

# Глубокое обучение

Введение

# План курса. Лекции и семинары

1. Бэкроп, оптимизаторы, PyTorch
2. Инициализация, регуляризация, best practices
3. Conv layers, Pooling, Архитектуры: vgg, resnet, mobile net
4. Перенос обучения. Pruning, quantization. Дистилляция знаний. trt, onnx. Интерпретация моделей, Shapley
5. Dense prediction, UNet
6. Object detection, метрики
7. Word Embeddings (TF-IDF, word2vec), Tokenizers, Lemma, Stop words, Rnn
8. Attention, Transformer, Translator
9. BERT, adapters, LoRa, QLoRa, NER
10. LLM
11. ViT, DETR, CLIP, Swin, SAM
12. GenAI: GAN, VAE, диффузии
13. Representation learning
14. RecSys. Collaborative filtering, ALS.
15. Профориентация

# План курса. Домашки и система оценивания

1. Базовые функции pytorch,  
классификация/регрессия, методы  
регуляризации.

2. Сегментация. Ускорение модели.

3. NLP. Классификацию текстов.

4. GAN/VAE

5. Representation learning.

6. RecSys. ALS.

+ Экзамен

# Про авторов

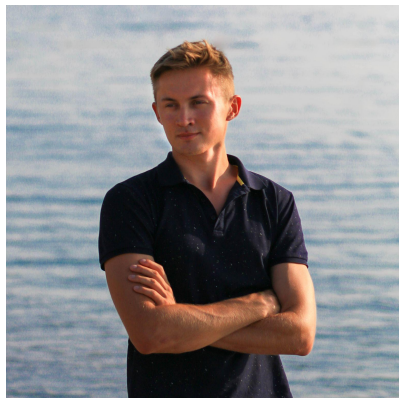


Батраков Юрий

@BatrakovYury

Выпускник МФТИ ФПМИ'23, ШАД'23

DS engineer Avito



Евдокимов Егор

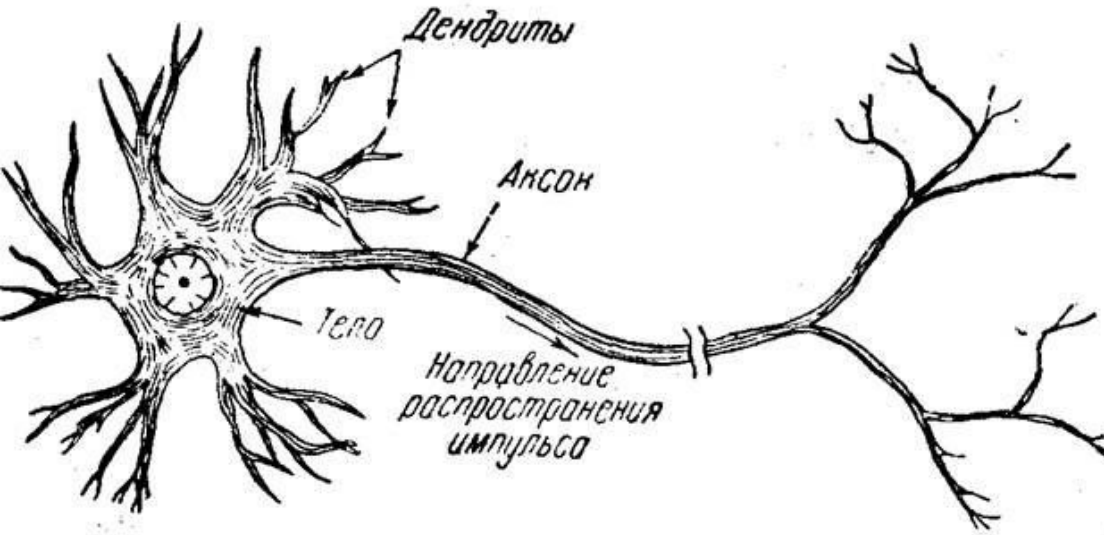
@ea\_evdokimov

Выпускник МФТИ ФПМИ'23, ШАД'25

DL engineer AutoTech

# Нейронные сети

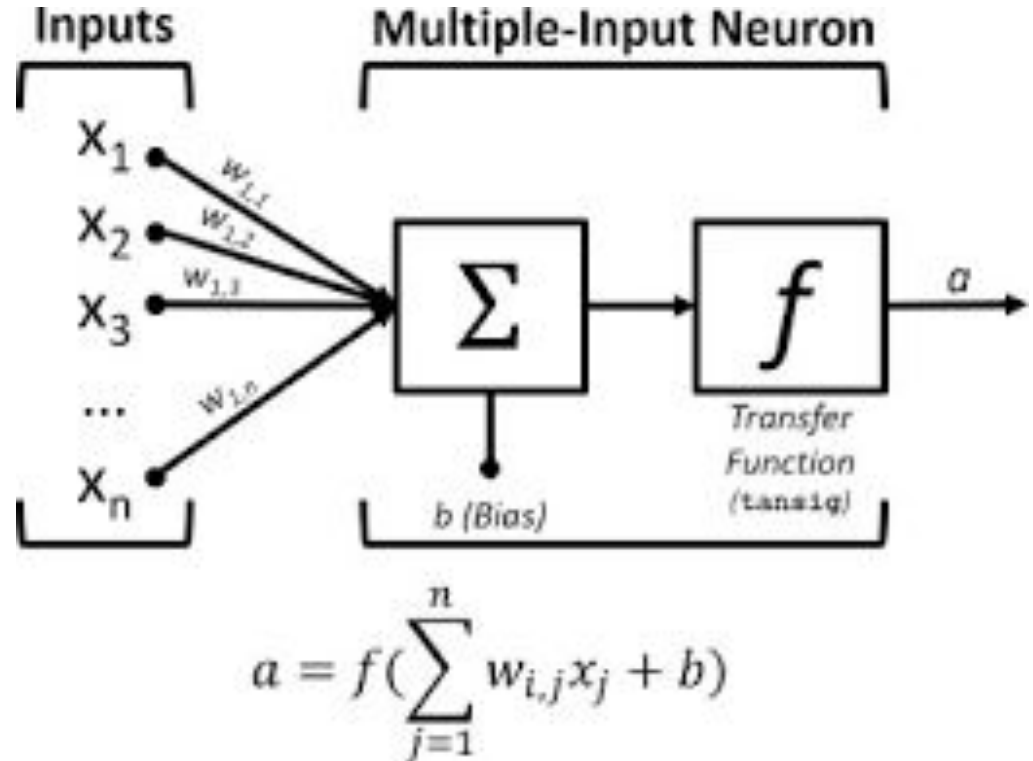
# Причем тут нейроны?



- Дендрит (приёмник), ядро (тело), аксон (передатчик)
- При активации, передаётся дальше
- Нейронов очень много (около 86 млрд)

# Причем тут нейроны?

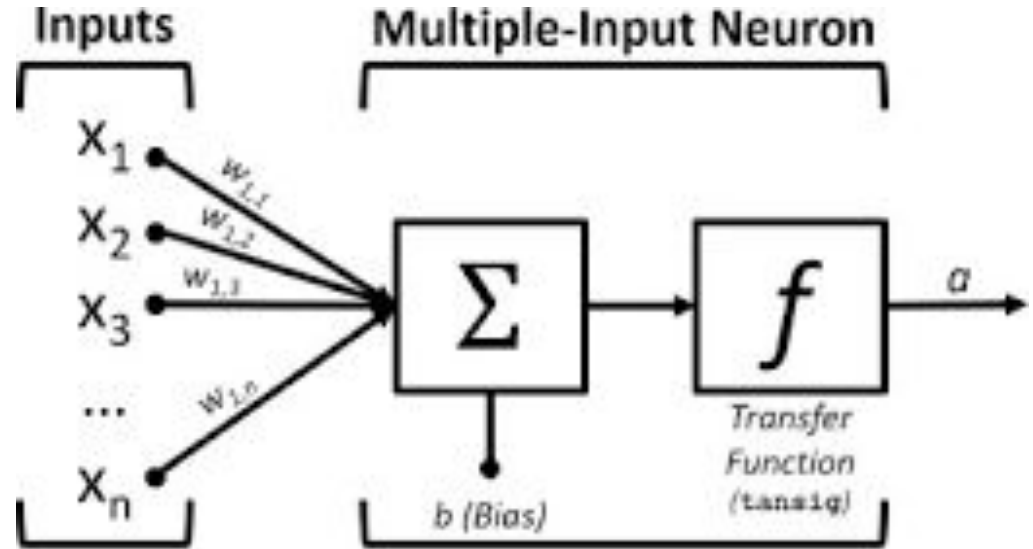
$(x_1, \dots, x_n)$  - векторное  
представление объекта,  $n$  -  
количество признаков  
 $(w_1, \dots, w_n)$  - коэффициенты  
 $b$  - смещение(bias)  
Легче представлять, что  
векторы  
 $(x_1, \dots, x_n, 1)$  для удобства  
умножения  
 $f$  – какая-то функция  
(функция активации)



