

# **Оптимизация. Обучение нейронных сетей**

# **Recap**

# Neural networks: the original linear classifier

**(Before)** Linear score function:  $f = Wx$

$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

# Neural networks: 2 layers

**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

# Neural networks: also called fully connected network

Полносвязная сеть - тип нейронной сети, в которой каждый нейрон одного слоя связан со всеми нейронами следующего слоя

**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

“Neural Network” is a very broad term; these are more accurately called “fully-connected networks” or sometimes “multi-layer perceptrons” (MLP)

# Neural networks: 3 layers

**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$   
or 3-layer Neural Network

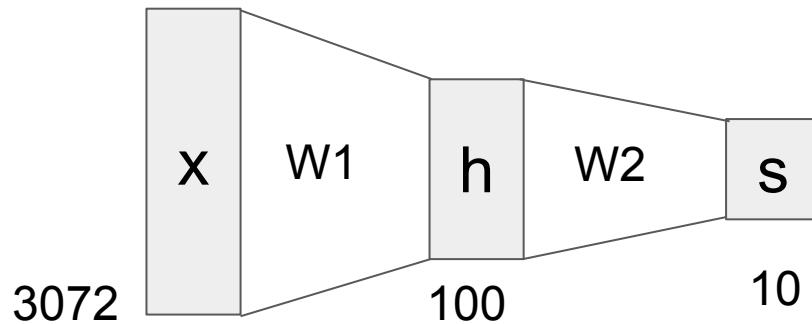
$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H_1 \times D}, W_2 \in \mathbb{R}^{H_2 \times H_1}, W_3 \in \mathbb{R}^{C \times H_2}$$

# Neural networks: hierarchical computation

(Before) Linear score function:  $f = Wx$

(Now) 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

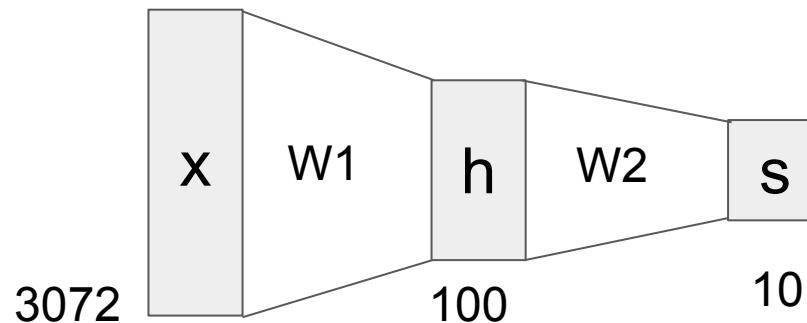


$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

# Neural networks: learning 100s of templates

(Before) Linear score function:  $f = Wx$

(Now) 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$



# Neural networks: why is max operator important?

(Before) Linear score function:  $f = Wx$

(Now) 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

The function  $\max(0, z)$  is called the **activation function**.

**Q:** What if we try to build a neural network without one?

$$f = W_2 W_1 x$$

# Neural networks: why is max operator important?

(Before) Linear score function:  $f = Wx$

(Now) 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

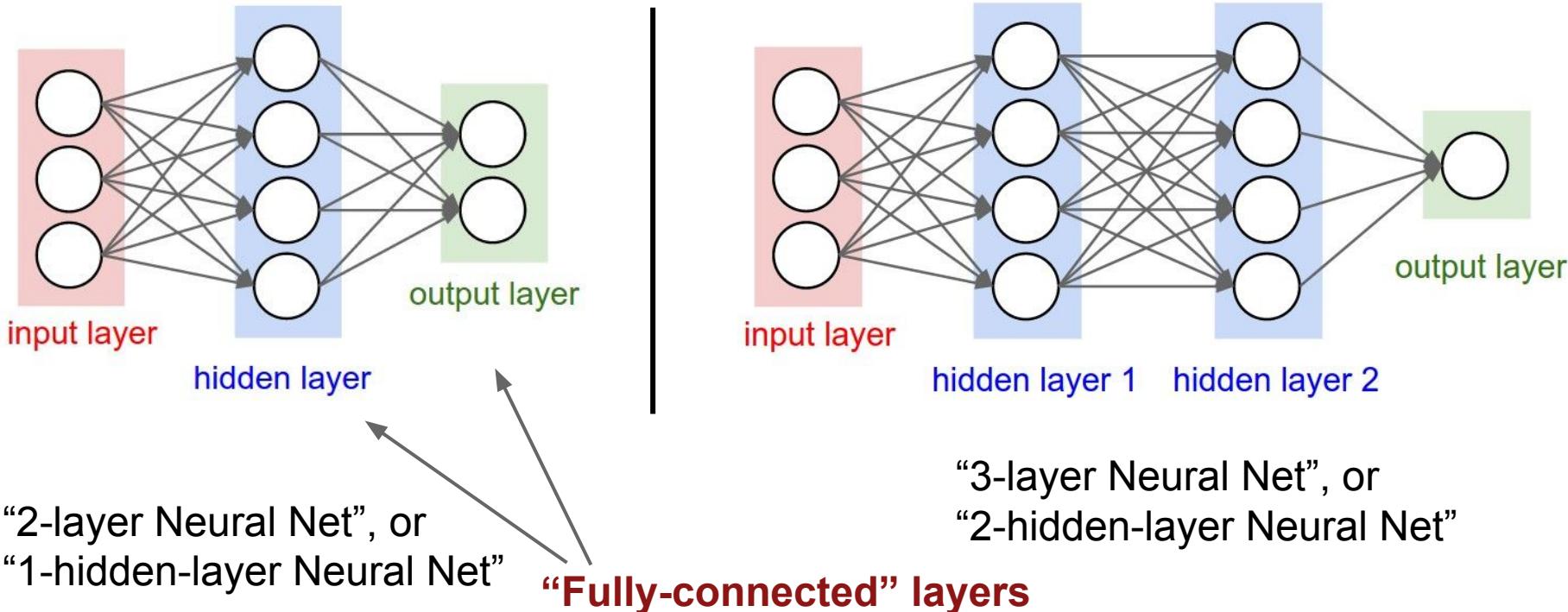
The function  $\max(0, z)$  is called the **activation function**.

Q: What if we try to build a neural network without one?

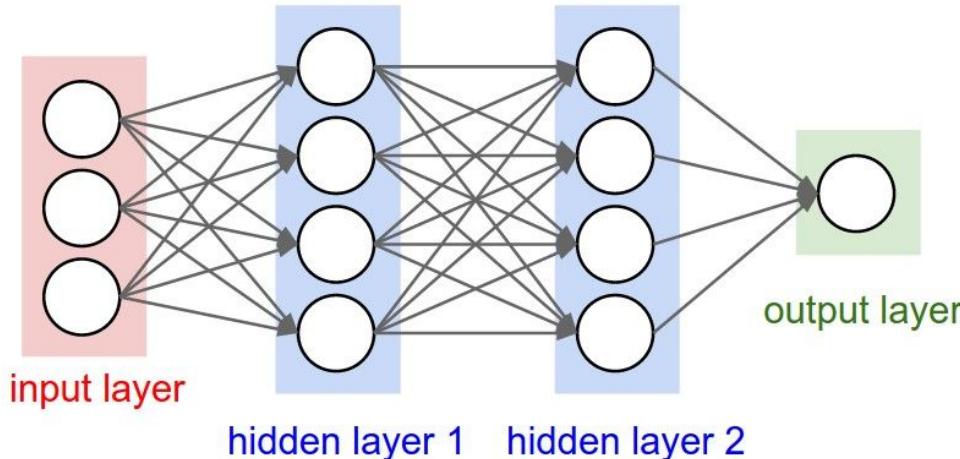
$$f = W_2 W_1 x \quad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

A: We end up with a linear classifier again!

# Neural networks: Architectures



# Example feed-forward computation of a neural network



```
# forward-pass of a 3-layer neural network:  
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)  
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)  
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)  
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)  
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

Gradient descent

# Setting the number of layers and their sizes

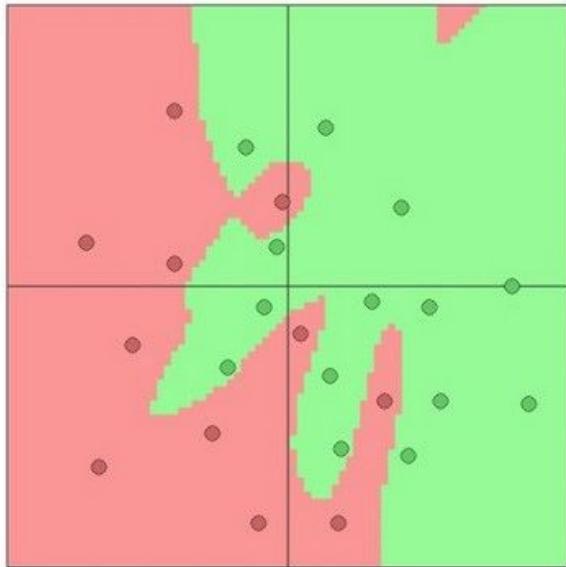


more neurons = more capacity

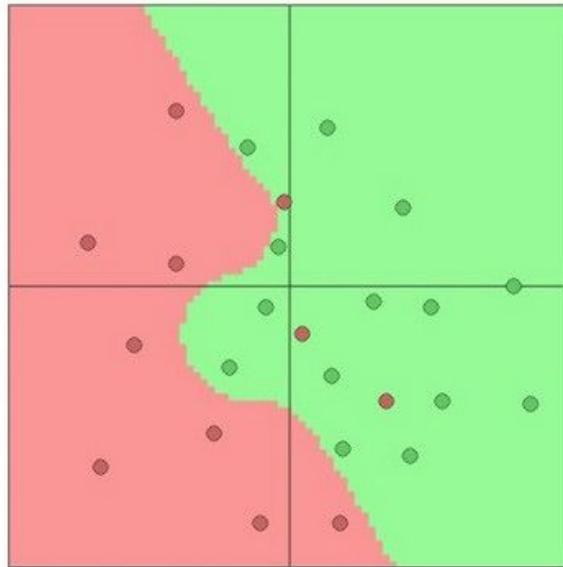
больше различных паттернов и зависимостей модель может захватить из данных.

Do not use size of neural network as a regularizer. Use stronger regularization instead:

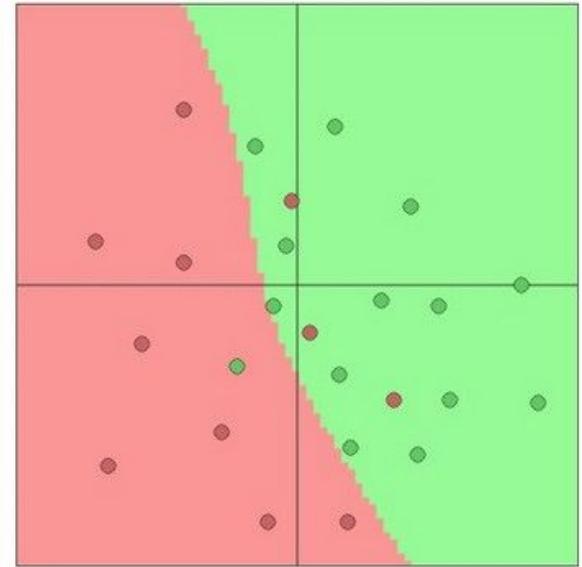
$\lambda = 0.001$



$\lambda = 0.01$



$\lambda = 0.1$



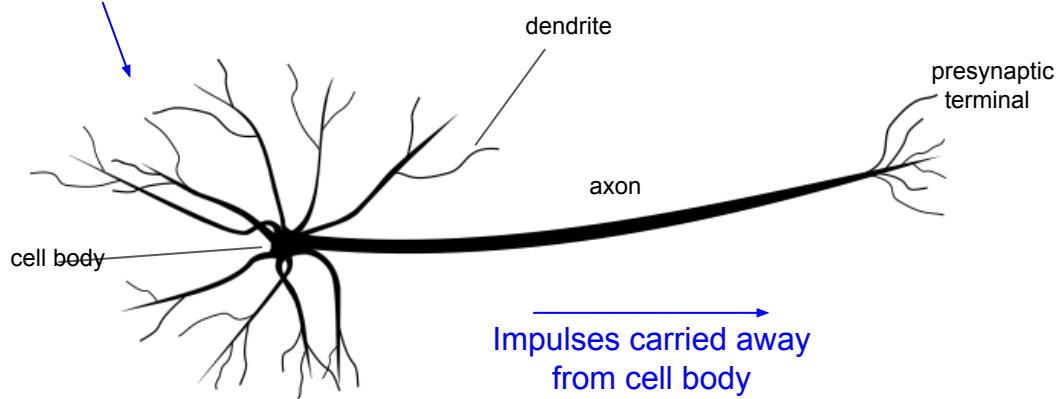
(Web demo with ConvNetJS:  
[http://cs.stanford.edu/people/karpathy/convnetjs/demo  
/classify2d.html](http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html))

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i) + \lambda R(W)$$



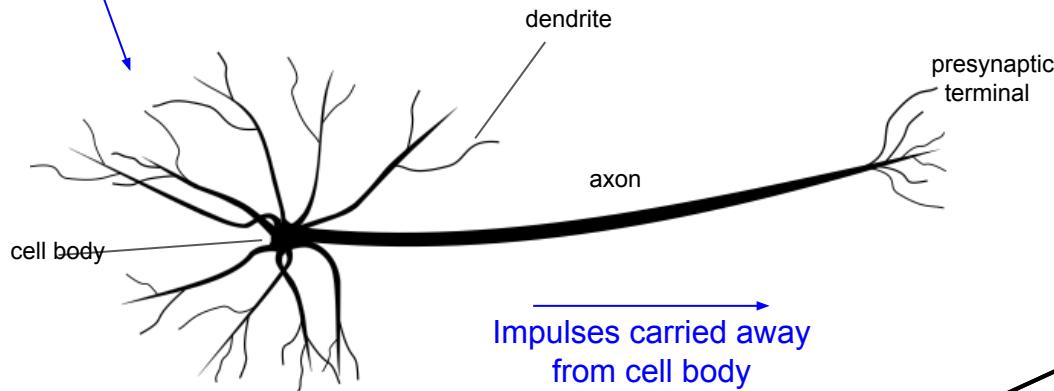
This image by [Fotis Bobolas](#) is  
licensed under [CC-BY 2.0](#)

Impulses carried toward cell body



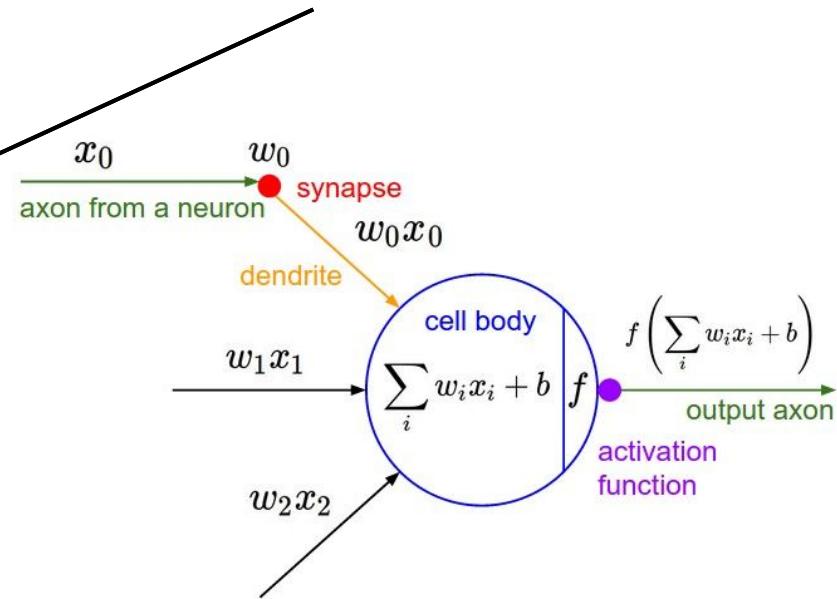
[This image](#) by Felipe Perucho  
is licensed under [CC-BY 3.0](#)

Impulses carried toward cell body

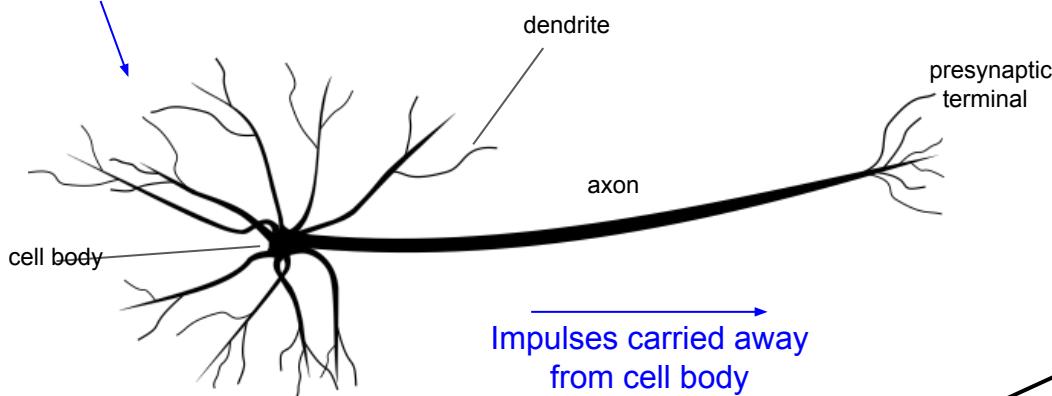


This image by Felipe Perucho  
is licensed under CC-BY 3.0

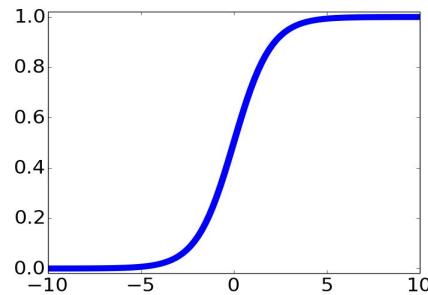
Impulses carried away  
from cell body



Impulses carried toward cell body



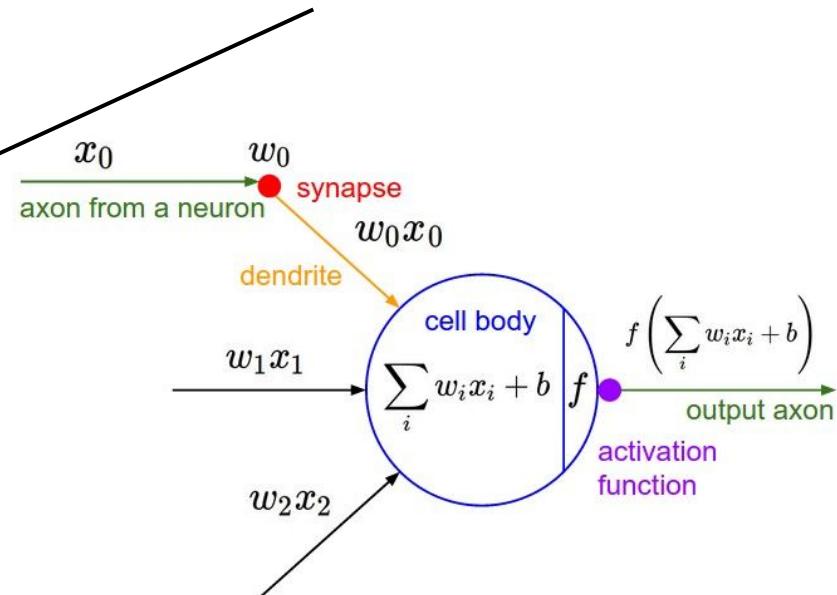
This image by Felipe Perucho  
is licensed under CC-BY 3.0



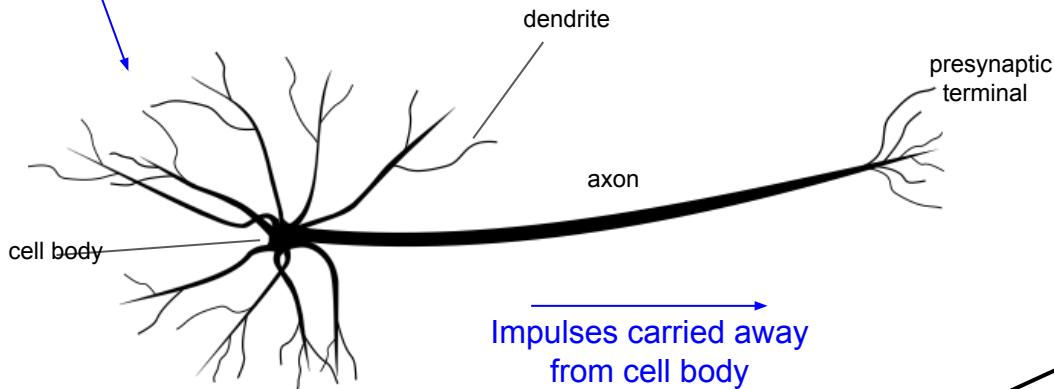
sigmoid activation function

$$\frac{1}{1 + e^{-x}}$$

Impulses carried away from cell body

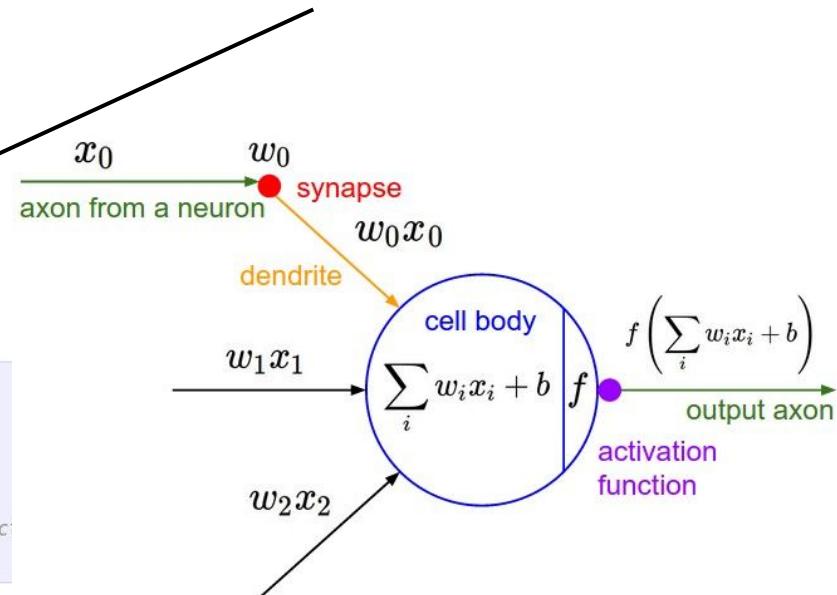


Impulses carried toward cell body

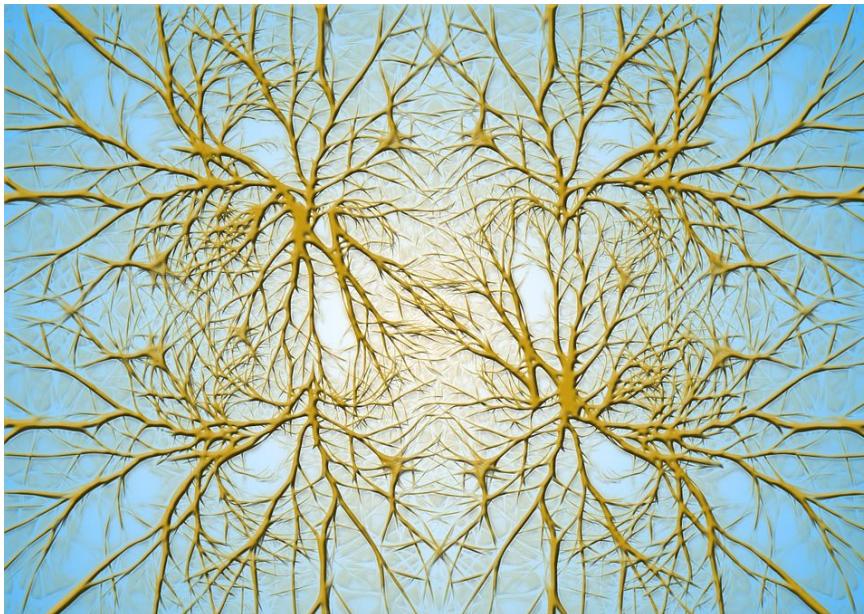


This image by Felipe Perucho  
is licensed under CC-BY 3.0

