

# Основы глубинного обучения

## Лекция 4

Оптимизация в глубинном обучении.

Евгений Соколов

[esokolov@hse.ru](mailto:esokolov@hse.ru)

НИУ ВШЭ, 2023

# Стохастический градиентный спуск

# Градиентный спуск

1. Начальное приближение:  $w^0$

2. Повторять:

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

3. Останавливаемся, если ошибка на тестовой выборке перестает убывать

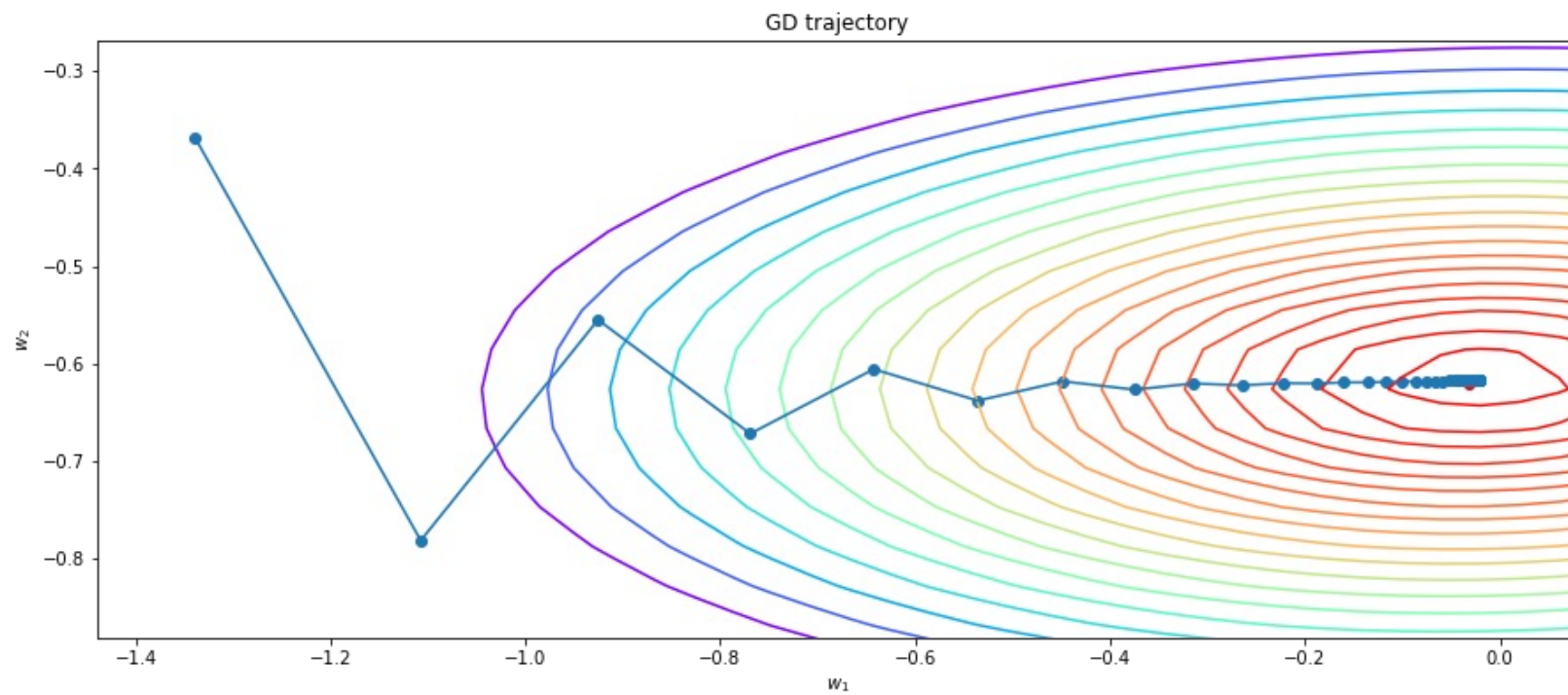
# Стохастический градиентный спуск

1. Начальное приближение:  $w^0$
2. Повторять, каждый раз выбирая случайный объект  $i_t$ :