# Причем тут нейроны?

$(x_1, \dots, x_n)$  - векторное  
представление объекта,  $n$  -  
количество признаков  
 $(w_1, \dots, w_n)$  - коэффициенты  
 $b$  - смещение(bias)  
Легче представлять, что  
векторы  
 $(1, x_1, \dots, x_n)$  для удобства  
умножения

**$f$**  – какая функция?



$$a = f\left(\sum_{j=1}^n w_{i,j}x_j + b\right)$$

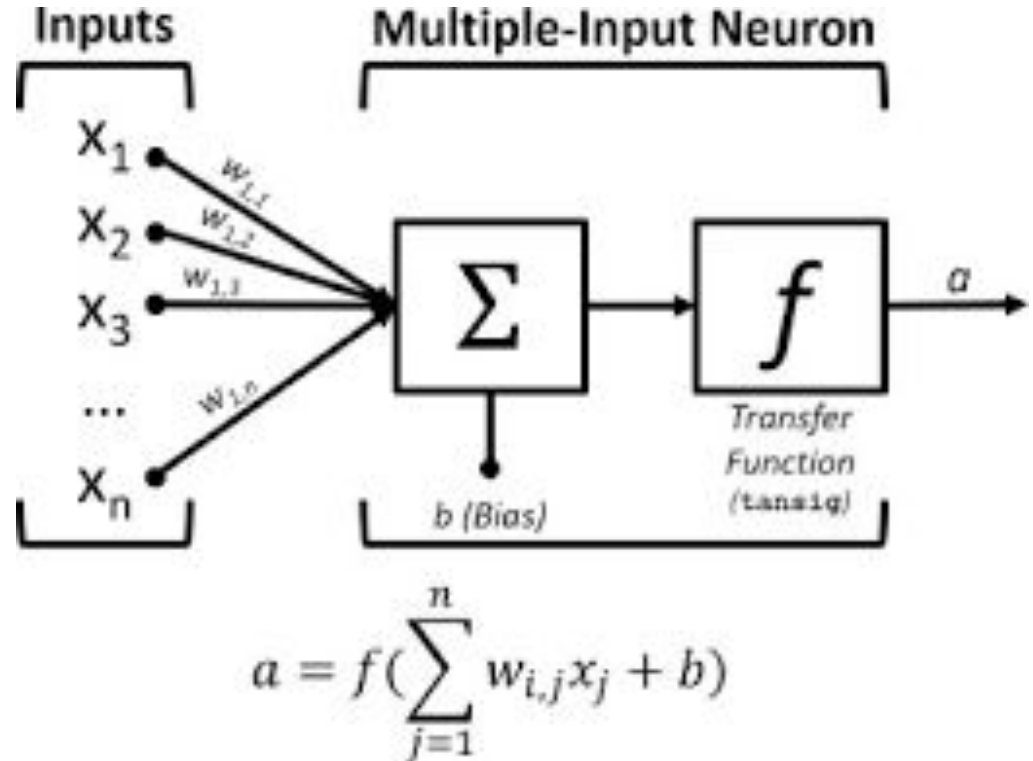


# Причем тут нейроны?

$(x_1, \dots, x_n)$  - векторное  
представление объекта,  $n$  -  
количество признаков  
 $(w_1, \dots, w_n)$  - коэффициенты  
 $b$  - смещение(bias)  
Легче представлять, что  
векторы  
 $(1, x_1, \dots, x_n)$  для удобства  
умножения

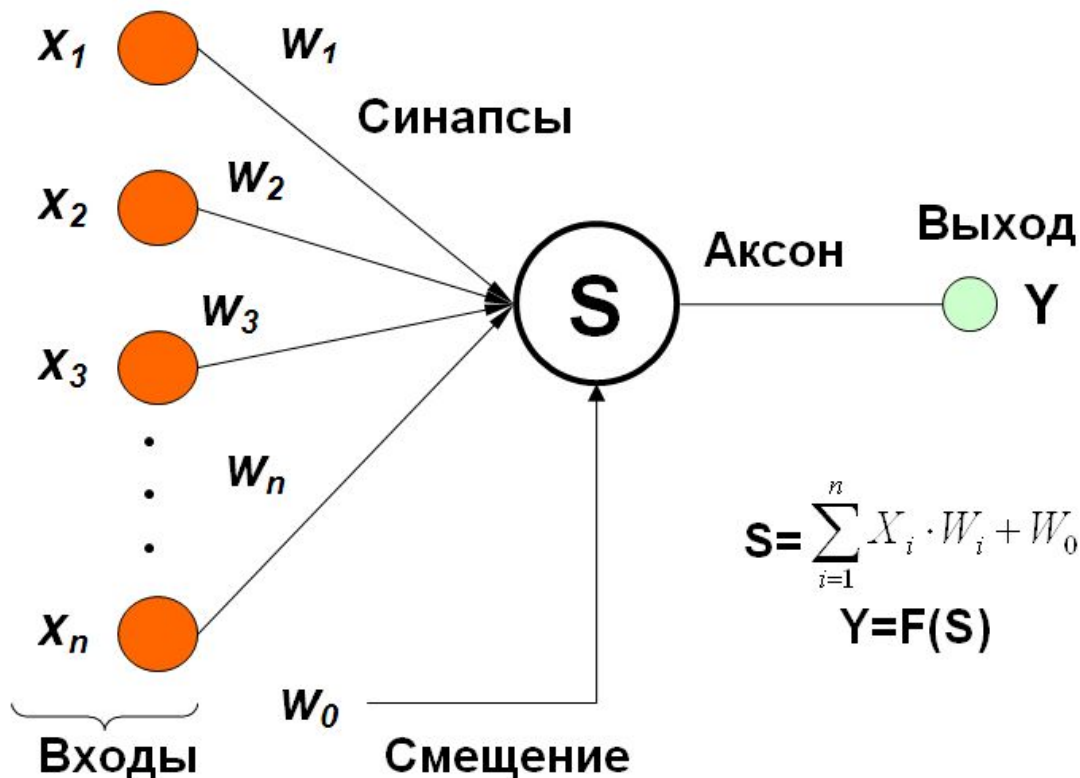
**$f$  – какая функция?**

кусочно-дифференцируемая,  
нелинейная



# Причем тут нейроны?

Полная аналогия с биологией

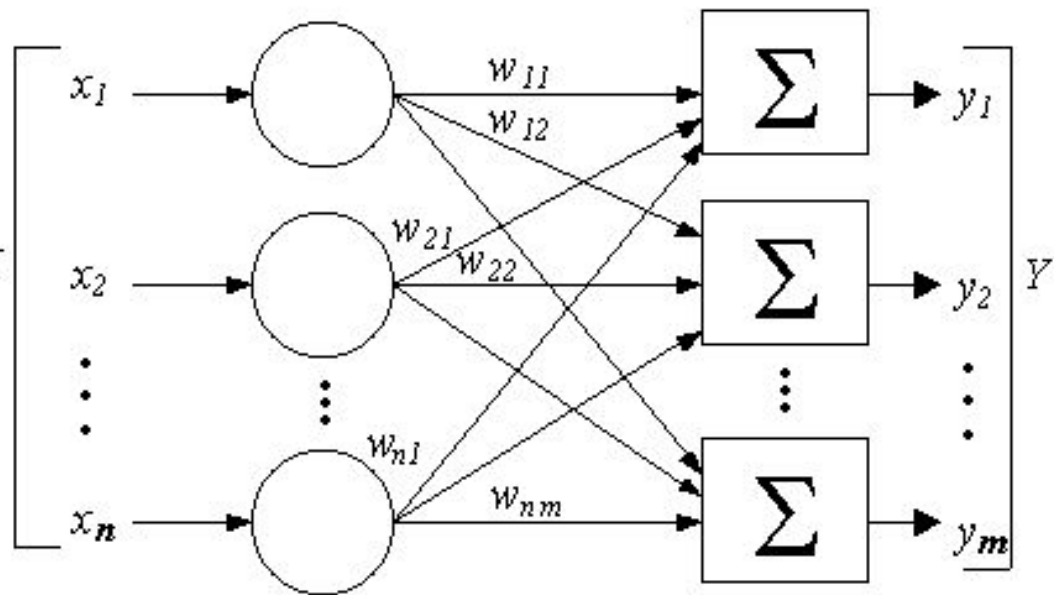


# Причем тут нейроны?

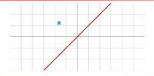


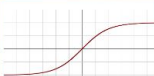




Пусть теперь у нас  $m$  выходов

Тогда эту операцию можно представить как

- Матрица весов:  $W = \begin{bmatrix} w_{0;1} & \cdots & w_{0;m} \\ \vdots & \ddots & \vdots \\ w_{d;1} & \cdots & w_{d;m} \end{bmatrix} \in \mathbb{R}^{(d+1) \times m}$ .
- $\sigma(z_1, \dots, z_m) \sim (\sigma(z_1), \dots, \sigma(z_m))$ .
- Тогда  $y = \sigma(x^T * W)$ .

