

# Нейронные сети

## Продолжение

Повторение



# Однослойная нейронная сеть

## Матричное представление

Пусть  $x = (x_1, x_2, \dots, x_d)$  — элемент выборки.

$s = (s_1, s_2, \dots, s_H)$  — выходы нейронов до применения функции активации.

$y = (y_1, y_2, \dots, y_H)$  — выход слоя.

Тогда работу слоя можно описать операциями:

$$1) s = (x_1, x_2, \dots, x_d) \times \begin{pmatrix} w_{11} & \dots & w_{1h} & \dots & w_{1H} \\ w_{21} & \dots & w_{2h} & \dots & w_{2H} \\ \dots & \dots & \dots & \dots & \dots \\ w_{d1} & \dots & w_{dh} & \dots & w_{dH} \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_H \end{pmatrix}^T$$
$$= x^T \cdot W + b^T$$

$$2) y = (y_1, y_2, \dots, y_H) = (\sigma(s_1), \sigma(s_2), \dots, \sigma(s_H)) = \sigma(s) = \boxed{\sigma(x^T W + b^T)}$$



# Функции активации

## Зачем нужны функции активации?

- Для того, чтобы делать нелинейные преобразования.
- Кроме того, по теореме Цыбенко, используя функции активации типа сигмолды мы можем аппроксимировать любую функцию используя двухслойную нейронную сеть.

# Обучение

Обозначим все параметры сети как  $\theta$ .

Пусть  $\mathcal{L}(\hat{y}_\theta, y)$  — **функция потерь** на объекте  $x$ .

Она сравнивает предсказания сети  $\hat{y}_\theta$  с откликом  $y$  на объекте  $x$ .

Минимизируем **эмпирический риск** по обучающей выборке  $x_1, \dots, x_n$ :

$$Q(\theta) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\hat{y}_{\theta,i}, y_i) \rightarrow \min_{\theta}$$

Решаем с помощью **градиентного спуска**

$$\theta_{t+1} = \theta_t - \eta \nabla Q(\theta_t), \quad \text{где } \eta \text{ — скорость обучения}$$

# Методы оптимизации

# Методы оптимизации

## Задача

$$f(x) \rightarrow \min_x$$

## Стохастический градиентный спуск

$$x_{t+1} = x_t - \eta \nabla f(x_t)$$

Приведем к записи вида

$$x_{t+1} = x_t + v_t,$$

где  $v_t = -\eta \nabla f(\theta_t)$  — аналог скорости.

*SGD не всегда оптимален.*

Существуют более быстрые методы.

## SGD + Momentum

Рассматривается смесь

- антиградиента
- шагов на предыдущих итерациях:

$$x_{t+1} = x_t + v_t$$

$$v_t = \mu v_{t-1} - \eta \nabla f(x_t),$$

где  $\mu$  — скорость затухания.

# Методы оптимизации

## Проблема

Компоненты вектора градиента могут иметь **разные масштабы**.

Поэтому по некоторым координатам уже давно сошлись,  
а по другим медленно двигаемся в сторону оптимума.

## Пояснение

В  $g$  хранится сумма квадратов частных производных.  
При шаге делаем нормировку градиента на корень из этой суммы.

## AdaGrad

Чтобы исправить это будем делать **нормировку**.

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{g_t + \varepsilon}} \odot \nabla f(x_t)$$

$$g_t = g_{t-1} + \nabla f(x_t) \odot \nabla f(x_t)$$

где  $\odot$  — поэлементное умножение,

$\varepsilon$  — сглаживающий параметр, необходимый,  
чтобы избежать деления на 0.

## Преимущества

- У часто обновляющихся параметров знаменатель будет больше, а поэтому обновление будет не сильным.
- Слабо обновляющиеся параметры обновятся больше.
- Скорость обучения (learning rate) автоматически затухает при увеличении итерации.



# Методы оптимизации

## Проблема

В AdaGrad  $g$  может увеличиваться сколько угодно, что через некоторое время приводит к **слишком маленьким обновлениям весов** и параличу сети.

## RMSProp

Будем использовать не сумму как в AdaGrad, а **экспоненциальное затухающее среднее**:

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{g_t + \varepsilon}} \odot \nabla f(x_t)$$

$$g_t = \mu g_{t-1} + (1 - \mu) \nabla f(x_t) \odot \nabla f(x_t)$$

$v$  и  $g$  будут долго накапливаться вначале, для этого искусственно увеличиваем их на первых шагах

## Adam

Комбинация RMSProp + Momentum

$$\theta_{t+1} = \theta_t - \frac{\eta}{\frac{\sqrt{g_t + \varepsilon}}{1 - \mu^t}} \odot \frac{v_{t+1}}{1 - \beta^t}$$

$$v_t = \mu v_{t-1} + (1 - \mu) \nabla f(x_t)$$

$$g_t = \mu g_{t-1} + (1 - \mu) \nabla f(x_t) \odot \nabla f(x_t)$$

**Adam** – самый популярный оптимизатор

# Методы оптимизации

## **Другие методы оптимизации**

Кроме рассмотренных нами методов, существуют и другие методы оптимизации.

Например, Nesterov Momentum, AdaDelta, LBFGS и др.

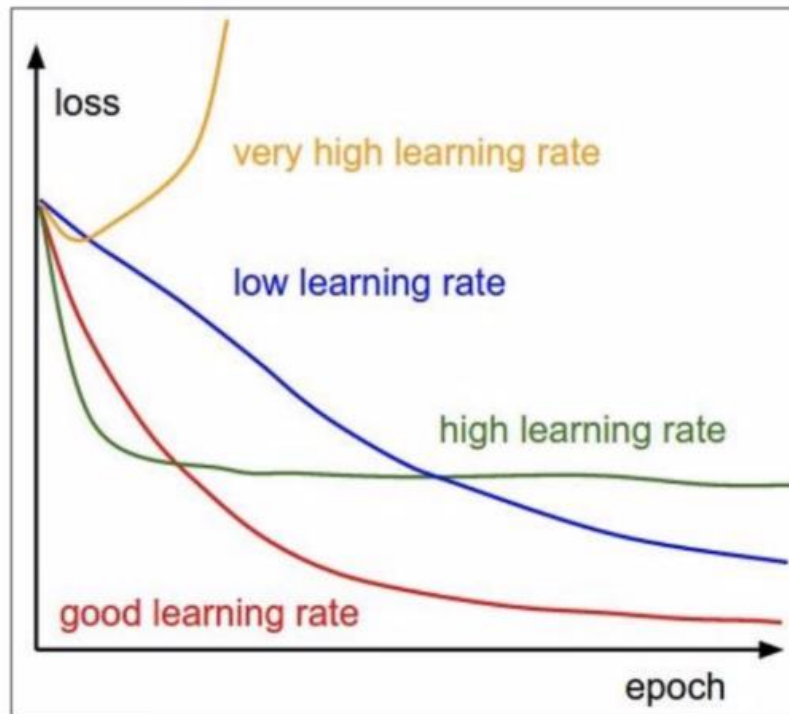
## **Почему мы рассматривали только методы первого порядка?**

Потому что методы второго порядка гораздо более вычислительно затратны.

