Обучение нейронных сетей

Лекция 3

План лекции

- Регуляризация
- Инициализация весов
- Нормализация по мини-батчам
- Варианты градиентного спуска

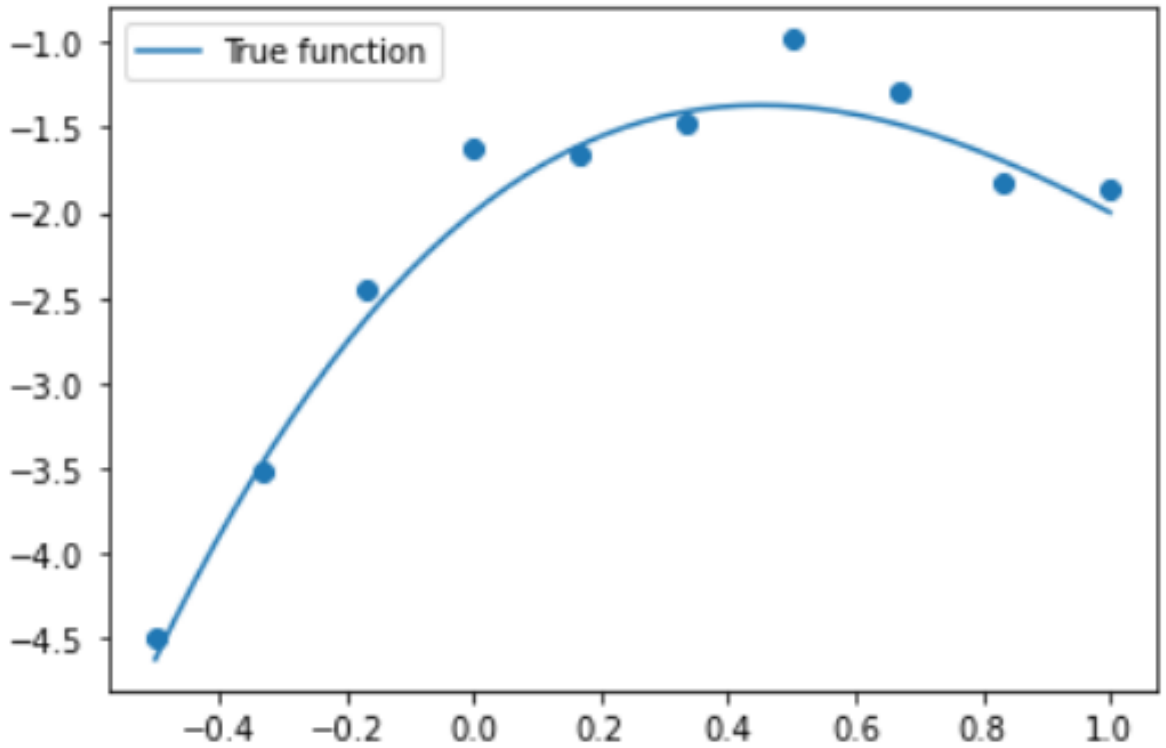
Регуляризация

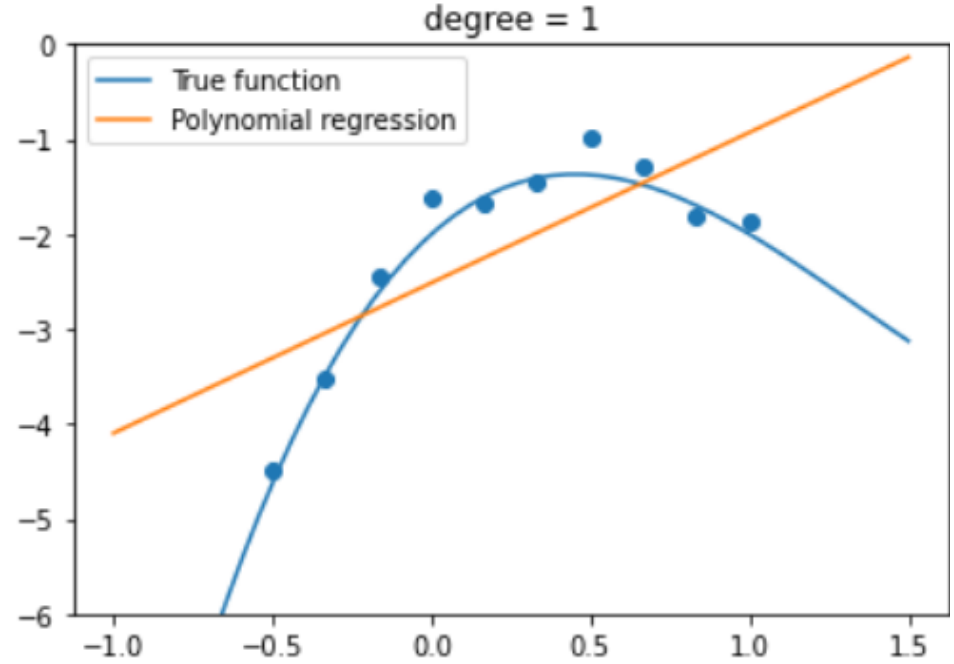
- Сокращение весов (Weight decay)
  - L1-регуляризация
  - L2-регуляризация
- Ранняя остановка (Early stopping)
- Дропаут (Dropout)

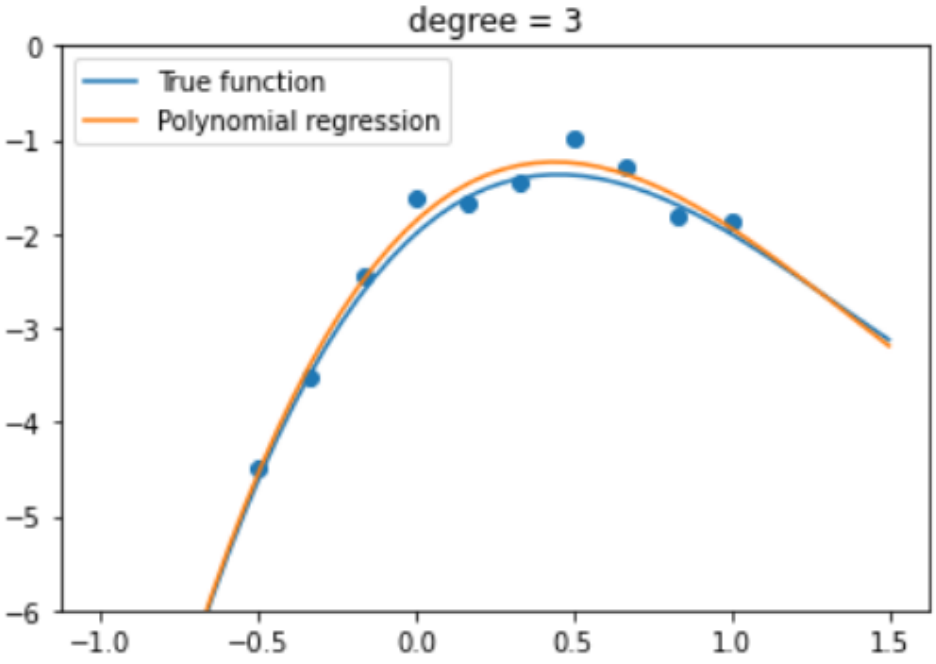
Регуляризация

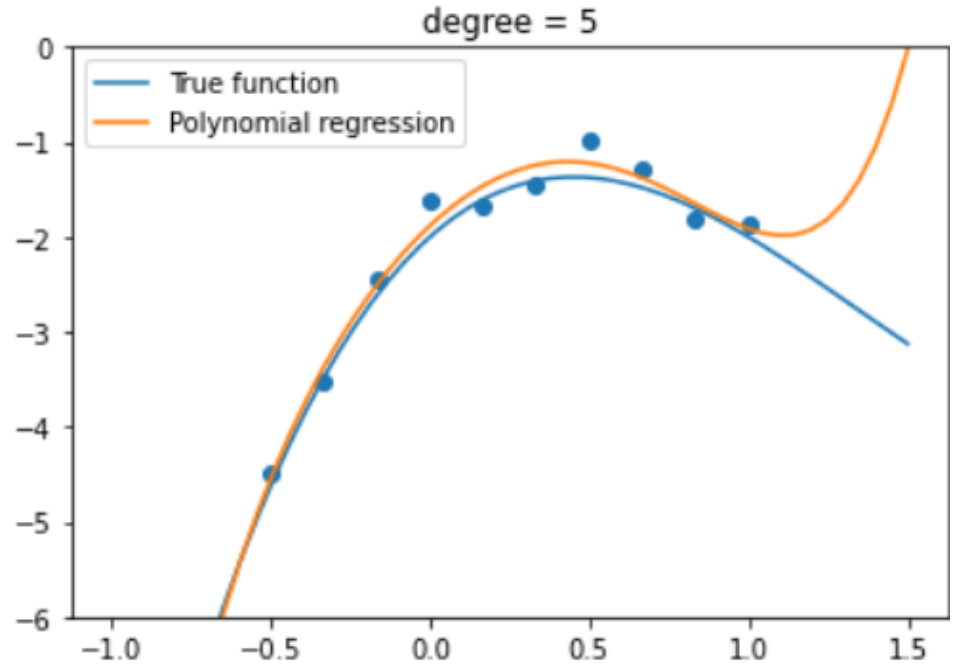
$$f(x) = x^3 - 4x^2 + 3x - 2$$

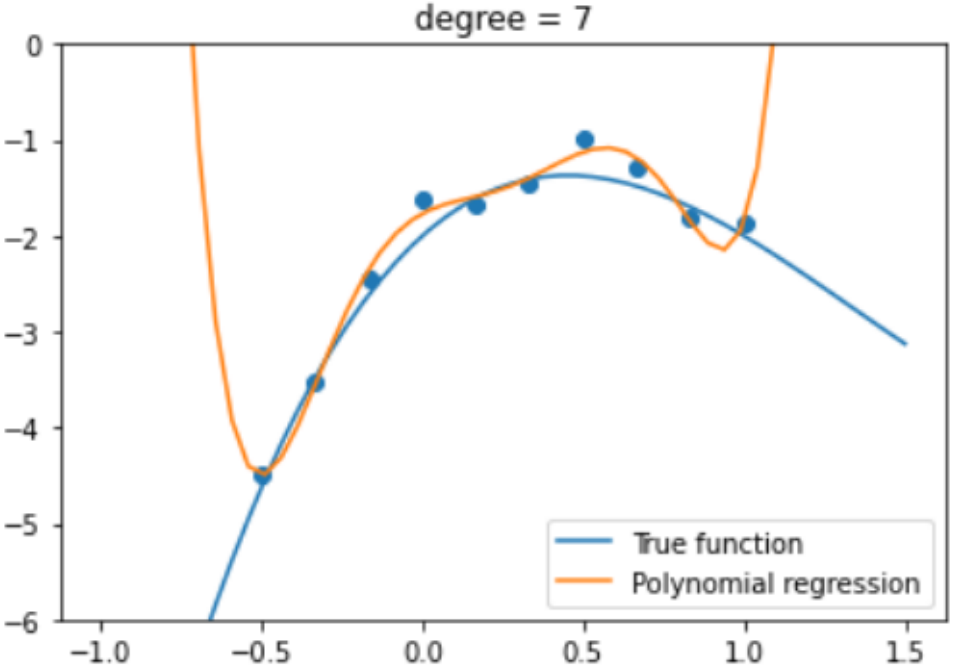
+ нормально распределенный шум $N(0, 0.25)$




degree = 1
True function
Polynomial regression
0
−1
−2
−3
−4
−5
−6
−1.0
−0.5
0.0
0.5
1.0
1.5


degree = 3
True function
Polynomial regression
0
−1
−2
−3
−4
−5
−6
−1.0
−0.5
0.0
0.5
1.0
1.5


degree = 5
True function
Polynomial regression
0
−1
−2
−3
−4
−5
−6
−1.0
−0.5
0.0
0.5
1.0
1.5


degree = 7
True function
Polynomial regression
0
−1
−2
−3
−4
−5
−6
−1.0
−0.5
0.0
0.5
1.0
1.5

Регуляризация

Весовые коэффициенты:

- Degree = 1: [–2.5, 1.6]
- Degree = 3: [–1.9, 3.1, –4.1, 1.0]
- Degree = 5: [–1.9, 3.1, –3.7, 0.5, –1.3, 1.4]
- Degree = 7: [–1.8, 1.8, –7.5, 18.5, 8.4, –51.4, 20.4, 9.8]
- Degree = 9: [–1.6, 1.5, –19.2, 36.8, 127.6, –280.0, 186.1, 653.5, –352.4, 18.2]