```
class Neuron:  
    # ...  
    def neuron_tick(inputs):  
        """ assume inputs and weights are 1-D numpy arrays and bias is a number """  
        cell_body_sum = np.sum(inputs * self.weights) + self.bias  
        firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function  
        return firing_rate
```

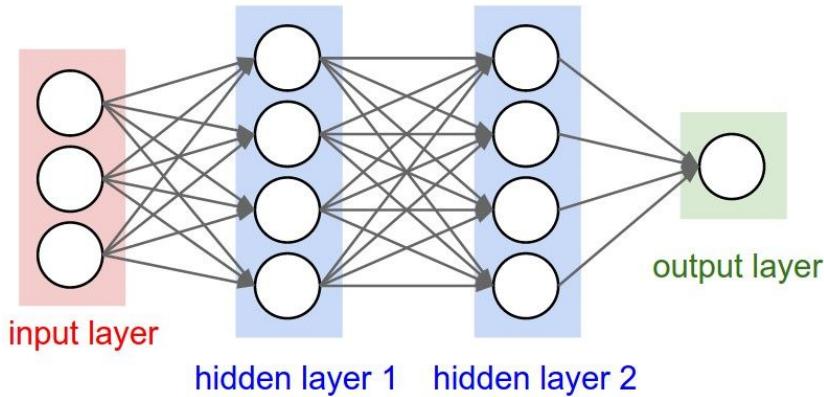


## Biological Neurons: Complex connectivity patterns

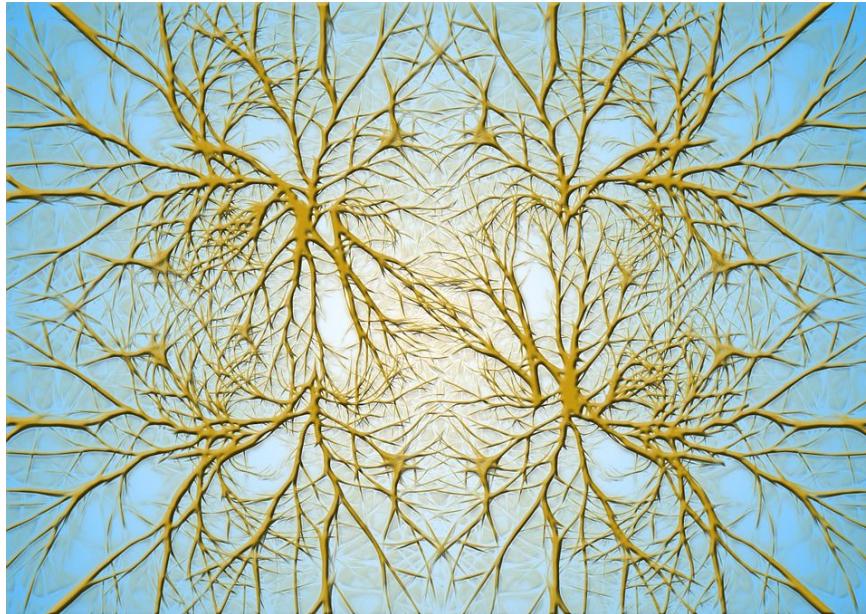


[This image is CC0 Public Domain](#)

Neurons in a neural network:  
Organized into regular layers for  
computational efficiency

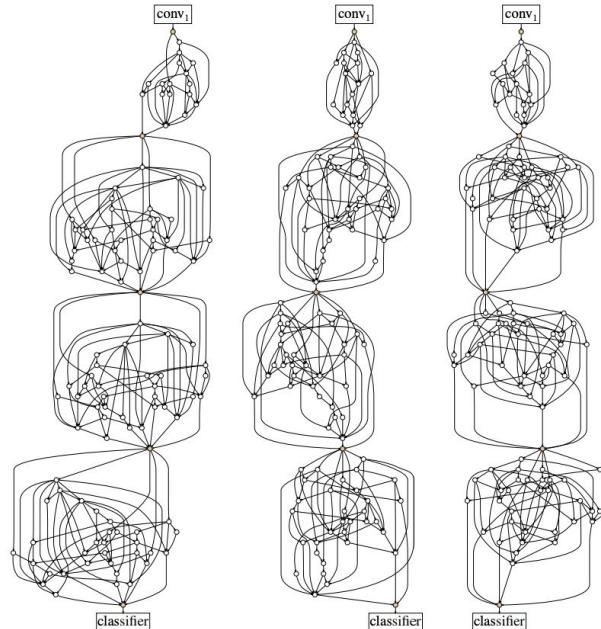


# Biological Neurons: Complex connectivity patterns



[This image is CC0 Public Domain](#)

# But neural networks with random connections can work too!



Xie et al, "Exploring Randomly Wired Neural Networks for Image Recognition", arXiv 2019

# Be very careful with your brain analogies!

## Biological Neurons:

- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system

[Dendritic Computation. London and Häusser]

# Plugging in neural networks with loss functions

$$s = f(x; W_1, W_2) = W_2 \max(0, W_1 x) \quad \text{Nonlinear score function}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM Loss on predictions}$$

$$R(W) = \sum_k W_k^2 \quad \text{Regularization}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W_1) + \lambda R(W_2) \quad \text{Total loss: data loss + regularization}$$

# Problem: How to compute gradients?

$$s = f(x; W_1, W_2) = W_2 \max(0, W_1 x) \quad \text{Nonlinear score function}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM Loss on predictions}$$

$$R(W) = \sum_k W_k^2 \quad \text{Regularization}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W_1) + \lambda R(W_2) \quad \text{Total loss: data loss + regularization}$$

If we can compute  $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}$  then we can learn  $W_1$  and  $W_2$

# (Bad) Idea: Derive $\nabla_W L$ on paper

$$s = f(x; W) = Wx$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$= \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda \sum_k W_k^2$$

$$= \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2$$

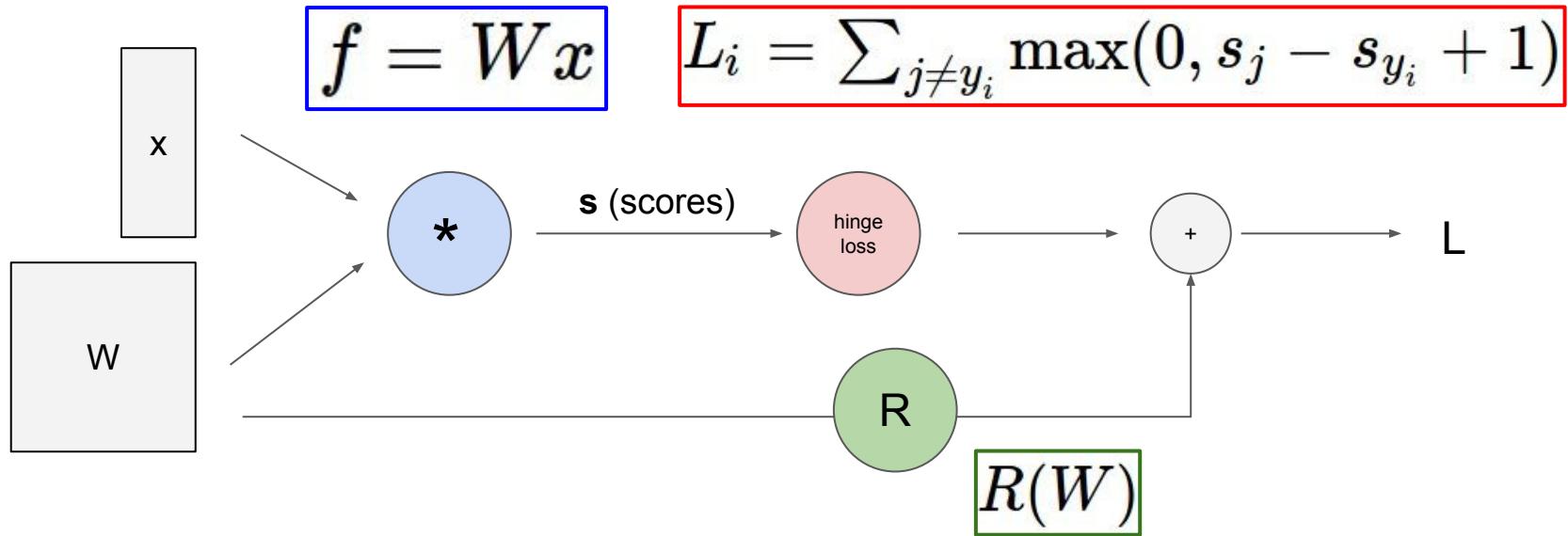
$$\nabla_W L = \nabla_W \left( \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$$

**Problem:** Very tedious: Lots of matrix calculus, need lots of paper

**Problem:** What if we want to change loss? E.g. use softmax instead of SVM? Need to re-derive from scratch =(

**Problem:** Not feasible for very complex models!

# Better Idea: Computational graphs + Backpropagation



# **Оптимизация**

# Stochastic Gradient Descent

$$\frac{1}{n} \sum_{i=1}^n L(y_i, f(x_i, \theta)) \rightarrow \min_{\theta}$$

Градиентный спуск:  $\theta_{t+1} = \theta_t - \alpha \frac{dL}{d\theta}$

# Stochastic Gradient Descent

$$\frac{1}{n} \sum_i L(y_i, f(x_i, \theta)) \rightarrow \min_{\theta}$$

Градиентный спуск:

$$\theta_{t+1} = \theta_t - \alpha \frac{dL}{d\theta}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{n} \sum_{i=1}^n \frac{dL(y_i, f(x_i, \theta))}{d\theta}$$

одно обновление -  
один проход по данным  
**долго, но точно**

# Stochastic Gradient Descent

$$\frac{1}{n} \sum_i L(y_i, f(x_i, \theta)) \rightarrow \min_{\theta}$$

Градиентный спуск:

$$\theta_{t+1} = \theta_t - \alpha \frac{dL}{d\theta}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{n} \sum_{i=1}^n \frac{dL(y_i, f(x_i, \theta))}{d\theta}$$

Стохастический  
градиентный спуск:

$$\theta_{t+1} = \theta_t - \alpha \frac{dL(y_i, f(x_i, \theta))}{d\theta}$$

В отличие от стандартного градиентного спуска, который вычисляет градиент функции потерь на всем обучающем наборе данных, SGD обновляет параметры модели на основе одного (или небольшого количества) случайно выбранного примера за итерацию.

одно обновление -  
один пример  
**быстро, но не так точно**

# Stochastic Gradient Descent

$$\frac{1}{n} \sum_i L(y_i, f(x_i, \theta)) \rightarrow \min_{\theta}$$

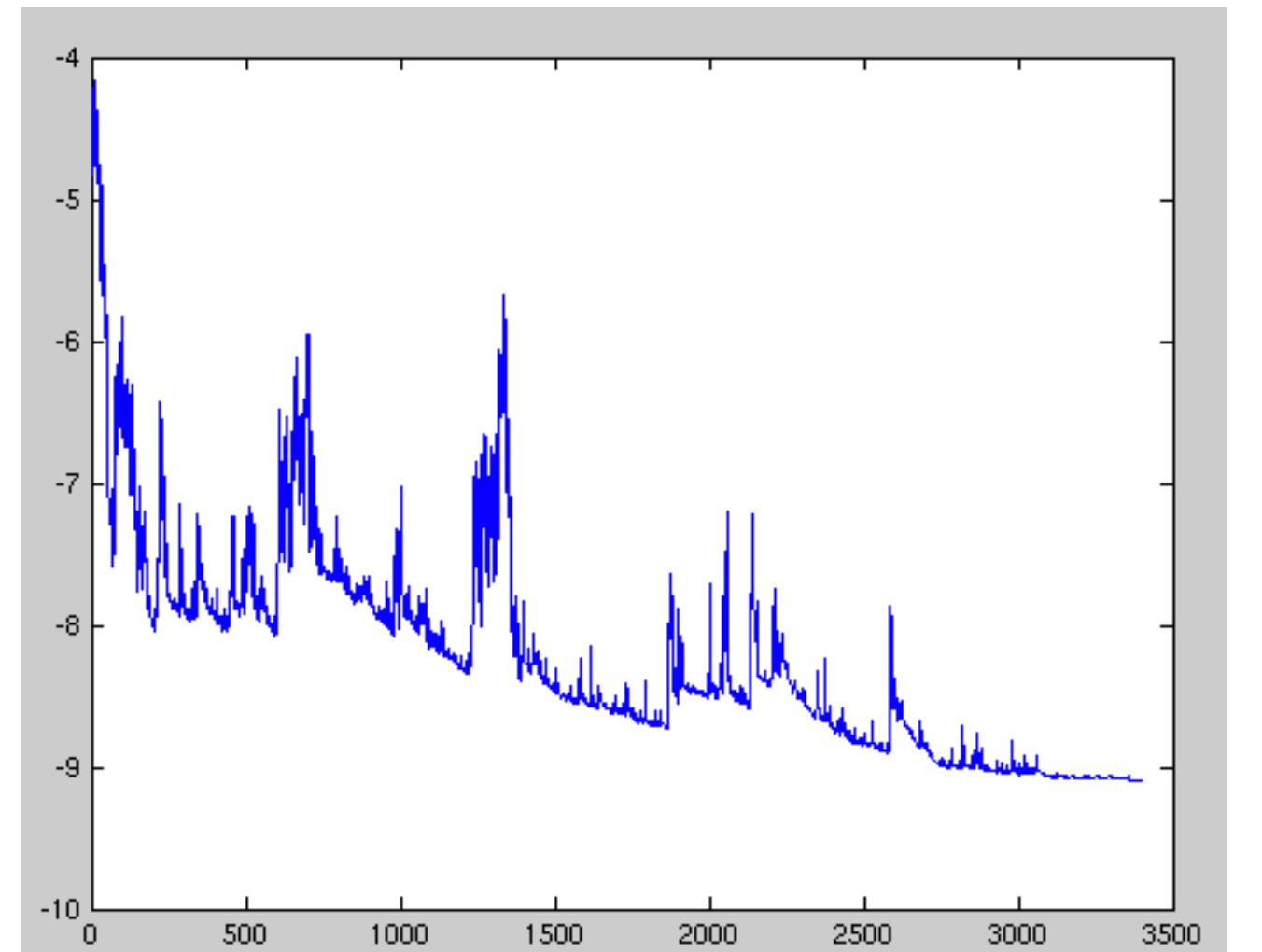
Градиентный спуск:

$$\theta_{t+1} = \theta_t - \alpha \frac{dL}{d\theta}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{n} \sum_{i=1}^n \frac{dL(y_i, f(x_i, \theta))}{d\theta}$$

Стохастический  
градиентный спуск:

$$\theta_{t+1} = \theta_t - \alpha \frac{dL(y_i, f(x_i, \theta))}{d\theta}$$



[Image credit](#)

# Stochastic Gradient Descent

$$\frac{1}{n} \sum_i L(y_i, f(x_i, \theta)) \rightarrow \min_{\theta}$$

Mini-batch SGD:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{m} \sum_{i=1}^m \frac{dL(y_i, f(x_i, \theta))}{d\theta}$$

одно обновление -  
 $m$  примеров (батч)

# Stochastic Gradient Descent

$$\frac{1}{n} \sum_i L(y_i, f(x_i, \theta)) \rightarrow \min_{\theta}$$

Mini-batch SGD:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{m} \sum_{i=1}^m \frac{dL(y_i, f(x_i, \theta))}{d\theta}$$

одно обновление –  
 $m$  примеров (батч)

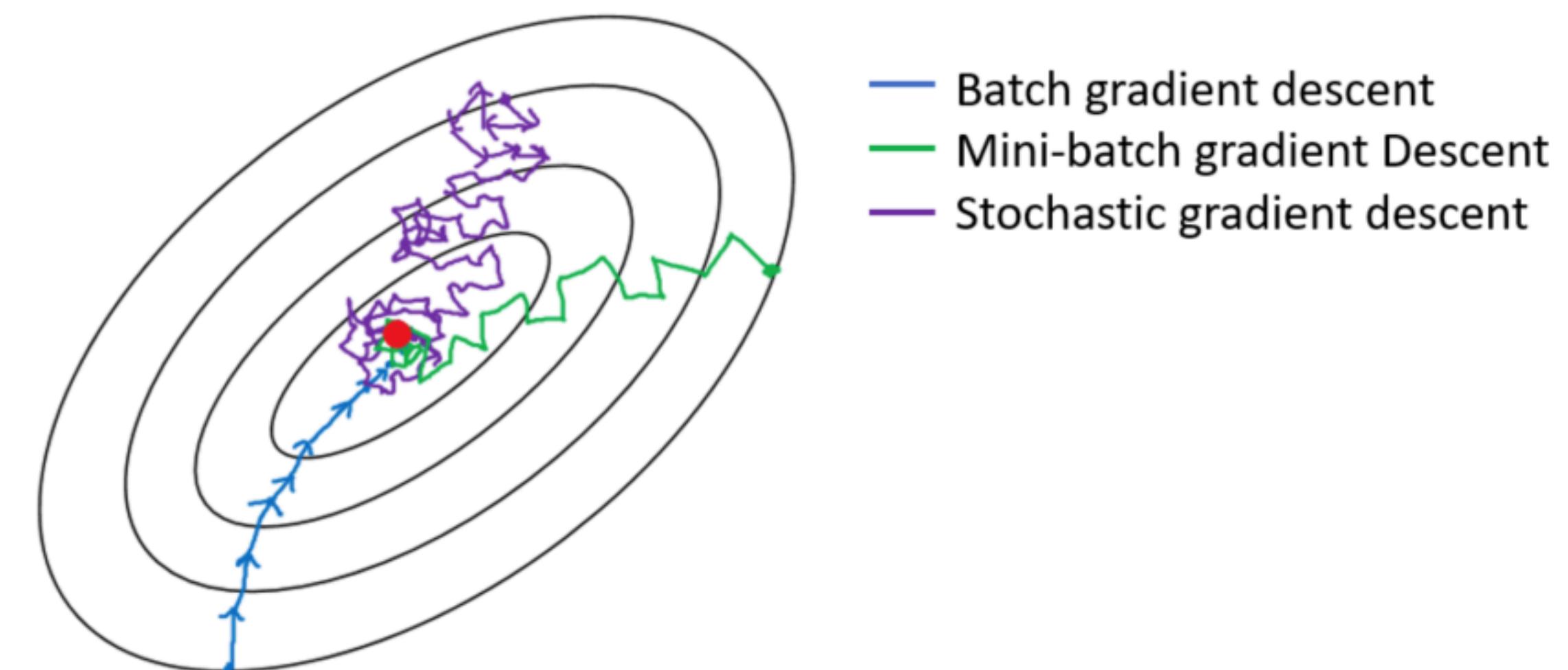


Image credit

# Stochastic Gradient Descent

$$\frac{1}{n} \sum_i L(y_i, f(x_i, \theta)) \rightarrow \min_{\theta}$$

Mini-batch SGD:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{m} \sum_{i=1}^m \frac{dL(y_i, f(x_i, \theta))}{d\theta}$$

одно обновление –  
 $m$  примеров (батч)

Теория: найдем глобальный минимум для выпуклых  $L$ , иначе локальный

Вместо того чтобы обновлять параметры модели на основе одного примера (как в SGD) или на основе всего набора данных (как в стандартном градиентном спуске), mini-batch SGD обновляет параметры на основе небольших подмножеств данных

# Stochastic Gradient Descent

$$\frac{1}{n} \sum_i L(y_i, f(x_i, \theta)) \rightarrow \min_{\theta}$$

Mini-batch SGD:

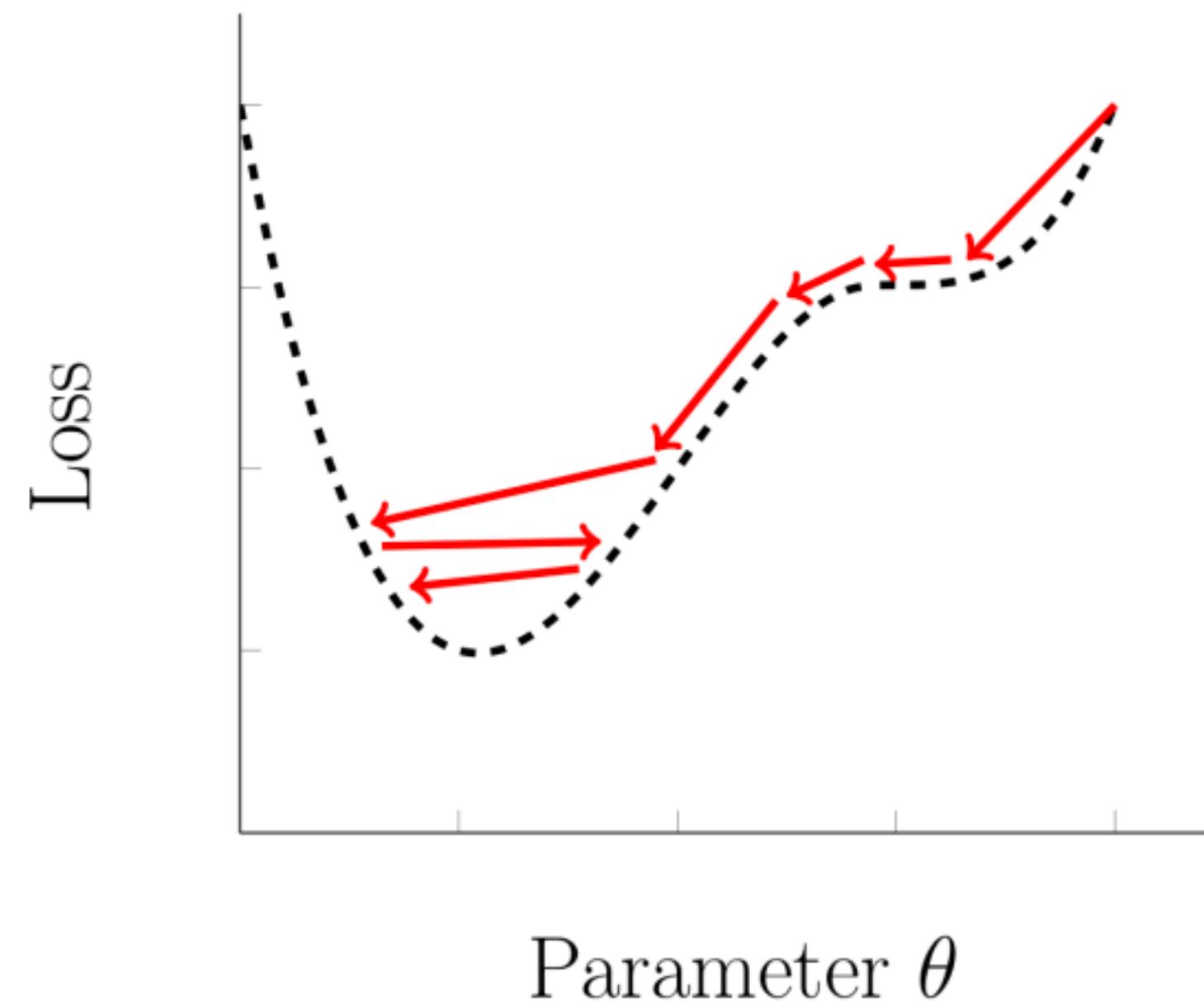
$$\theta_{t+1} = \theta_t - \frac{\alpha}{m} \sum_{i=1}^m \frac{dL(y_i, f(x_i, \theta))}{d\theta}$$

learning rate

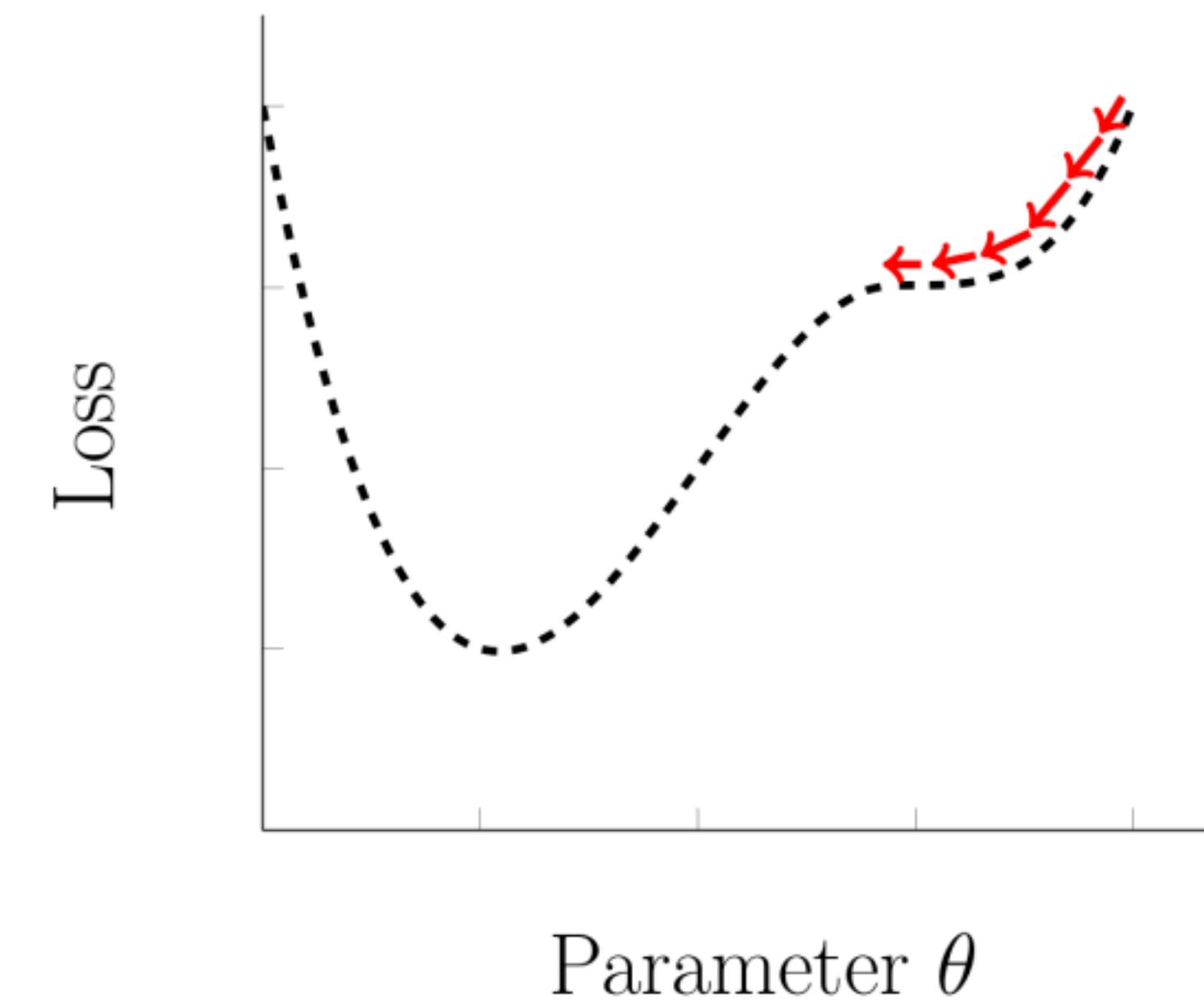
одно обновление -  
 $m$  примеров (батч)

# Stochastic Gradient Descent

High Learning Rate



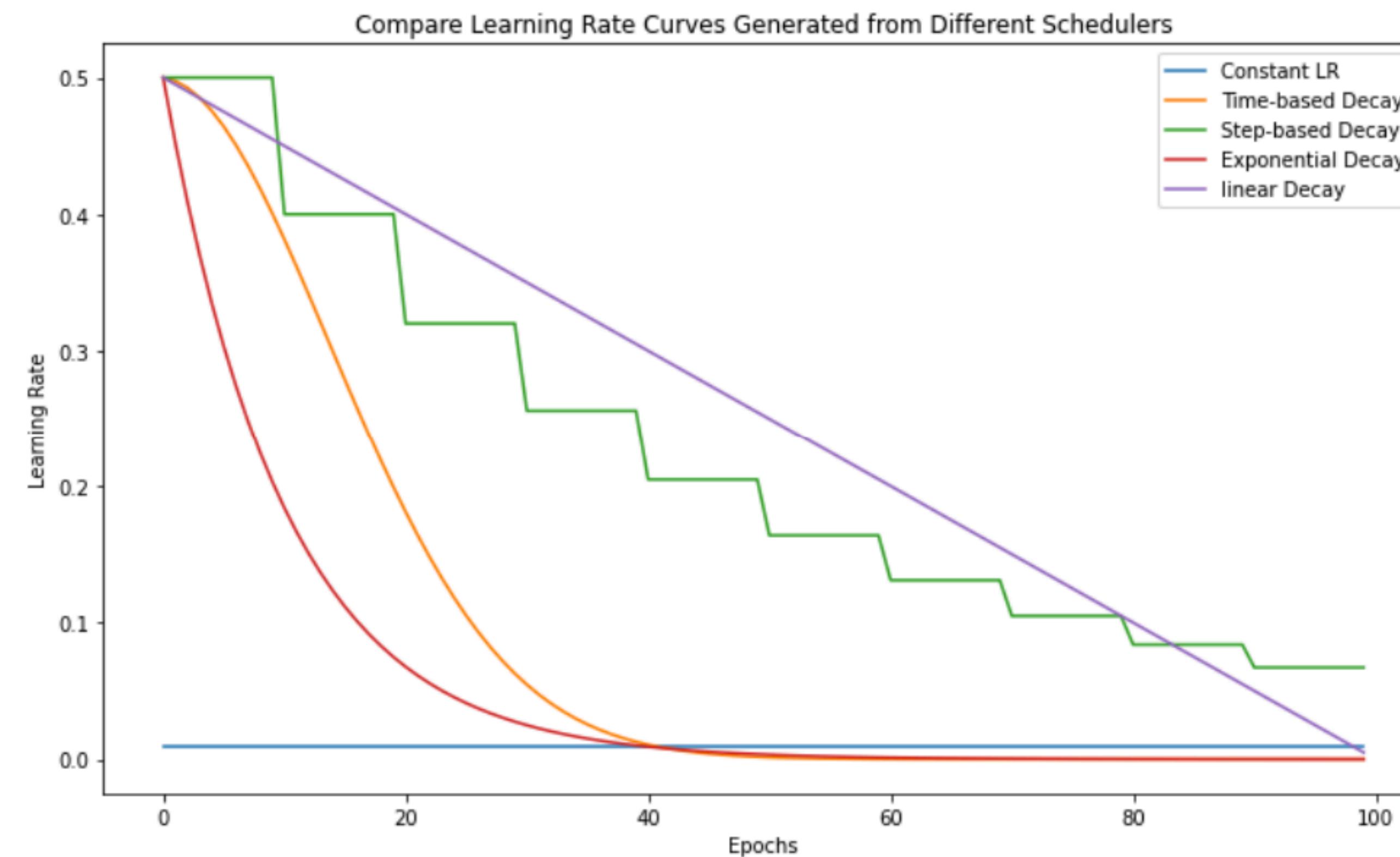
Low Learning Rate



[Image credit](#)

# Stochastic Gradient Descent

Можно выбирать разные lr на разных эпохах - расписание lr (scheduler)



[Image credit](#)

# Stochastic Gradient Descent

## Проблемы:

- Градиент может быть шумным
- LR одинаковый для всех параметров и данных
- Можно застрять в локальном минимуме или седловой точке