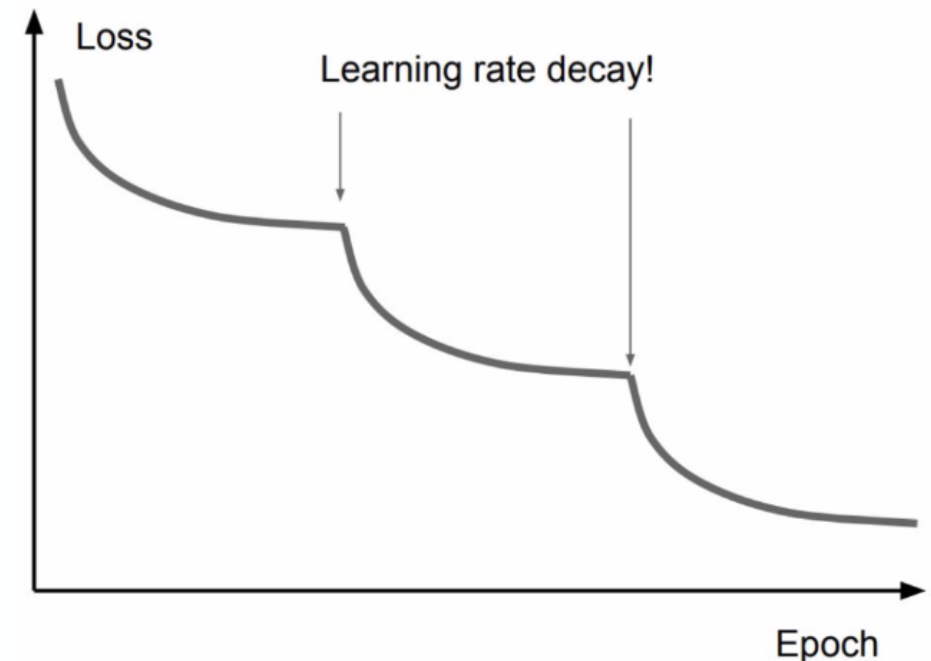
# Методы оптимизации

## Learning rate

- Нужно выбирать не слишком большим и не слишком маленьким.
- Стандартное значение – 0.001 для Adam.



Когда ошибка на валидации перестала уменьшаться на протяжении нескольких эпох можно уменьшить  $lr$  в несколько раз. Это обычно дает небольшой прирост качества.



# Методы оптимизации

Существует много способов изменения learning rate по ходу обучения:

- с разогревом (warmstart),
- ступенчатые,
- циклические и т.д.

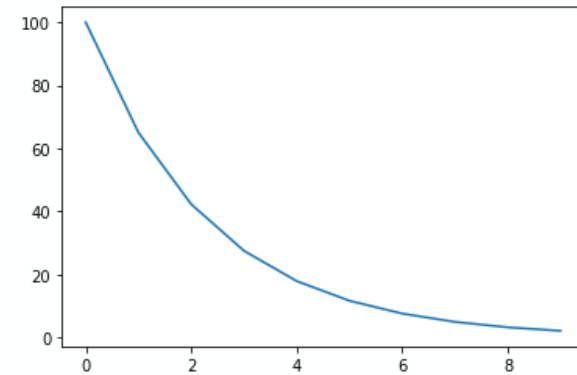
Часть из них реализована в [pytorch](#): [обзор с визуализацией](#).

Для некоторых сложных архитектур изменение стратегии изменения learning rate очень важно. Например, при обучении трансформеров.

## Примеры

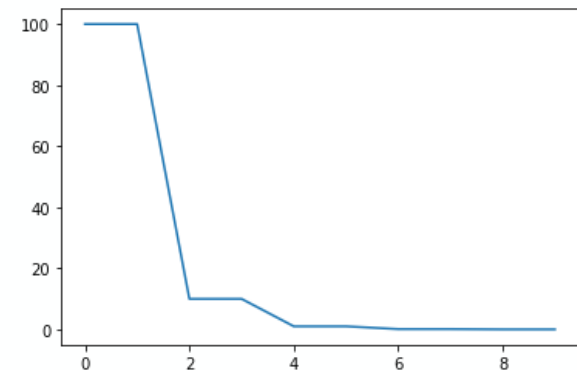
### LAMBDA LR

$$\eta_t = \eta_0 \lambda_t$$



### StepLR

$$\eta_t = \begin{cases} \gamma \eta_{t-1}, & \text{если } t \% k = 0 \\ \eta_{t-1}, & \text{иначе} \end{cases}$$



Инициализация

# Инициализация

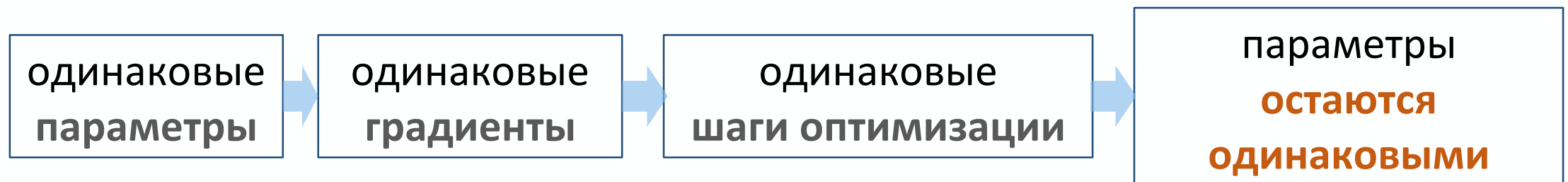
Решаем с помощью **градиентного спуска**



Необходимо **инициализировать параметры**.

Хорошо ли инициализировать все одной константой?

**Плохо**



# Инициализация

Инициализировать **случайно** небольшими значениями из нормального или равномерного распределений?

# Инициализация

Инициализировать **случайно** небольшими значениями из нормального или равномерного распределений?

**Проблема** рассмотрим прямой проход

$$u = \sum_{j=1}^d w_j x_j \quad \mathbf{D}(u) = \mathbf{D}\left(\sum_{j=1}^d w_j x_j\right) = \sum_{j=1}^d \mathbf{D}(w_j) \cdot x_j^2$$

⇒ из-за суммирования дисперсия увеличивается в  $d$  раз.

- Большая дисперсия может привести к численным ошибкам или насыщению ф-й активации *tanh* и *sigmoid*.
- Маленькая — к околонулевым промежуточным представлениям.



Инициализировать случайно небольшими значениями из нормального или равномерного распределений с дисперсией  $1/d$ .



# Инициализация

## Более сложные методы

Во время *прямого прохода* дисперсия выхода слоя увеличивается в  $d_{in}$  раз, где  $d_{in}$  — размерность входа слоя.

**Проблема** рассмотрим *обратный проход*.

Дисперсия градиента увеличивается в  $d_{out}$  раз, где  $d_{out}$  — размерность выхода слоя.



## Инициализация Ксавьера (Xavier)

Инициализировать случайно небольшими значениями из нормального или равномерного распределений с дисперсией

$$\frac{2}{d_{in} + d_{out}}$$

[статья](#)

# Инициализация

## Более сложные методы

Мы ранее опирались на то, что функция активации будет tanh или sigmoid.

**Проблема** рассмотрим функцию активации *ReLU*.

Она имеет смещенную относительно нуля область значений.