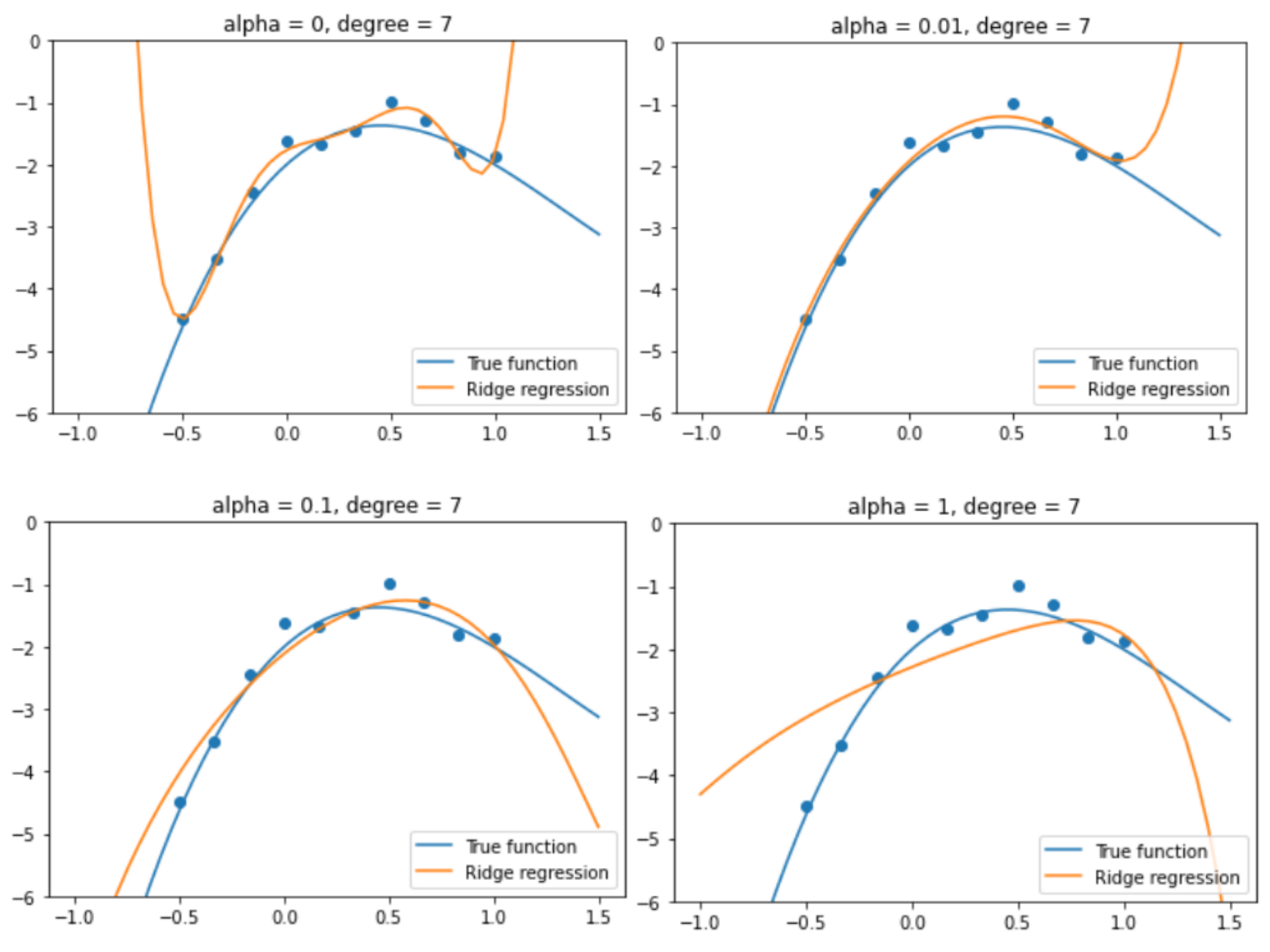
Регуляризация: weight decay

- L_2-регуляризатор – гребневая регрессия (ridge regression):

$$R(\vec{w}) = \|\vec{w}\|_2^2 = \sum_{i=1}^{d} w_i^2$$

- L_1-регуляризатор – Lasso regression (least absolute shrinkage and selection operator):

$$R(\vec{w}) = \|\vec{w}\|_1 = \sum_{i=1}^{d} |w_i|$$


alpha = 0, degree = 7
alpha = 0.01, degree = 7
alpha = 0.1, degree = 7
alpha = 1, degree = 7
True function
Ridge regression

Регуляризация: weight decay

Весовые коэффициенты (degree = 7):

- $\alpha = 0$: [–1.8, 1.8, –7.5, 18.5, 8.4, –51.4, 20.4, 9.8]
- $\alpha = 0.01$: [–1.9, 3.1, –3.3, 0.5, –1.1, –0.1, 0.1, 0.8]
- $\alpha = 0.1$: [–2.1, 2.7, –2.0, 0.2, –0.7, –0.1, –0.1, 0.1]
- $\alpha = 1.0$: [–2.3, 1.4, –0.4, 0.1, –0.2, –0.1, –0.2, –0.1]

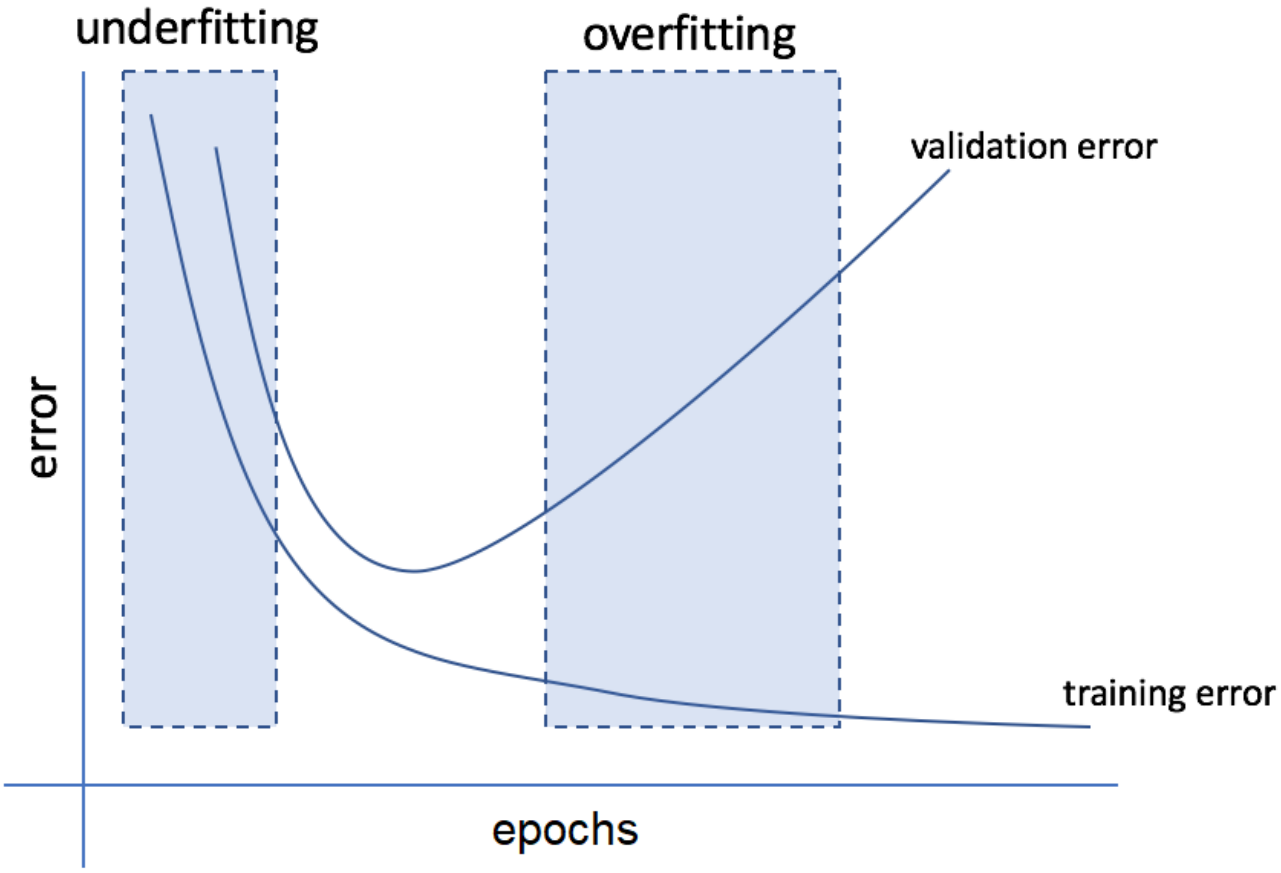
Регуляризация: weight decay – PyTorch

- L2-регуляризация: параметр `weight_decay` в оптимизаторах
  - По умолчанию = 0
- L1-регуляризация:

```
l1_alpha = 0.01
reg_loss = 0
for param in model.parameters():
     reg_loss += torch.norm(param, 1)
loss += l1_alpha * reg_loss
```

Регуляризация: early stopping

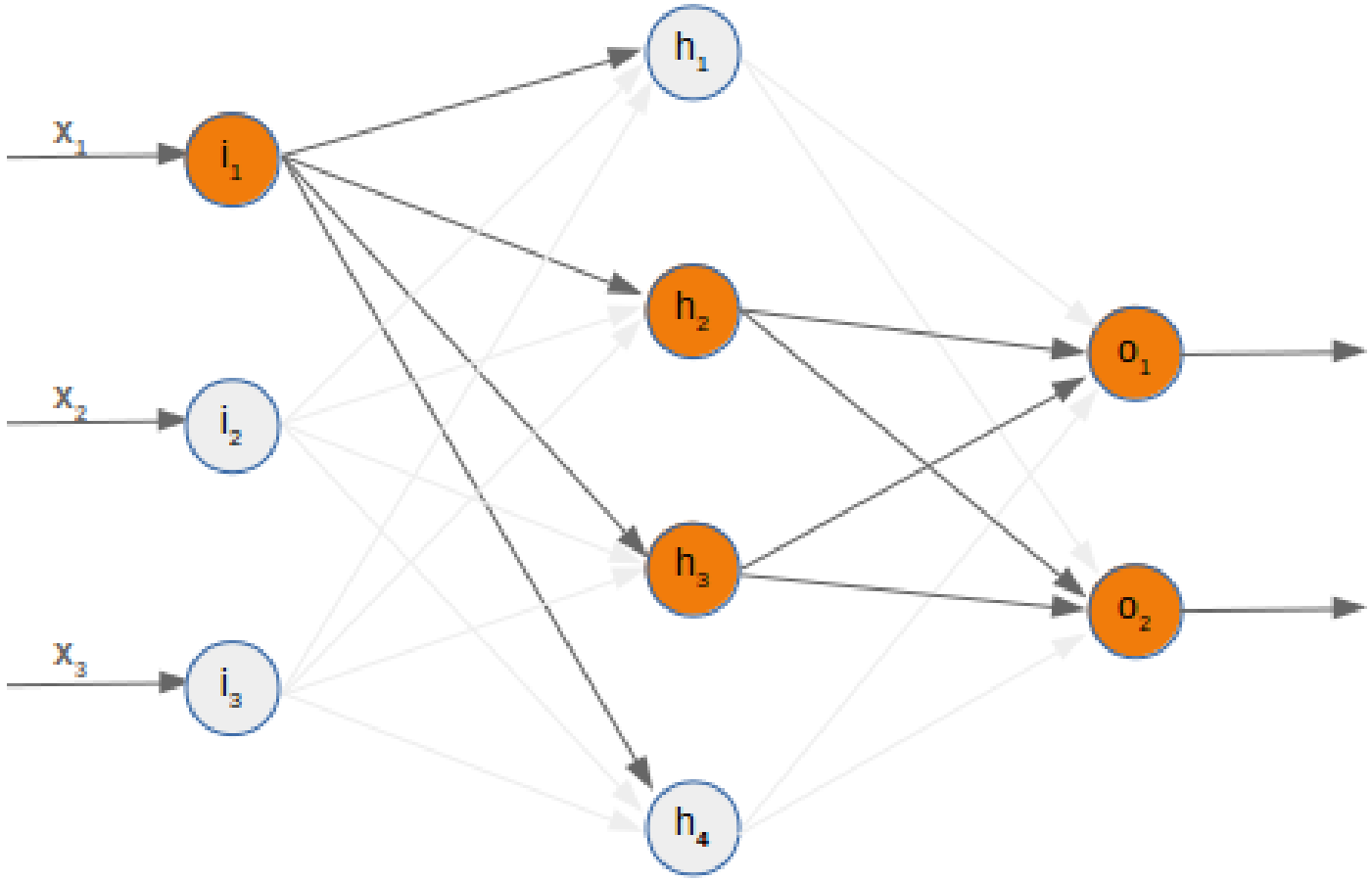
- Training dataset $\rightarrow$ training dataset + validation dataset



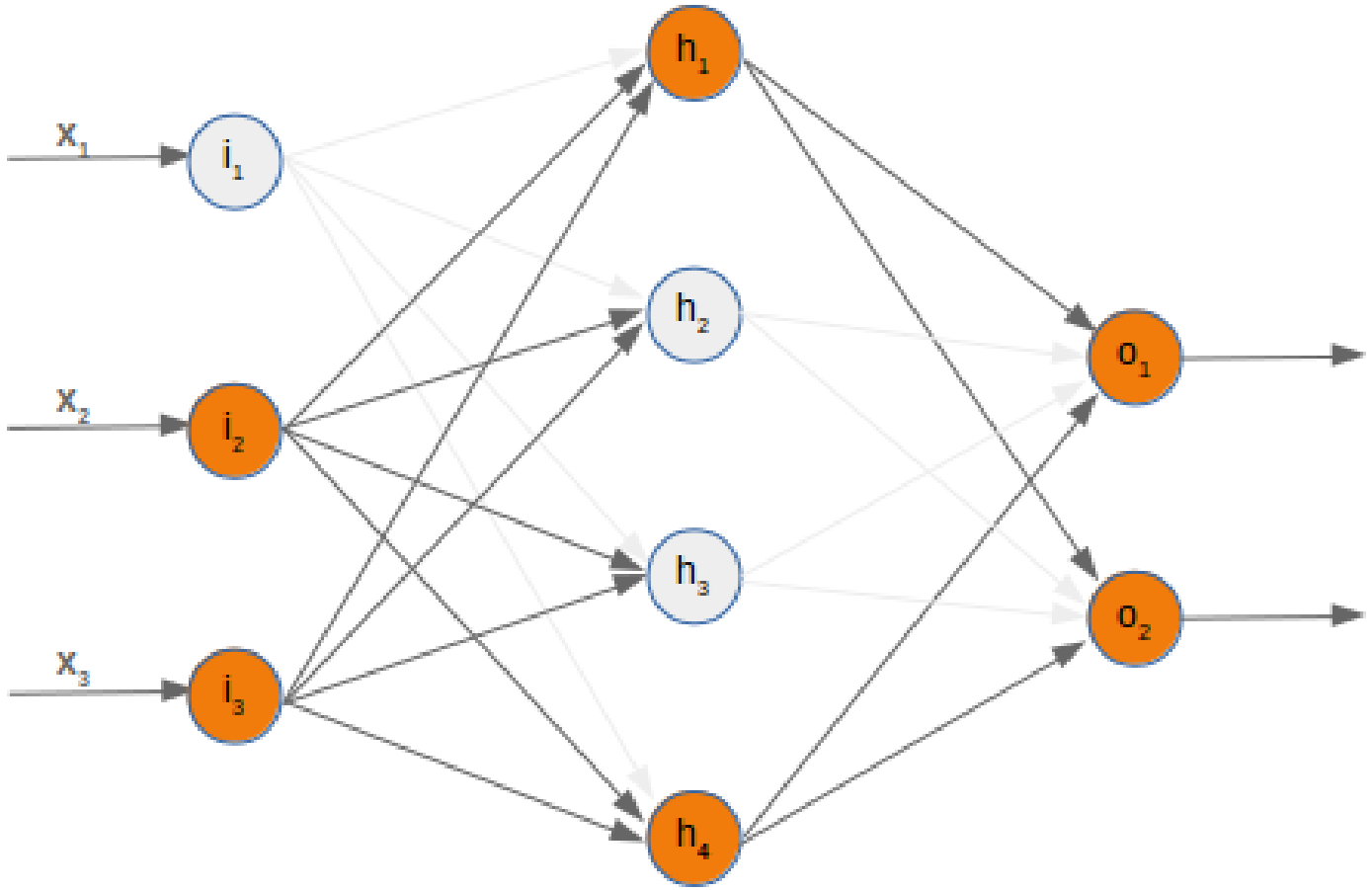
Регуляризация: early stopping – PyTorch

- В PyTorch поддержка отсутствует
- Сторонние библиотеки:
  - `PyTorch Lightning` (ссылка)
  - `ignite` (ссылка1, ссылка2)