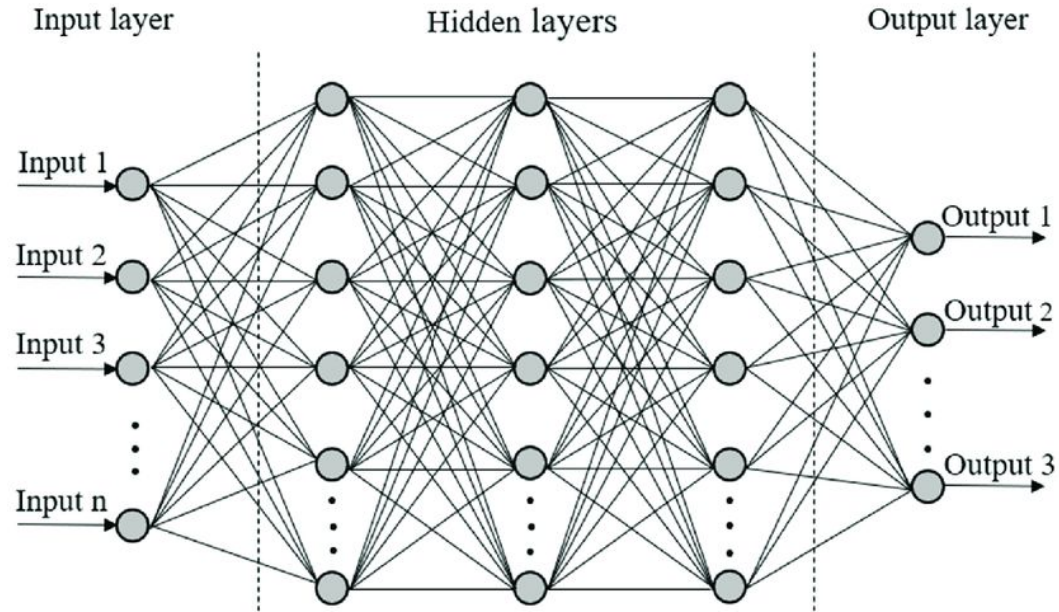


# Функции активации

| ACTIVATION FUNCTION          | PLOT   | EQUATION   | DERIVATIVE  | RANGE               |
|------------------------------|--|--|---|---------------------|
| Linear                       |   | $f(x) = x$   | $f'(x) = 1$   | $(-\infty, \infty)$ |
| Binary Step                  |   | $f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$               | $f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$  | $\{0, 1\}$          |
| Sigmoid                      |   | $f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$  | $f'(x) = f(x)(1 - f(x))$  | $(0, 1)$            |
| Hyperbolic Tangent(tanh)     |   | $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$  | $f'(x) = 1 - f(x)^2$  | $(-1, 1)$           |
| Rectified Linear Unit(ReLU)  |   | $f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$               | $f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$                   | $[0, \infty)$       |
| Softplus                     |   | $f(x) = \ln(1 + e^x)$  | $f'(x) = \frac{1}{1 + e^{-x}}$  | $(0, 1)$            |
| Leaky ReLU                   |   | $f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$           | $f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$  | $(-1, 1)$           |
| Exponential Linear Unit(ELU) |  | $f(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ | $f'(x) = \begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$ | $[0, \infty)$       |

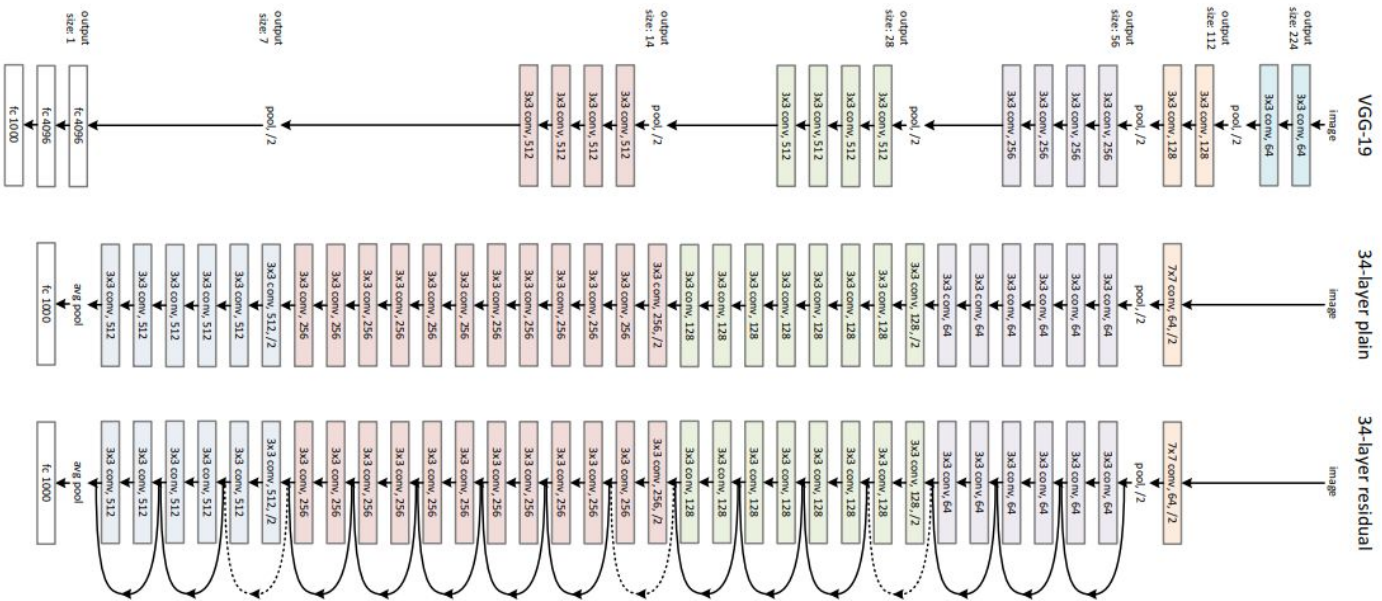
# Почему сети?

Обычно в сетях не один, а гораздо больше слоев



# Почему глубокие сети?

Обычно в сетях не один, а гораздо БОЛЬШЕ слоев



# Что известно из теории? Теорема Цыбенко

## Формальное изложение [\[ править \]](#) [\[ править код \]](#)

Пусть  $\varphi$  любая непрерывная [сигмоидная функция](#), например,  $\varphi(\xi) = 1/(1 + e^{-\xi})$ . Тогда, если дана любая непрерывная функция действительных переменных  $f$  на  $[0, 1]^n$  (или любое другое компактное подмножество  $\mathbb{R}^n$ ) и  $\varepsilon > 0$ , то существуют векторы  $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_N$ ,  $\alpha$  и  $\theta$  и параметризованная функция  $G(\cdot, \mathbf{w}, \alpha, \theta) : [0, 1]^n \rightarrow \mathbb{R}$  такая, что для всех  $\mathbf{x} \in [0, 1]^n$  выполняется

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

где

$$G(\mathbf{x}, \mathbf{w}, \alpha, \theta) = \sum_{i=1}^N \alpha_i \varphi(\mathbf{w}_i^T \mathbf{x} + \theta_i),$$

и  $\mathbf{w}_i \in \mathbb{R}^n$ ,  $\alpha_i, \theta_i \in \mathbb{R}$ ,  $\mathbf{w} = (\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_N)$ ,  $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_N)$ , и  $\theta = (\theta_1, \theta_2, \dots, \theta_N)$ .