## Инициализация Каминга (Kaiming)

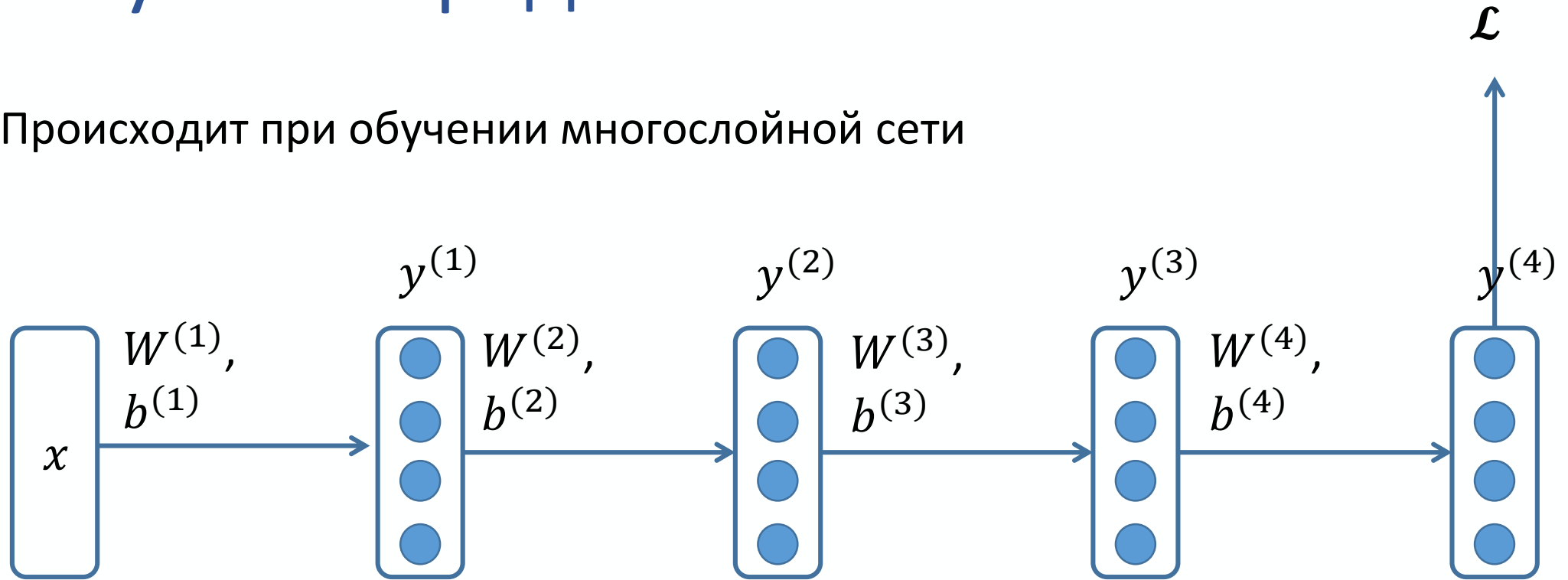
Инициализировать случайно небольшими значениями

из нормального или равномерного распределений с дисперсией  $\frac{2}{d_{in}}$

# Затухание и взрыв градиента

# Затухание градиента

Происходит при обучении многослойной сети



Сделали forward pass.

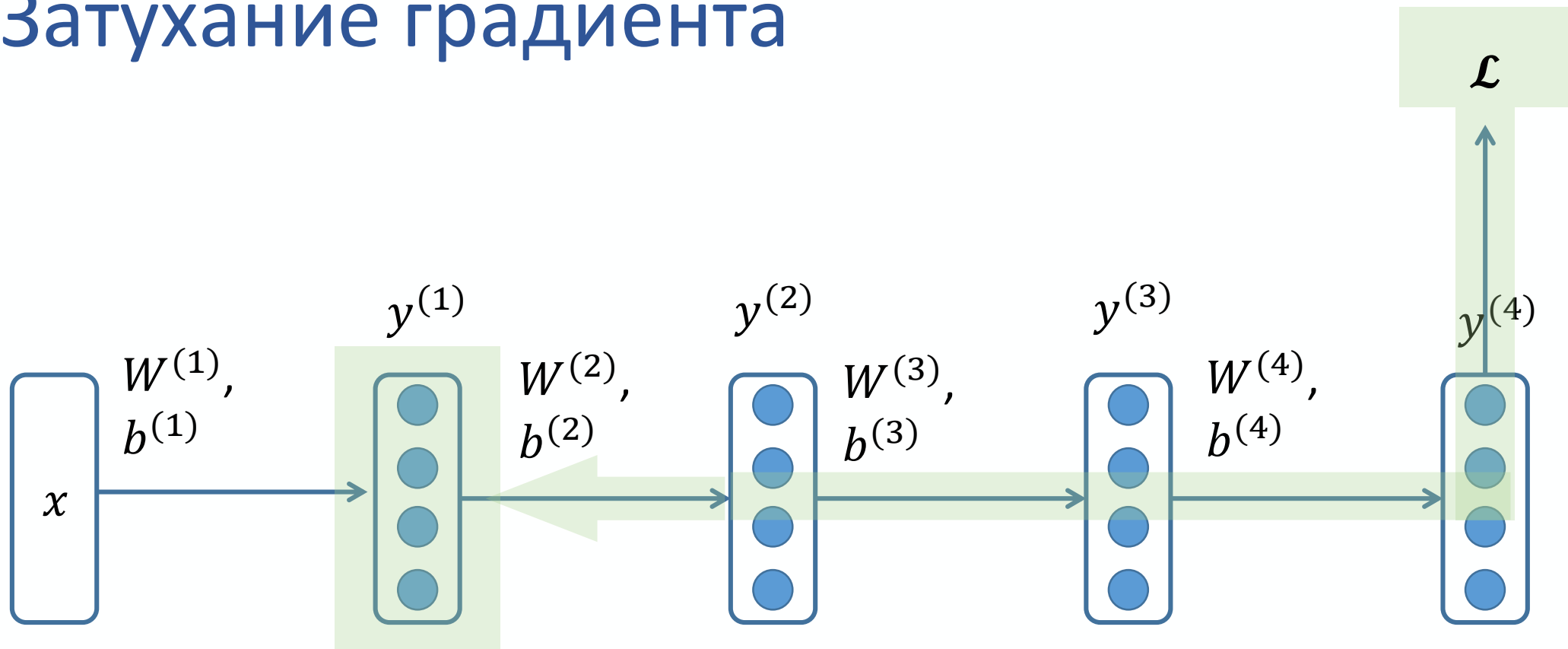
Теперь хотим обновить параметры.

Для этого делаем backward pass.