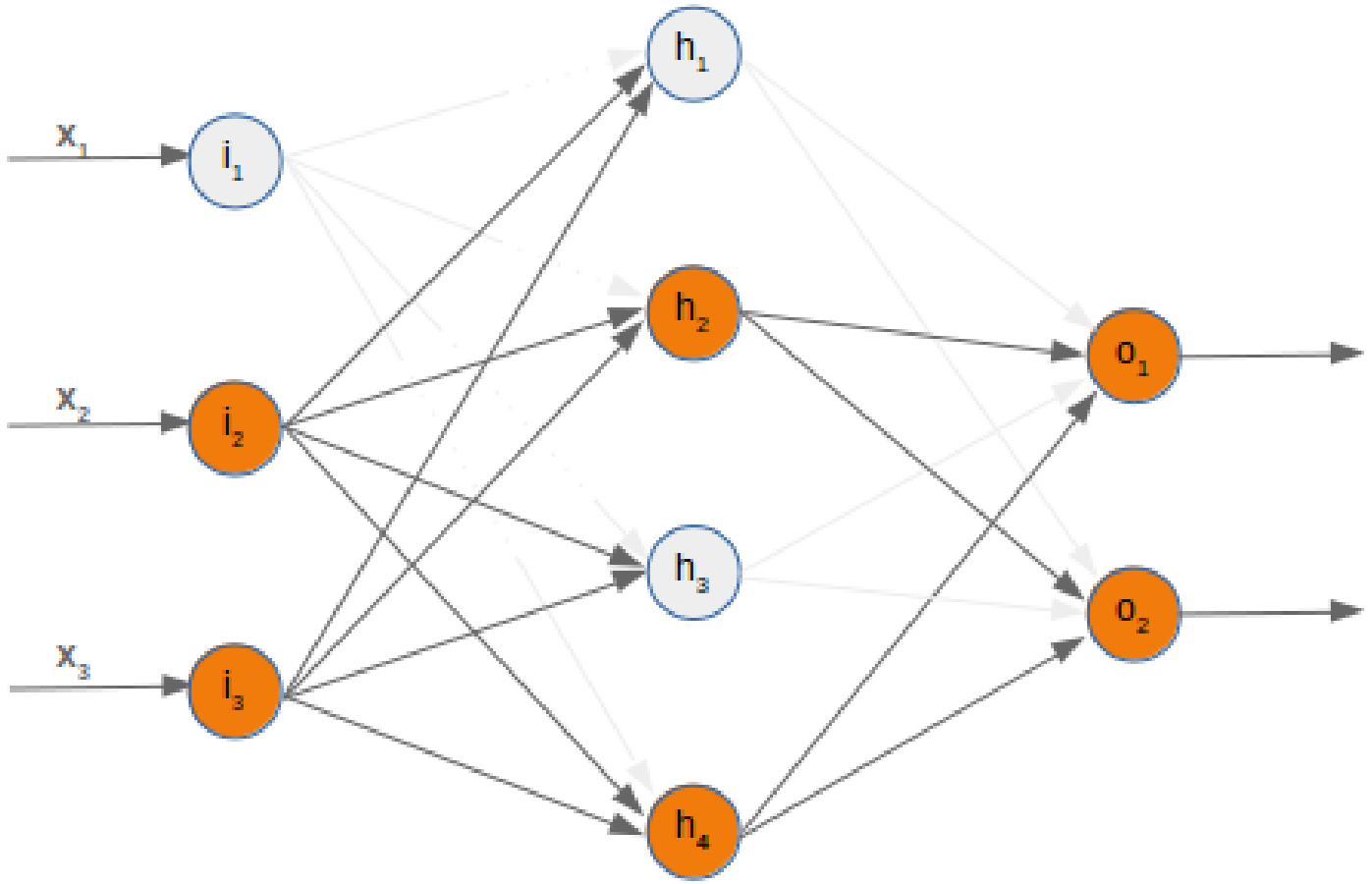
Регуляризация: dropout



Регуляризация: dropout



Регуляризация: dropout

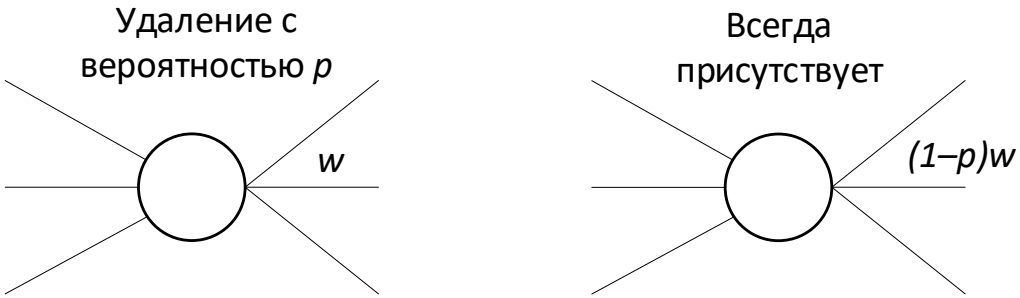


Регуляризация: dropout

- Обучение (training phase): на каждом обучающем примере каждый нейрон (кроме нейронов выходного слоя) исключается из сети с вероятностью p (распределение Бернулли)
  - Для входов тоже можно применять dropout, но есть риск потерять важную информацию
- Предсказание (inference phase): выход каждого нейрона умножается на вероятность $1 - p$
- Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors // arXiv:1207.0580. 2012
  - https://arxiv.org/abs/1207.0580

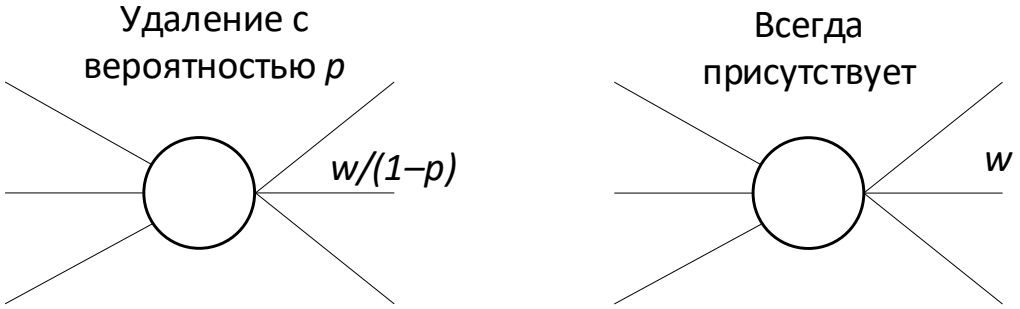
Регуляризация: dropout

Dropout



Обучение Предсказание

Inverted dropout



Обучение Предсказание

Регуляризация: dropout

- Дропаут эквивалентен усреднению 2^N моделей, где N – количество нейронов, для которых применяется дропаут
- Рекомендуемые значения p:
  - входной слой: $[0.1 \dots 0.2]$
  - скрытые слои: $[0.2 \dots 0.5]$
- PyTorch: `torch.nn.Dropout(p=0.5)`

Регуляризация: dropout

```
p = 0.5
```

```
dropout = nn.Dropout(p)
```

```
a = torch.tensor([1.0, 2.0])
```

```
for i in range(10):
    print(dropout(a))
```

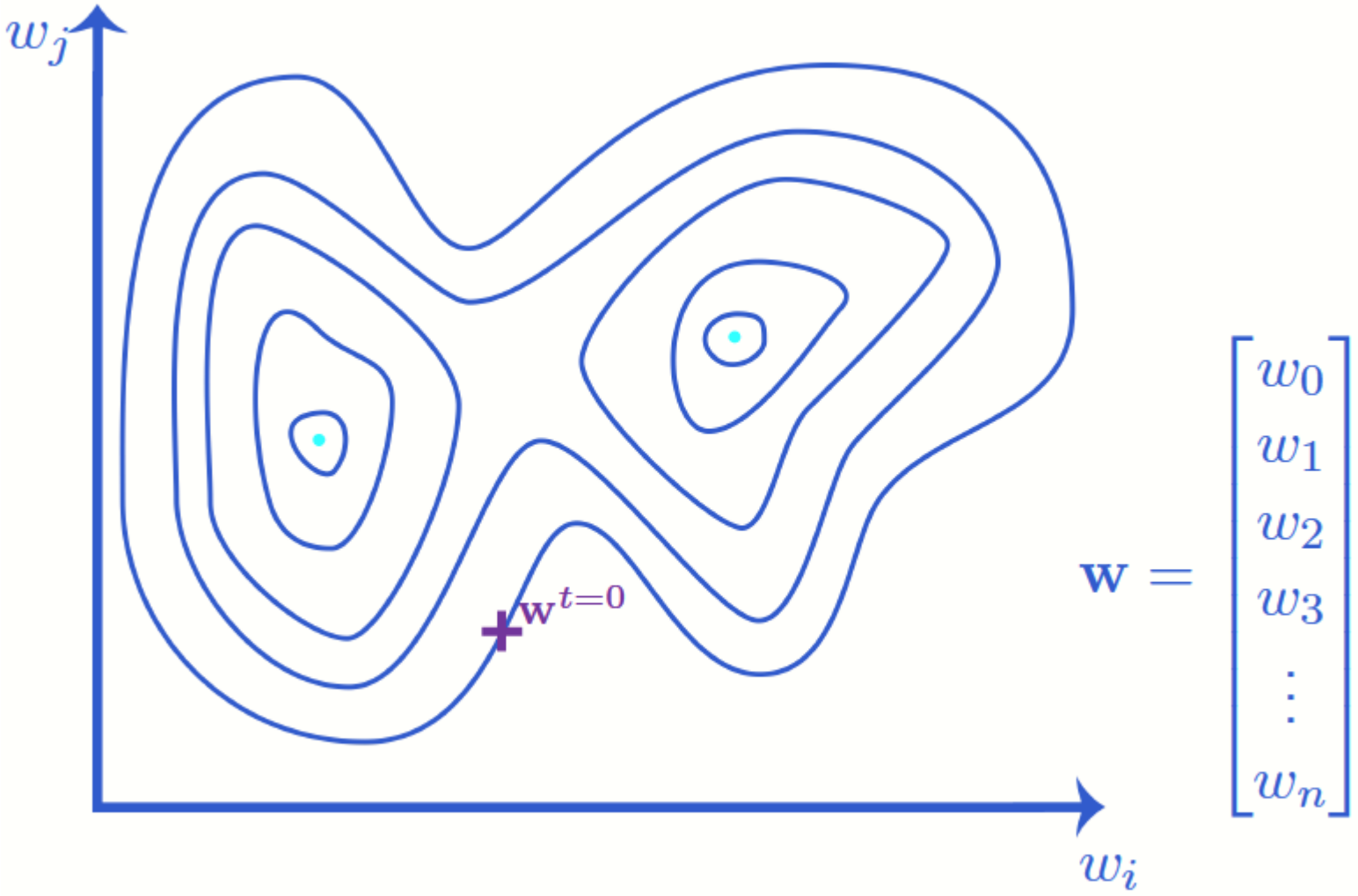
```
tensor([2., 4.])
tensor([0., 4.])
tensor([0., 4.])
tensor([0., 0.])
tensor([2., 0.])
tensor([0., 0.])
tensor([2., 0.])
tensor([0., 4.])
tensor([0., 0.])
tensor([2., 0.])
```

```
dropout.training = False
```

```
for i in range(10):
    print(dropout(a))
```

```
tensor([1., 2.])
tensor([1., 2.])
tensor([1., 2.])
tensor([1., 2.])
tensor([1., 2.])
tensor([1., 2.])
tensor([1., 2.])
tensor([1., 2.])
tensor([1., 2.])
tensor([1., 2.])
```

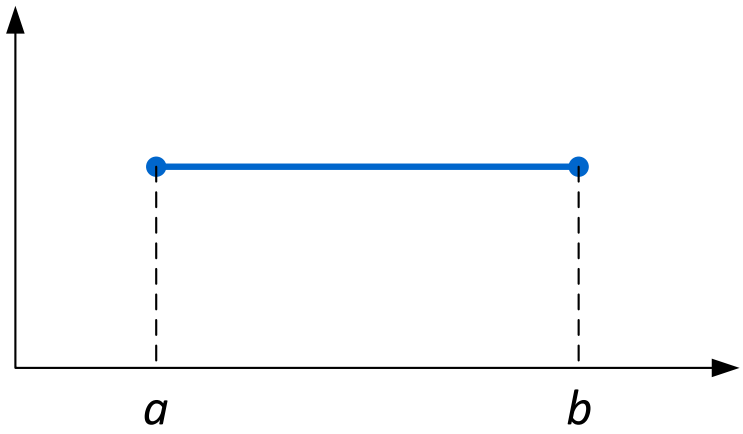
Инициализация весов



Инициализация весов: предобучение

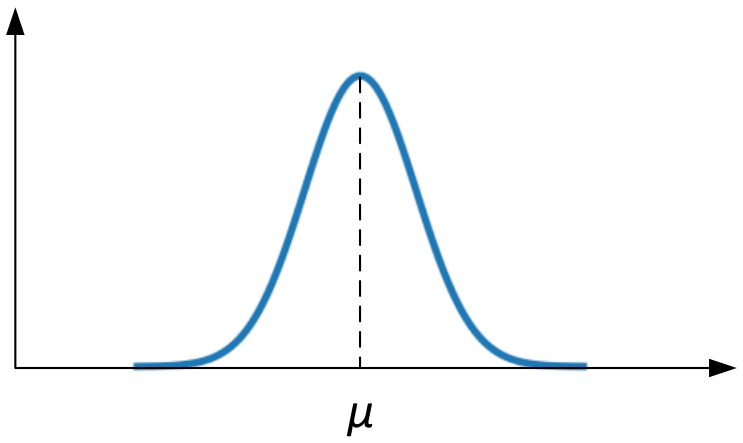
- *Предобучение* (pretraining) – обучение простой модели на простой задаче перед обучением желаемой модели на желаемой задаче
- *Тонкая настройка, дообучение* (fine-tuning) – обучение заранее предобученной модели на желаемой задаче
- *Перенос обучения* (transfer learning) =
  предобучение + тонкая настройка