# Что известно из теории?

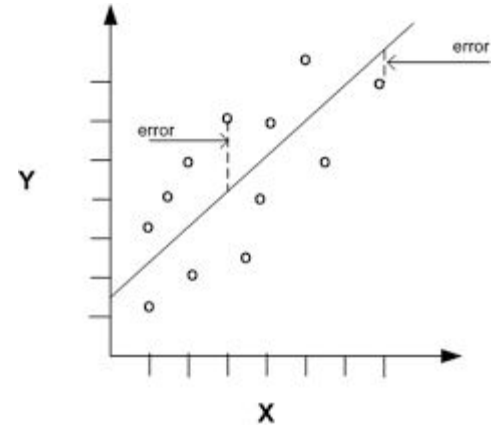
То есть нейронная сеть с одним скрытым слоем может аппроксимировать любую непрерывную функцию многих переменных с любой точностью. Условиями являются: достаточное количество нейронов скрытого слоя, удачный подбор весов.



Обучение сети

# Как оценить качество имеющегося решения?

| Task           | Error type                  | Loss function   |
|----------------|-----------------------------|---|
| Regression     | Mean-squared error          | $\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$                                      |
|                | Mean absolute error         | $\frac{1}{n} \sum_{i=1}^n  y_i - \hat{y}_i $  |
| Classification | Cross entropy =<br>Log loss | $-\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] =$ |



Как это обучать?

$$w^* = \arg \min_w L(w)$$

# Методы оптимизации

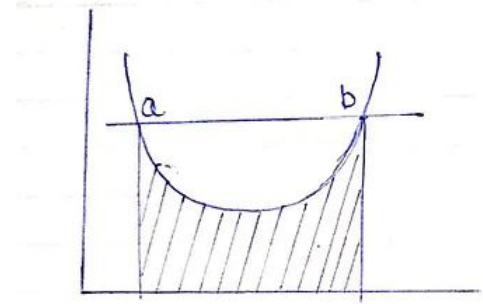
Найти глобальный минимум невыпуклой функции – очень трудная задача, но зачастую нам хватает локального, который является, в частности, стационарной точкой: такой, в которой производная равна нулю.

большинство алгоритмов оптимизации, придуманных для выпуклого случая, дословно перешли в невыпуклый

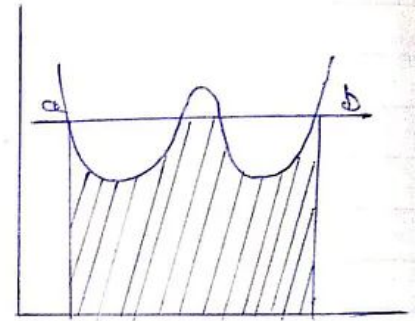
Причина номер 1: сойтись в локальный минимум лучше, чем никуда. Об этом речь уже шла.

Причина номер 2: в окрестности локального минимума функция становится выпуклой, и там мы сможем быстро сойтись.

Причина номер 3: иногда невыпуклая функция является в некотором смысле «зашумленной» версией выпуклой или похожей на выпуклую.

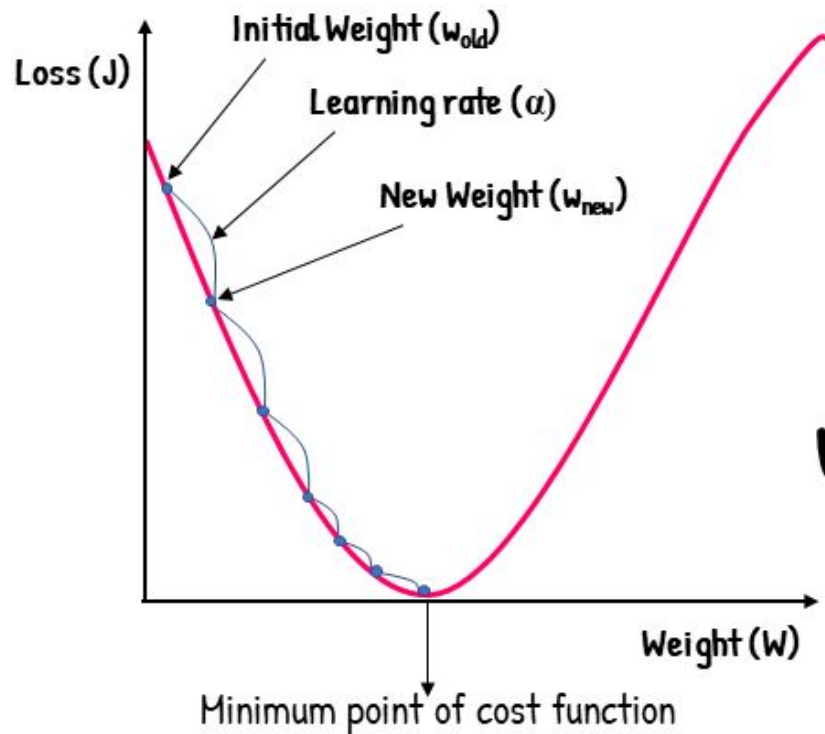


convex  
function  
CS Scanned with CamScanner



not convex  
function

# Градиентный спуск (GD)



$$w_{new} = w_{old} - \alpha \frac{\delta J}{\delta w}$$

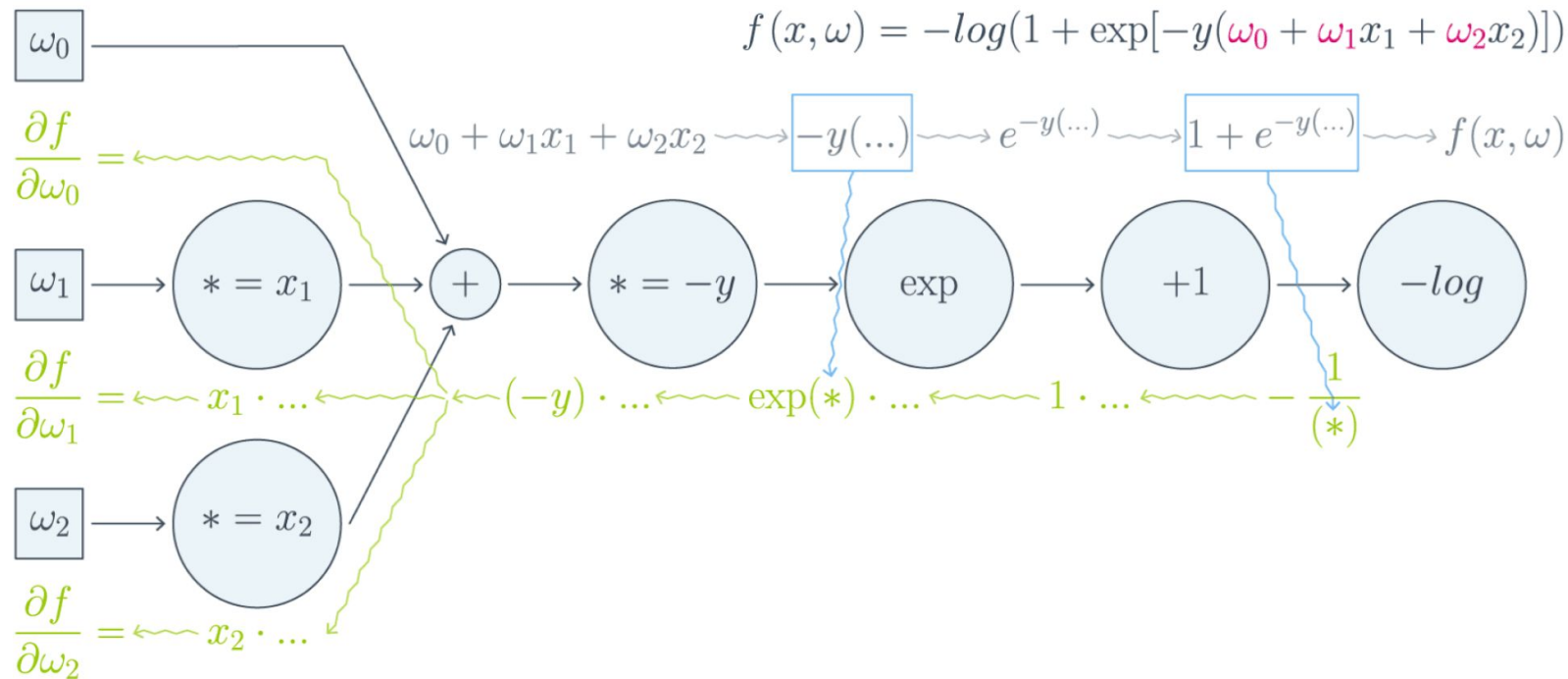
## Как посчитать производную? Backpropagation