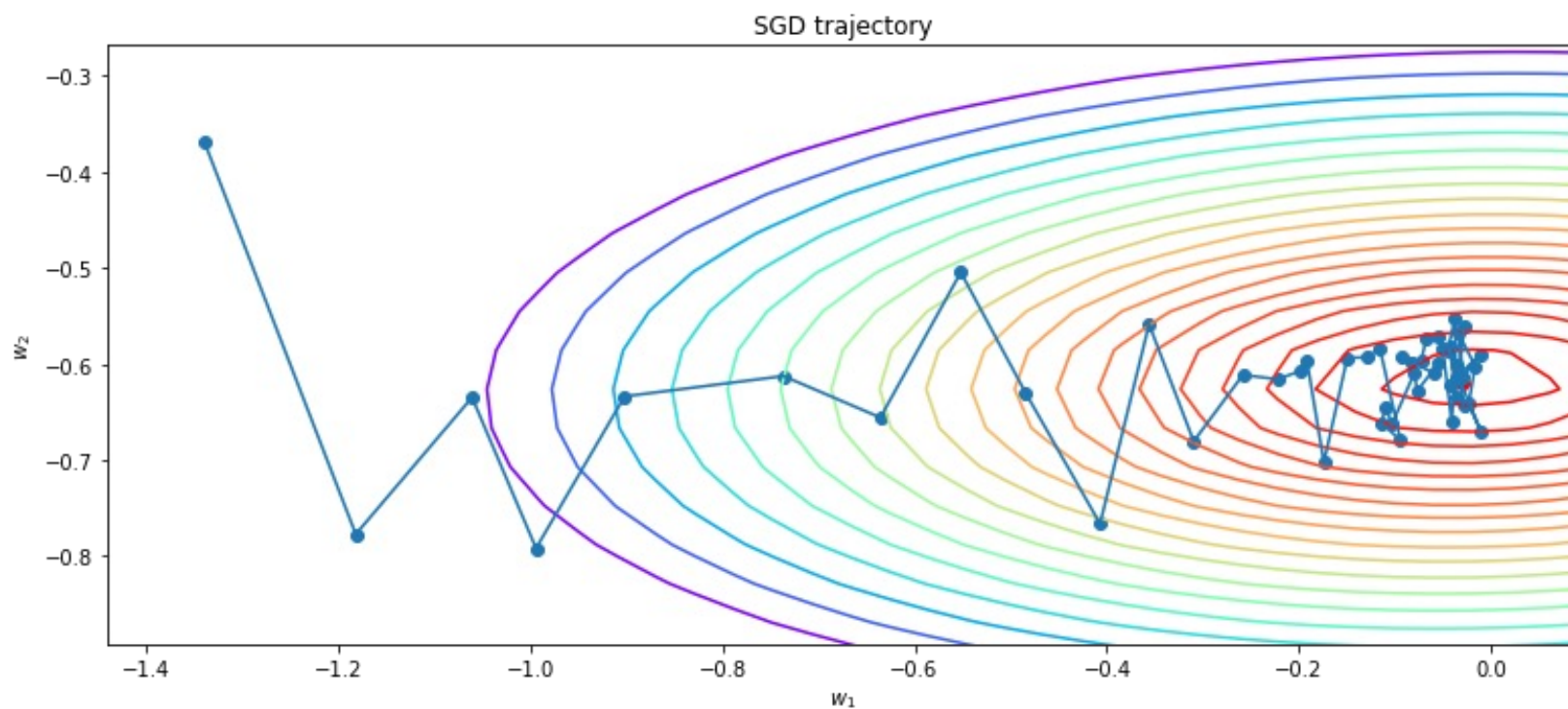
$$w^t = w^{t-1} - \eta \nabla L(y_{i_t}, a(x_{i_t}))$$