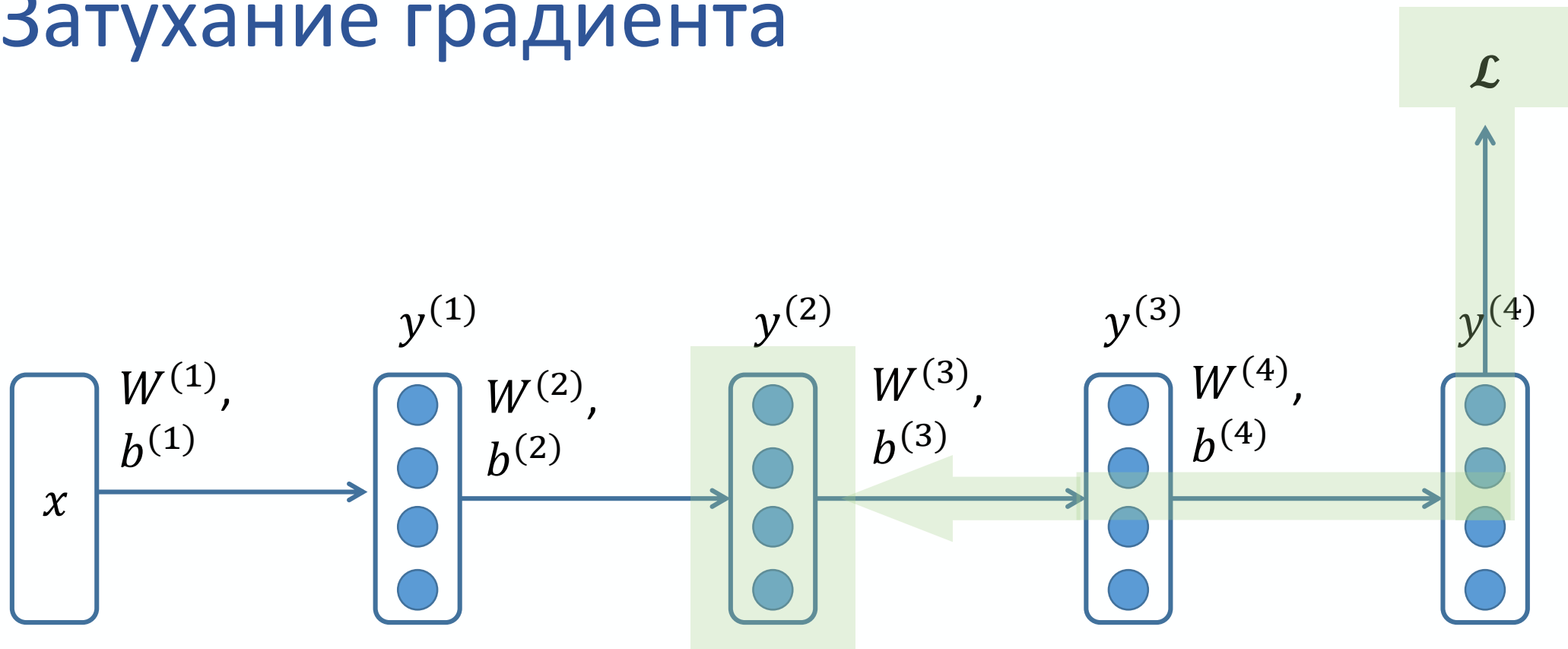
Посмотрим, как будут считаться градиенты для  $y^{(1)}$ .

# Затухание градиента

$$\frac{\partial \mathcal{L}}{\partial y^{(1)}}$$

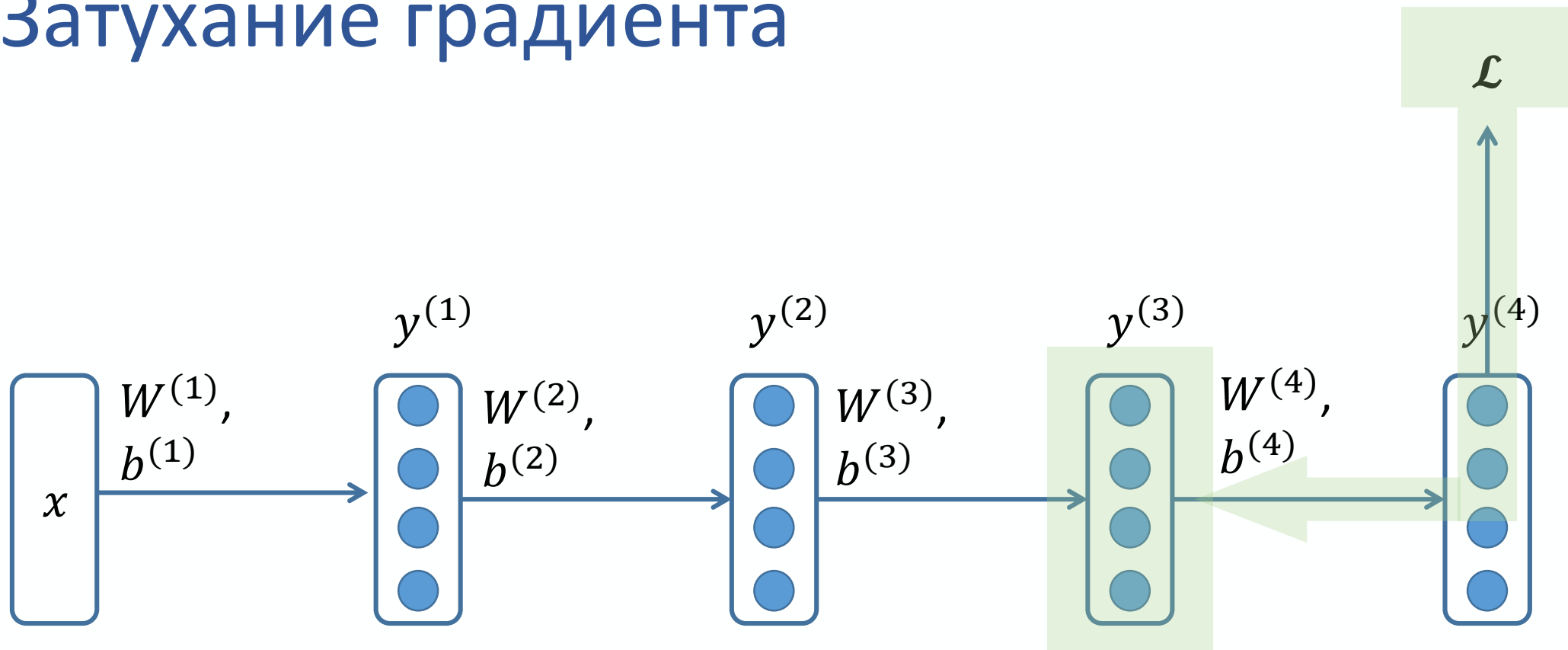


# Затухание градиента



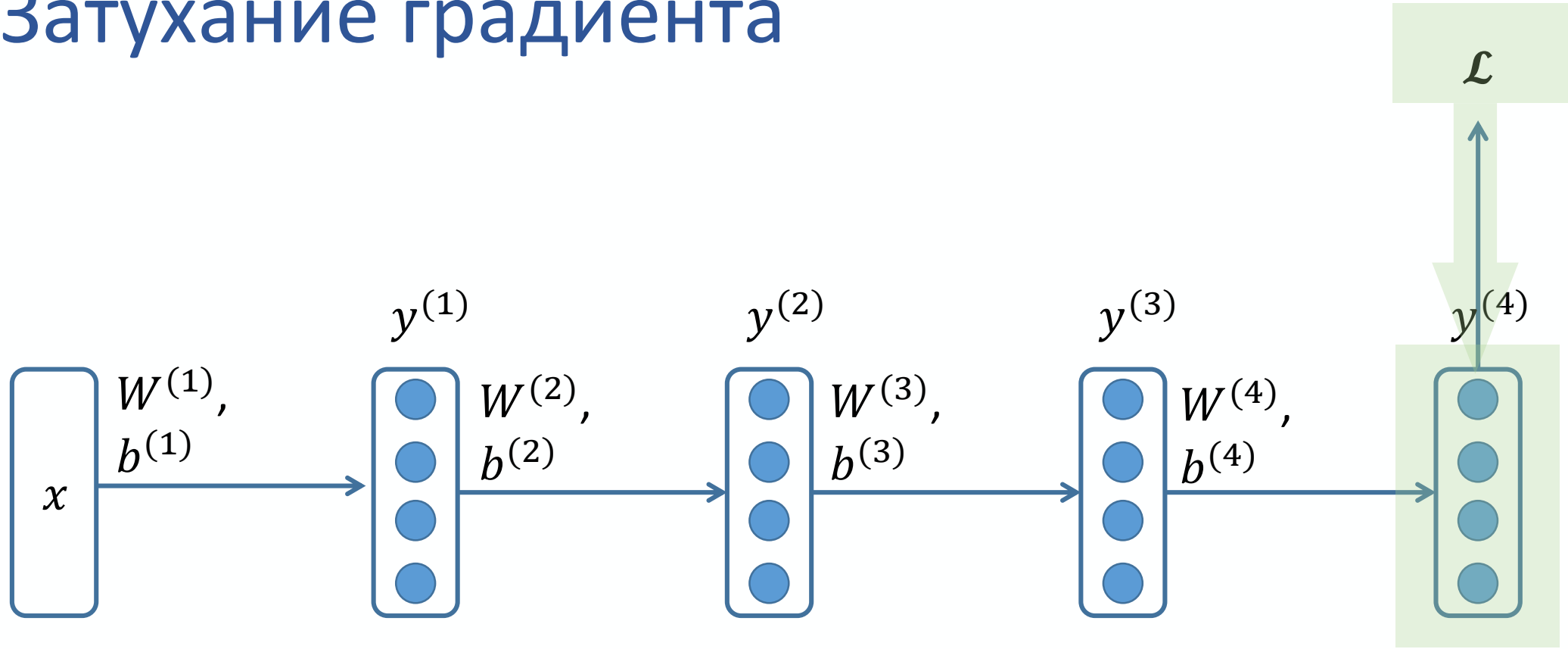
$$\frac{\partial \mathcal{L}}{\partial y^{(1)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(2)}}$$