$$f(w_0) = g_m(g_{m-1}(\dots g_1(w_0) \dots))$$

$$f'(w_0) = g'_m(g_{m-1}(\dots g_1(w_0) \dots)) \cdot g'_{m-1}(g_{m-2}(\dots g_1(w_0) \dots)) \cdot \dots \cdot g'_1(w_0)$$

$$g_1(w_0), g_2(g_1(w_0)), \dots, g_{m-1}(\dots g_1(w_0) \dots)$$

# Как посчитать производную? Backpropagation



## Как посчитать производную? Backpropagation

$$\frac{\partial f}{\partial w_0} = (-y) \cdot e^{-y(w_0 + w_1 x_1 + w_2 x_2)} \cdot \frac{-1}{1 + e^{-y(w_0 + w_1 x_1 + w_2 x_2)}}$$

$$\frac{\partial f}{\partial w_1} = x_1 \cdot (-y) \cdot e^{-y(w_0 + w_1 x_1 + w_2 x_2)} \cdot \frac{-1}{1 + e^{-y(w_0 + w_1 x_1 + w_2 x_2)}}$$

$$\frac{\partial f}{\partial w_2} = x_2 \cdot (-y) \cdot e^{-y(w_0 + w_1 x_1 + w_2 x_2)} \cdot \frac{-1}{1 + e^{-y(w_0 + w_1 x_1 + w_2 x_2)}}$$



# Backpropagation. В чем проблема?

Рассмотрим простой пример. Допустим, что  $X^r$  и

$X^{r+1}$  — два последовательных промежуточных представления  $N \times M$  и  $N \times K$ , связанных функцией  $X^{r+1} = f^{r+1}(X^r)$ . Предположим, что мы как-то посчитали производную  $\frac{\partial \mathcal{L}}{\partial X_{ij}^{r+1}}$

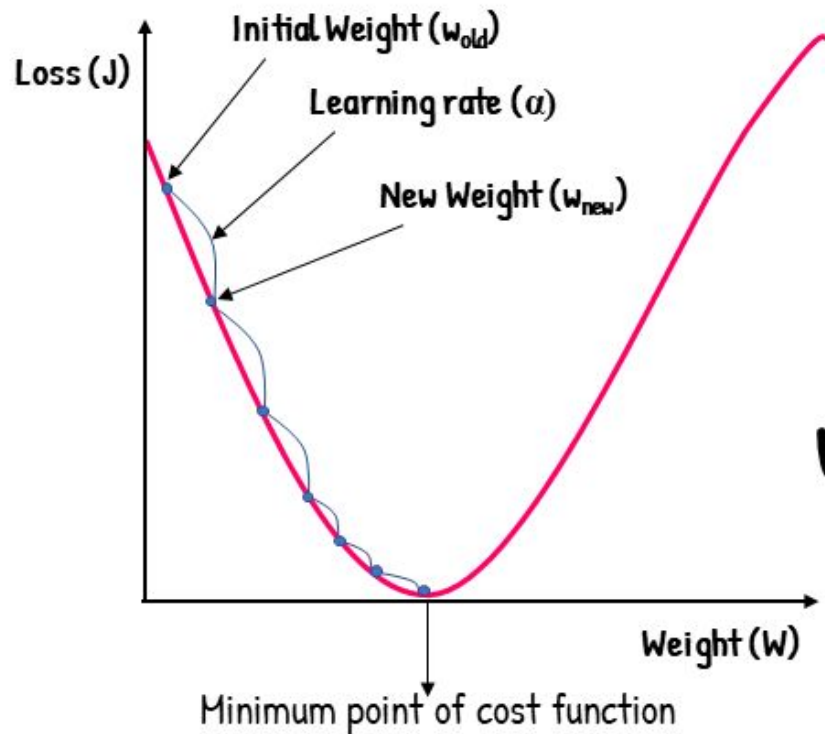
функции потерь  $\mathcal{L}$ , тогда

$$\frac{\partial \mathcal{L}}{\partial X_{st}^r} = \sum_{i,j} \frac{\partial f_{ij}^{r+1}}{\partial X_{st}^r} \frac{\partial \mathcal{L}}{\partial X_{ij}^{r+1}}$$

И мы видим, что, хотя оба градиента  $\frac{\partial \mathcal{L}}{\partial X_{ij}^{r+1}}$  и  $\frac{\partial \mathcal{L}}{\partial X_{st}^r}$  являются просто матрицами, в ходе вычислений возникает «четырёхмерный кубик»  $\frac{\partial f_{ij}^{r+1}}{\partial X_{st}^r}$ , даже хранить который весьма болезненно: уж больно много памяти он требует ( $N^2 MK$  по сравнению с безобидными  $NM + NK$ , требуемыми для хранения градиентов). Поэтому хочется промежуточные производные  $\frac{\partial f_{ij}^{r+1}}{\partial X_{st}^r}$  рассматривать не как вычисляемые объекты  $\frac{\partial f_{ij}^{r+1}}{\partial X_{st}^r}$ , а как преобразования, которые превращают  $\frac{\partial \mathcal{L}}{\partial X_{ij}^{r+1}}$  в  $\frac{\partial \mathcal{L}}{\partial X_{st}^r}$ .

# Методы выпуклой оптимизации

# Градиентный спуск (GD)



$$w_{new} = w_{old} - \alpha \frac{\delta J}{\delta w}$$

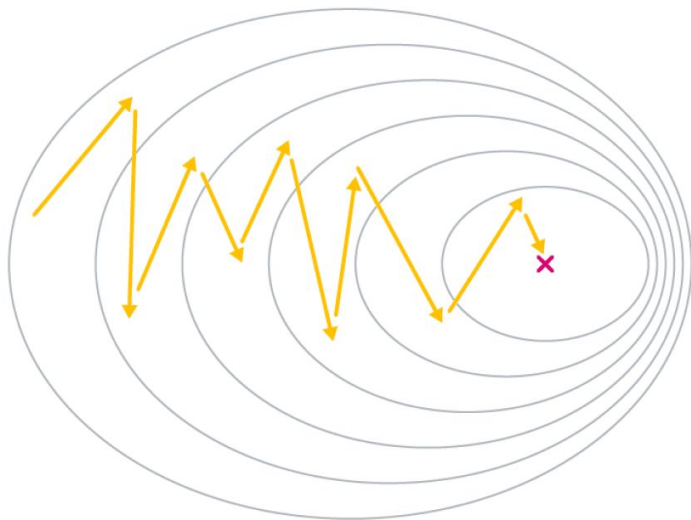
# Стохастический градиентный спуск (SGD)

В SGD вместо использования всего набора данных для каждой итерации выбирается только один случайный обучающий пример (или небольшой batch) для вычисления градиента и обновления параметров модели. Этот случайный выбор привносит случайность в процесс оптимизации, отсюда и термин “стохастический”.

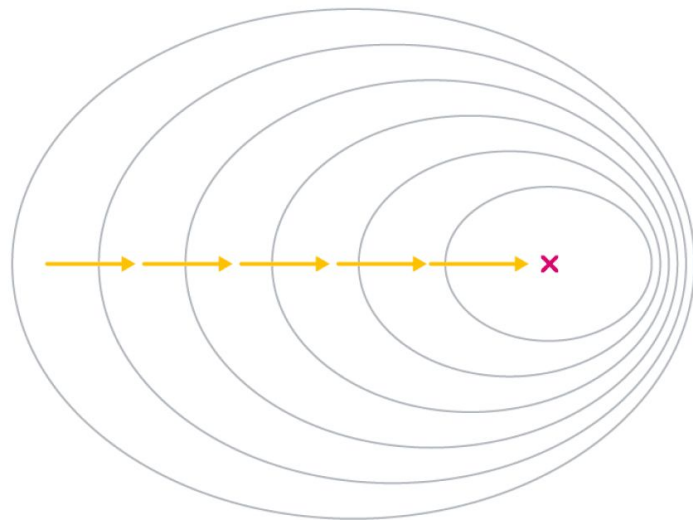
Преимуществом использования SGD является его вычислительная эффективность, особенно при работе с большими наборами данных. При использовании одного примера или небольшого батча вычислительных затрат на итерацию значительно снижаются по сравнению с традиционными методами градиентного спуска, которые требуют обработки всего набора данных.