Инициализация весов: случайная



Равномерное распределение (Uniform):

$$U(a, b)$$



Нормальное распределение (Normal):

$$N(\mu, \sigma^2)$$

μ – математическое ожидание,
σ^2 – дисперсия

Инициализация весов: случайная

- Инициализация Яна Лекуна
  - LeCun Y.A., Bottou L., Orr G.B., Müller K.-R. Efficient BackProp // Neural Networks: Tricks of the Trade. 1998. P. 9–50.
- Один из часто используемых вариантов инициализации:

$$w_i \sim U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right),$$

где m – количество входов некоторого слоя

Инициализация весов: случайная

- Инициализация Ксавье Глоро
  (Xavier initialization, Glorot initialization)
  - Glorot Xavier, Bengio Yoshua. Understanding the Difficulty of Training Deep Feedforward Neural Networks // International Conference on Artificial Intelligence and Statistics. 2010. P. 249–256.

$$w_i \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right),$$

где m – количество входов слоя,
n – количество выходов слоя

- Дисперсия активаций при прямом распространении и дисперсия градиентов при обратном распространении будут одинаковы и примерно равны единице $\rightarrow$ сигналы не затухают

Инициализация весов: случайная

- Инициализация Каймина Хе для несимметричных функций активации (Kaiming initialization, He initialization)
  - He Kaiming, Zhang Xiangyu, Ren Shaoqing, Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification // Proceedings of the IEEE International Conference on Computer Vision (ICCV). 2015 – P. 1026–1034.

$$w_i \sim N\left(0, \frac{2}{m}\right),$$

где m – количество входов слоя, $\frac{2}{m}$ – дисперсия

Инициализация весов: случайная

Рекомендации

- Симметричные функции: инициализация Ксавье Глоро
  - Логистический сигмоид, гиперболический тангенс
  - Xavier initialization, Glorot initialization

$$w_i \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

- Несимметричные функции: инициализация Каймина Хе
  - ReLU, ELU, Leaky ReLU, SeLU
  - Kaiming initialization, He initialization

$$w_i \sim N\left(0, \frac{2}{m}\right)$$

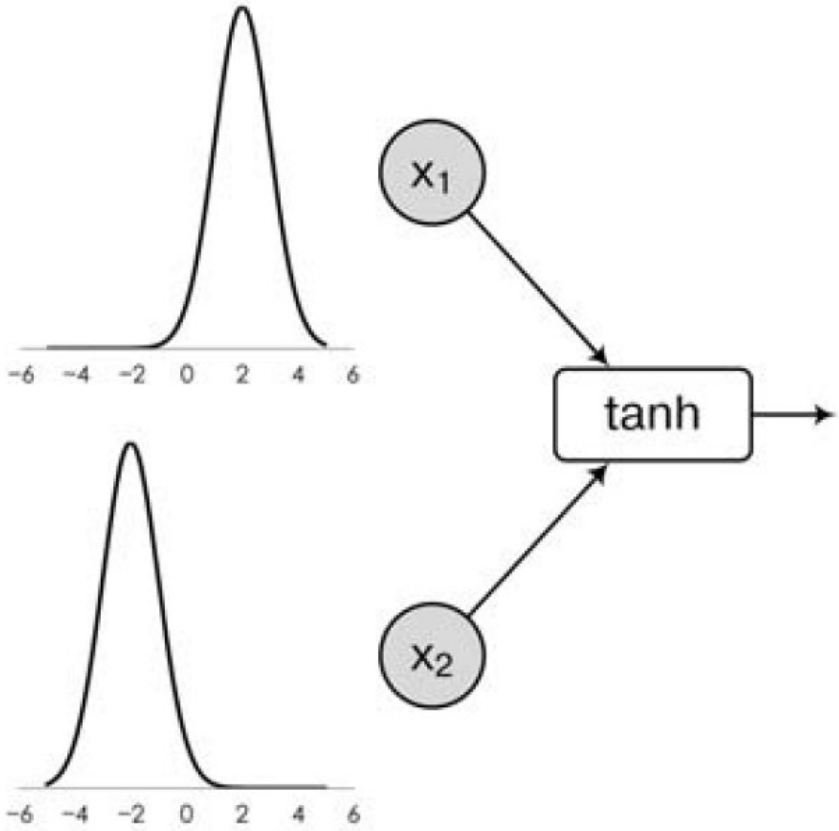
Инициализация весов: случайная

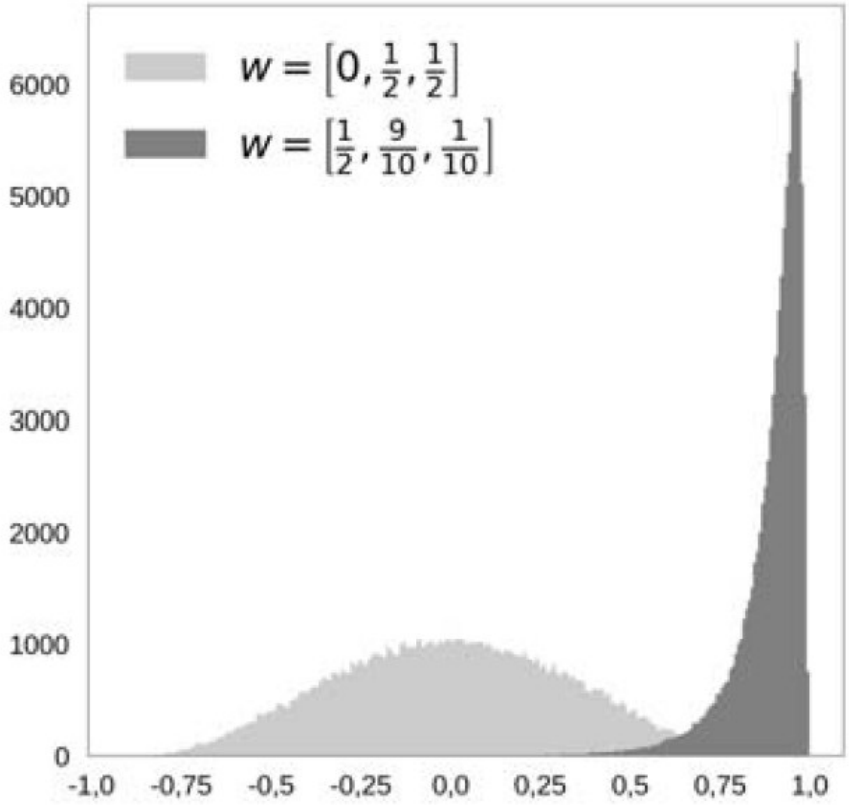
- `torch.nn.init.uniform_(tensor, a=0.0, b=1.0)`
- `torch.nn.init.normal_(tensor, mean=0.0, std=1.0)`
- `torch.nn.init.xavier_uniform_(tensor, gain=1.0)`
- `torch.nn.init.xavier_normal_(tensor, gain=1.0)`
- `torch.nn.init.kaiming_uniform_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu‘)`
- `torch.nn.init.kaiming_normal_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu’)`

- `gain` – опциональный масштабирующий коэффициент
  - `torch.nn.init.calculate_gain(nonlinearity)`

Нормализация по мини-батчам

$$y = \tanh(w_0 + w_1 x_1 + w_2 x_2)$$





Внутренний сдвиг переменных (internal covariance shift)

Нормализация по мини-батчам

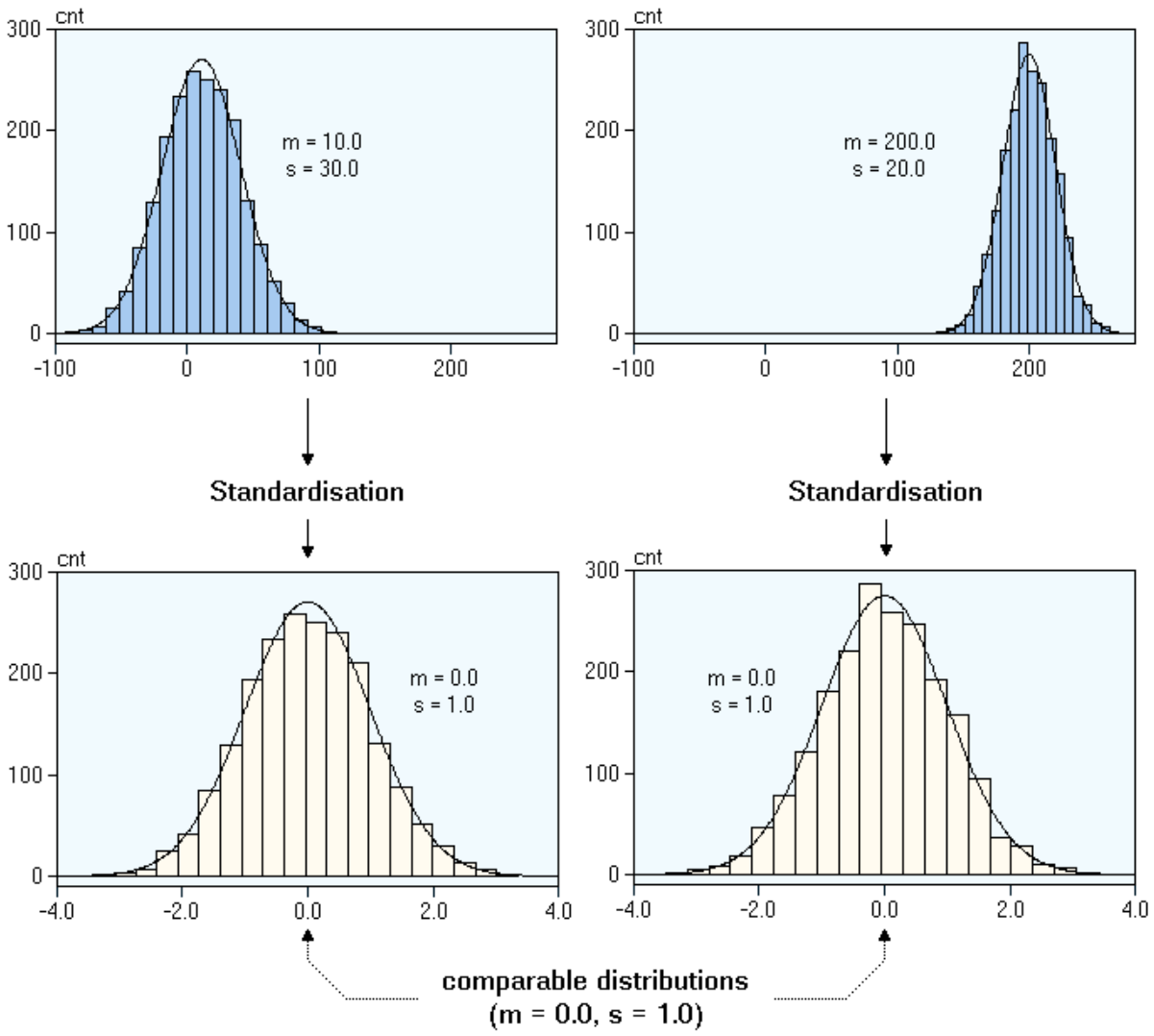
- Standard scaling (Z-score normalization) – мера относительного разброса значения признака, которая показывает, сколько стандартных отклонений составляет его разброс относительного среднего значения признака:

$$z = \frac{x - \mu}{\sigma}, \qquad \mu = M[x] = \frac{1}{N}\sum_{i}^{N} x_i\,, \qquad \sigma = \sqrt{D[x]} = \sqrt{\frac{1}{N}\sum_{i}^{N}(x_i - \mu)^2}$$

  где μ – мат. ожидание признака, σ – стандартное отклонение

- Standard scaling приводит произвольное распределение к распределению со средним значением 0 и дисперсией 1
- «Правило трёх сигм» – с вероятностью 0,9973 значение **нормально** распределённой случайной величины лежит в интервале $(\mu - 3\sigma, \mu + 3\sigma)$

Нормализация по мини-батчам



Нормализация по мини-батчам

- Batch Normalization
  - Ioffe S., Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift // Proc. 32nd ICML, 2015. – P. 448–456.
- Нормализованные входные векторы каждого слоя:

$$\hat{\vec{x}} = (\hat{x}_1, \dots, \hat{x}_d), \qquad \hat{x}_k = \frac{x_k - M[x_k]}{\sqrt{D[x_k]}}$$

  где $M[x_k]$ и $D[x_k]$ вычисляются по мини-батчу

- Проблема: пропадает нелинейность (например, для сигмоиды значения оказываются в линейной области)

Нормализация по мини-батчам

- Решение: дополнительные параметры γ_k и β_k, которые настраиваются в процессе обучения:

$$y_k = \gamma_k \hat{x}_k + \beta_k = \gamma_k \frac{x_k - M[x_k]}{\sqrt{D[x_k]}} + \beta_k$$

- Например, тождественная функция реализуется при:

$$\gamma_k = \sqrt{D[x_k]}, \qquad \beta_k = M[x_k]$$

Нормализация по мини-батчам

Слой нормализации по мини-батчам

- Вход: мини-батч $B = \{\vec{x}_1, \dots, \vec{x}_m\}$
- Вычисление базовых статистик:

$$\mu_B = \frac{1}{m}\sum_i^m \vec{x}_i\,, \qquad \sigma_B^2 = \frac{1}{m}\sum_i^m (x_i - \mu_B)^2$$