# Затухание градиента



$$\frac{\partial \mathcal{L}}{\partial y^{(1)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(2)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial y^{(3)}}{\partial y^{(2)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(3)}}$$

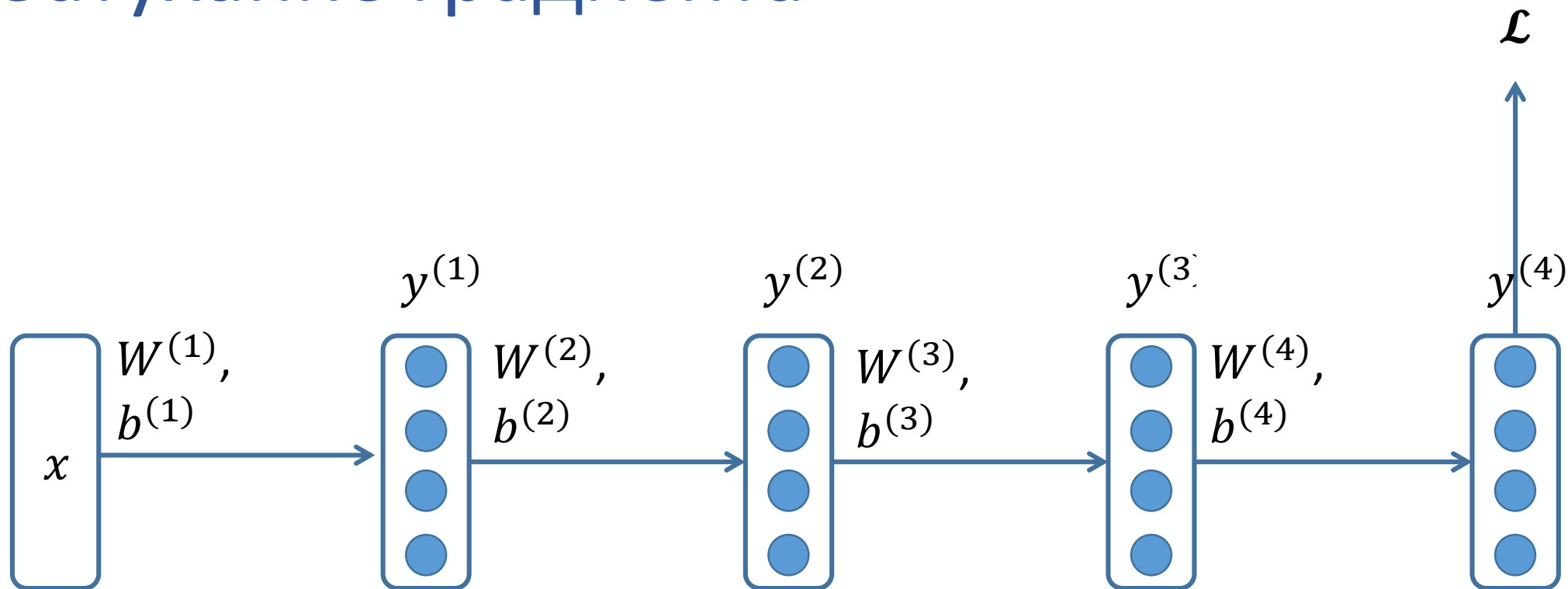
# Затухание градиента



$$\frac{\partial \mathcal{L}}{\partial y^{(1)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(2)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial y^{(3)}}{\partial y^{(2)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(3)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial y^{(3)}}{\partial y^{(2)}} \cdot \frac{\partial y^{(4)}}{\partial y^{(3)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(4)}}$$



# Затухание градиента



$$\frac{\partial \mathcal{L}}{\partial y^{(1)}} = \frac{\partial y^{(2)}}{\partial y^{(1)}} \cdot \frac{\partial y^{(3)}}{\partial y^{(2)}} \cdot \frac{\partial y^{(4)}}{\partial y^{(3)}} \cdot \frac{\partial \mathcal{L}}{\partial y^{(4)}} \sim 0$$

Если градиенты очень маленькие, то из-за пределов вычислительной точности они превращаются в 0.

# Затухание градиента

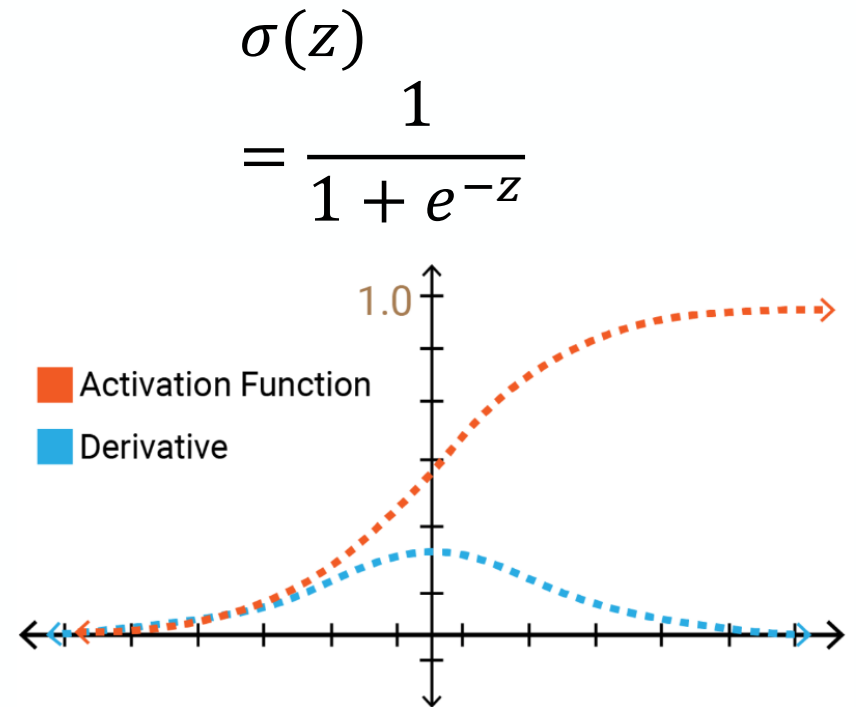
Причины, почему производная может быть близкой к 0

- Неудачная **функция активации**.
- При **приближении к локальному оптимуму** в большинстве случаев функции меняются слабо. Это приводит к небольшим изменениям градиентов.
- Неудачная **начальная инициализация**.