Особенно на небольших партиях данных (мини-батчах) градиенты могут иметь высокий уровень шума (который мешает точному движению к минимуму функции потерь, т. е. нестабильное движение к min). Это может замедлять обучение или приводить к непредсказуемым результатам. Существуют методы сглаживания, такие как использование большего размера батча или методов вроде сглаживания моментов (momentum), RMSProp или Adam, которые корректируют шаг обучения на основе скорости изменения градиентов. Шумный градиент может иногда помочь избежать застревания в плохих локальных минимумах, так как случайные флуктуации могут вытолкнуть модель из локального минимума. С другой стороны, чрезмерный шум может замедлить сходимость или даже привести к нестабильности в процессе обучения, когда модель постоянно "скачет" вокруг хороших решений, не приближаясь к оптимуму.

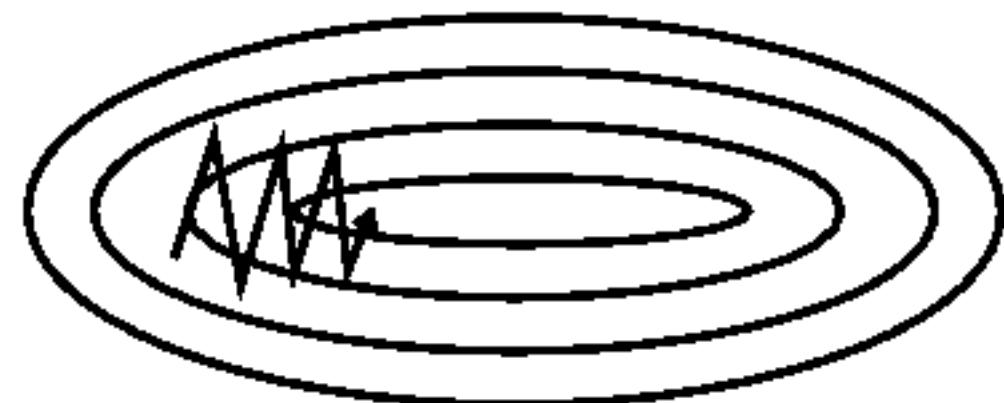
Стандартный SGD использует один и тот же learning rate для всех весов модели, что может быть неэффективным, так как разные параметры могут требовать разных скоростей обновления. Это решается адаптивными методами, такими как Adam, AdaGrad или RMSProp, которые настраивают LR индивидуально для каждого параметра на основе его предыдущих изменений.

В глубоких сетях часто можно застрять не в настоящих локальных минимумах, а в седловых точках, где градиент равен нулю в одном направлении, но не в других. Это серьезная проблема, так как модель может застрять в этих точках на длительное время. Использование методов, таких как ускоряющие оптимизаторы (например, Adam или Nesterov momentum), может помочь модели преодолевать седловые точки.

# Stochastic Gradient Descent

Проблемы:

- Градиент может быть шумным
- LR одинаковый для всех параметров и данных
- Можно застрять в локальном минимуме или седловой точке



SGD

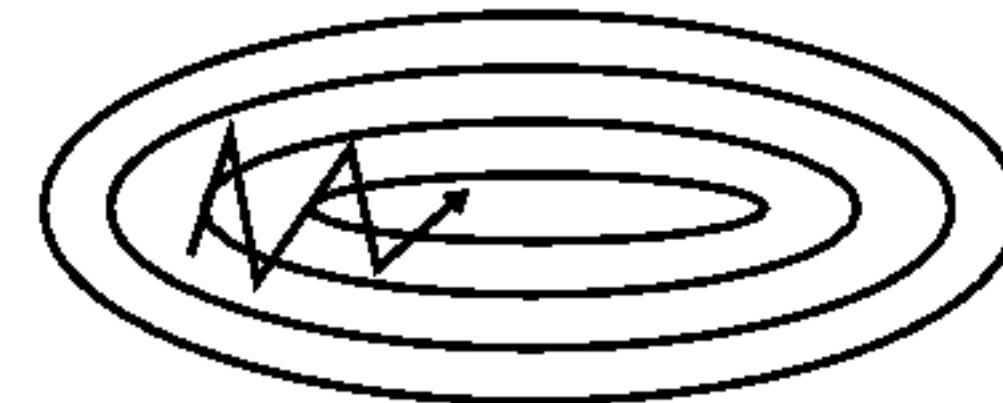
# Stochastic Gradient Descent

Проблемы:

- Градиент может быть шумным
- LR одинаковый для всех параметров и данных
- Можно застрять в локальном минимуме или седловой точке



SGD



SGD + momentum

# Stochastic Gradient Descent + Momentum

SGD

$$\theta_{t+1} = \theta_t - \alpha \frac{dL(\theta)}{d\theta}$$

SGD + momentum

$$v_t = \gamma v_{t-1} + \alpha \frac{dL(\theta)}{d\theta}$$

$$\theta_{t+1} = \theta_t - v_t$$

# Stochastic Gradient Descent + Momentum

SGD

$$\theta_{t+1} = \theta_t - \alpha \frac{dL(\theta)}{d\theta}$$

SGD + momentum

$$v_t = \gamma v_{t-1} + \alpha \frac{dL(\theta)}{d\theta}$$

$$\theta_{t+1} = \theta_t - v_t$$

$v$  - “скорость”

$\gamma$  - “трение”, обычно = 0.9 ... 0.99

# Stochastic Gradient Descent + Momentum

SGD

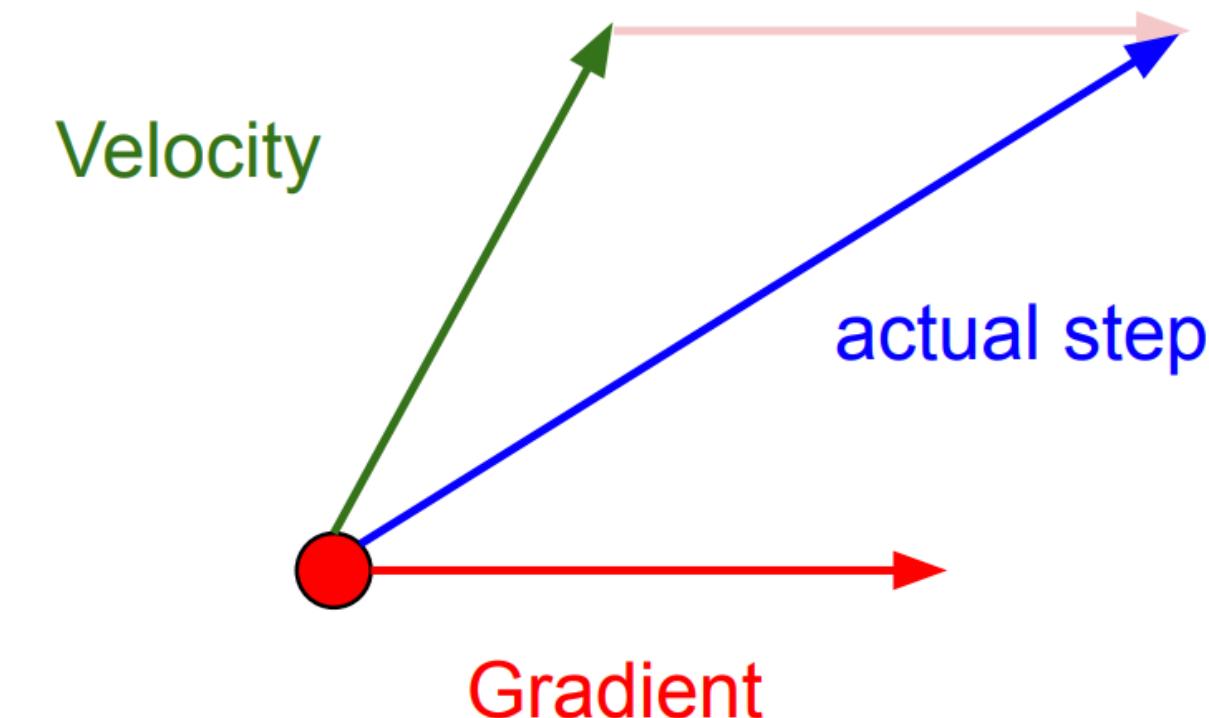
$$\theta_{t+1} = \theta_t - \alpha \frac{dL(\theta)}{d\theta}$$

SGD + momentum

$$v_t = \gamma v_{t-1} + \alpha \frac{dL(\theta)}{d\theta}$$

$$\theta_{t+1} = \theta_t - v_t$$

Метод момента помогает сгладить обновления параметров, учитывая не только текущий градиент, но и предыдущие градиенты. Это позволяет "разгонять" обновления в направлении, в котором модель уже движется, что может помочь избежать колебаний и ускорить сходимость.



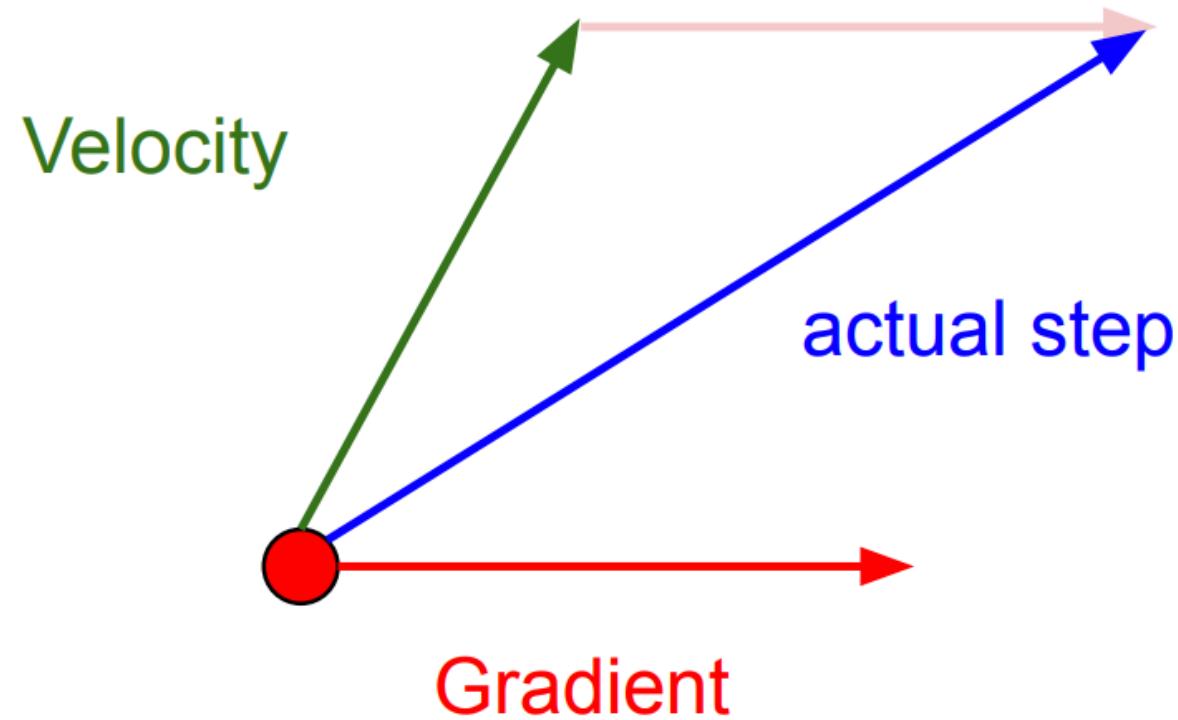
[Image credit](#)

# Nesterov Momentum

SGD + momentum

$$v_t = \gamma v_{t-1} + \alpha \frac{dL(\theta)}{d\theta}$$

$$\theta_{t+1} = \theta_t - v_t$$

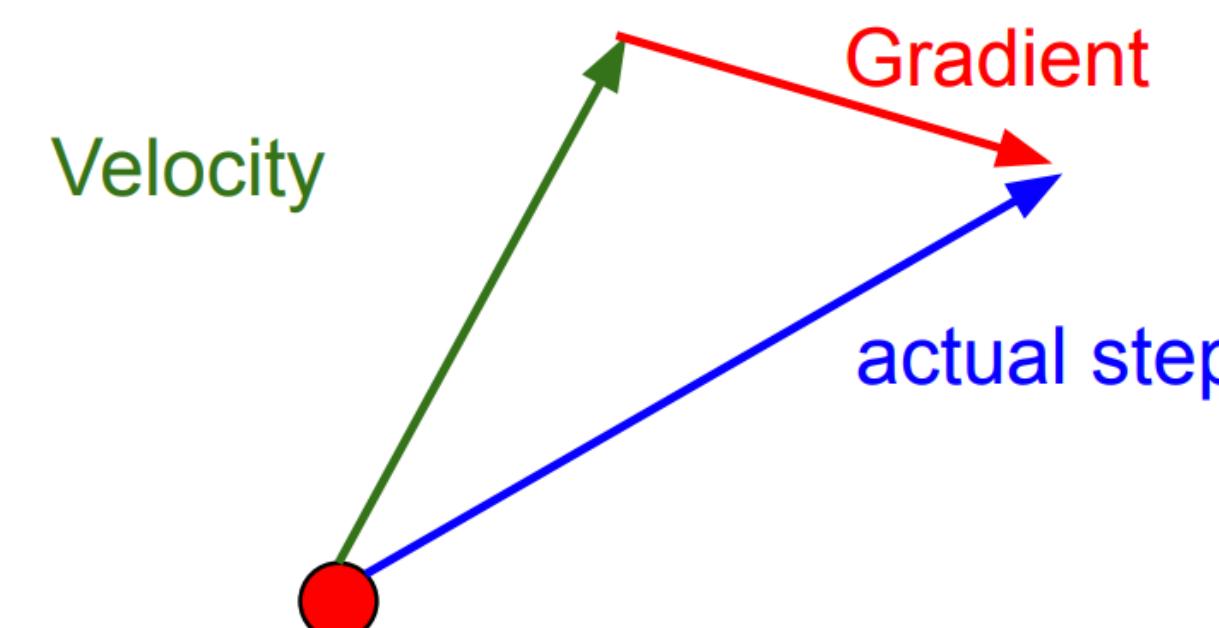


В отличие от обычного момента, который использует только предыдущие градиенты, Nesterov Momentum делает "предварительный шаг" в направлении предыдущего обновления перед вычислением текущего градиента. Это позволяет более точно оценить, куда движется функция потерь.

SGD + Nesterov momentum