# Стохастический градиентный спуск (SGD)

Stochastic Gradient Descent



Gradient Descent



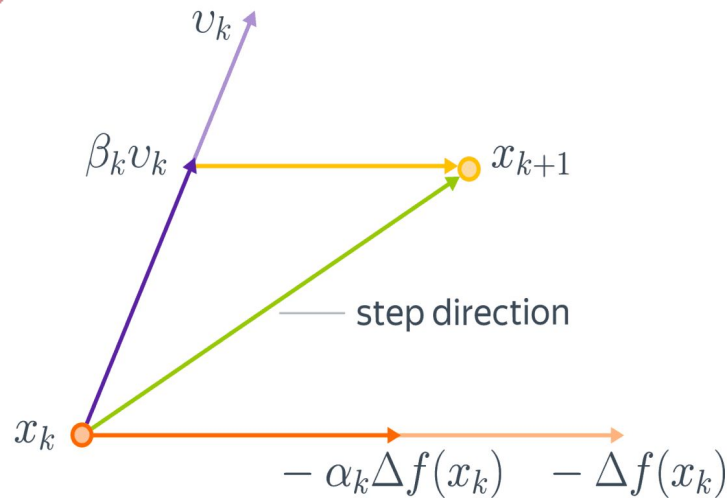
# Метод инерции, momentum

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) + \beta_k (x_k - x_{k-1})$$

Выгода от метода инерции довольно понятна – мы будем пропускать паразитные локальные минимумы и седла и продолжать движение вниз.

$$v_{k+1} = \beta_k v_k - \alpha_k \nabla f(x_k)$$

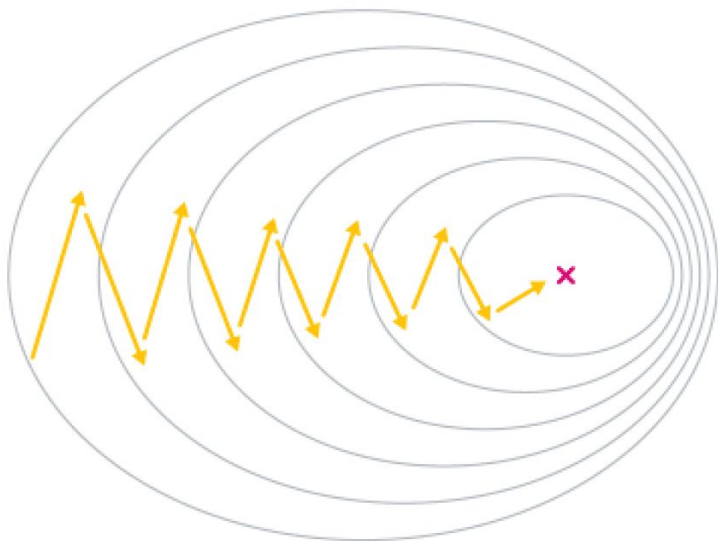
$$x_{k+1} = x_k + v_{k+1}.$$



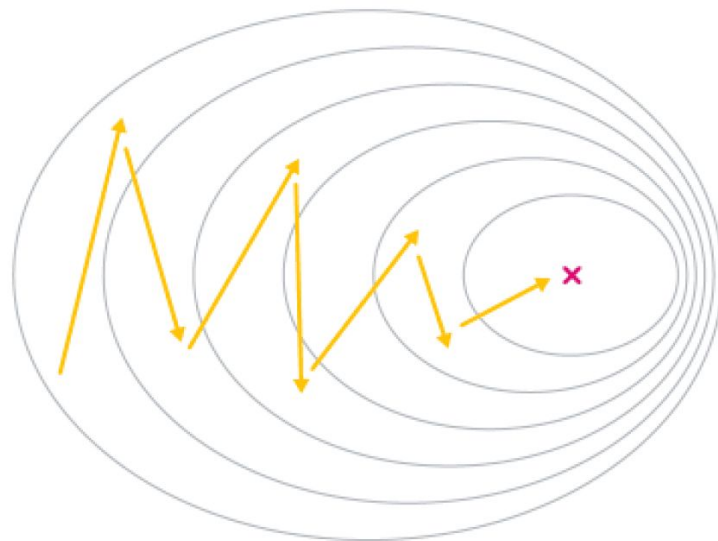
**Momentum**

# Метод инерции, momentum

SGD without momentum

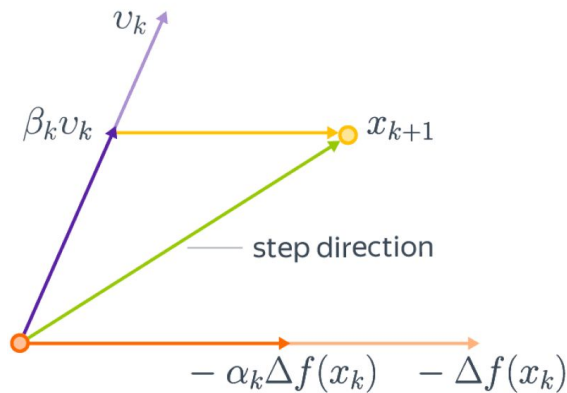


SGD with momentum

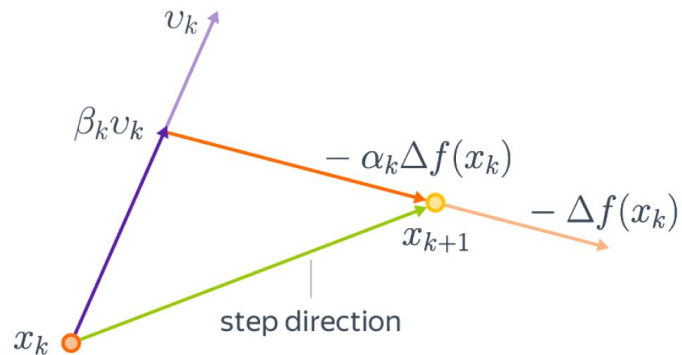


# Accelerated Gradient Descent (Nesterov Momentum)

$$v_{k+1} = \beta_k v_k - \alpha_k \nabla f(x_k + \beta_k v_k)$$
$$x_{k+1} = x_k + v_{k+1}$$



Momentum



Nesterov momentum



# Accelerated Gradient Descent (Nesterov Momentum)

Больше объяснений и визуализаций по ссылке

<https://towardsdatascience.com/gradient-descent-explained-9b953fc0d2c>

# Адаптивный подбор размера шага. Adagrad

Идея следующая: если мы вышли на плато по какой-то координате и соответствующая компонента градиента начала затухать, то нам нельзя уменьшать размер шага слишком сильно, поскольку мы рискуем на этом плато остаться, но в то же время уменьшать надо, потому что это плато может содержать оптимум. Если же градиент долгое время довольно большой, то это может быть знаком, что нам нужно уменьшить размер шага, чтобы не пропустить оптимум. Поэтому мы стараемся компенсировать слишком большие или слишком маленькие координаты градиента.

$$G_{k+1} = G_k + (\nabla f(x_k))^2$$
$$x_{k+1} = x_k - \frac{\alpha}{\sqrt{G_{k+1} + \varepsilon}} \nabla f(x_k).$$

# Адаптивный подбор размера шага. RMSProp

будем не просто складывать нормы градиентов, а усреднять их в *скользящем режиме*:

$$G_{k+1} = \gamma G_k + (1 - \gamma)(\nabla f(x_k))^2$$
$$x_{k+1} = x_k - \frac{\alpha}{\sqrt{G_{k+1} + \varepsilon}} \nabla f(x_k).$$