# Затухание градиента

## Сигмоида

- Имеет горизонтальные асимптоты.  
Для больших значений аргумента производная имеет очень маленькие значения.
- Многие из производных выхода по входу  $\frac{\partial a}{\partial u}$  могут иметь маленькие значения.
- При перемножении производных вида  $\frac{\partial a}{\partial u}$  произведение может занулиться.
- Обновление весов практически не изменит веса. Наступит “паралич сети”.

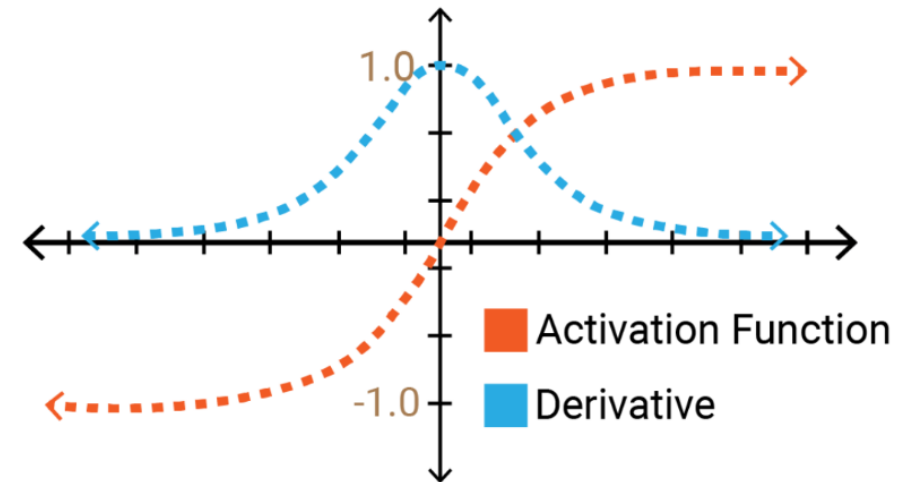


# Затухание градиента

## Гиперболический тангенс

Такая же ситуация, как и с сигмоидальной функцией.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$$

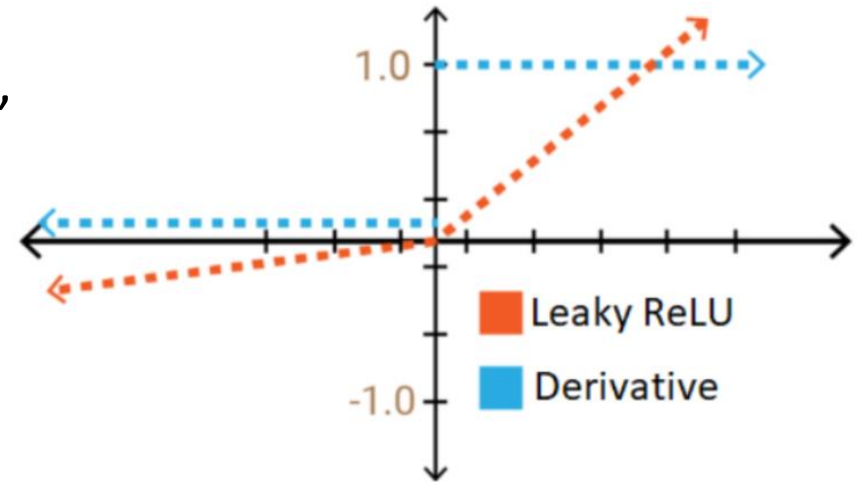


# Затухание градиента

## Leaky ReLU

- Производная либо равна 1, либо  $\alpha$ .
- Матрица производных  $\frac{\partial a}{\partial u}$  будет состоять только из чисел 1 и  $\alpha$ .
- Паралич сети может возникнуть только в случае, если почти вся матрица  $\frac{\partial a}{\partial u}$  состоит из  $\alpha$ , Это произойдет с очень маленькой вероятностью. На практике встречается только в очень глубоких сетях.

$$lrelu(z) = z \cdot I\{z > 0\} + \alpha z \cdot I\{z \leq 0\}$$



## ReLU

То же что Leaky ReLU при  $\alpha = 0$ .

- Матрица производных  $\frac{\partial a}{\partial u}$  будет состоять только из чисел 1 и 0.
- Паралич сети может возникнуть.

# Затухание градиента

## Решение проблемы затухания градиента

- Использование функций активаций без горизонтальных асимптот. То есть не использовать sigmoid, tanh и прочие.

- Residual connections (shortcuts)

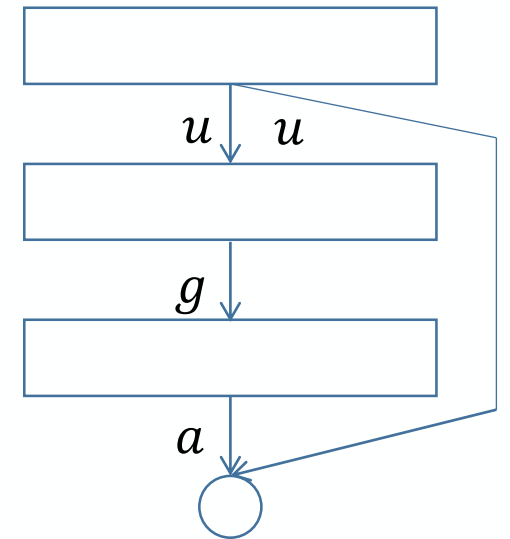
К выходу какого-то слоя прибавляем выход какого-то из предыдущих слоев. Пусть skip-connection делается через два слоя. Рассматриваем 3 слоя.

Пусть

- $u$  - выход первого из них,
- $g$  - выход среднего слоя,
- $a$  - выход последнего слоя,
- $a'$  - результат после прибавления  $u$ .

$\frac{\partial a'}{\partial g} \cdot \frac{\partial g}{\partial u}$  - произведение, присутствующее в BackProp.

$$\frac{\partial a'}{\partial g} \cdot \frac{\partial g}{\partial u} = \frac{\partial a'}{\partial u} = \frac{\partial a}{\partial u} + 1 = \frac{\partial a}{\partial g} \frac{\partial g}{\partial u} + 1 > 0 \text{ при } \frac{\partial a}{\partial g} \frac{\partial g}{\partial u} \sim 0.$$



$$a' = a + u$$

# Взрыв градиента

## Проблемы

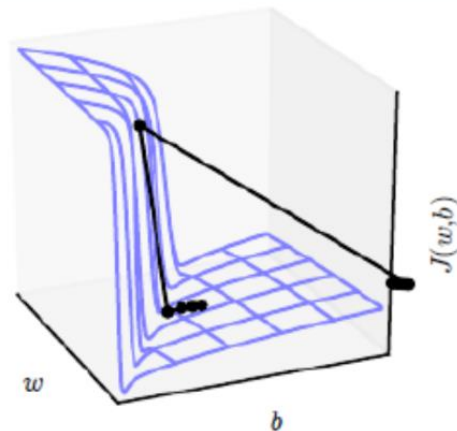
- При подсчете градиента друг на друга умножаются

градиенты выхода по входу  $\frac{\partial a}{\partial u}$  от разных слоев.