$$v_t = \gamma v_{t-1} + \alpha \frac{dL(\theta - \gamma v_{t-1})}{d\theta}$$

$$\theta_{t+1} = \theta_t - v_t$$



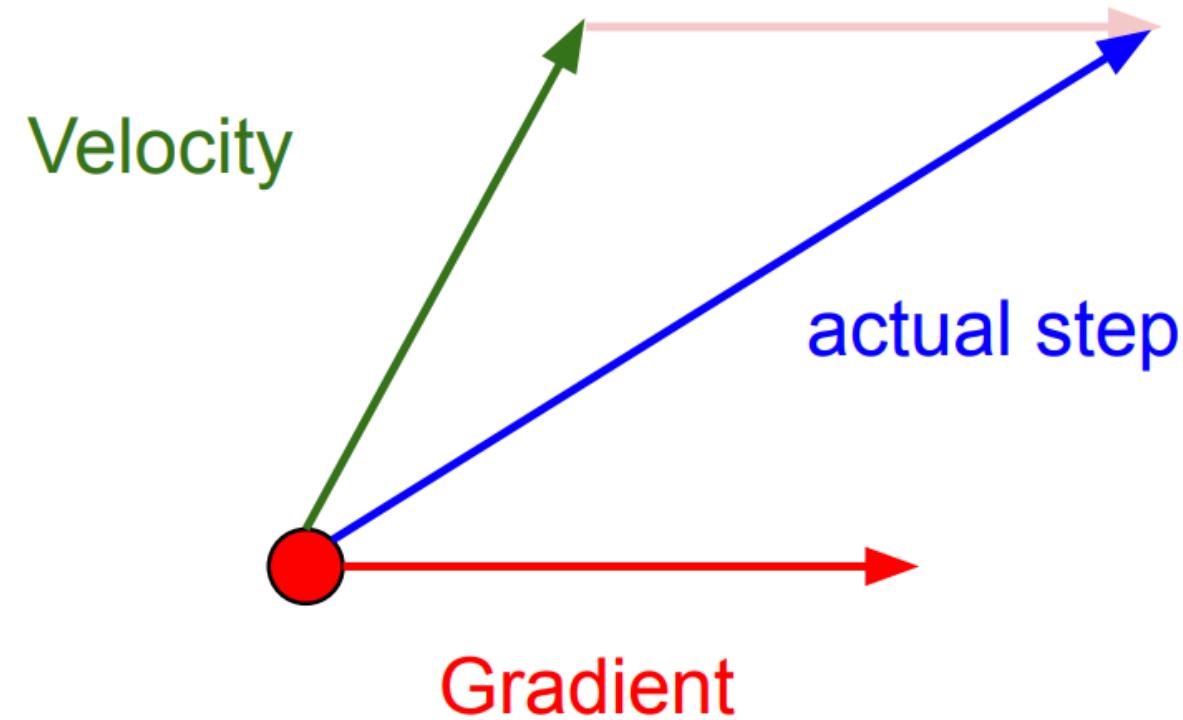
[Image credit](#)

# Nesterov Momentum

SGD + momentum

$$v_t = \gamma v_{t-1} + \alpha \frac{dL(\theta)}{d\theta}$$

$$\theta_{t+1} = \theta_t - v_t$$



SGD + Nesterov momentum

$$v_t = \gamma v_{t-1} + \alpha \frac{dL(\theta - \gamma v_{t-1})}{d\theta}$$

$$\theta_{t+1} = \theta_t - v_t$$

оцениваем, какие  
параметры будут

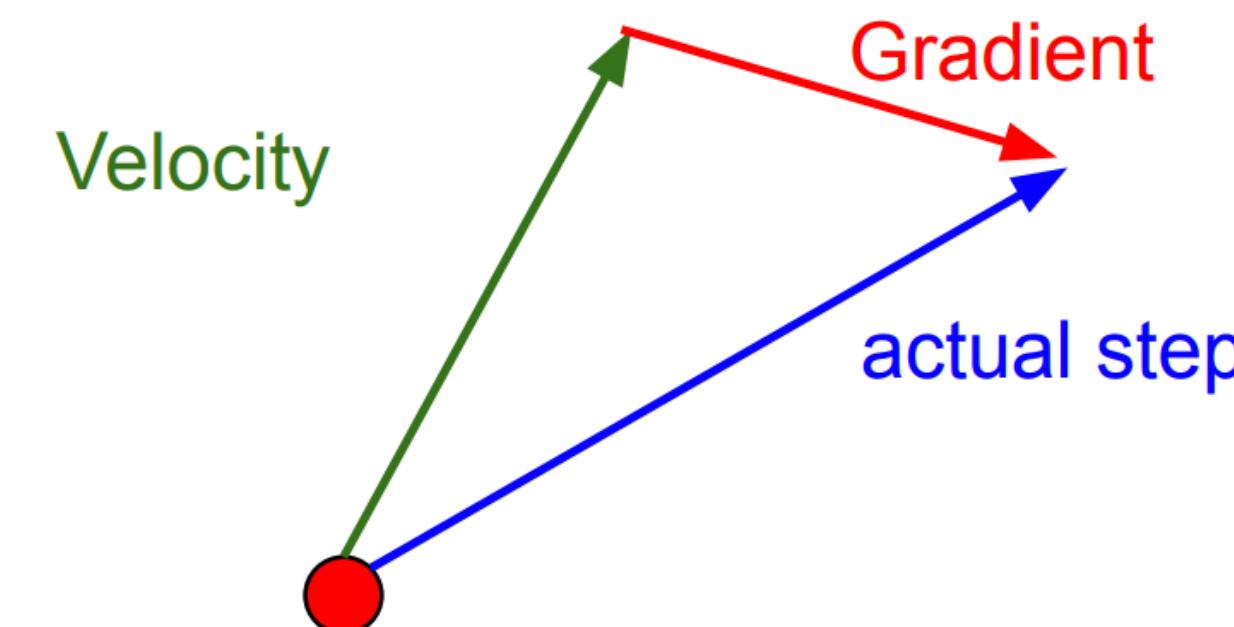
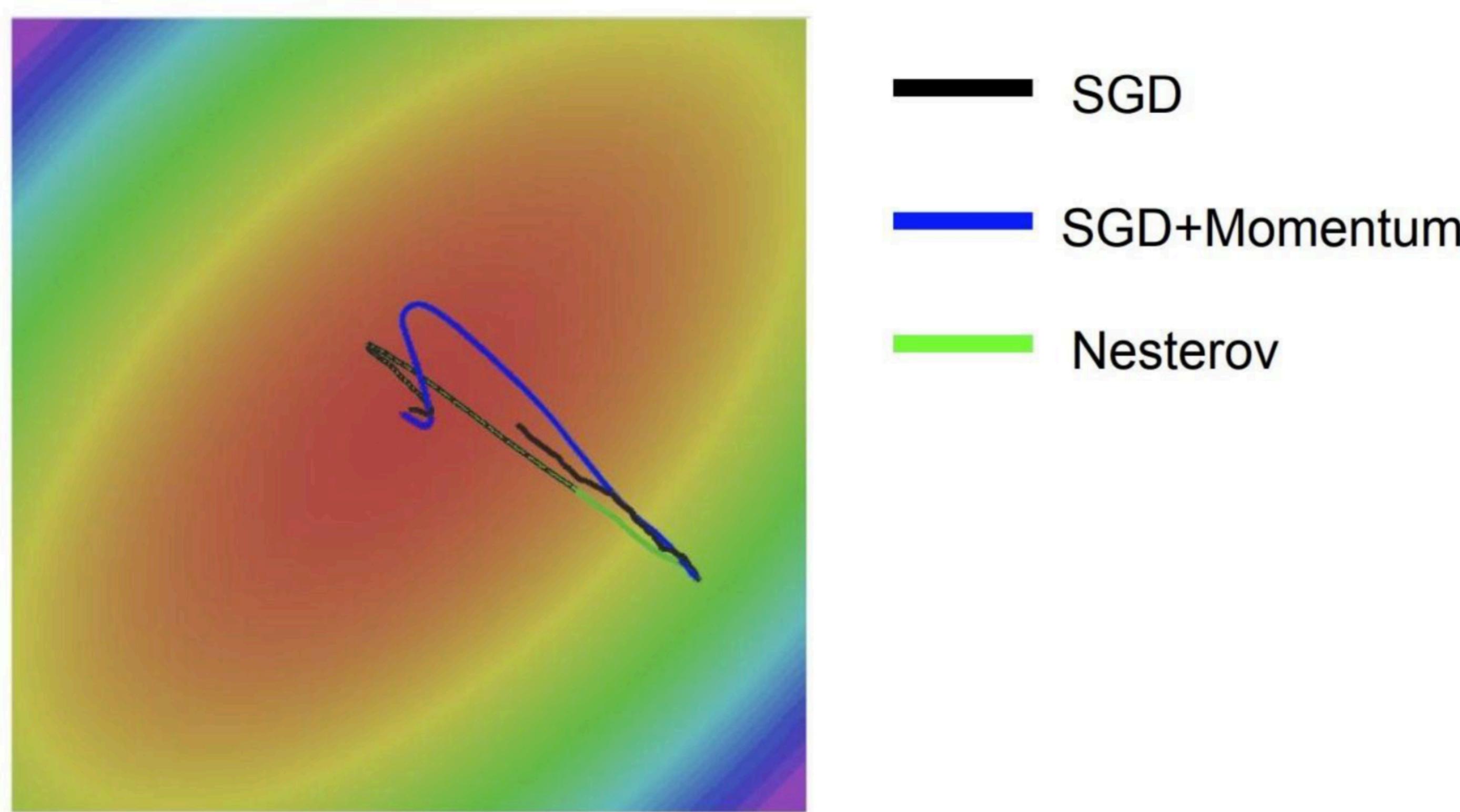


Image credit

# Nesterov Momentum



[Image credit](#)

# Stochastic Gradient Descent

Проблемы:

- Градиент может быть шумным
- LR одинаковый для всех параметров и данных
- Можно застрять в локальном минимуме или седловой точке

# AdaGrad

Идея: адаптивный learning rate

- небольшой lr для часто обновляемых параметров и большой для редких

# AdaGrad

Идея: адаптивный learning rate

- небольшой lr для часто обновляемых параметров и большой для редких

$$g_{t,i} := \frac{dL(\theta_{t,i})}{d\theta_{t,i}}$$

Градиент для  $i$ -го параметра  
на шаге  $t$

# AdaGrad

Идея: адаптивный learning rate

- небольшой lr для часто обновляемых параметров и большой для редких

$$g_{t,i} := \frac{dL(\theta_{t,i})}{d\theta_{t,i}}$$

Градиент для  $i$ -го параметра  
на шаге  $t$

$$G_{t,i} = \sum_{\tau=1}^t g_{\tau,i}^2$$

“кэш” градиентов

# AdaGrad

Идея: адаптивный learning rate

- небольшой lr для часто обновляемых параметров и большой для редких

$$g_{t,i} := \frac{dL(\theta_{t,i})}{d\theta_{t,i}}$$

Градиент для  $i$ -го параметра  
на шаге  $t$

$$G_{t,i} = \sum_{\tau=1}^t g_{\tau,i}^2$$

“кэш” градиентов

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

# AdaGrad

Идея: адаптивный learning rate

- небольшой lr для часто обновляемых параметров и большой для редких

$$g_{t,i} := \frac{dL(\theta_{t,i})}{d\theta_{t,i}}$$

Градиент для  $i$ -го параметра  
на шаге  $t$

$$G_{t,i} = \sum_{\tau=1}^t g_{\tau,i}^2$$

“кэш” градиентов

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

# AdaGrad

Идея: адаптивный learning rate

- небольшой lr для часто обновляемых параметров и большой для редких

$$g_{t,i} := \frac{dL(\theta_{t,i})}{d\theta_{t,i}}$$

Градиент для  $i$ -го параметра  
на шаге  $t$

$$G_{t,i} = \sum_{\tau=1}^t g_{\tau,i}^2$$

“кэш” градиентов

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

может стать  $\sim 0$

# RMSProp

Идея: адаптивный learning rate

- небольшой lr для часто обновляемых параметров и большой для редких

$$g_{t,i} := \frac{dL(\theta_{t,i})}{d\theta_{t,i}}$$

Градиент для  $i$ -го параметра  
на шаге  $t$

$$G_{t,i} = \beta G_{t-1,i} + (1 - \beta) g_{t,i}^2$$

“кэш” градиентов, **exponential smoothing**

$$\beta = 0.9$$

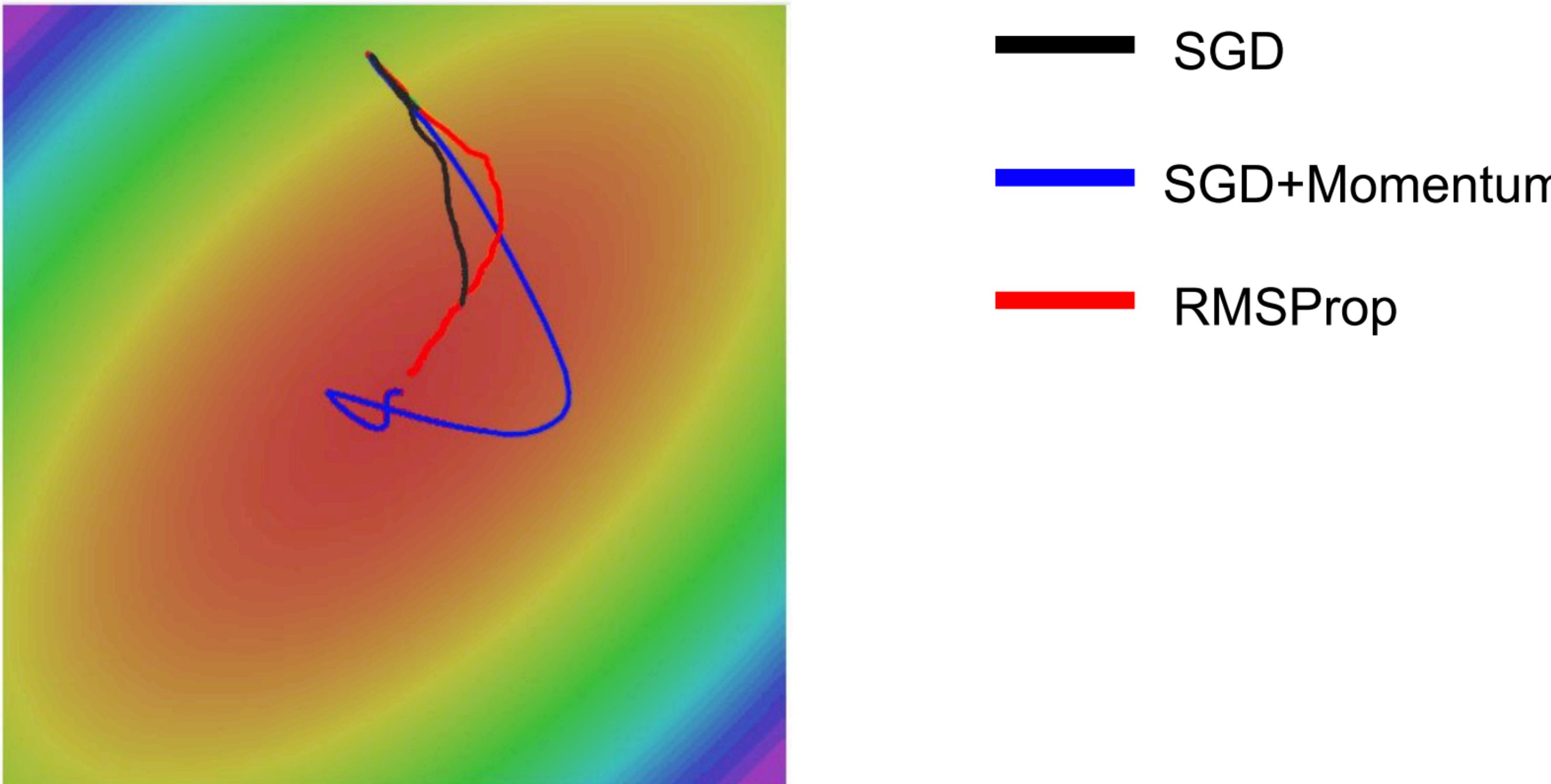
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

Root Mean Square Propagation

RMSProp адаптирует скорость обучения для каждого параметра модели, основываясь на среднеквадратичном значении градиентов.

Сглаживание градиентов: RMSProp использует экспоненциальное скользящее среднее для градиентов, что помогает сгладить их и уменьшить влияние резких изменений.

# RMSProp



[Image credit](#)

# Adam

Соединим идеи AdaGrad/RMSProp и Momentum

RMSProp

$$g_{t,i} := \frac{dL(\theta_{t,i})}{d\theta_{t,i}}$$