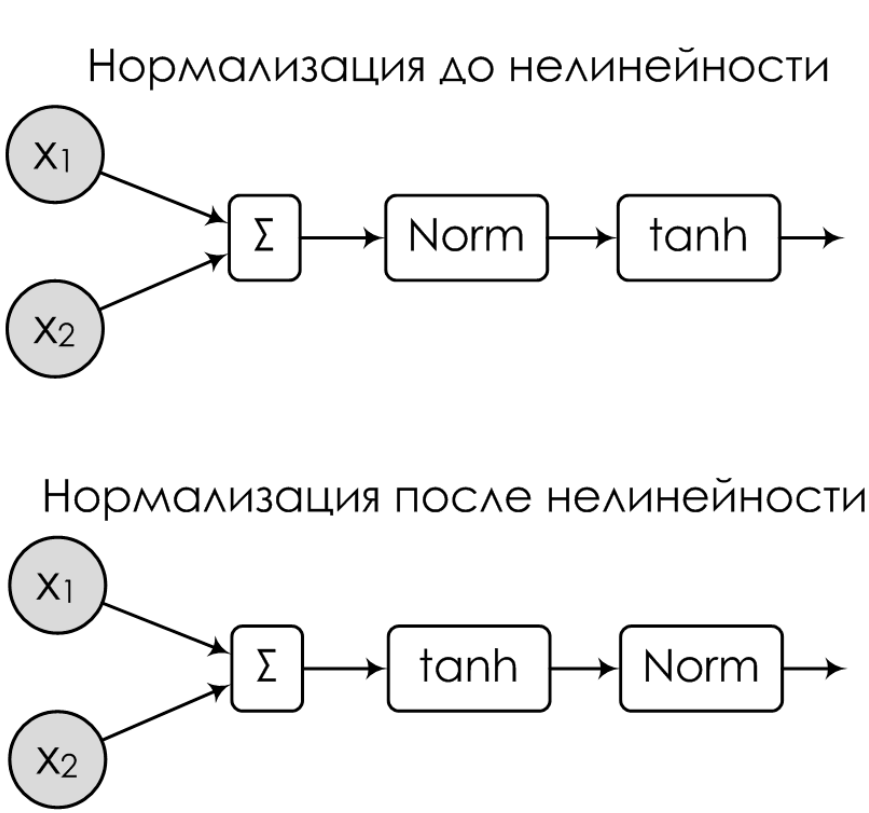
- Нормализация входов:

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

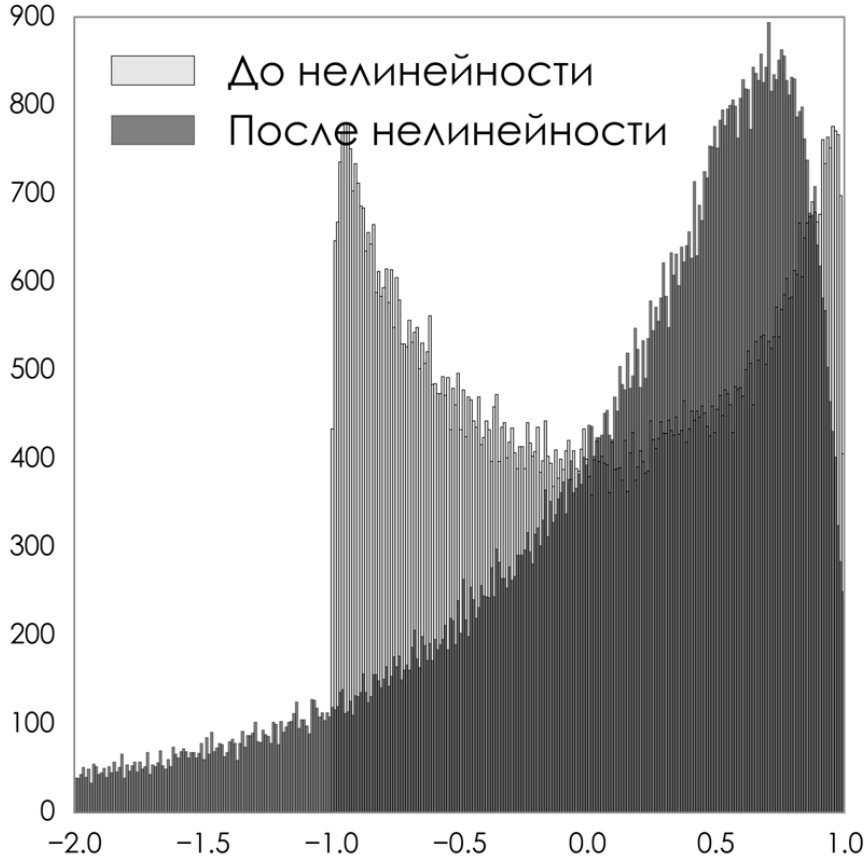
- Вычисление результата:

$$\vec{y}_i = \vec{\gamma}\hat{\vec{x}}_i + \vec{\beta}$$

Нормализация по мини-батчам

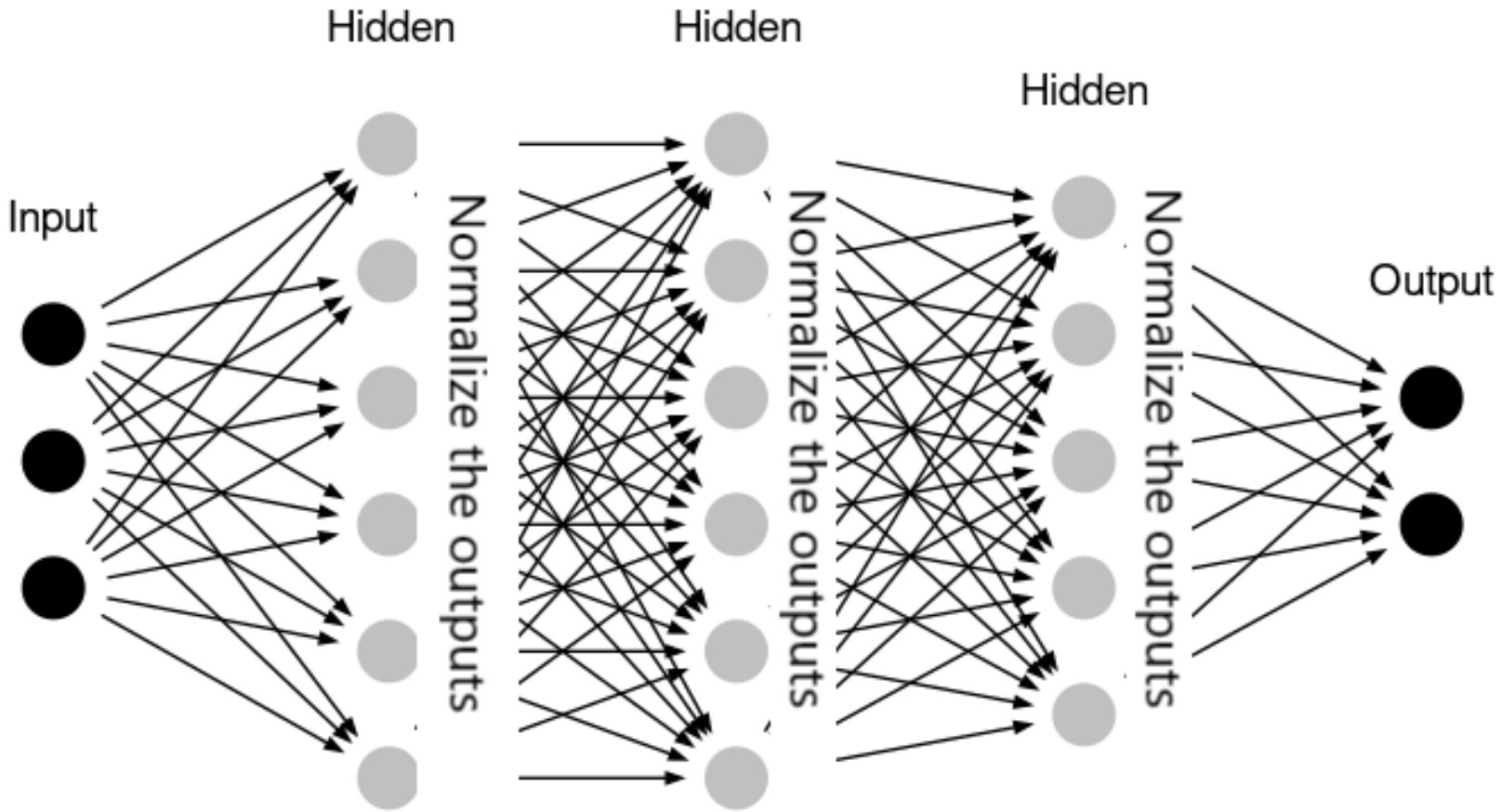


(а)



(б)

Нормализация по мини-батчам

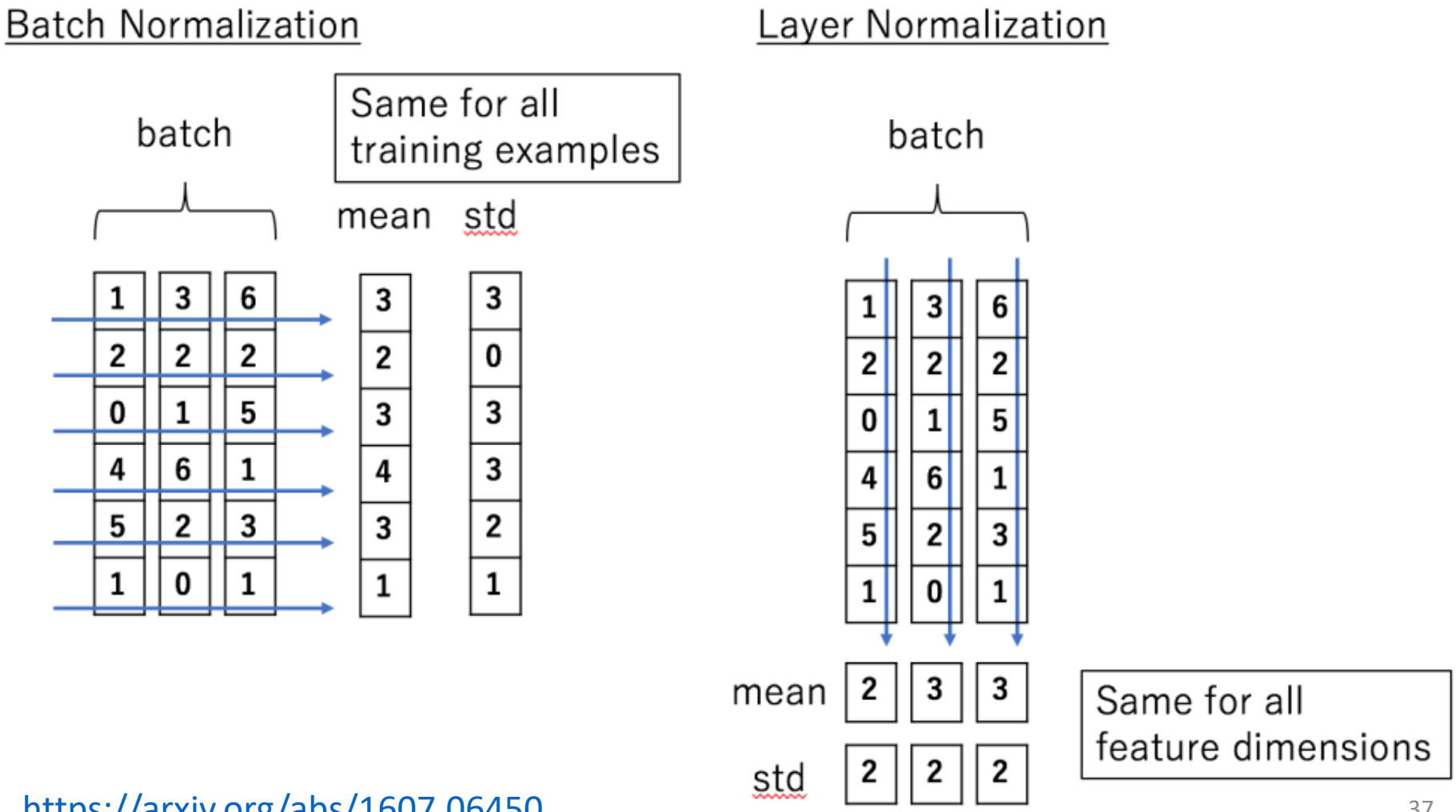


Нормализация по мини-батчам

- PyTorch:

```
torch.nn.BatchNorm1d(
        num_features,
        eps=1e-05,
        momentum=0.1,
        affine=True,
        track_running_stats=True
)
```

Layer Normalization



https://arxiv.org/abs/1607.06450

Layer Normalization

- PyTorch:

```
torch.nn.LayerNorm(
        normalized_shape,
        eps=1e-05,
        elementwise_affine=True
)
```

Градиентный спуск

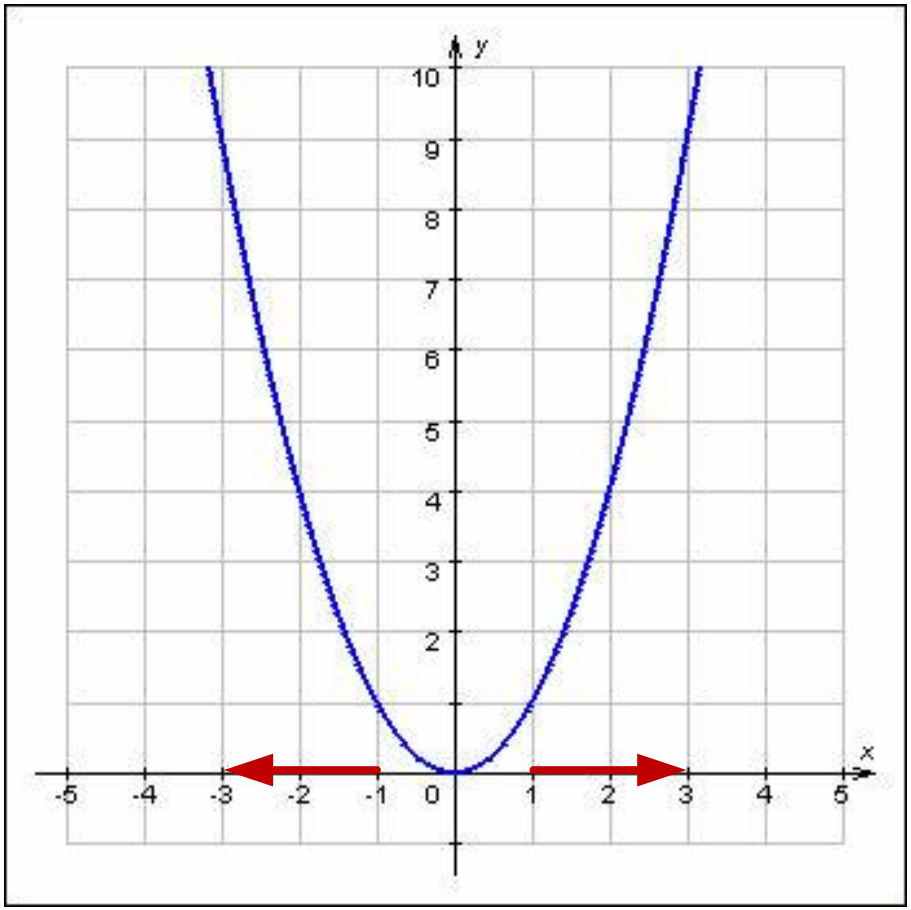
- *Градиентом* функции $f: \mathbb{R}^d \to \mathbb{R}$ называется вектор её частных производных (∇ – оператор набла, оператор Гамильтона):

$$\nabla f(x_1, \dots, x_d) = \left(\frac{\partial f}{\partial x_j}\right)_{j=1}^{d} = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_d}\right)$$