- Если каждый градиент достаточно большой, то при вычислении умножения

может произойти переполнение.

Если даже переполнение не произошло, то при обновлении весов веса сильно изменятся, что приводит к нестабильности сети.



# Взрыв градиента

## Решения

- **Добавление к функции потерь регуляризации весов (weight decay)**

Раньше веса обновлялись как  $\theta_t = \theta_{t-1} - \eta \nabla Q(\theta_t)$ .

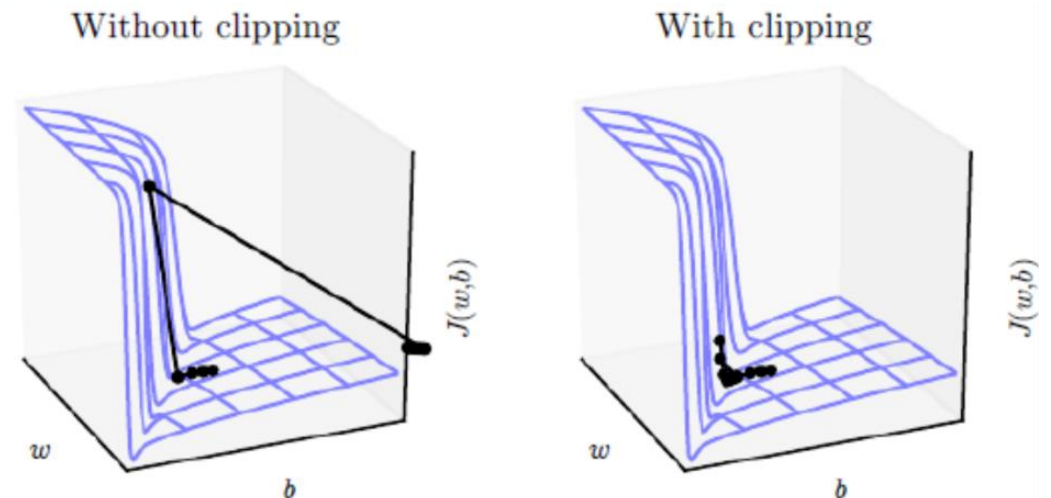
Добавляем регуляризацию (напр. L2) к лоссу.  $\tilde{Q} = Q + \frac{\lambda}{2} \theta^2$ .

Теперь веса обновляются как  $\theta_t = \theta_{t-1} - \eta \nabla Q(\theta_t) - \eta \lambda \theta_t$ .

- **Gradient clipping**

Устанавливается гиперпараметр `threshold` и если норма градиента больше `threshold`, то градиент масштабируется.

$$\|G\| > threshold \Rightarrow G = \frac{threshold \cdot G}{\|G\|}$$





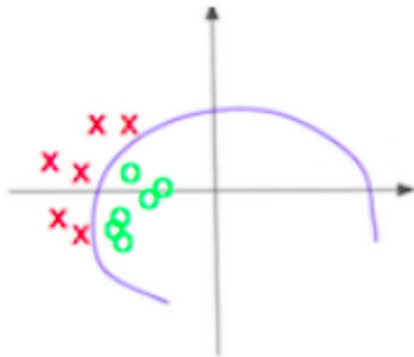
# Пакетная нормализация / Batch Normalization

[Статья на русском про Batch-нормализацию](#)

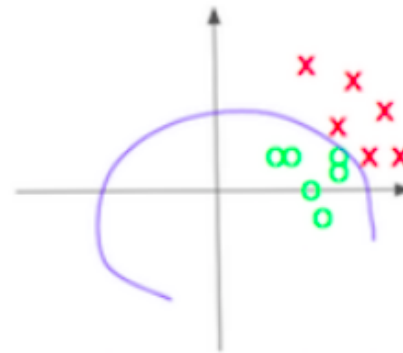
# Ковариантный сдвиг

**Ковариантный сдвиг** — это ситуация, когда распределения значений **признаков** в обучающей и тестовой выборке имеют разные параметры (математическое ожидание, дисперсия и т.д.).

обучающая выборка



тестовая выборка



# Ковариантный сдвиг

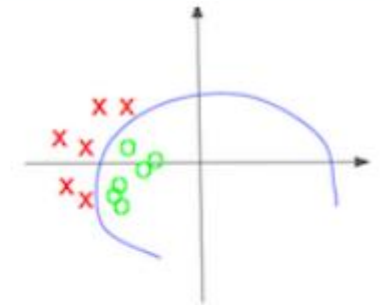
- В обучающей выборке только красные розы.



Роза  
( $y=1$ )



Не роза  
( $y=0$ )



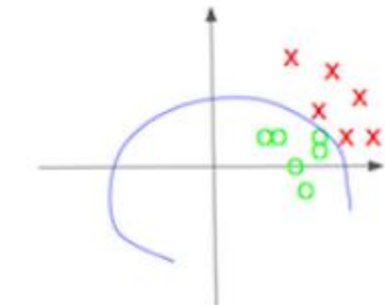
- В тестовой — розы разных цветов. Параметры распределения данных изменились.



Роза  
( $y=1$ )



Не роза  
( $y=0$ )



Проблема :(

# Ковариантный сдвиг

Ковариантный сдвиг будет незначительным, если распределение признаков в обучающем и тестовом наборе данных будет практически одинаковым.



## **Простое решение**

Перемешаем данные случайным образом.

Тогда распределение везде будет примерно одинаковым.

# Ковариантный сдвиг и нейронные сети

Самый распространенный способ обновления весов нейронной сети —

**Mini-batch Gradient Descent.**

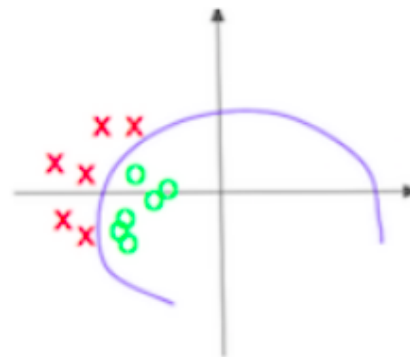
Разбиваем данные на блоки (батчи).  $(x_{i_1}, \dots, x_{i_b})$  — текущий батч.

Для каждого блока считаем градиент и обновляем параметры.

Теперь **ковариантный сдвиг** может возникнуть между двумя **батчами**.

$$\theta_t = \theta_{t-1} - \eta \nabla \left( \frac{1}{b} \sum_{k=1}^b \mathcal{L}(\hat{y}_{\theta_{t-1}, i_k} \ y_{i_k}) \right)$$

первый батч



второй батч



# Ковариантный сдвиг и нейронные сети

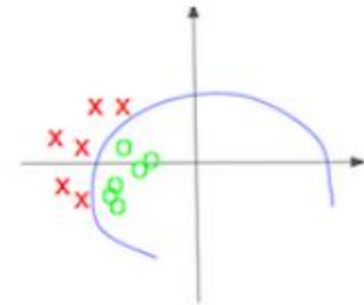
- В первом батче только красные розы. Модель изменила параметры под первый батч.