$$G_{t,i} = \beta G_{t-1,i} + (1 - \beta) g_{t,i}^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

SGD + momentum

$$v_t = \gamma v_{t-1} + \alpha \frac{dL(\theta)}{d\theta}$$

$$\theta_{t+1} = \theta_t - v_t$$

Adaptive Moment Estimation

Adam использует адаптивные скорости обучения для каждого параметра, основываясь на оценках первого и второго моментов градиентов. Adam использует экспоненциальное скользящее среднее для градиентов и их квадратов, что помогает сгладить обновления и уменьшить влияние резких изменений.

# Adam

Соединим идеи AdaGrad/RMSProp и Momentum

**RMSProp**

$$g_{t,i} := \frac{dL(\theta_{t,i})}{d\theta_{t,i}}$$

$$G_{t,i} = \beta G_{t-1,i} + (1 - \beta) g_{t,i}^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

**SGD + momentum**

$$v_t = \gamma v_{t-1} + \alpha \frac{dL(\theta)}{d\theta}$$

$$\theta_{t+1} = \theta_t - v_t$$

**Adam**

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} v_t$$

# Adam

Соединим идеи AdaGrad/RMSProp и Momentum

**RMSProp**

$$g_{t,i} := \frac{dL(\theta_{t,i})}{d\theta_{t,i}}$$

$$G_{t,i} = \beta G_{t-1,i} + (1 - \beta) g_{t,i}^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

**SGD + momentum**

$$v_t = \gamma v_{t-1} + \alpha \frac{dL(\theta)}{d\theta}$$

$$\theta_{t+1} = \theta_t - v_t$$

**Adam**

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} v_t$$

# Adam

Соединим идеи AdaGrad/RMSProp и Momentum

**RMSProp**

$$g_{t,i} := \frac{dL(\theta_{t,i})}{d\theta_{t,i}}$$

$$G_{t,i} = \beta G_{t-1,i} + (1 - \beta) g_{t,i}^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

**SGD + momentum**

$$v_{t+1} = \gamma v_t + \alpha \frac{dL(\theta)}{d\theta}$$

$$\theta_{t+1} = \theta_t - v_t$$

**Adam**

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t$$

$$G_t = \beta_2 G_{t-1} + (1 - \beta_2) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} v_t$$

# Adam

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t$$

$$\beta_1 = 0.9$$

$$\beta_2 = 0.999$$

$$G_t = \beta_2 G_{t-1} + (1 - \beta_2) g_t^2$$

$$v_0 = G_0 = 0$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} v_t$$

Какие будут первые шаги?

# Adam

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t$$

$$\beta_1 = 0.9$$

$$\beta_2 = 0.999$$

$$G_t = \beta_2 G_{t-1} + (1 - \beta_2) g_t^2$$

$$v_0 = G_0 = 0$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_1^t} \quad \hat{G}_t = \frac{G_t}{1 - \beta_2^t}$$

bias correction

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{G}_t + \epsilon}} \hat{v}_t$$

# Adam

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t$$

$$\beta_1 = 0.9$$

$$\beta_2 = 0.999$$

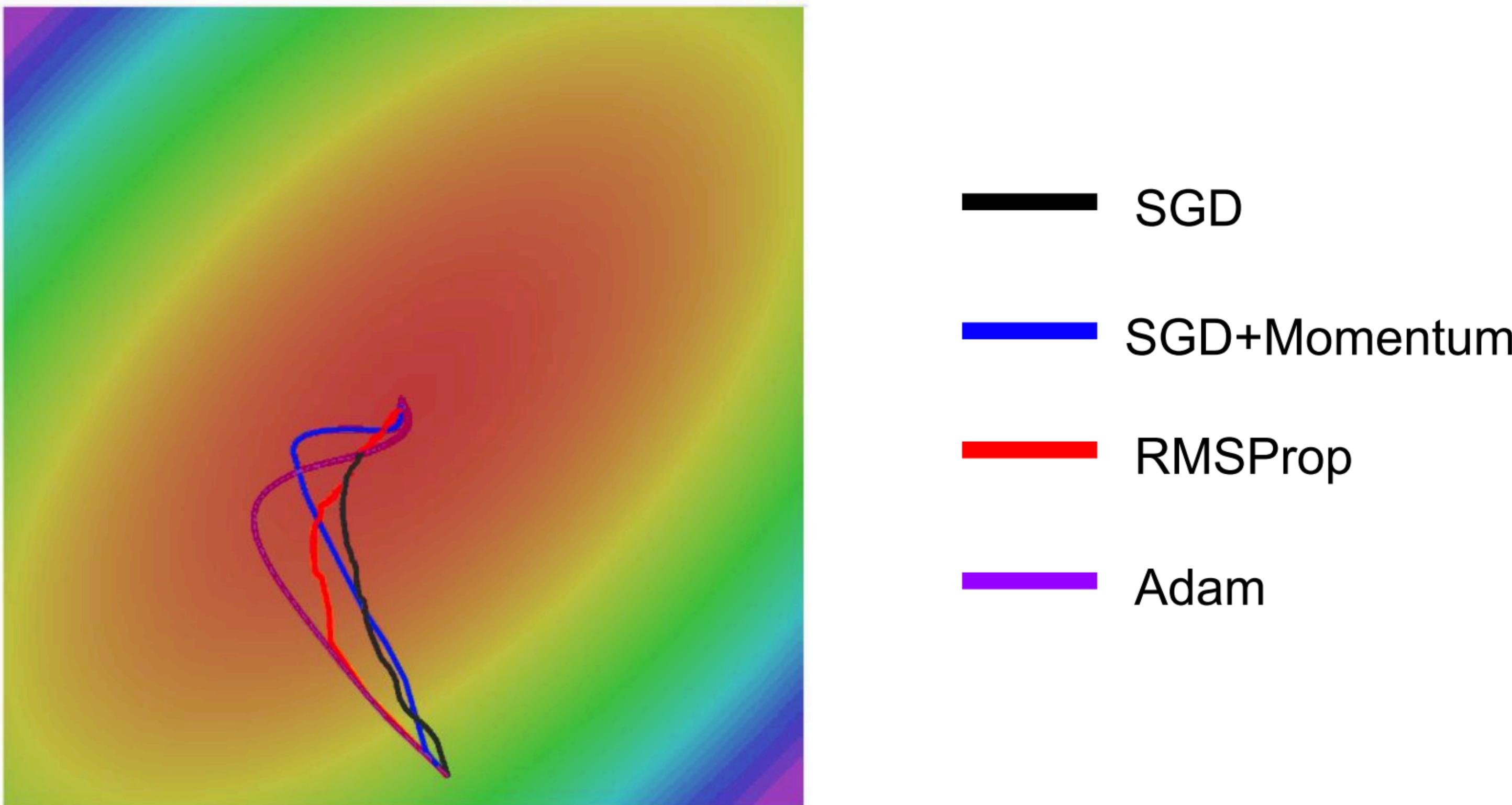
$$G_t = \beta_2 G_{t-1} + (1 - \beta_2) g_t^2$$

$$v_0 = G_0 = 0$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_1^t} \quad \quad \hat{G}_t = \frac{G_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{G}_t + \epsilon}} \hat{v}_t$$

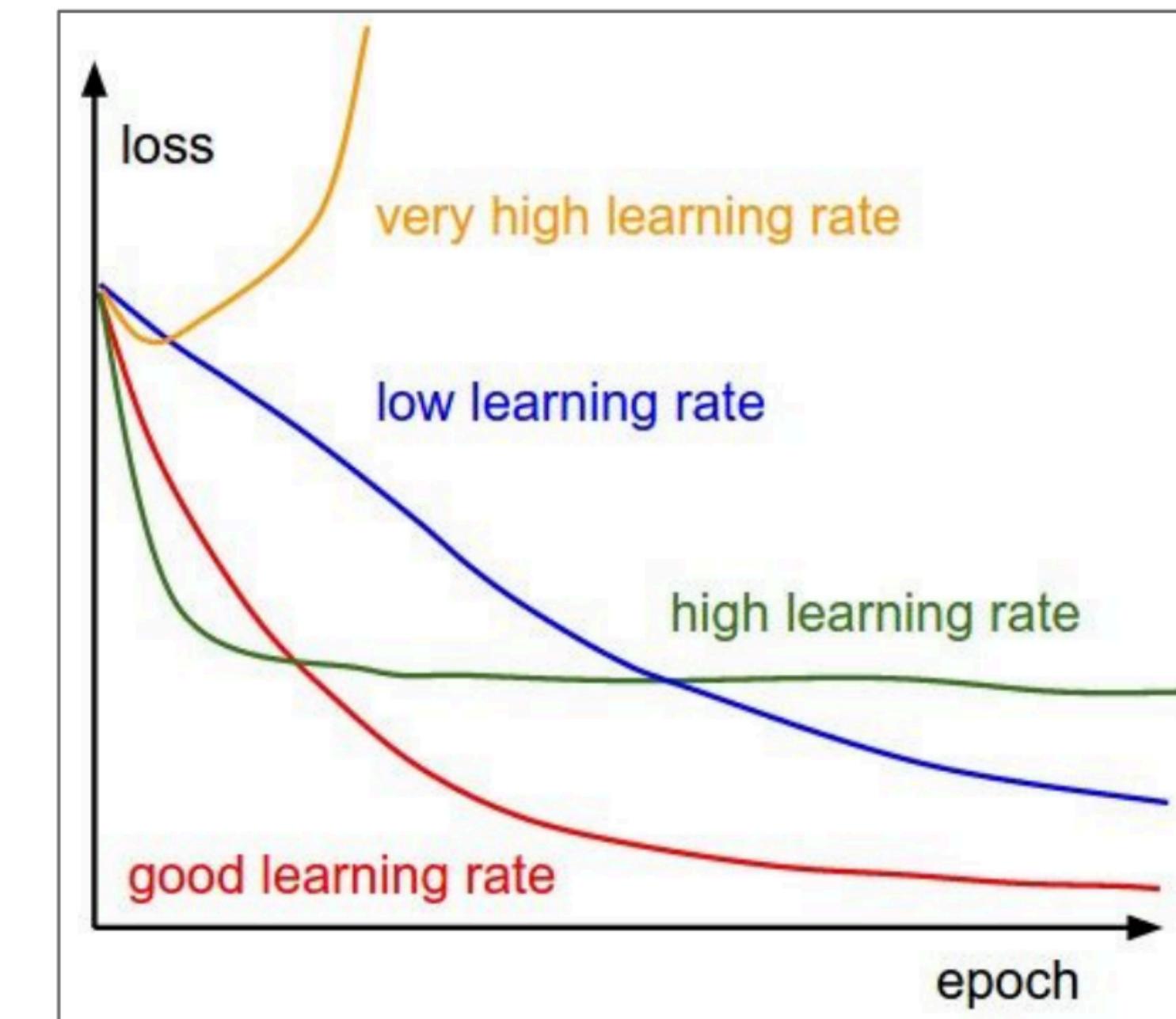
# Adam



[Image credit](#)

# Выводы

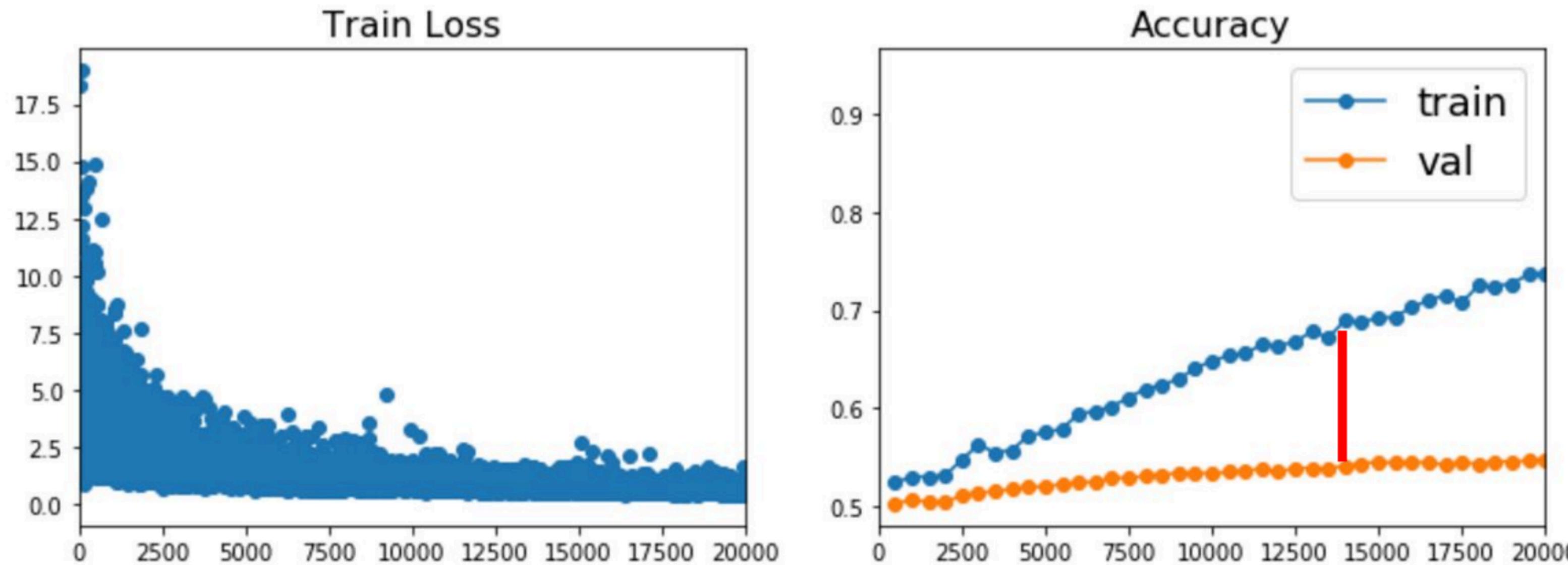
- Adam - хороший выбор для начала
- Learning rate - важный параметр
- LR Scheduler может помочь



[Image credit](#)

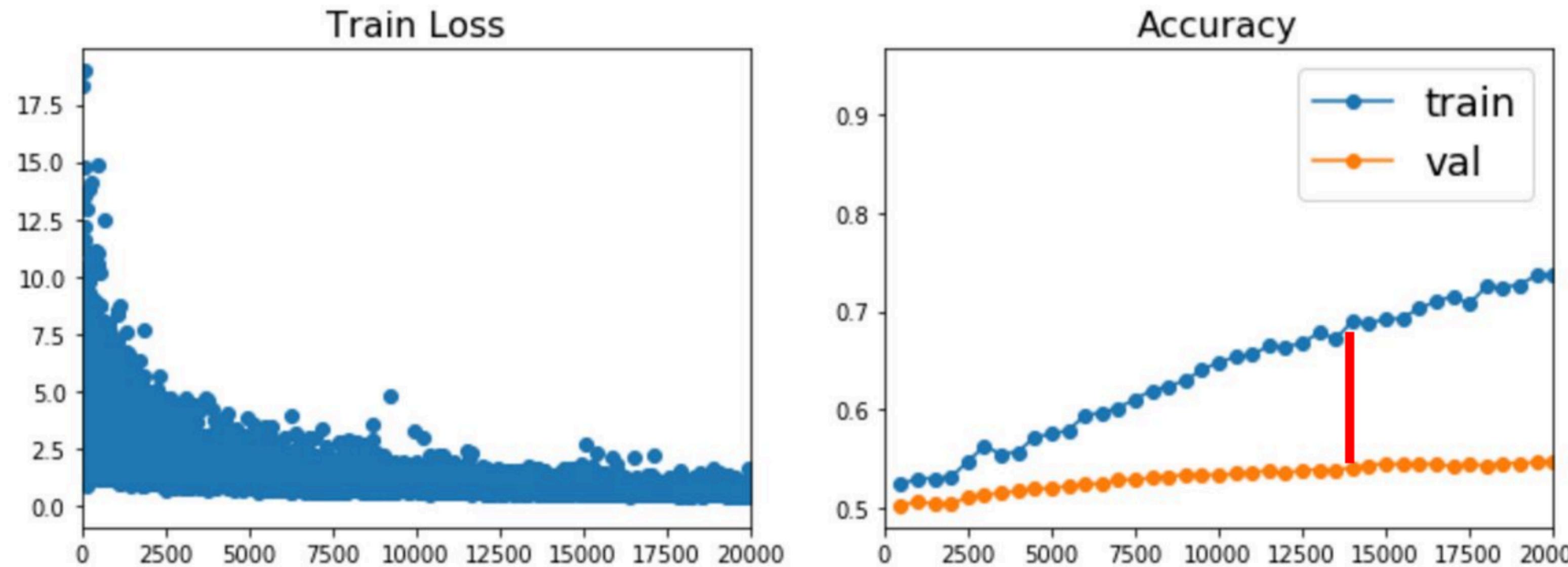
# **Обучение нейросетей**

# Обучение



Train loss падает, метрика на train улучшается - модель обучается!

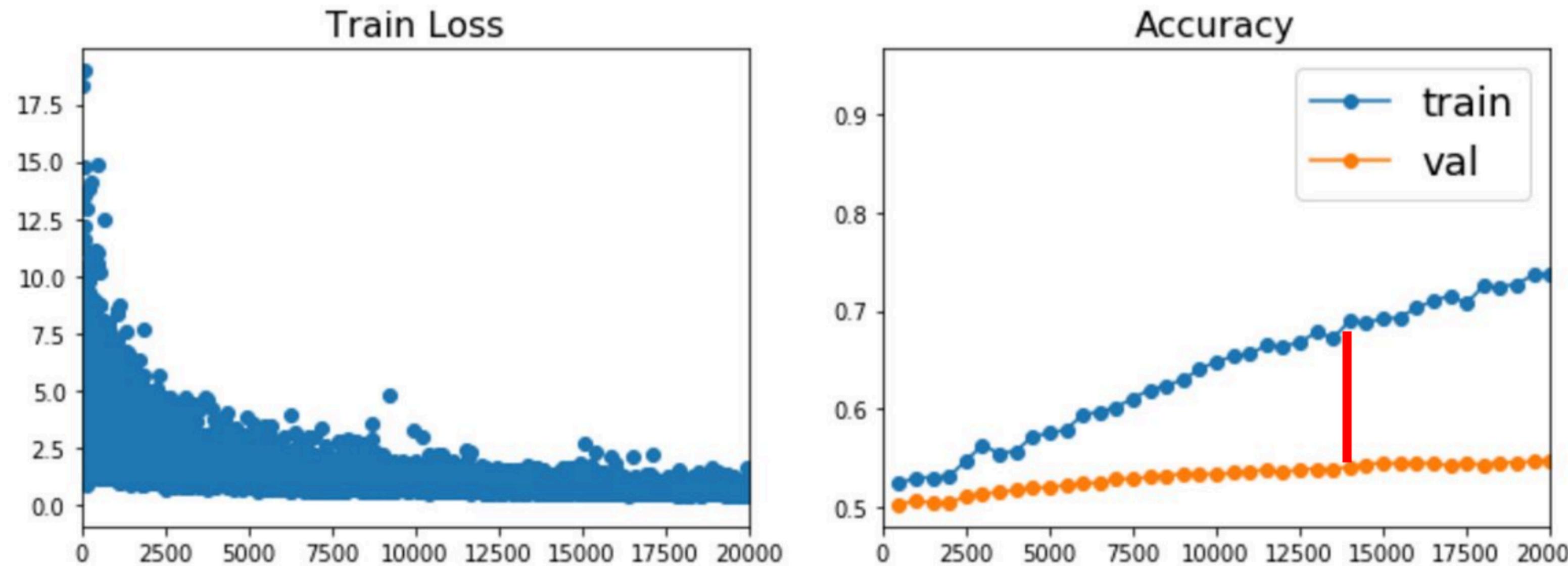
