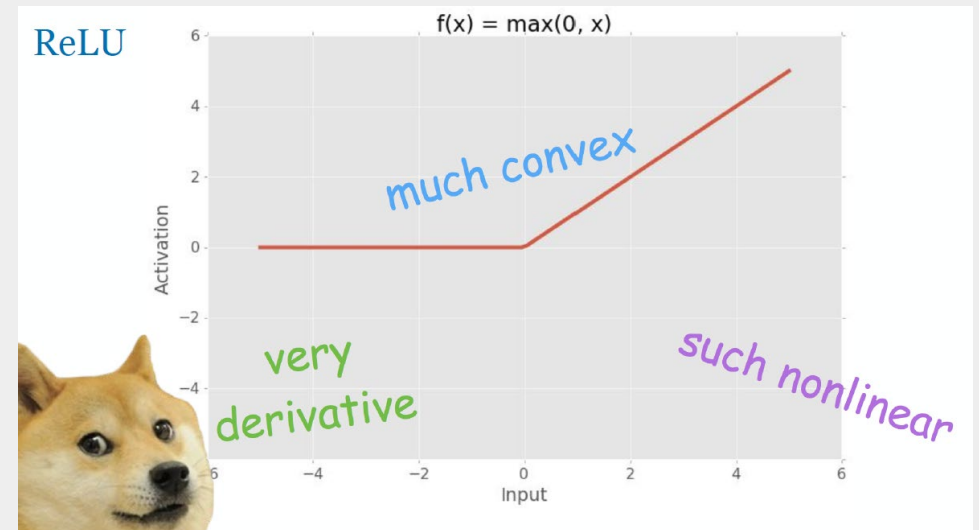
- Центрирован относительно нуля
- Все еще похож на сигмоиду
- Все равно будет затухать градиент, даже еще сильнее

Но тангенс используется в рекуррентных сетках.



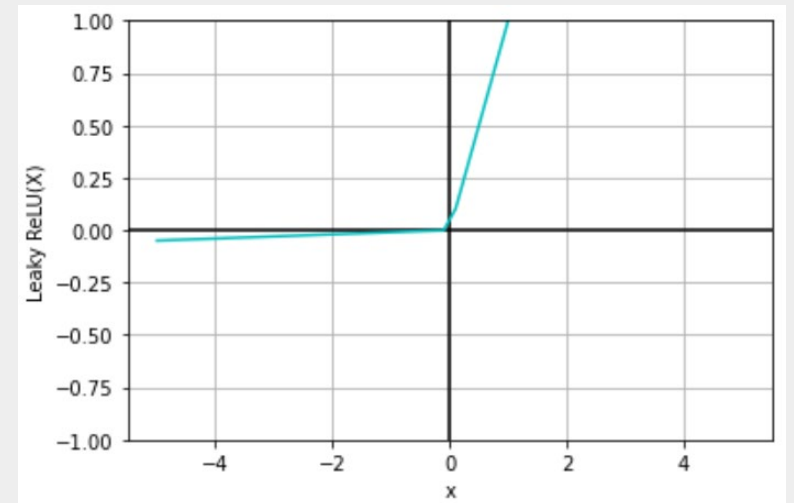
ReLU

- Быстро вычисляется
- Градиент не затухает
- Сходимость сеток ускоряется
- Сетка может умереть, если активация занулится на всех нейронах
- Не центрирован относительно нуля
- Если w_0 инициализировано большим отрицательным числом, нейрон сразу умирает \Rightarrow надо аккуратно инициализировать веса



Leaky ReLU

- Как ReLU, но не умирает, всё ещё легко считается
- Производная может быть любого знака
- Важно, чтобы $a \neq 1$, иначе линейность