Роза  
( $y=1$ )



Не роза  
( $y=0$ )



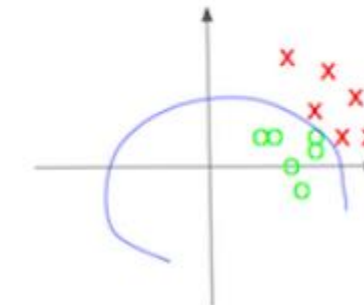
- Во втором — розы разных цветов. Параметры распределения данных существенно изменились. Модели теперь нужно сильно изменять параметры под второй батч. Она начнет «скакать» в пространстве параметров



Роза  
( $y=1$ )



Не роза  
( $y=0$ )



# Ковариантный сдвиг и нейронные сети

## Простое решение

Перемешаем данные случайным образом.

Тогда распределение везде будет примерно одинаковым.

Но так мы изменим **только входные данные**.

Проблема ковариантного сдвига может наблюдаться  
**для входов каждого слоя нейронной сети!**

Мы не можем воспользоваться простым решением,  
т. к. распределение входных данных для каждого узла скрытых слоев  
изменяется каждый раз, когда происходит обновление параметров в предыдущем  
слое.

Эта проблема называется **внутренним ковариантным сдвигом**.  
Эту проблему можно решить понижением скорости обучения,  
регуляризацией или **батч-нормализацией**.

# Пакетная нормализация / Batch Normalization

Батч-нормализация представляется в виде обучаемого слоя нейронной сети.

Вход:  $(x_{i_1}, \dots, x_{i_b})$  — текущий батч.

Обучаемые параметры:  $\gamma, \beta$

Константа:  $\epsilon$

Выходы:  $(y_{i_1}, \dots, y_{i_b})$

- Даем возможность нейросети выучить нужный для нее масштаб и сдвиг.
  - Возможно для разных слоев нужны разные сдвиги и масштабы.
  - Можно подобрать такие  $\gamma$  и  $\beta$ , что нормализации не будет.
- Модель сама определит, какие нужны параметры, чтобы не потерять в точности.

**Алгоритм вычисления  $y_{i_k}$  при обучении**

- $\mu_i = \frac{1}{b} \sum_{k=1}^b x_{i_k}$  — матем. ожидание по батчу
- $\sigma_i^2 = \frac{1}{b} \sum_{k=1}^b (x_{i_k} - \mu_i)^2$  — дисперсия по батчу
- $\hat{x}_{i_k} = \frac{x_{i_k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$  — нормализация
- $y_{i_k} = \gamma \hat{x}_{i_k} + \beta$  — масштабирование и сдвиг

Все операции дифференцируемы, значит можем делать backprop.



# Пакетная нормализация / Batch Normalization

**Почему батч-нормализация  
— это решение проблемы?**

Батч-нормализация

нормирует данные внутри батча.

Благодаря чему сдвиг и дисперсия  
для всех батчей становятся

одинаковыми.

**Алгоритм вычисления  $y_{i_k}$  при обучении**

- $\mu_i = \frac{1}{b} \sum_{k=1}^b x_{i_k}$  — матем. ожидание по батчу
- $\sigma_i^2 = \frac{1}{b} \sum_{k=1}^b (x_{i_k} - \mu_i)^2$  — дисперсия по батчу
- $\hat{x}_{i_k} = \frac{x_{i_k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$  — нормализация
- $y_{i_k} = \gamma \hat{x}_{i_k} + \beta$  — масштабирование и сдвиг

# Пакетная нормализация / Batch Normalization

## Алгоритм вычисления $y_{i_k}$ при обучении

- $\mu_i = \frac{1}{b} \sum_{k=1}^b x_{i_k}$  — матем. ожидание по батчу
- $\sigma_i^2 = \frac{1}{b} \sum_{k=1}^b (x_{i_k} - \mu_i)^2$  — дисперсия по батчу
- $\hat{x}_{i_k} = \frac{x_{i_k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$  — нормализация
- $y_{i_k} = \gamma \hat{x}_{i_k} + \beta$  — масштабирование и сдвиг
- $\mu_t \leftarrow \alpha \mu_i + (1 - \alpha) \mu_{t-1}$  ←
- $\sigma_t^2 \leftarrow \alpha \sigma_i^2 + (1 - \alpha) \sigma_{t-1}^2$

## Тестирование

При тестировании модели, мы не можем считать среднее и дисперсию по батчу.

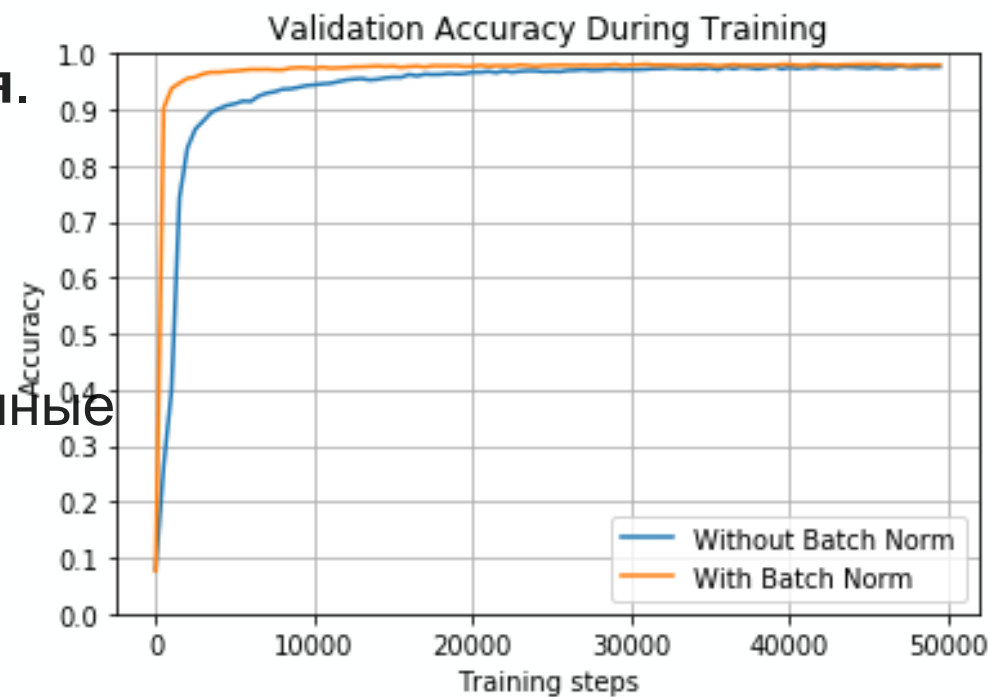
Иначе для разных способов разбиения тестовых данных по батчам будут получаться разные результаты.

Будем использовать  $\mu, \sigma$ , оцененные по всей обучающей выборке. В остальном алгоритм тот же.

Считаем экспоненциальное среднее для  $\mu, \sigma$  во время обучения

# Пакетная нормализация / Batch Normalization

- Решается проблема **ковариационно сдвига**.
- Достигается более **быстрая сходимость** моделей, несмотря на выполнение дополнительных вычислений.
- Можно ставить **большую скорость обучения**.
- Позволяет каждому слою сети обучаться более независимо от других слоев, т.к. слои теперь получают нормализованные данные



[Пример взят отсюда](#)

# Пакетная нормализация / Batch Normalization

## Другое мнение

В [статье](#) критикуется идея, что Batch Norm решает проблему ковариантного сдвига.

Полезность метода поясняется тем, благодаря нормализации выходов градиенты более предсказуемо себя ведут, что обеспечивает более быструю и эффективную оптимизацию.



**ВСЁ!**