# Обучение



Train loss падает, метрика на train улучшается - модель обучается!

Проблема: на валидации качество сильно хуже

# Обучение



Train loss падает, метрика на train улучшается - модель обучается!

Проблема: на валидации качество сильно хуже

Переобучение, нужна регуляризация

# Регуляризация

$$\frac{1}{n} \sum_i L(y_i, f(x_i, \theta)) + \lambda R(\theta) \rightarrow \min_{\theta}$$

$$R(\theta) = \sum_l \|\theta_l\|^2$$

L2-регуляризация (weight decay)

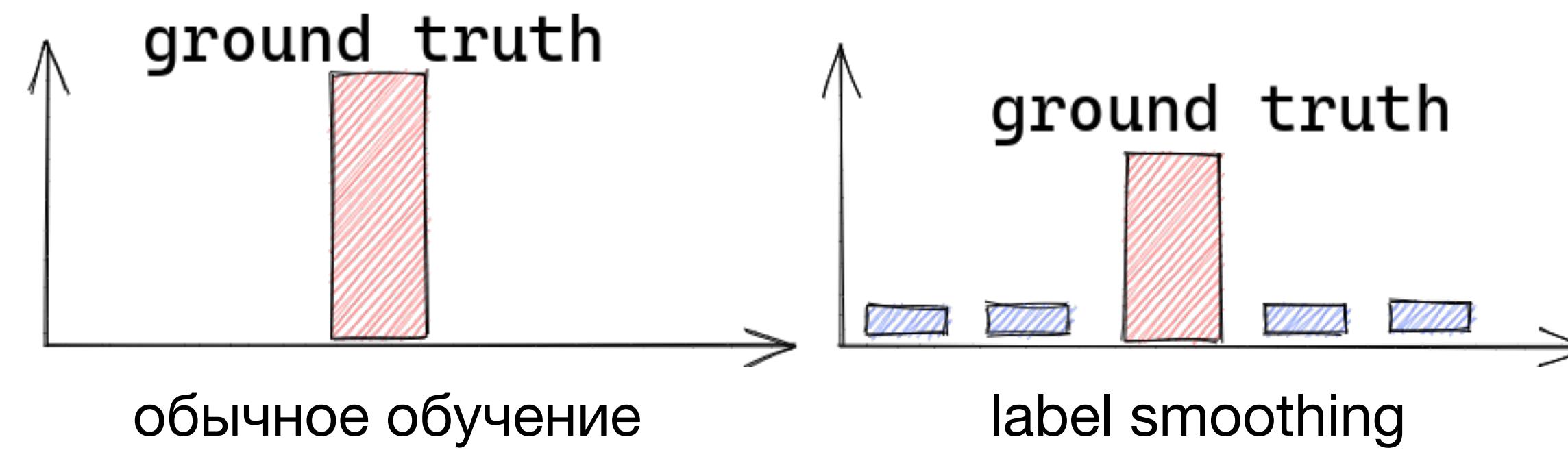
$$R(\theta) = \sum_l |\theta_l|$$

L1-регуляризация

$$R(\theta) = \sum_l |\theta_l| + \|\theta_l\|^2$$

Elastic Net (L1 + L2 -регуляризация)

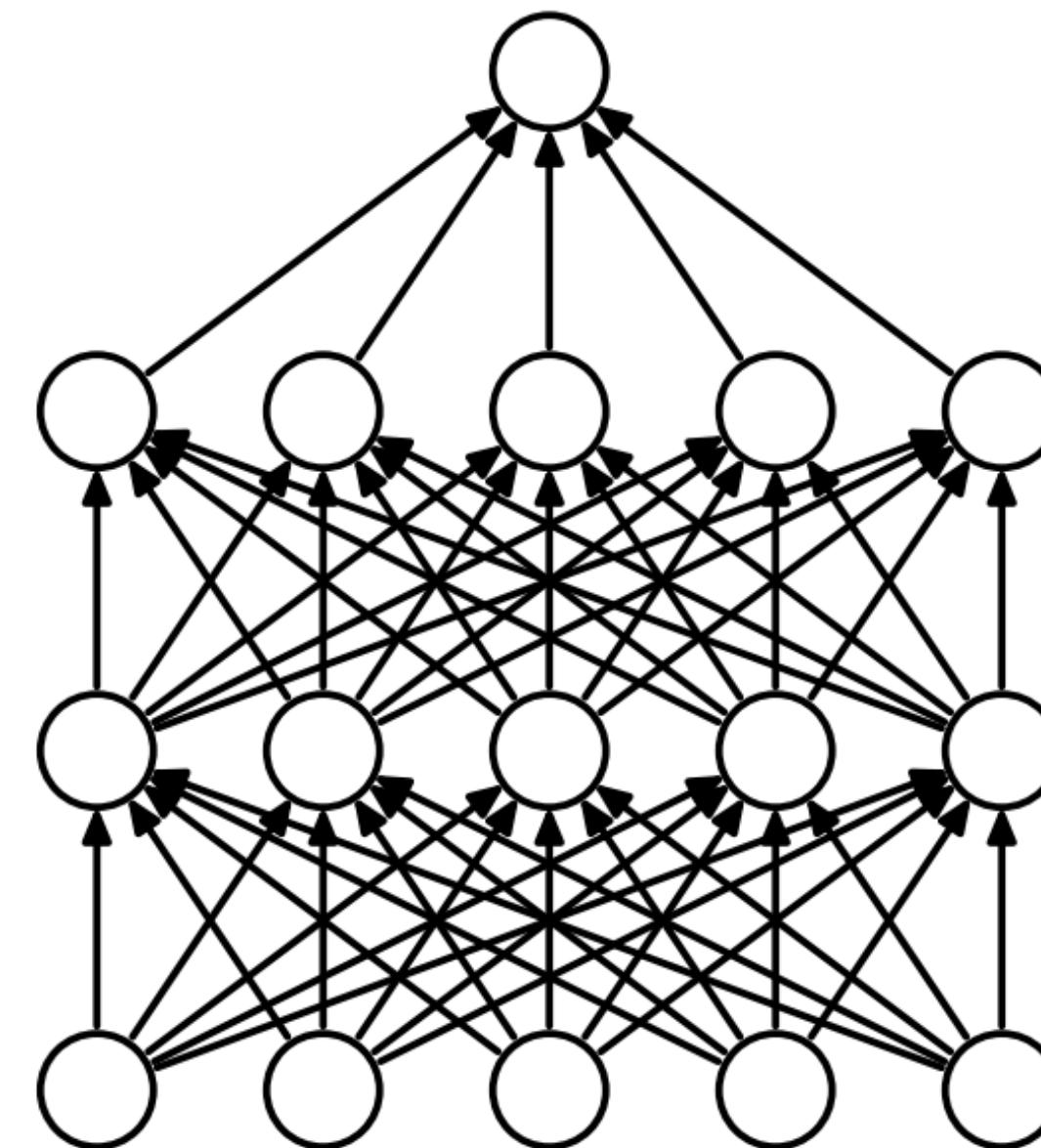
# Регуляризация: label smoothing



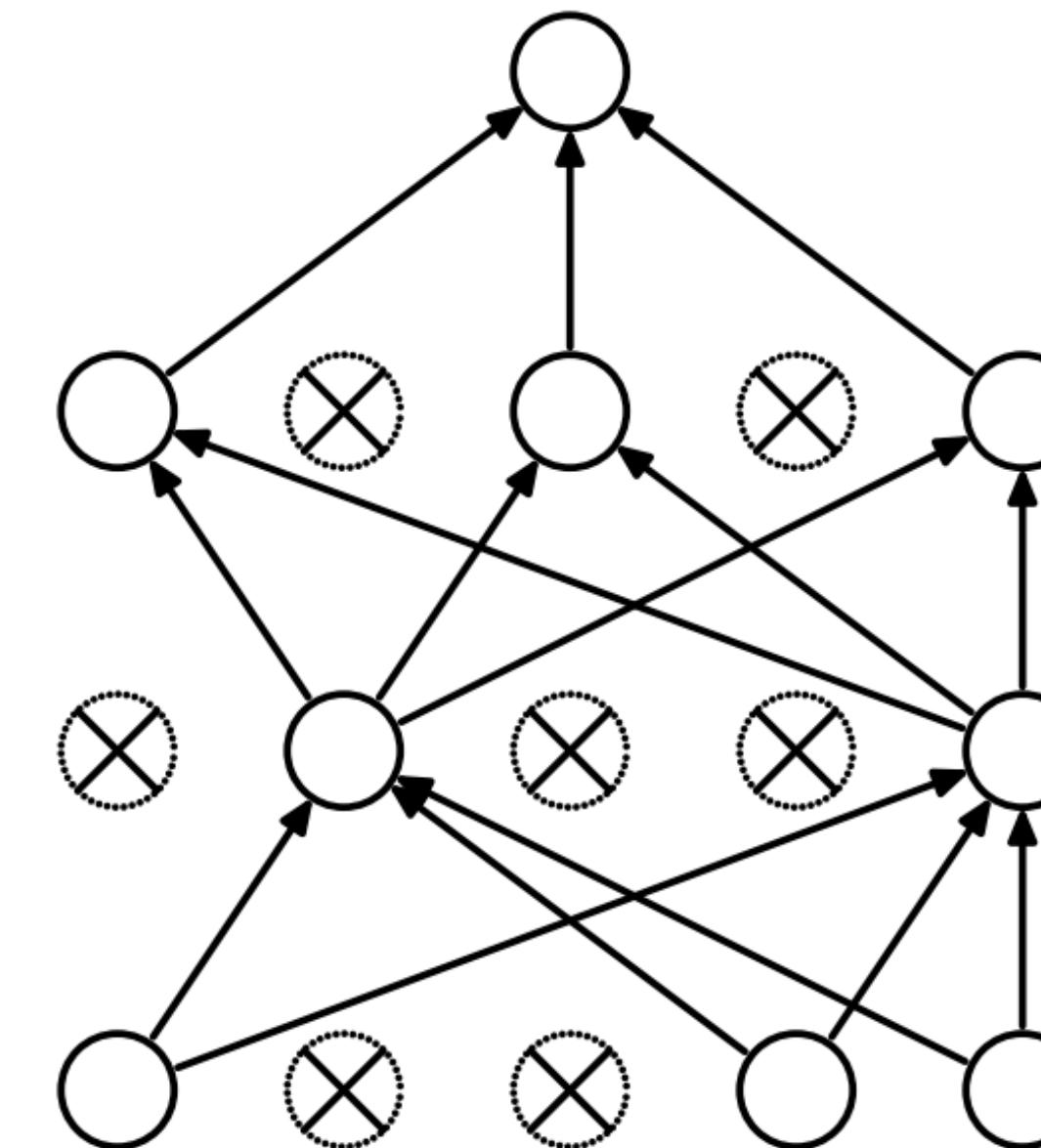
Target распределение: вероятность правильной метки  $\alpha$ , остальных  $1 - \alpha$

Вместо того чтобы использовать жесткие метки (например, 0 и 1) для классов, label smoothing заменяет их на "размягченные" метки. Это означает, что вместо того, чтобы назначать метке класса значение 1, мы назначаем ей значение, немного меньшее, а всем остальным классам — значения, немного больше нуля. Это помогает избежать чрезмерной уверенности модели в своих предсказаниях.

# Регуляризация: dropout



(a) Standard Neural Net

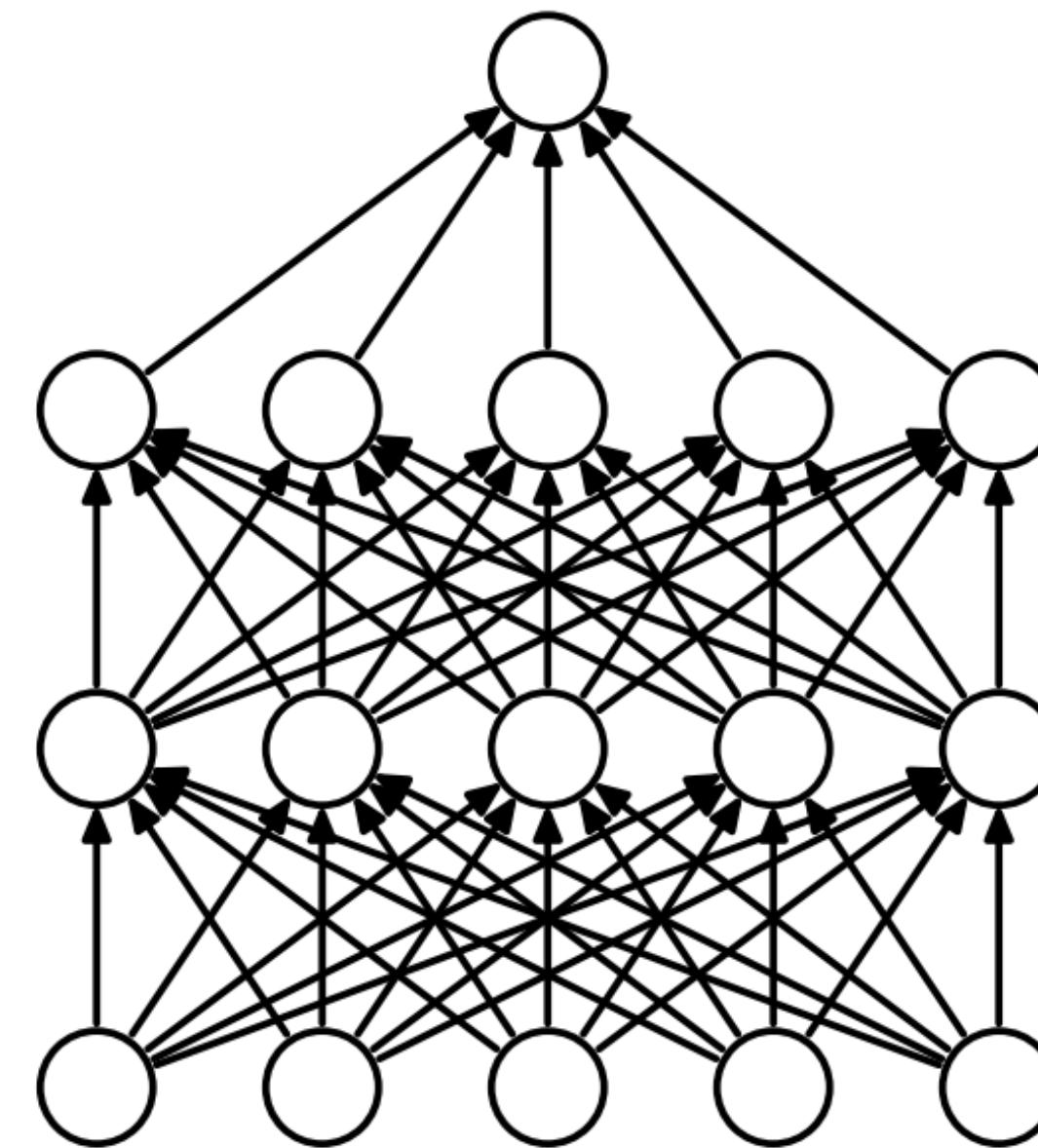


(b) After applying dropout.

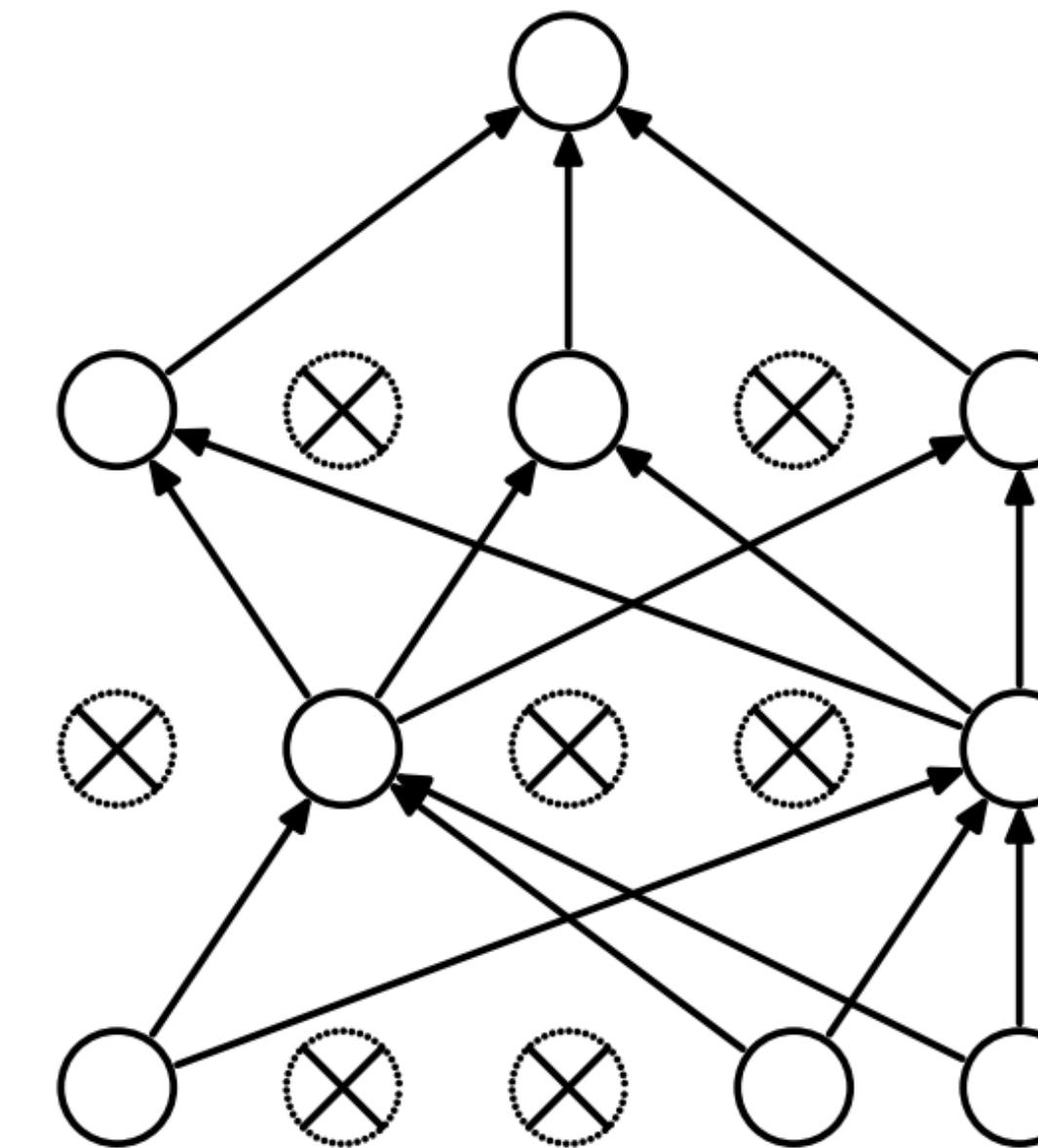
[Image credit](#)

Случайно выкидываем нейроны на forward pass с **вероятностью  $p$**

# Регуляризация: dropout



(a) Standard Neural Net

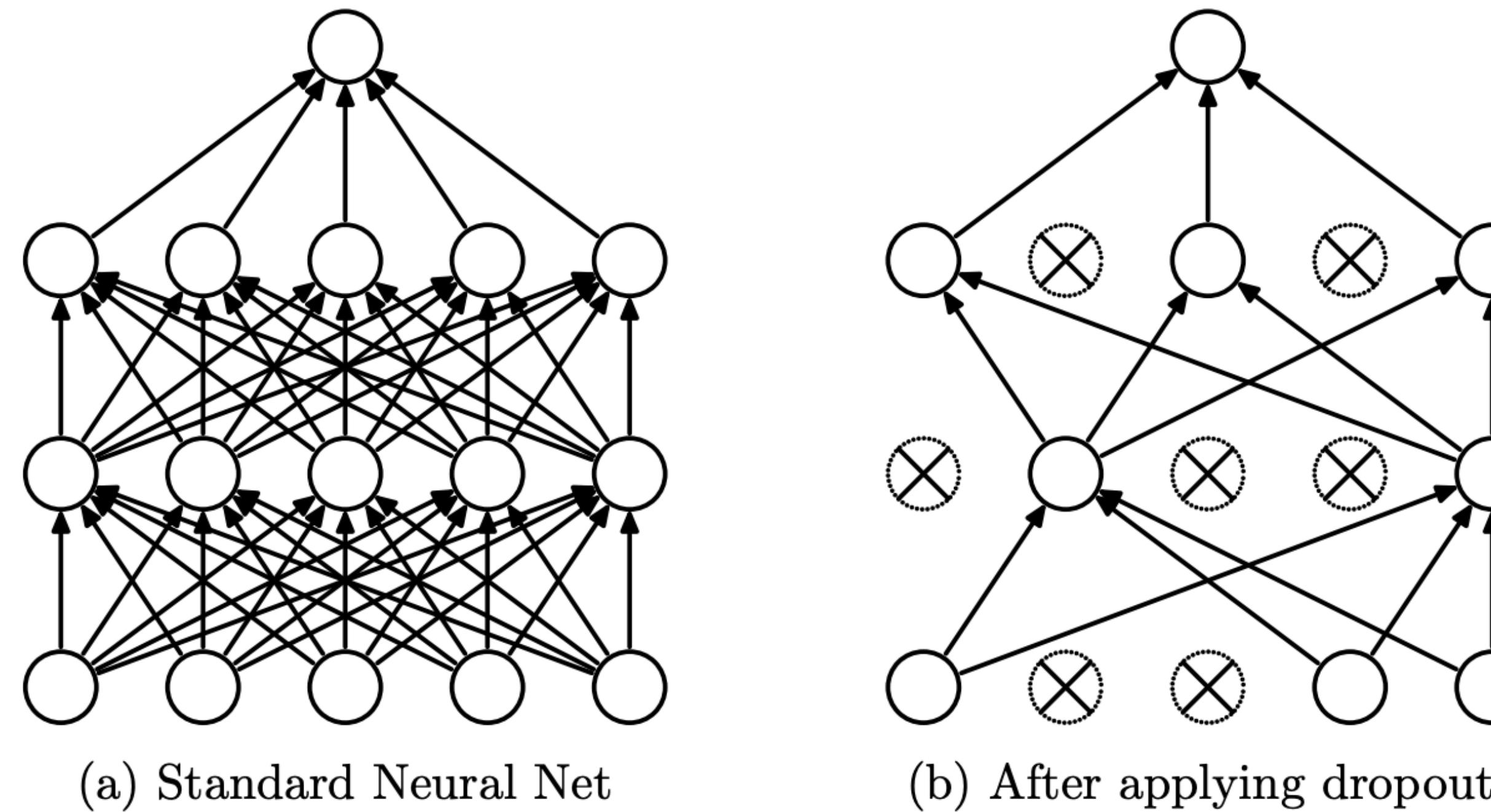


(b) After applying dropout.

[Image credit](#)

Случайно выкидываем нейроны на forward pass с **вероятностью**  $p$   
Во время обучения и теста работает по-разному!

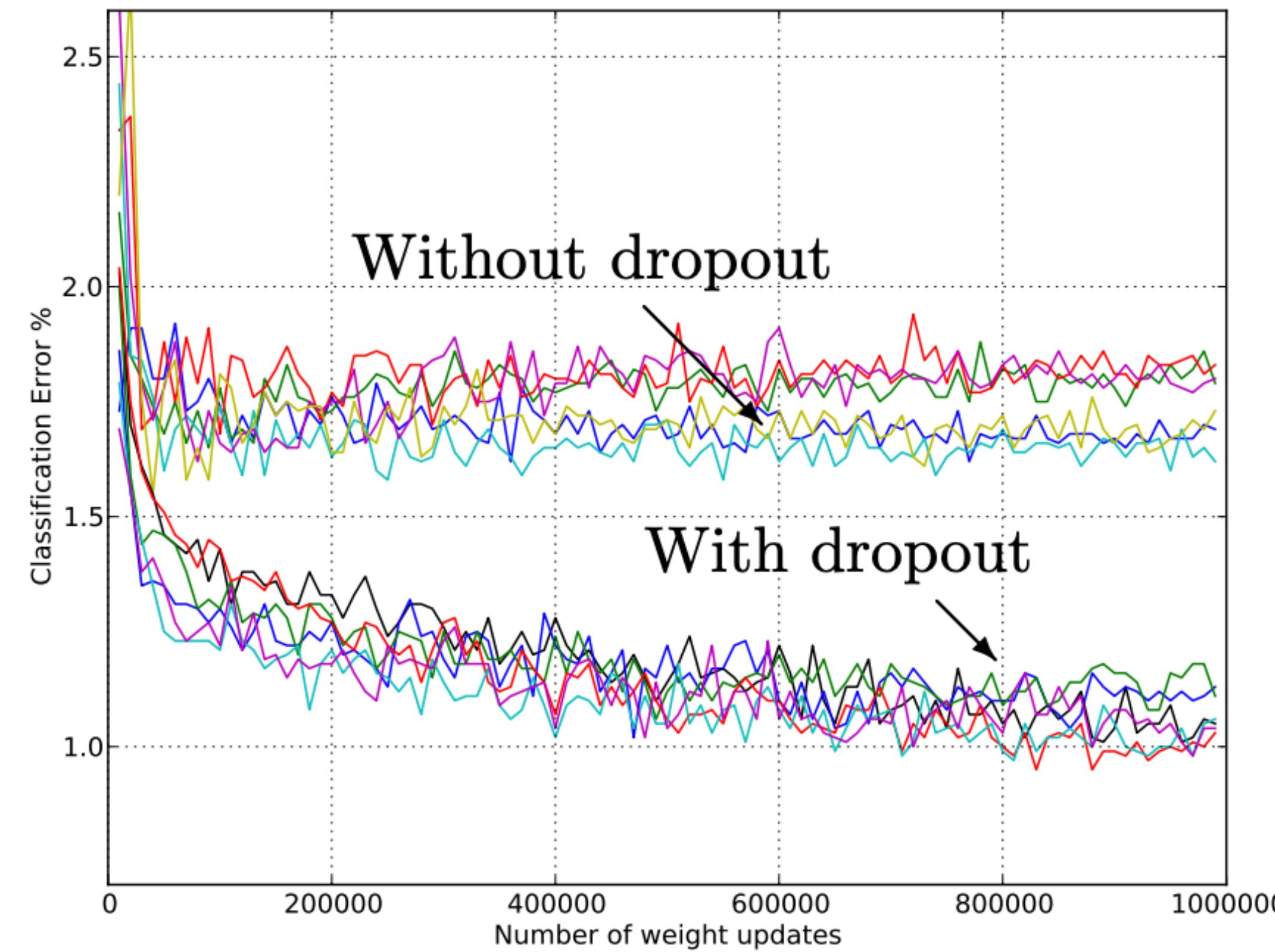
# Регуляризация: dropout



Случайно выкидываем нейроны на forward pass с **вероятностью**  $p$   
Обучение: умножаем оставшиеся нейроны на  $\frac{1}{1-p}$

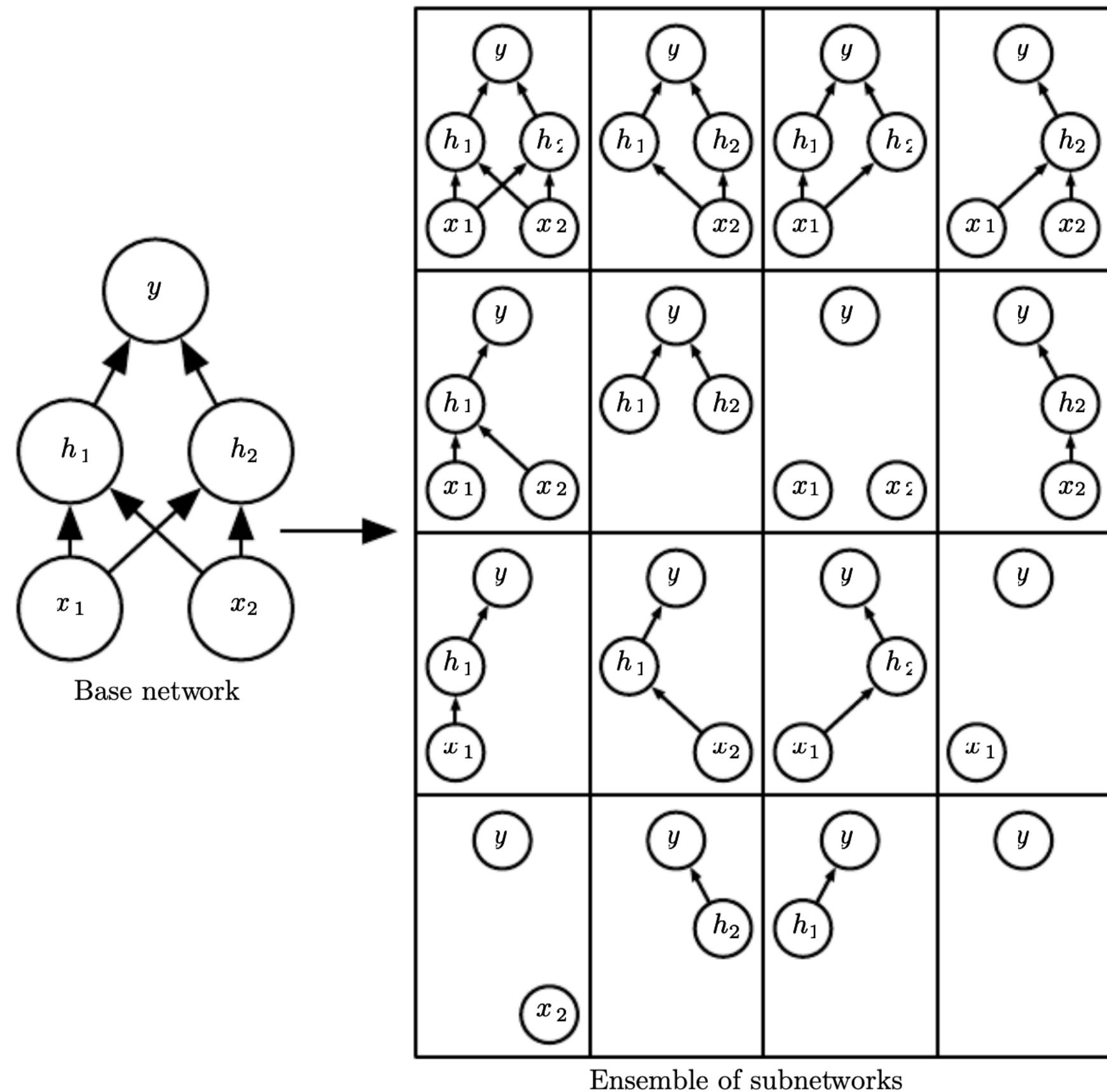
[Image credit](#)

# Регуляризация: dropout



[Image credit](#)

# Регуляризация: dropout



Обучение ансамбля нейросетей  
(shared параметры)

Image credit

# Регуляризация

- Из классического ML (L1/L2 регуляризация, label smoothing)
- Early stopping

Early Stopping (ранняя остановка) — это метод регуляризации, используемый для предотвращения переобучения нейронной сети. Он заключается в прекращении обучения модели, когда её производительность на валидационном наборе данных перестает улучшаться
- Dropout

Дополнительные задачи (или auxiliary tasks) — это дополнительные задачи, которые могут быть использованы для улучшения обучения модели. Они помогают модели лучше понять данные и могут привести к улучшению производительности на основной задаче.  
Примеры:

  - Смешение задач: Одновременное обучение модели нескольким связанным задачам, например, классификация и сегментация.
  - Предсказание атрибутов: Например, в задаче распознавания лиц можно одновременно предсказывать возраст и пол.
- Больше данных, меньше модель

Больше данных, меньше модель — это принцип, который указывает на то, что увеличение объёма обучающих данных может позволить использовать менее сложные модели без риска переобучения.
- Аугментации

Аугментация данных — это метод, используемый для искусственного увеличения объёма обучающего набора данных путём применения различных преобразований к имеющимся данным.