- Градиент является направлением наискорейшего роста функции, а *антиградиент* ($-\nabla f$) – направлением наискорейшего убывания

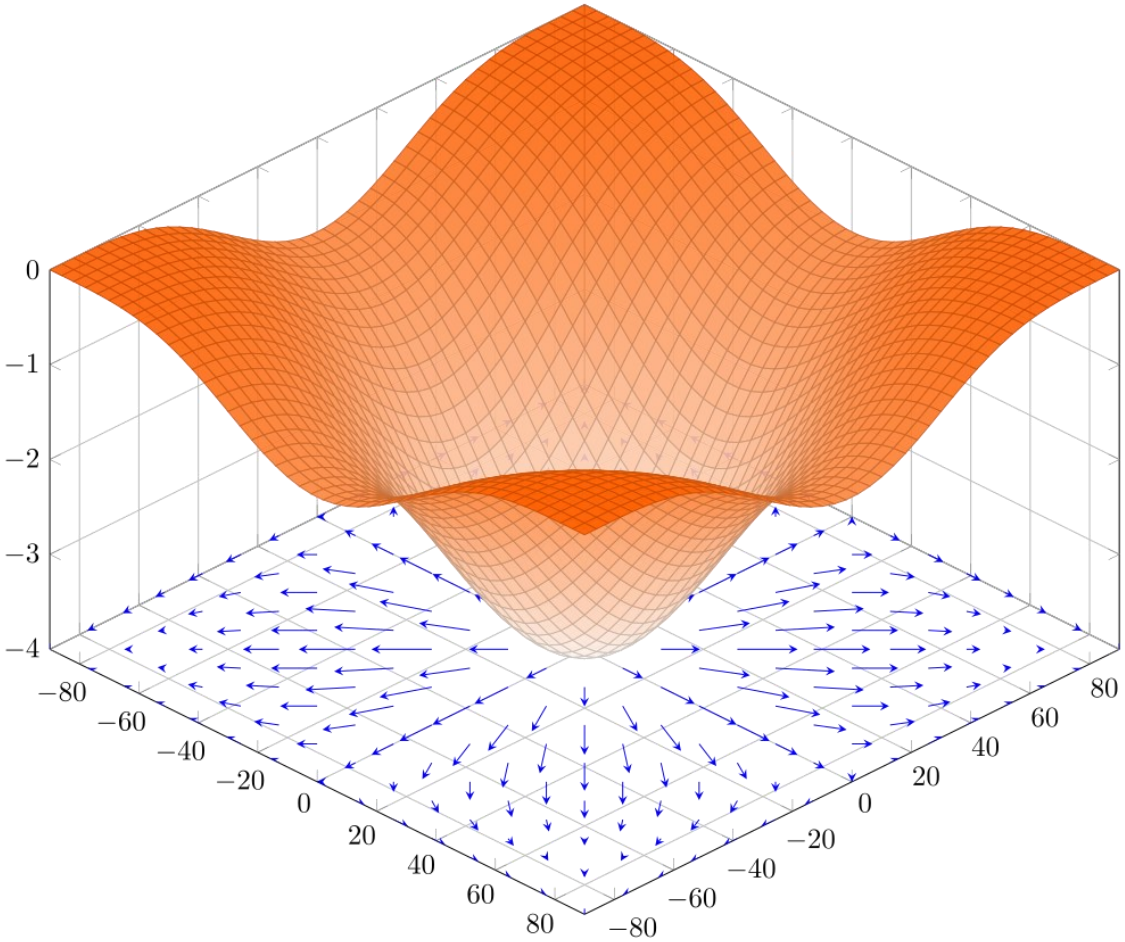
Градиентный спуск

- *Пример*: функция $y = x^2$, производная $\frac{dy}{dx} = 2x$



Градиентный спуск

- *Пример*: функция $y = -(\cos^2 x_1 + \cos^2 x_2)^2$



Градиентный спуск

Алгоритм градиентного спуска:

1. Выбрать начальную точку $\vec{w}^{(0)}$
2. Повторять до сходимости:

$$\vec{w}^{(k)} = \vec{w}^{(k-1)} - \eta_k \nabla Q\left(\vec{w}^{(k-1)}\right),$$

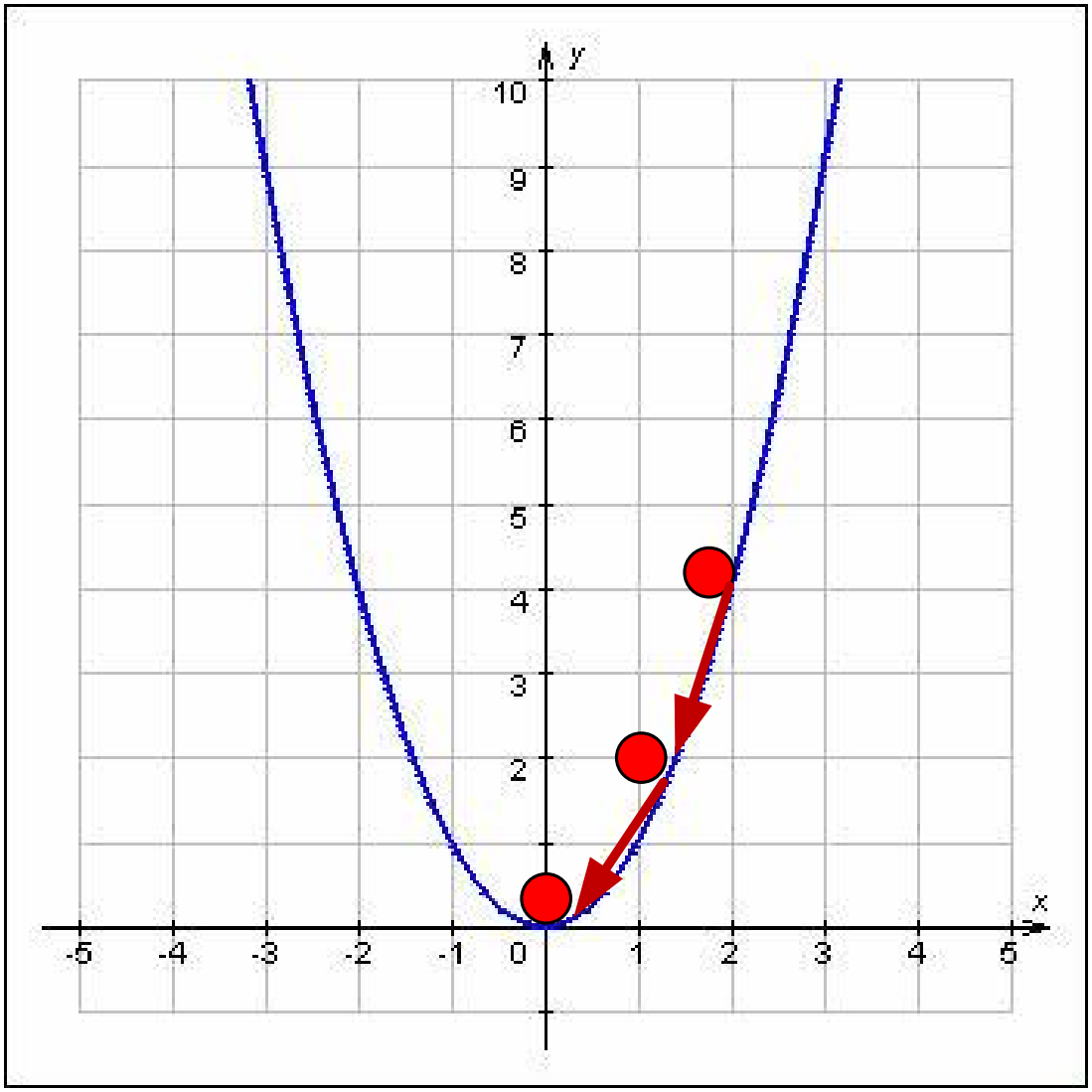
где k – номер шага,

$Q(\vec{w})$ – функционал ошибки для набора параметров $\vec{w}$,

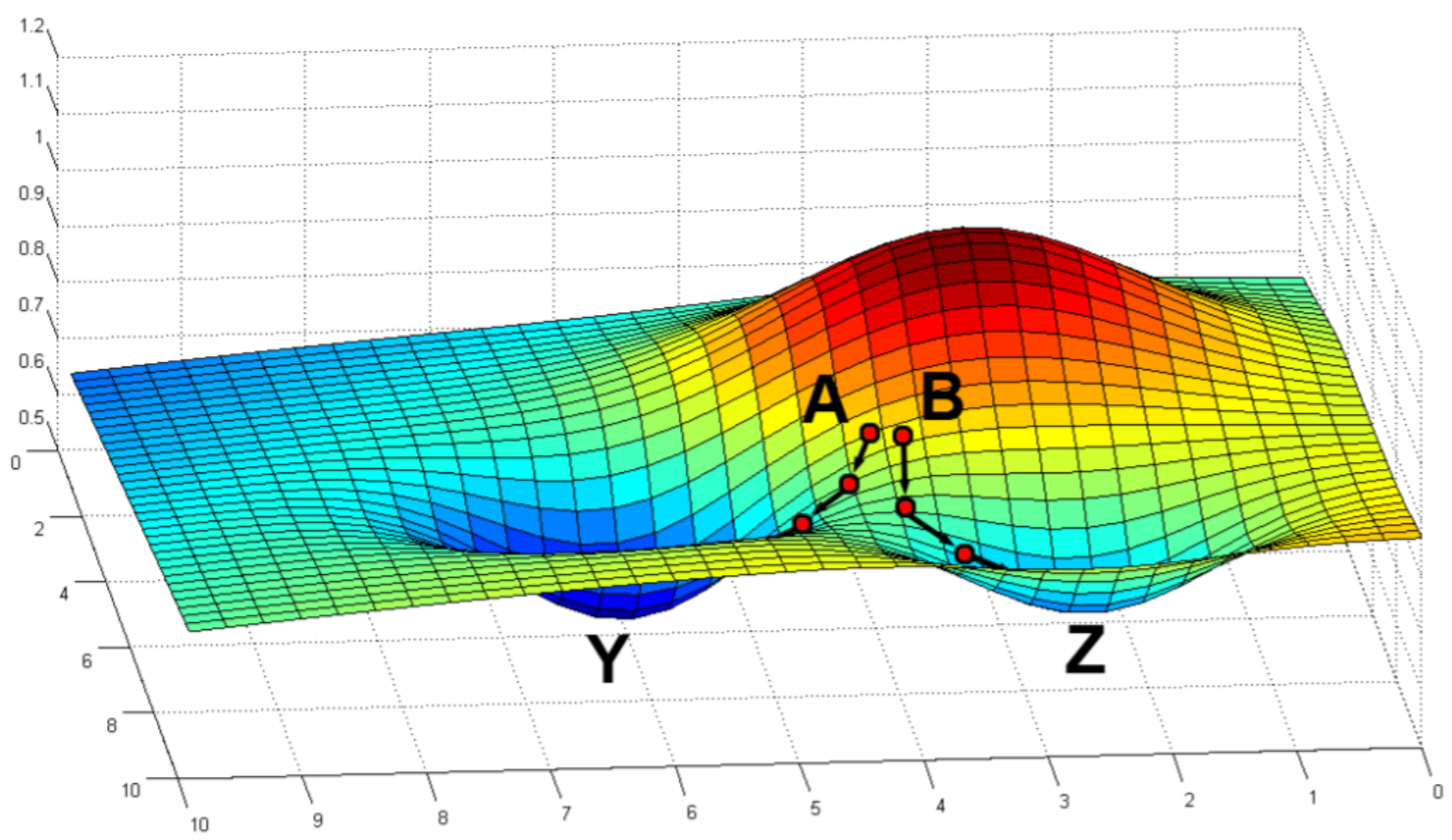
η_k – скорость спуска (длина k-го шага).