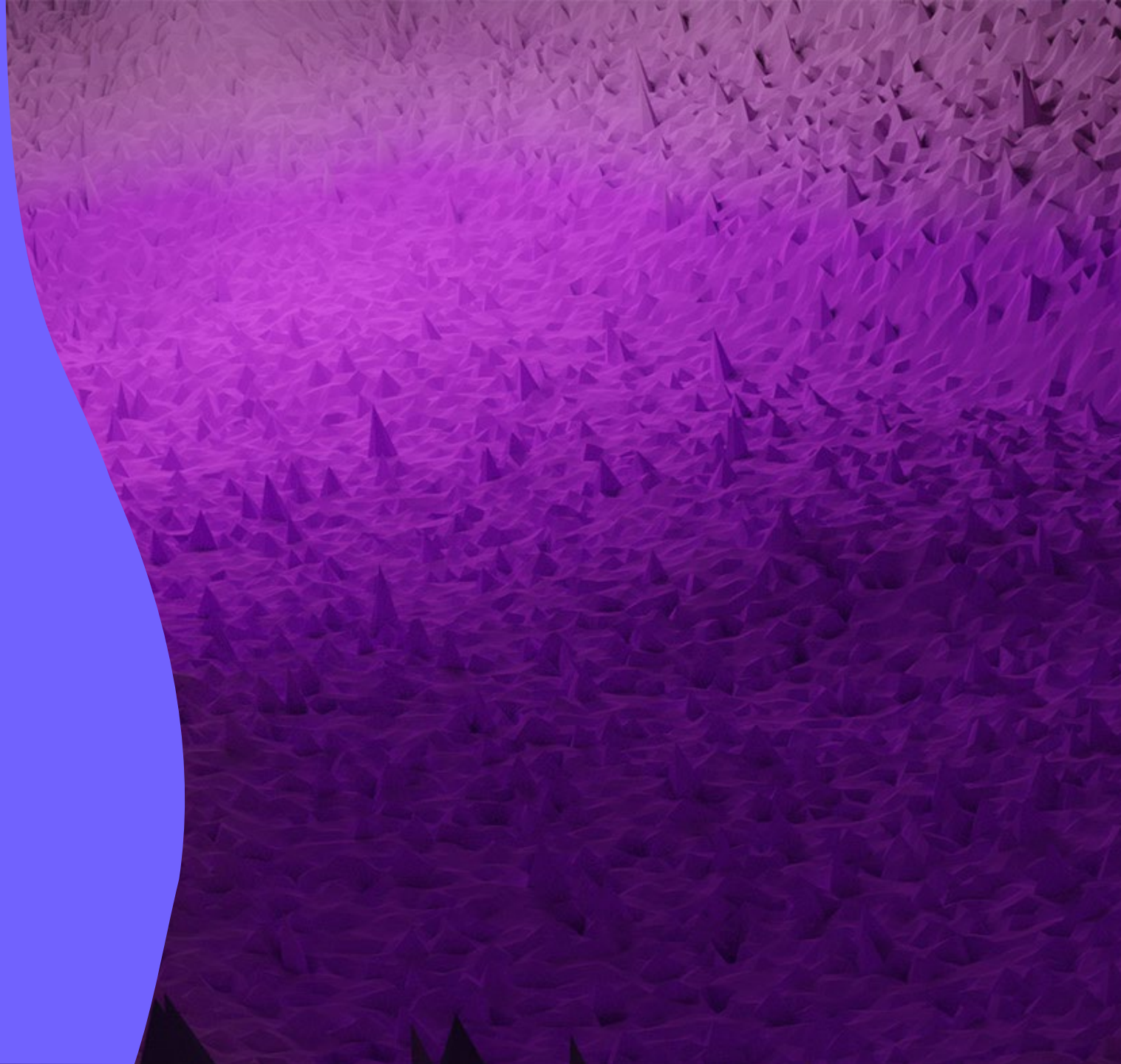


Как выбрать?

- Обычно начинают с *ReLU*, если сетка умирает, берут *LeakyReLU*
- *ReLU* – стандартный выбор для свёрточных сетей
- В рекуррентных сетках чаще всего предпочитается *tanh*
- На самом деле это не очень важно, нужно держать в голове свойства функций, о которых выше шла речь, и понимать, что от перебора функций обычно выигрыш в качестве довольно низкий. Но есть и исключения ...

[статья с обзором](#)

Инициализация весов



Инициализация весов

Зависит от того, симметричная у нас функция активации или нет.