# Batch Normalization

Рассмотрим один (не первый) слой нейросети:

во время обучения меняется распределение входа (**internal covariate shift**)

- меняется вход в саму нейросеть (разные объекты)
- меняются веса предыдущих входов

# Batch Normalization

Рассмотрим один (не первый) слой нейросети:

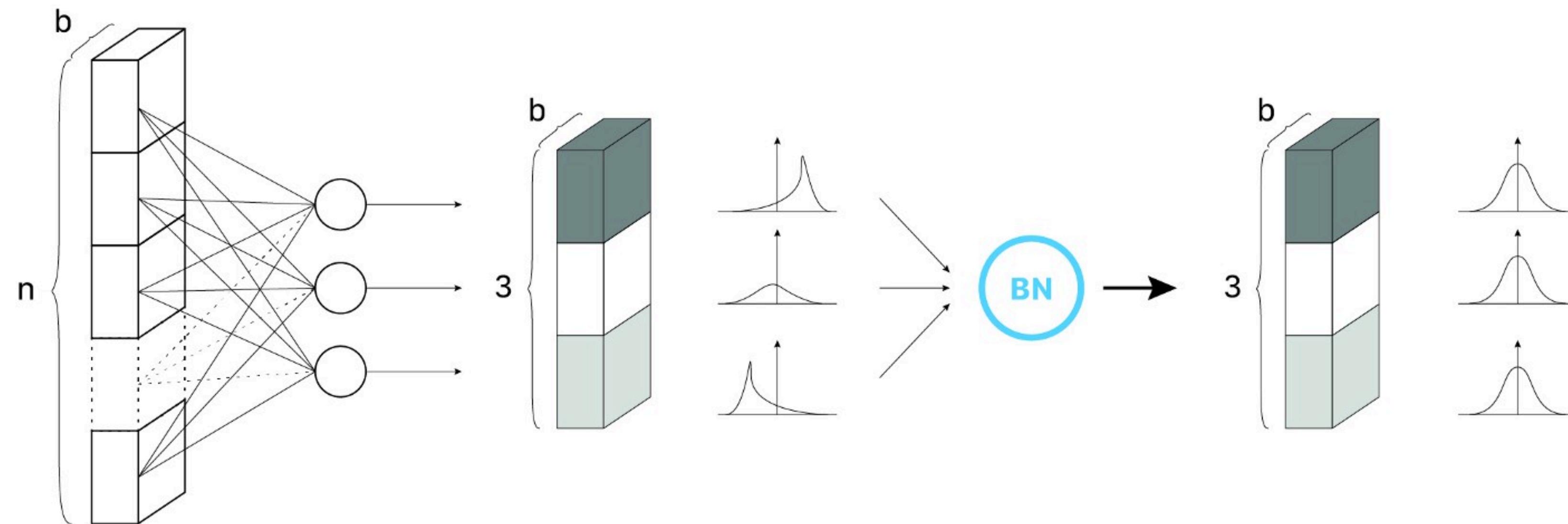
во время обучения меняется распределение входа (**internal covariate shift**)

- меняется вход в саму нейросеть (разные объекты)
- меняются веса предыдущих входов

**Batch normalization: будем нормировать среднее и дисперсию входов**

Batch Normalization нормализует входы каждого слоя нейронной сети, чтобы они имели нулевое среднее и единичную дисперсию. Это помогает устранить проблему, известную как "внутренний ковариационный сдвиг", когда распределение входов для каждого слоя меняется во время обучения, что может замедлить процесс обучения.

# Batch Normalization



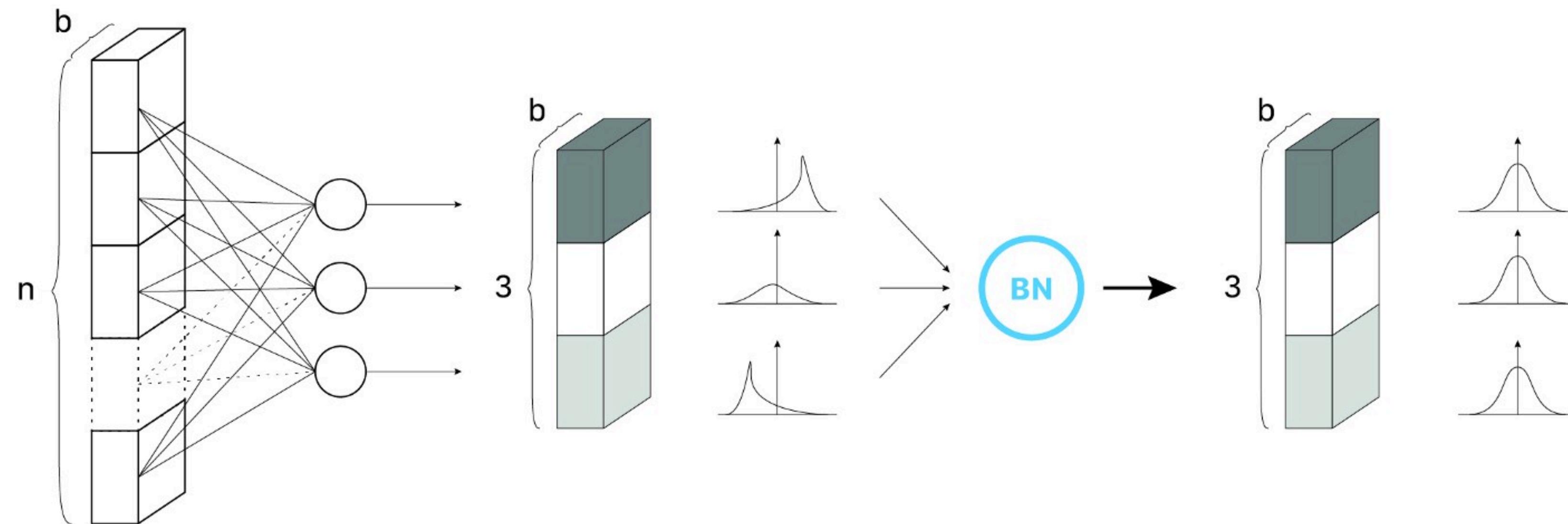
[Image credit](#)

Batch normalization: будем нормировать среднее и дисперсию входов

$x_1, \dots, x_m$  input mini-batch

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

# Batch Normalization



[Image credit](#)

Batch normalization: будем нормировать среднее и дисперсию входов

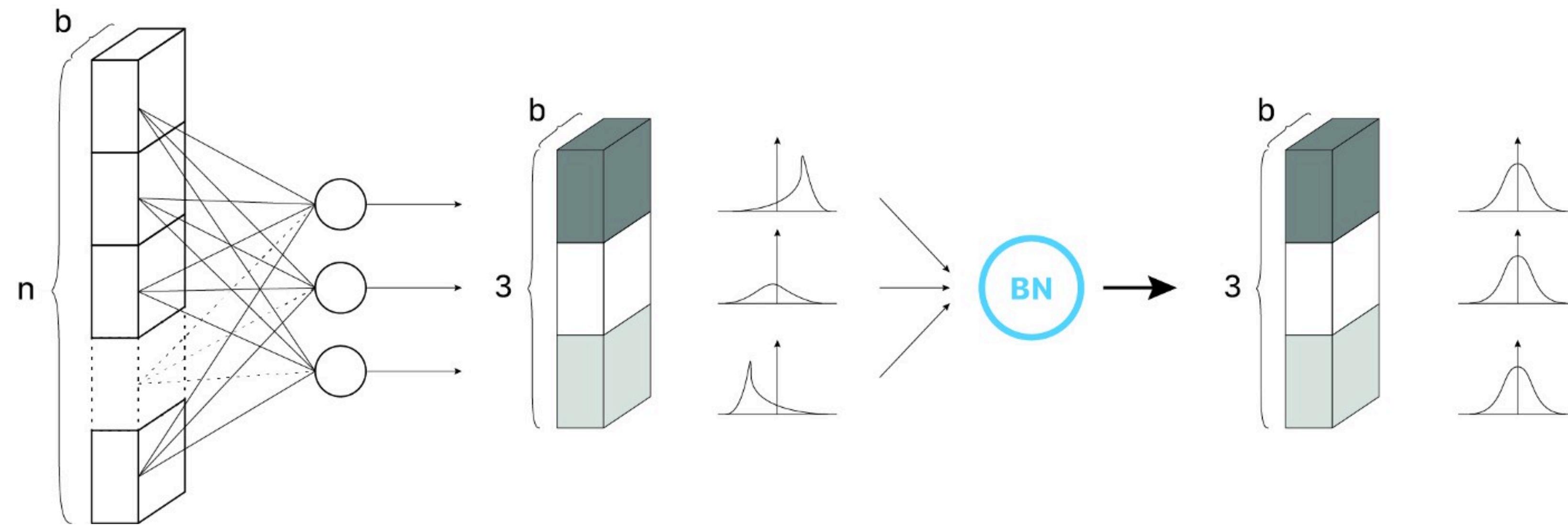
$x_1, \dots, x_m$  input mini-batch

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$



$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

# Batch Normalization



[Image credit](#)

Batch normalization: будем нормировать среднее и дисперсию входов

$x_1, \dots, x_m$  input mini-batch

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$



$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Identity transformation?

# Batch Normalization

$x_1, \dots, x_m$  input mini-batch

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta \quad \gamma, \beta \text{ - обучаемые параметры}$$

# Batch Normalization

$x_1, \dots, x_m$  input mini-batch

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i$$

- зависит от батча,  
как использовать на teste?

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta \quad \gamma, \beta \text{ - обучаемые параметры}$$

# Batch Normalization

$x_1, \dots, x_m$  input mini-batch

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad \hat{\mu} = \alpha \mu_B + (1 - \alpha) \hat{\mu}$$

- для обучения

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \quad \hat{\sigma}^2 = \alpha \sigma_B^2 + (1 - \alpha) \hat{\sigma}^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

# Batch Normalization

$x_1, \dots, x_m$  input mini-batch

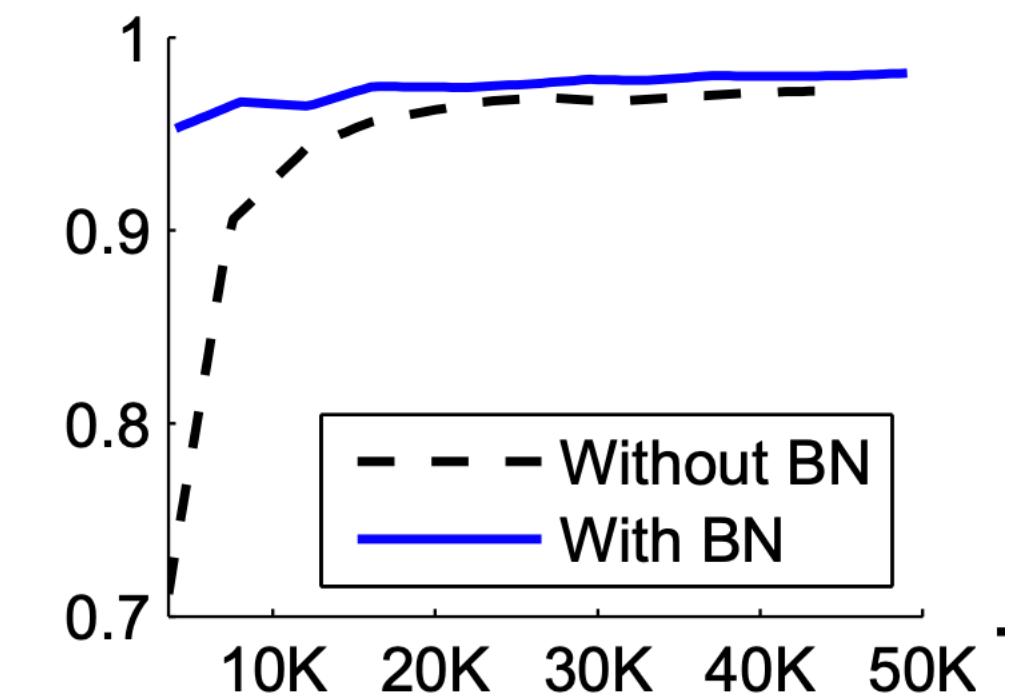
$$\hat{x}_i = \frac{x_i - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}}$$

- для теста

$$y_i = \gamma \hat{x}_i + \beta$$

# Batch Normalization

- Быстрее сходимость
- Позволяет использовать значения  $\gamma$  больше
- Нейросети менее чувствительны к инициализации



Обычно используется между Linear/Conv и нелинейностью

# Batch Normalization

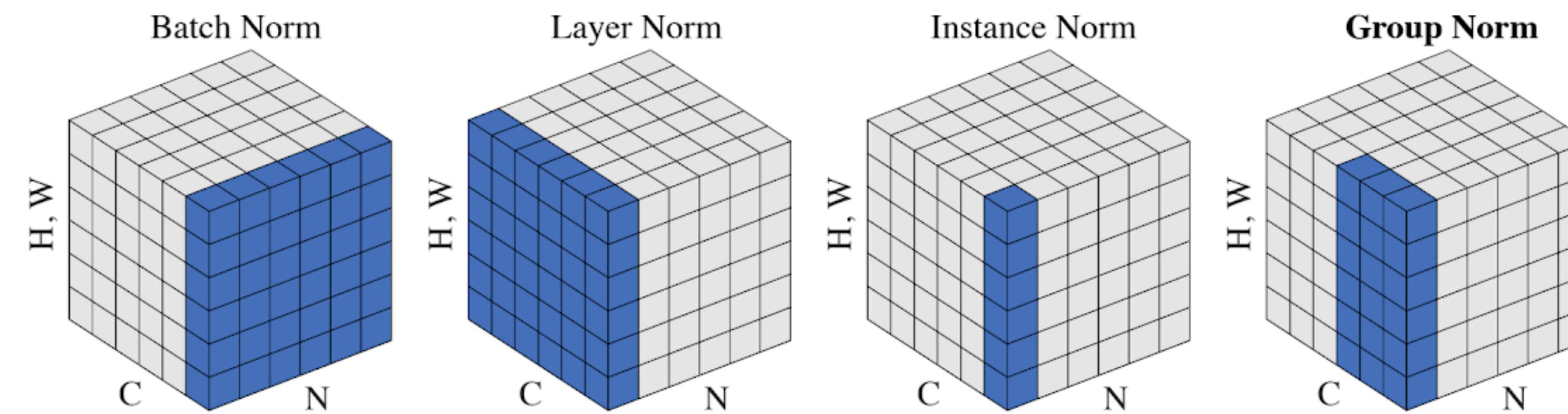


Figure 2. **Normalization methods.** Each subplot shows a feature map tensor, with  $N$  as the batch axis,  $C$  as the channel axis, and  $(H, W)$  as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.