- Условия останова:
  - вектор весов почти перестает изменяться
  - достигнуто максимальное число итераций

Градиентный спуск

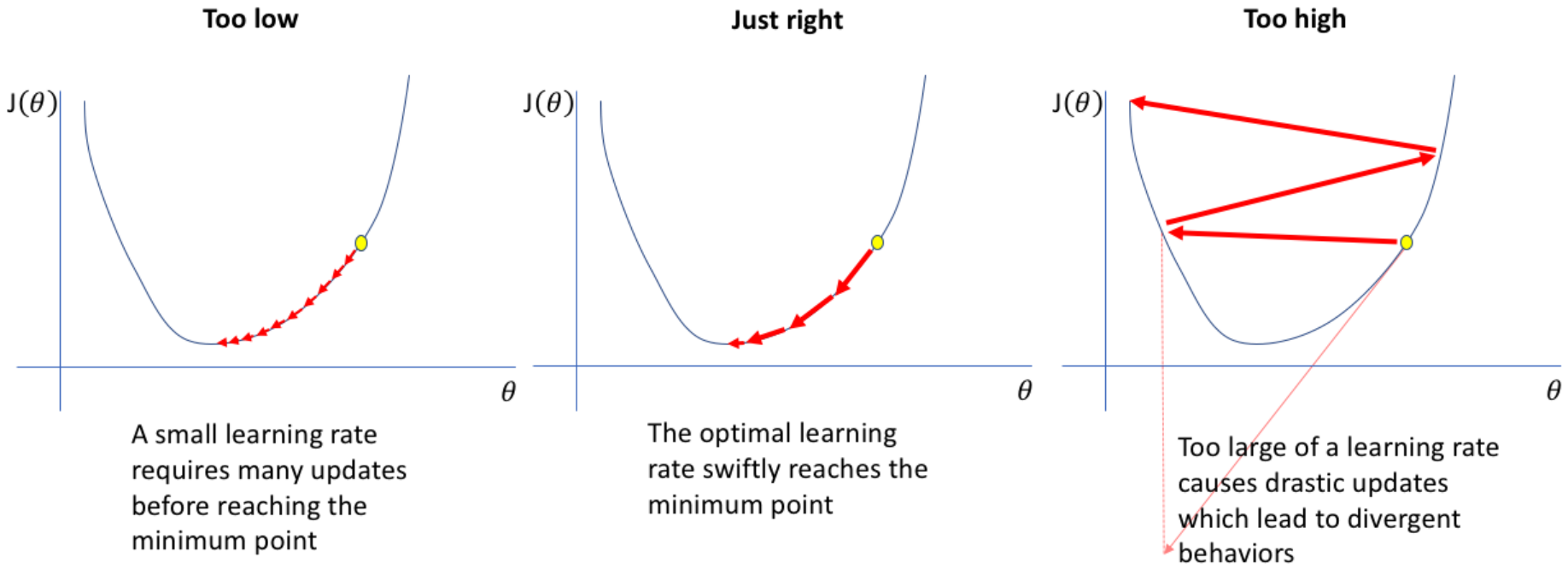


Градиентный спуск



Градиентный спуск: скорость спуска

- Скорость спуска:
  - слишком высокая → переход через минимум
  - слишком низкая → медленная сходимость



Градиентный спуск: скорость спуска

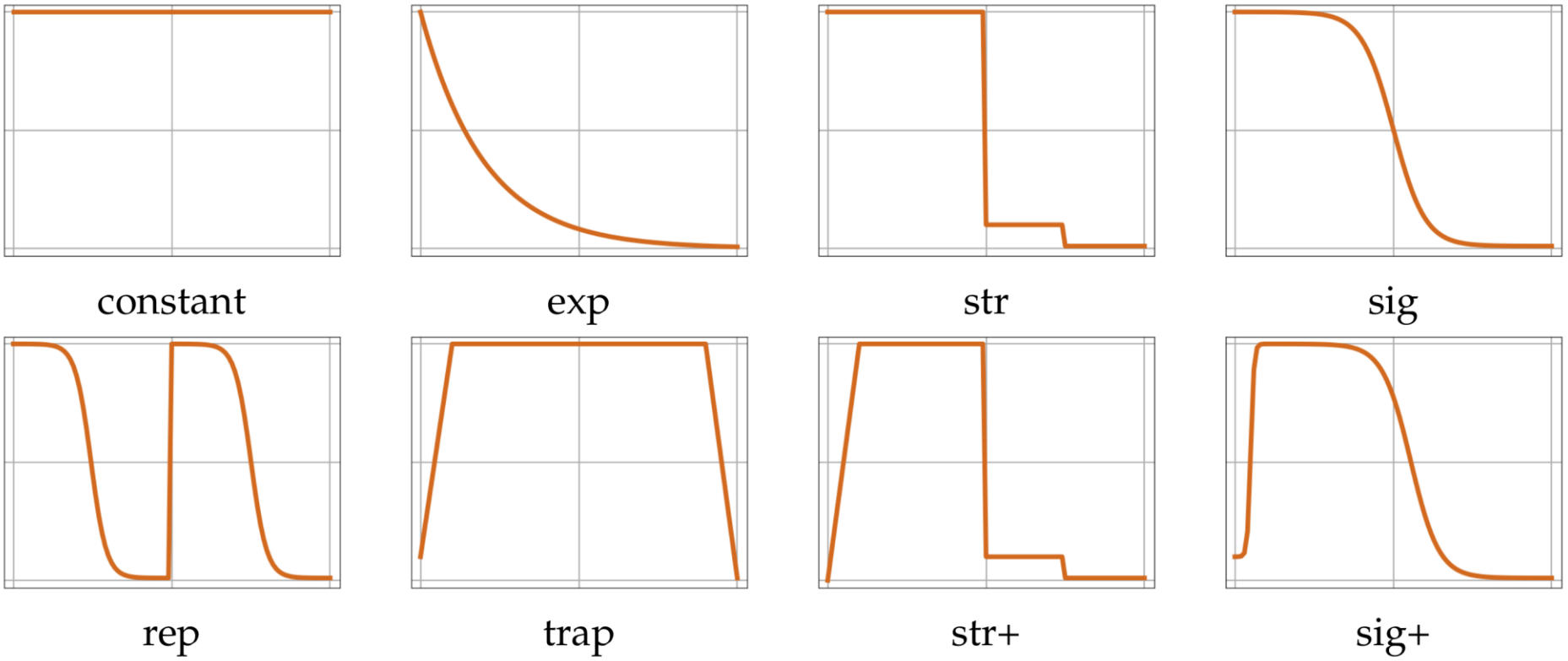
Варианты изменения скорости спуска:

- константная: $\eta_k = const$
- затухание (learning rate decay/annealing):
  - линейное затухание (linear decay): $\eta_k = \eta_0 \left(1 - \frac{k}{K}\right)$
  - экспоненциальное затухание (exponential decay):

$$\eta_k = \eta_0 e^{-\frac{k}{K}}$$

- повышение (learning rate warmup)

Градиентный спуск: скорость спуска



https://www.mdpi.com/2079-9292/10/16/2029/htm

Градиентный спуск: скорость спуска

- PyTorch ("How to adjust learning rate"):

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = ExponentialLR(optimizer, gamma=0.9)

for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step()
```

Стохастический градиентный спуск

- Функционал ошибки представим в виде суммы l функций ошибок:

$$Q(\vec{w}) = \frac{1}{l}\sum_{i=1}^{l} L_i(\vec{w})$$

- При градиентном спуске необходимо вычислять градиент всей суммы:

$$\nabla Q(\vec{w}) = \frac{1}{l}\sum_{i=1}^{l} \nabla L_i(\vec{w})$$

- Если выборка большая, вычисление градиента трудоемко

Стохастический градиентный спуск

- Оценить градиент суммы функций можно градиентом одного случайно взятого i_k слагаемого:

$$\nabla Q(\vec{w}) \approx \nabla L_{i_k}(\vec{w})$$

- Метод стохастического градиентного спуска (stochastic gradient descent, SGD):

$$\vec{w}^{(k)} = \vec{w}^{(k-1)} - \eta_k \nabla L_{i_k}\left(\vec{w}^{(k-1)}\right)$$

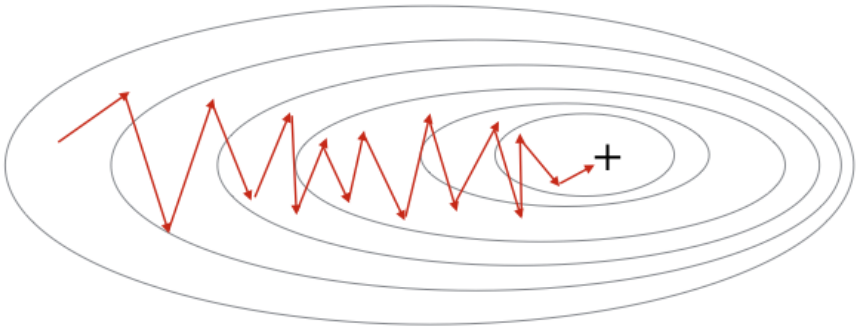
- Градиентный спуск по мини-батчам (mini-batch gradient descent):

$$\nabla Q(\vec{w}) \approx \frac{1}{n}\sum_{j=1}^{n} \nabla L_{i_{kj}}(\vec{w}),$$

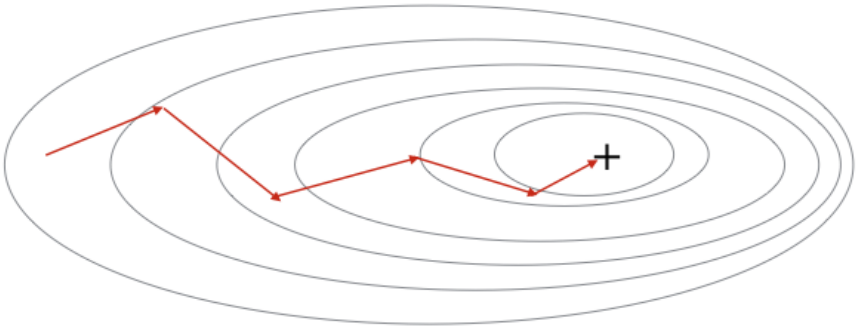
где i_{kj} – случайно выбранные номера слагаемых

Стохастический градиентный спуск

Stochastic Gradient Descent



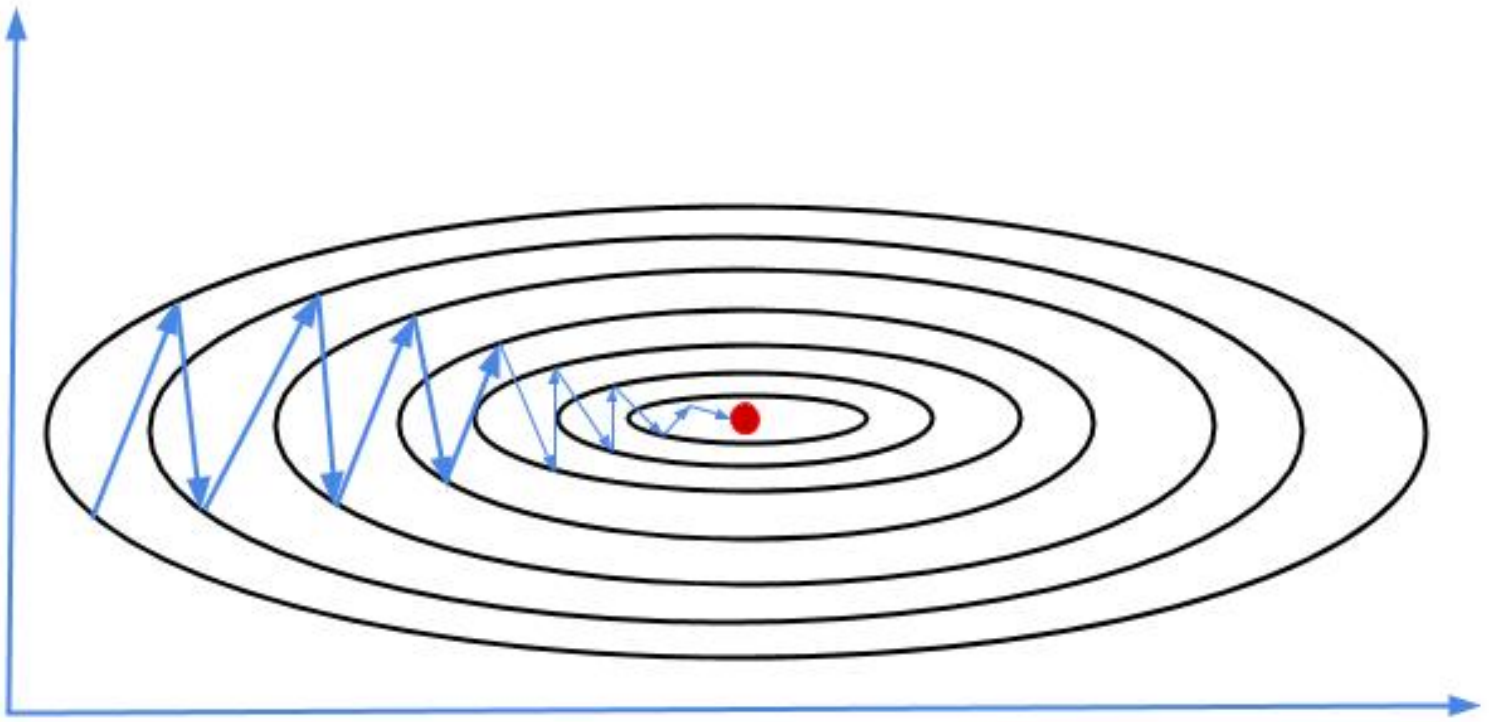
Mini-Batch Gradient Descent



Модификации градиентного спуска

Метод моментов

- Направление антиградиента может меняться на каждом шаге:



Модификации градиентного спуска

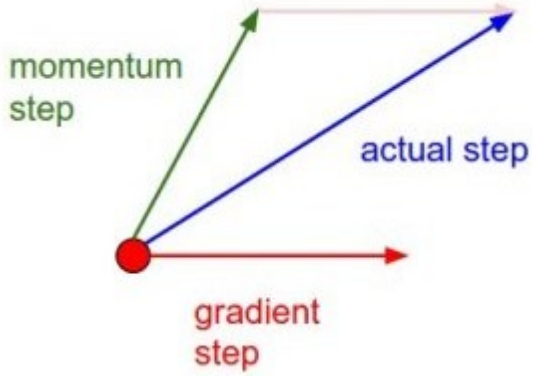
Метод моментов

- Можно усреднять векторы антиградиента с нескольких предыдущих шагов с помощью вектора инерции:

$$\vec{h}_0 = 0,$$

$$\vec{h}_k = \alpha\vec{h}_{k-1} + \eta_k \nabla Q\left(\vec{w}^{(k-1)}\right),$$

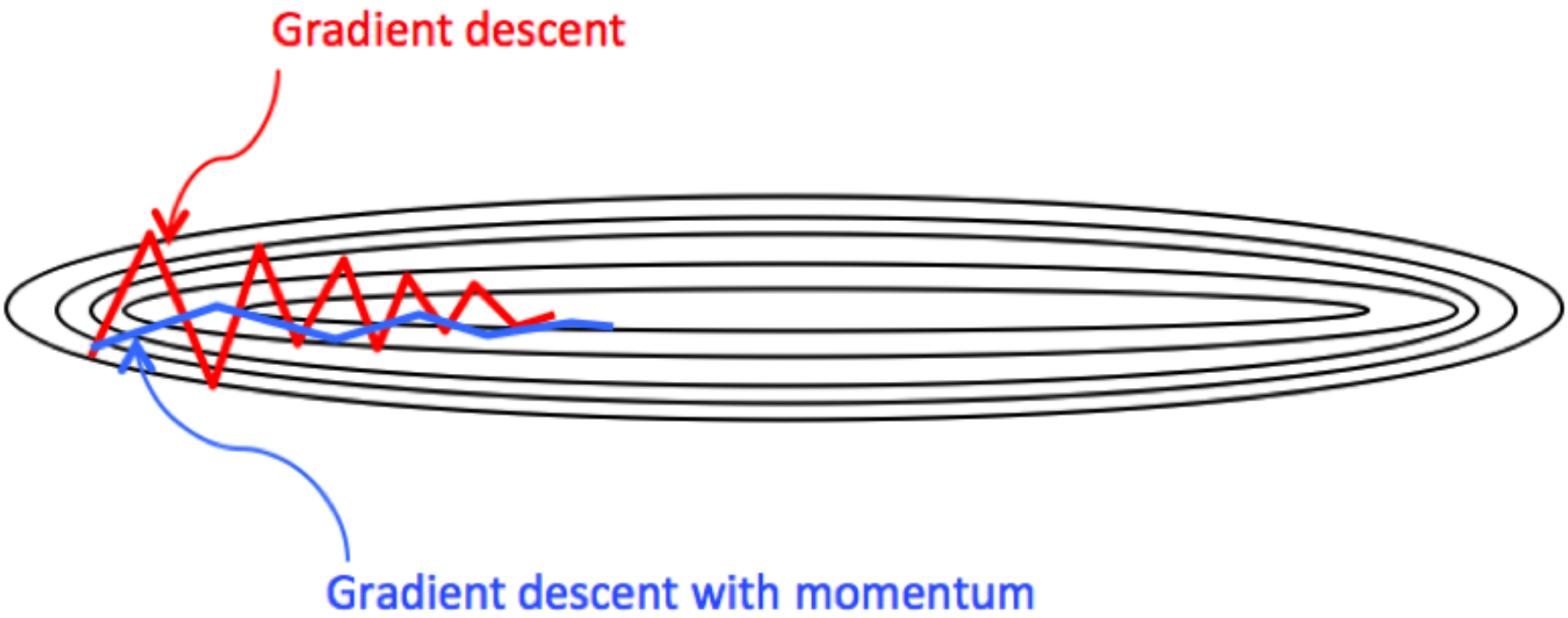
где α – коэффициент момента.



- Тогда обновление весов:

$$\vec{w}^{(k)} = \vec{w}^{(k-1)} - \vec{h}_k$$

Модификации градиентного спуска

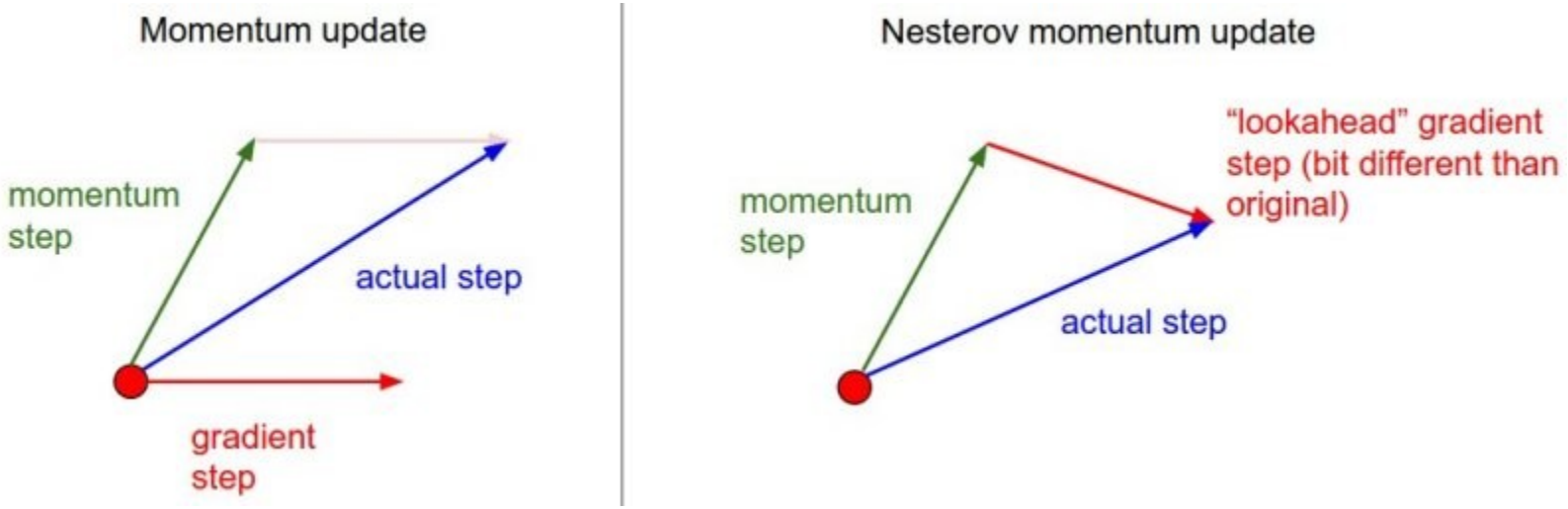


Модификации градиентного спуска

Метод Нестерова

- Можно вычислять градиент сразу в промежуточной точке:

$$\vec{w}^{(k)} = \vec{w}^{(k-1)} - \alpha\vec{h}_{k-1} - \eta_k \nabla Q\left(\vec{w}^{(k-1)} - \alpha\vec{h}_{k-1}\right)$$



Модификации градиентного спуска

- PyTorch:

```
torch.optim.SGD(
        params,
        lr=<required parameter>,
        momentum=0,
        weight_decay=0,
        nesterov=False
)
```

Модификации градиентного спуска

Метод AdaGrad

- Разное изменение скорости для разных компонентов вектора весов:

$$\nabla Q\left(\vec{w}^{(k-1)}\right) = \left(\mathrm{g}_1^{k-1}, \ldots, \mathrm{g}_d^{k-1}\right)$$

$$G_{kj} = G_{k-1,j} + \left(\mathrm{g}_j^{k-1}\right)^2$$

$$w_j^{(k)} = w_j^{(k-1)} - \frac{\eta_k}{\sqrt{G_{kj} + \varepsilon}} \mathrm{g}_j^{k-1}$$

- PyTorch: $\eta = 0.01$, $\varepsilon = 10^{-10}$

Модификации градиентного спуска

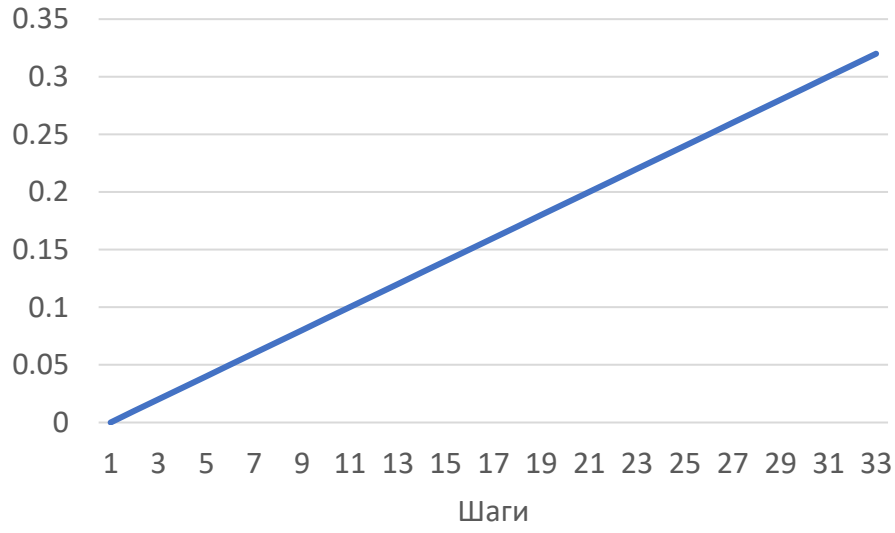
Метод AdaGrad

- Проблема: переменная G_{kj} монотонно растёт, из-за чего шаги становятся всё медленнее и могут остановиться ещё до того, как достигнут минимум функционала

$$\nabla Q\left(\vec{w}^{(k-1)}\right) = \left(\mathrm{g}_1^{k-1}, \ldots, \mathrm{g}_d^{k-1}\right)$$

$$G_{kj} = G_{k-1,j} + \left(\mathrm{g}_j^{k-1}\right)^2$$

$$w_j^{(k)} = w_j^{(k-1)} - \frac{\eta_k}{\sqrt{G_{kj} + \varepsilon}} \mathrm{g}_j^{k-1}$$



Модификации градиентного спуска

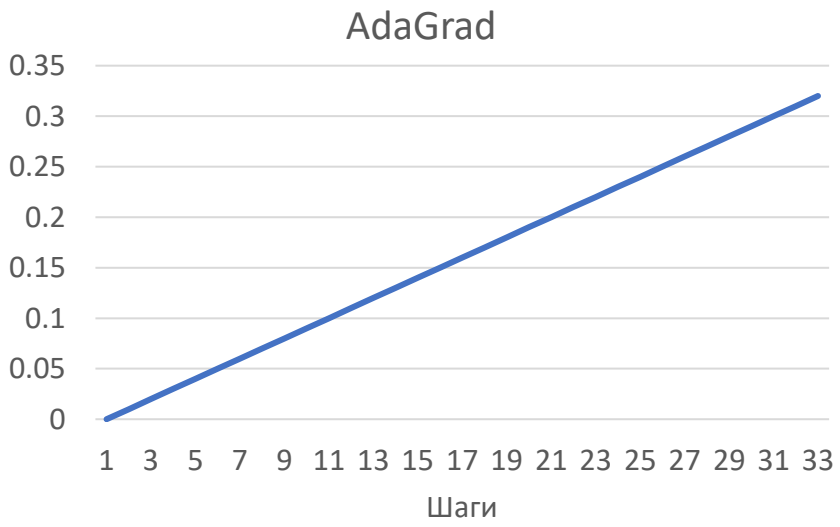
Метод AdaDelta

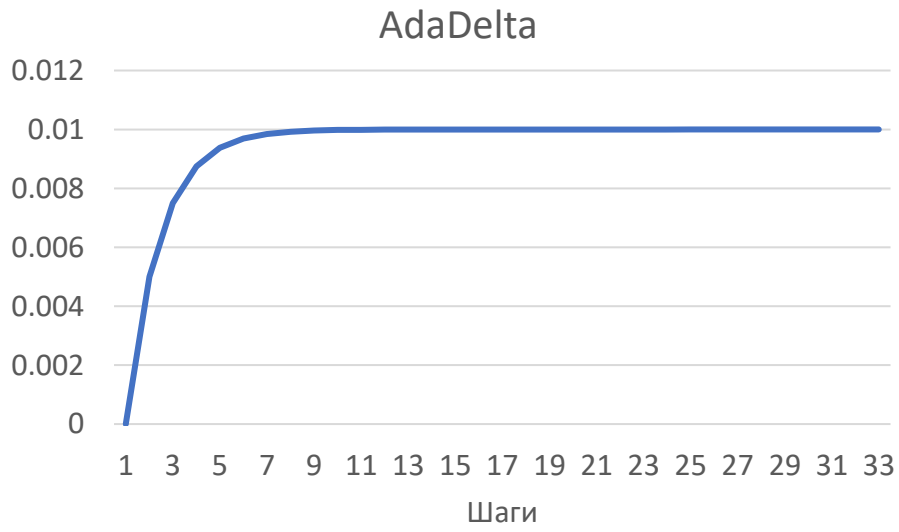
- Идея 1: используется экспоненциальное среднее градиентов:

$$G_{kj} = \rho G_{k-1,j} + (1-\rho)\left(\mathrm{g}_j^{k-1}\right)^2,$$

$$w_j^{(k)} = w_j^{(k-1)} - \frac{\eta_k}{\sqrt{G_{kj}+\varepsilon}}\mathrm{g}_j^{k-1}$$

где $\rho \in [0,1]$ – сглаживающая константа; PyTorch: $\rho = 0.9$





Модификации градиентного спуска

Метод AdaDelta

- Проблема: размерности при обновлении весов не совпадают:

$$w_j^{(k)} = w_j^{(k-1)} - \eta_k \frac{\partial Q}{\partial w_j}$$

- Идея 2: выравнивание размерностей:

$$\Delta w_j^{(k)} = \frac{\sqrt{E\left[\Delta w_j^2\right]_{k-1}}}{\sqrt{G_{kj} + \varepsilon}} \mathrm{g}_j^{(k-1)}$$

$$w_j^{(k)} = w_j^{(k-1)} - \Delta w_j^{(k)}$$

$$E\left[\Delta w_j^2\right]_k = \rho E\left[\Delta w_j^2\right]_{k-1} + (1-\rho)\left(\Delta w_j^{(k)}\right)^2$$

Модификации градиентного спуска

Метод RMSprop

- Предложен Джеффри Хинтоном независимо от AdaDelta
- Использует идею о квадратном корне из экспоненциального среднего квадратов градиентов (RMS – Root Mean Squares – среднее квадратическое):

$$G_{kj} = \alpha G_{k-1,j} + (1-\alpha)\left(\mathrm{g}_j^{k-1}\right)^2$$

$$w_j^{(k)} = w_j^{(k-1)} - \frac{\eta_k}{\sqrt{G_{kj} + \varepsilon}}\mathrm{g}_j^{k-1}$$

- PyTorch: $\eta = 0.01$, $\alpha = 0.99$

Модификации градиентного спуска

Метод Adam (adaptive moment estimation)

- Использует сглаженные версии среднего и среднеквадратичного градиентов:

$$m_k = \beta_1 m_{k-1} + (1 - \beta_1) g_k$$

$$v_k = \beta_2 v_{k-1} + (1 - \beta_2) g_k^2$$

$$w^{(k)} = w^{(k-1)} - \frac{\eta_k}{\sqrt{v_k + \epsilon}} m_k$$

- PyTorch: $\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$

Модификации градиентного спуска

Метод AdamW

- Добавление регуляризации в Adam
  (Adam with weight decay)

$$w^{(k)} = w^{(k-1)} - \eta_k \left(\frac{m_k}{\sqrt{v_k + \epsilon}} + \omega w^{(k-1)} \right)$$

- PyTorch: $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\omega = 0.01$