Если да, то используется инициализация Ксавье Глоро

Если нет, то используется инициализация Кайминга Хе

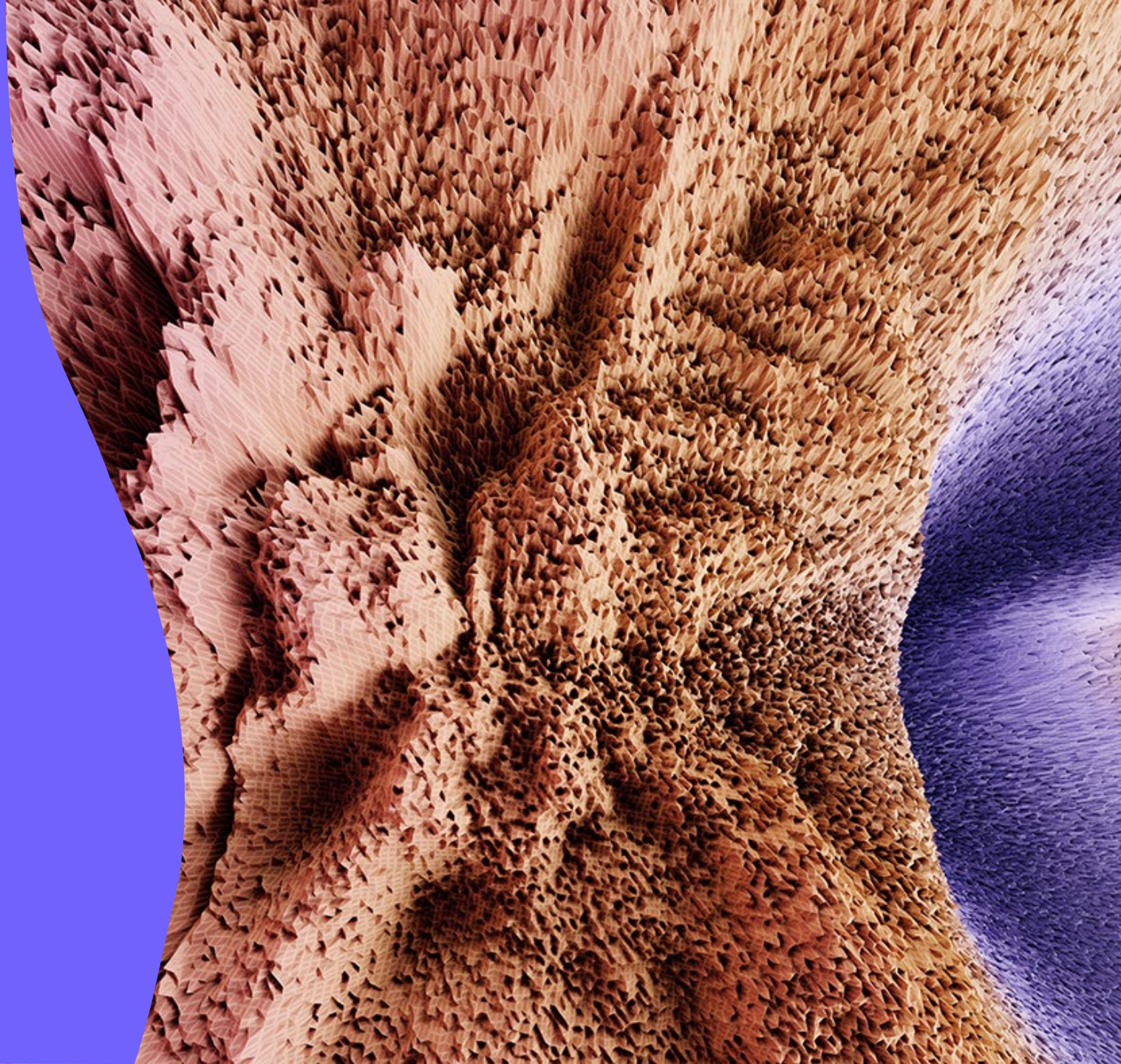
Для сверток используется инициализация ЛеКуна

Вам обычно не приходится об этом думать: у `torch` грамотно выставлены дефолтные значения.

Регуляризация и нормализация

Dropout

BatchNorm



Dropout

Наиболее частая проблема нейронных сетей – лютое, бешеное переобучение

Особенно если у нас полносвязные слои: вспоминаем, сколько у нас в сумме получается параметров

Как с этим бороться?