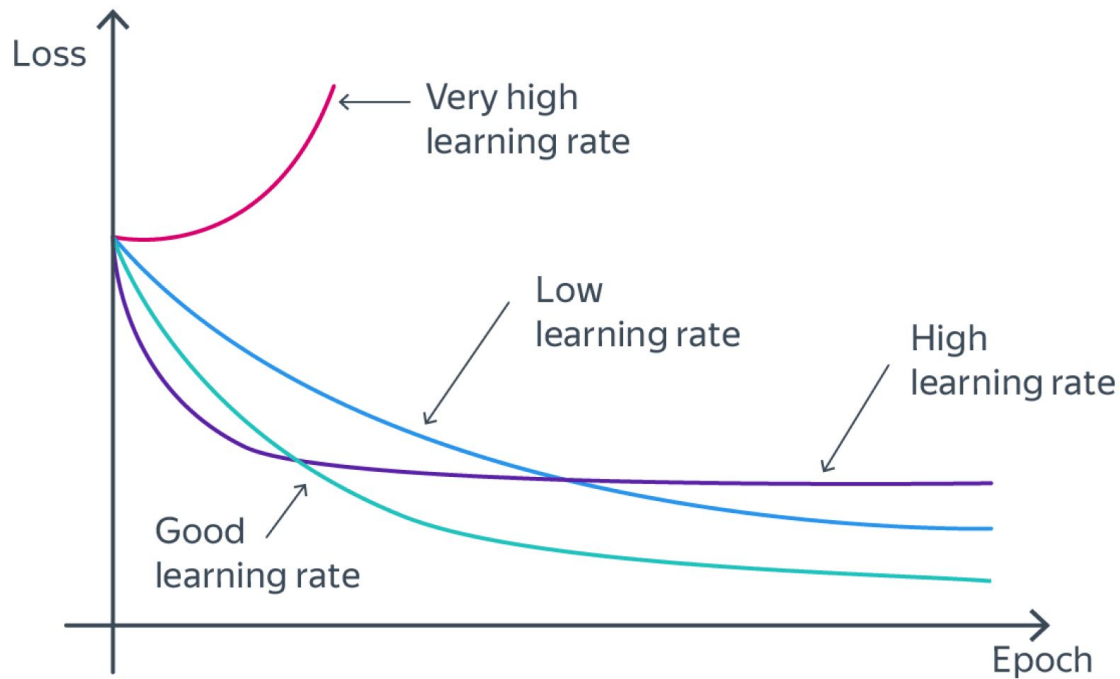
# Адаптивный подбор размера шага. Adam

ADaptive Momentum

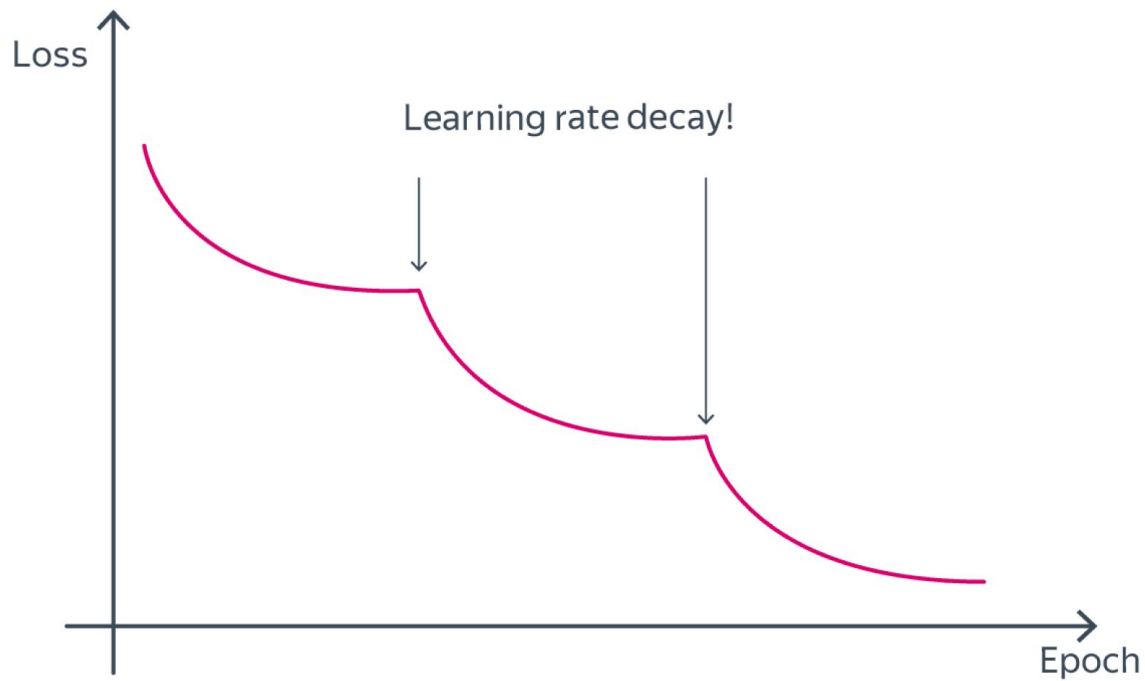
$$\begin{aligned}v_{k+1} &= \beta_1 v_k + (1 - \beta_1) \nabla f(x_k) \\G_{k+1} &= \beta_2 G_k + (1 - \beta_2) (\nabla f(x_k))^2 \\x_{k+1} &= x_k - \frac{\alpha}{\sqrt{G_{k+1} + \varepsilon}} v_{k+1}.\end{aligned}$$

# Нюансы обучения

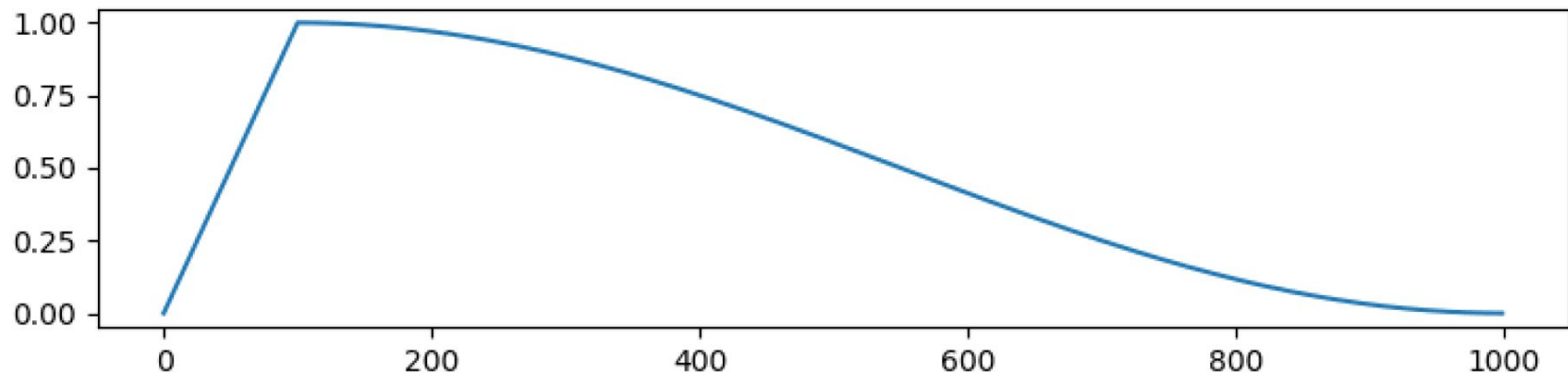
# Интерпретация learning curve



# LR decay

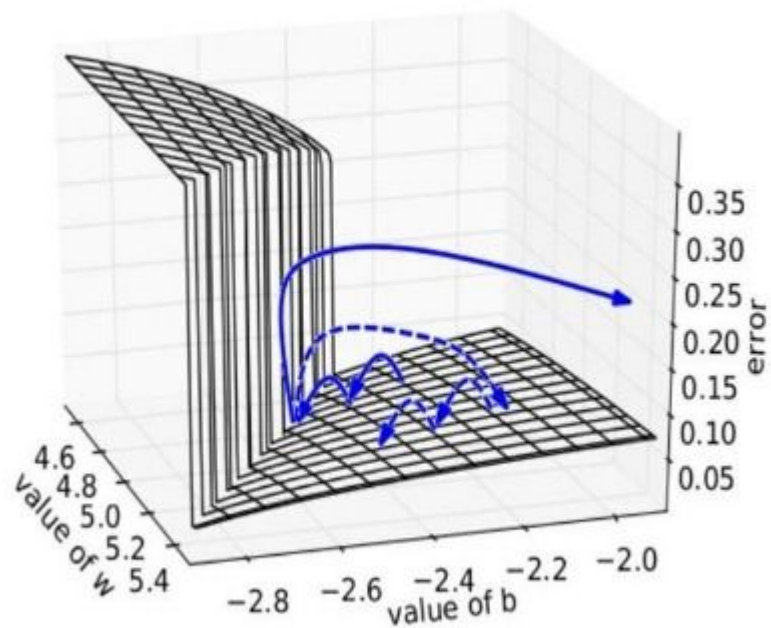


# Warmup

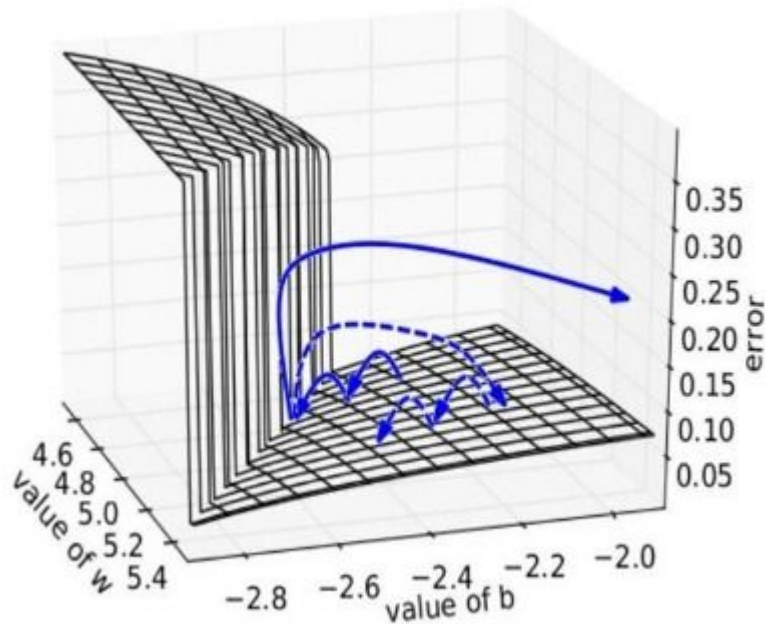




# Gradient explosion



# Gradient explosion



Решения:

1. Регуляризация
2. Gradient clipping

---

**Algorithm 1** Pseudo-code for norm clipping

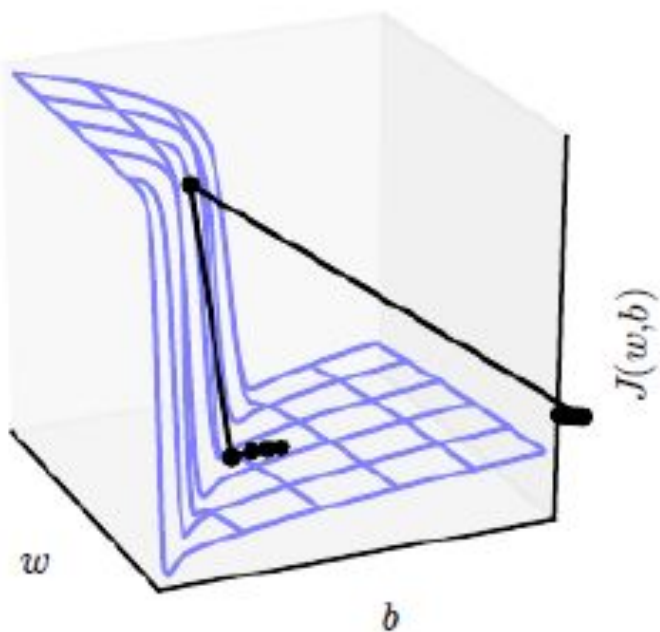
---

```
 $\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$   
if  $\|\hat{\mathbf{g}}\| \geq threshold$  then  
     $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$   
end if
```

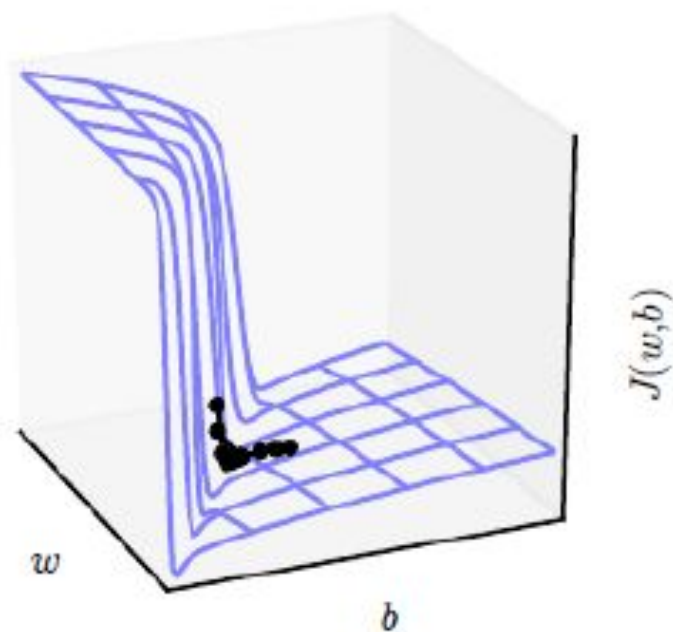
---

# Gradient explosion

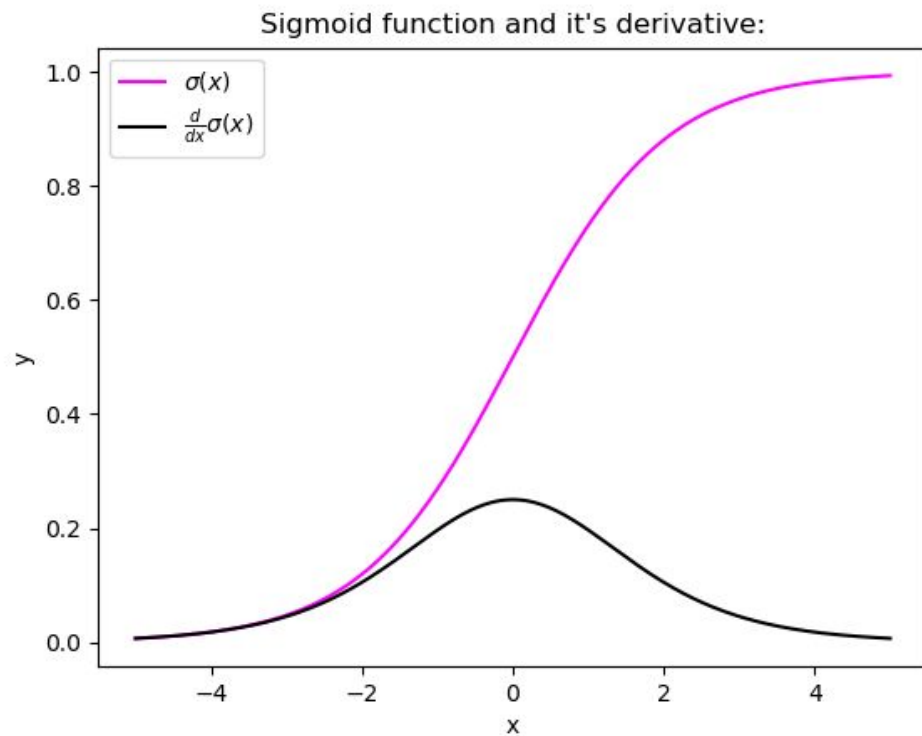
Without clipping



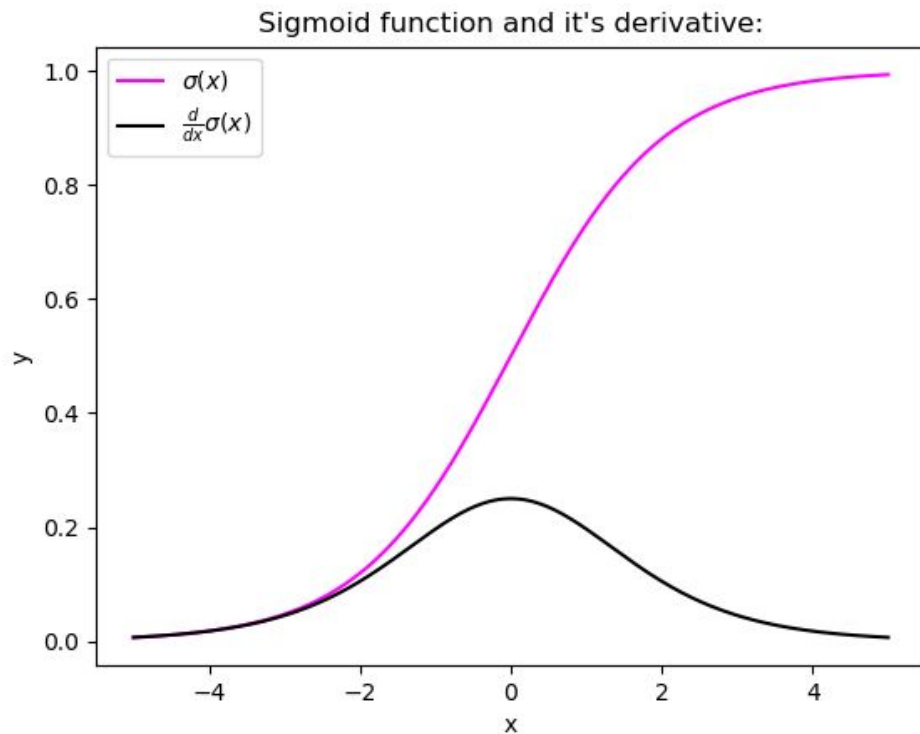
With clipping



# Gradient vanish



# Gradient vanish



Решения:

1. Не Sigmoid/Tanh, а ReLU
- ...
2. Skip connections