Дать сетке по башке! 👍

Dropout

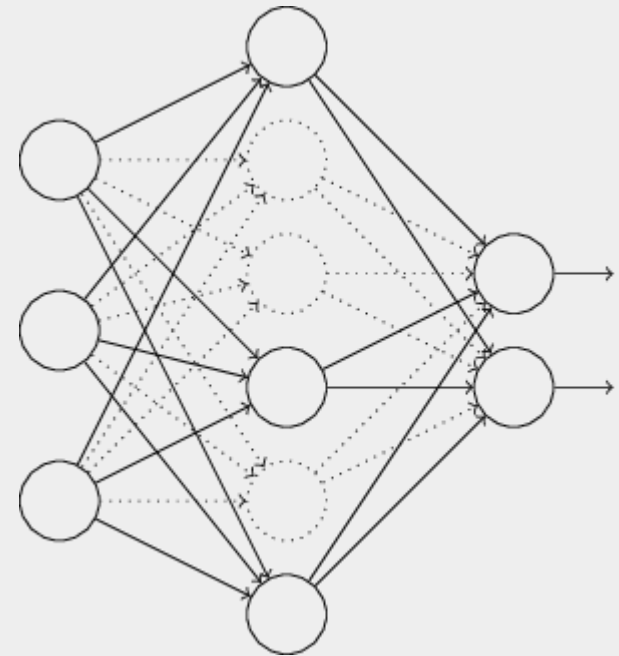
Дропаут: добавляем слой, в котором нет параметров, а его единственный гиперпараметр p – это вероятность зануления нейрона

На этапе обучения:

$$d(x) = \frac{1}{p} m \circ x$$

(m – вектор того же размера, что и x , элементы берутся из распределения $\text{Ber}(p)$)

Деление на p нужно для сохранения суммарного масштаба выходов



Dropout

- Интерпретация: мы обучаем все возможные архитектуры нейросетей, которые получаются из исходной выбрасыванием отдельных нейронов
- У всех этих архитектур общие веса
- На этапе применения (почти) усредняем прогнозы всех этих архитектур

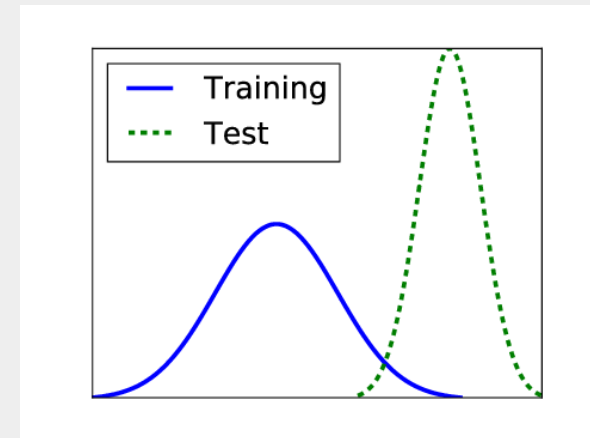
Covariate Shift

Covariate shift – изменение распределения данных.

Объекты могут быть по-разному распределены на обучении и на тестировании, распределение может изменяться после прохождения очередного слоя (internal covariate shift).

Методов решения много.

Internal Covariate Shift – изменение распределения данных между слоями.



BatchNorm

Придумали для этого делать Batch Normalisation:

- Реализуется как отдельный слой
- Вычисляется для текущего батча
- Оценим среднее и дисперсию каждой компоненты входного вектора:

$$\mu_B = \frac{1}{n} \sum_{j=1}^n x_{B,j}$$

$$\sigma_B^2 = \frac{1}{n} \sum_{j=1}^n (x_{b,j} - \mu_B)^2 : \text{покоординатно}$$

$x_{B,j}$ - j-й объект в батче

BatchNorm

Отмасштабируем все выходы:

$$\tilde{x}_{B,j} = \frac{x_{B,j} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Зададим нужные нам среднее и дисперсию:

$$z_{B,j} = \gamma \cdot \tilde{x}_{B,j} + \beta$$

Здесь γ и β – обучаемые параметры сети.

Итого:

- Позволяет увеличить длину шага в градиентном спуске
- *Не факт, что действительно устраняет covariance shift*

Куда лучше помещать?

Однозначного ответа на этот вопрос нет. Кто-то считает, что нужно помещать между слоем и активацией, кто-то говорит, что после активации.

В последнее время, судя по всему, преобладает второе мнение (но Женя Соколов считает, что до активации, а Дима Андросов и Влад Соловей считают, что после). Скорее всего, лучше проверять на практике, как будет работать.

Дропаут + нормализация: нормальная практика делать батчнорм, потом слой, потом дропаут без батчнорма