3. Останавливаемся, если ошибка на тестовой выборке перестает убывать

# Градиентный спуск



# Стохастический градиентный спуск



# Стохастический градиентный спуск

1. Начальное приближение:  $w^0$
2. Повторять, каждый раз выбирая случайный объект  $i_t$ :

$$w^t = w^{t-1} - \eta_t \nabla L(y_{i_t}, a(x_{i_t}))$$

3. Останавливаемся, если ошибка на тестовой выборке перестает убывать

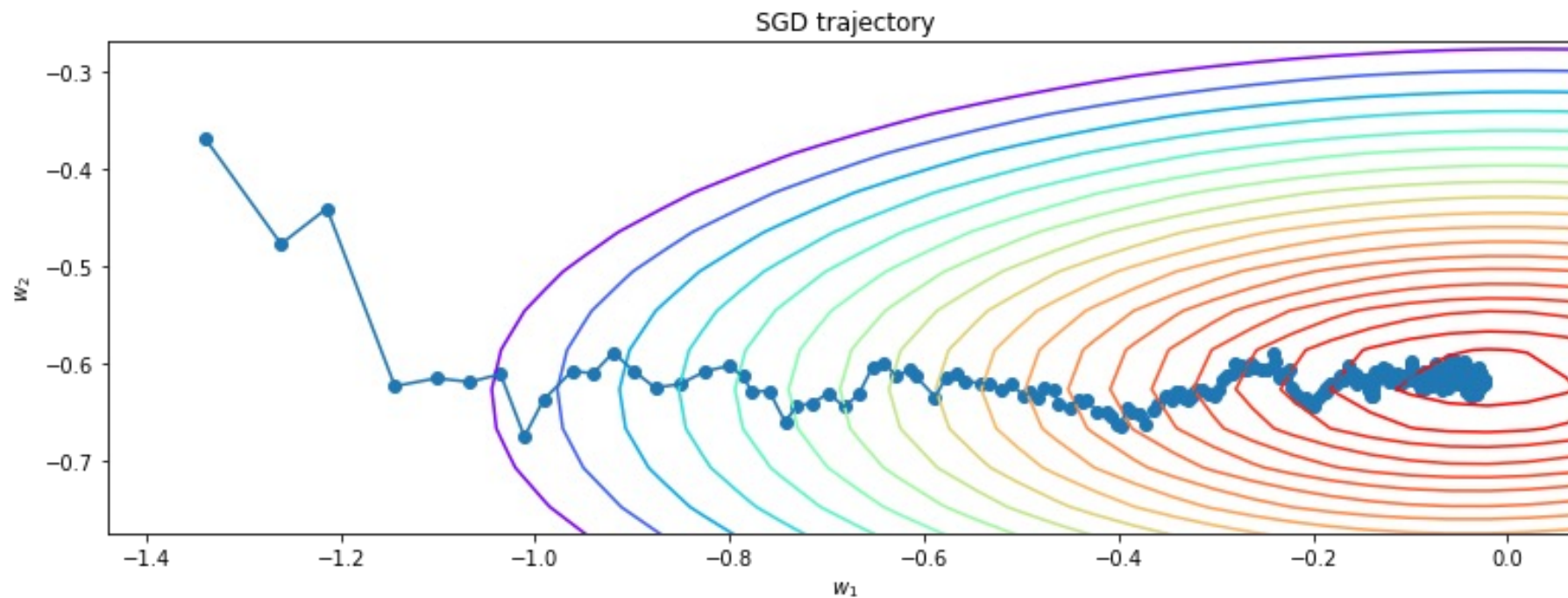
# Стохастический градиентный спуск

- Оценка по одному объекту **несмещённая**
- То есть в среднем мы идём в правильную сторону
- Даже в точке оптимума оценка по одному объекту вряд ли будет нулевой
- Поэтому важно, чтобы длина шага стремилась к нулю
- Сходимость к глобальному минимуму гарантируется только для выпуклых функций

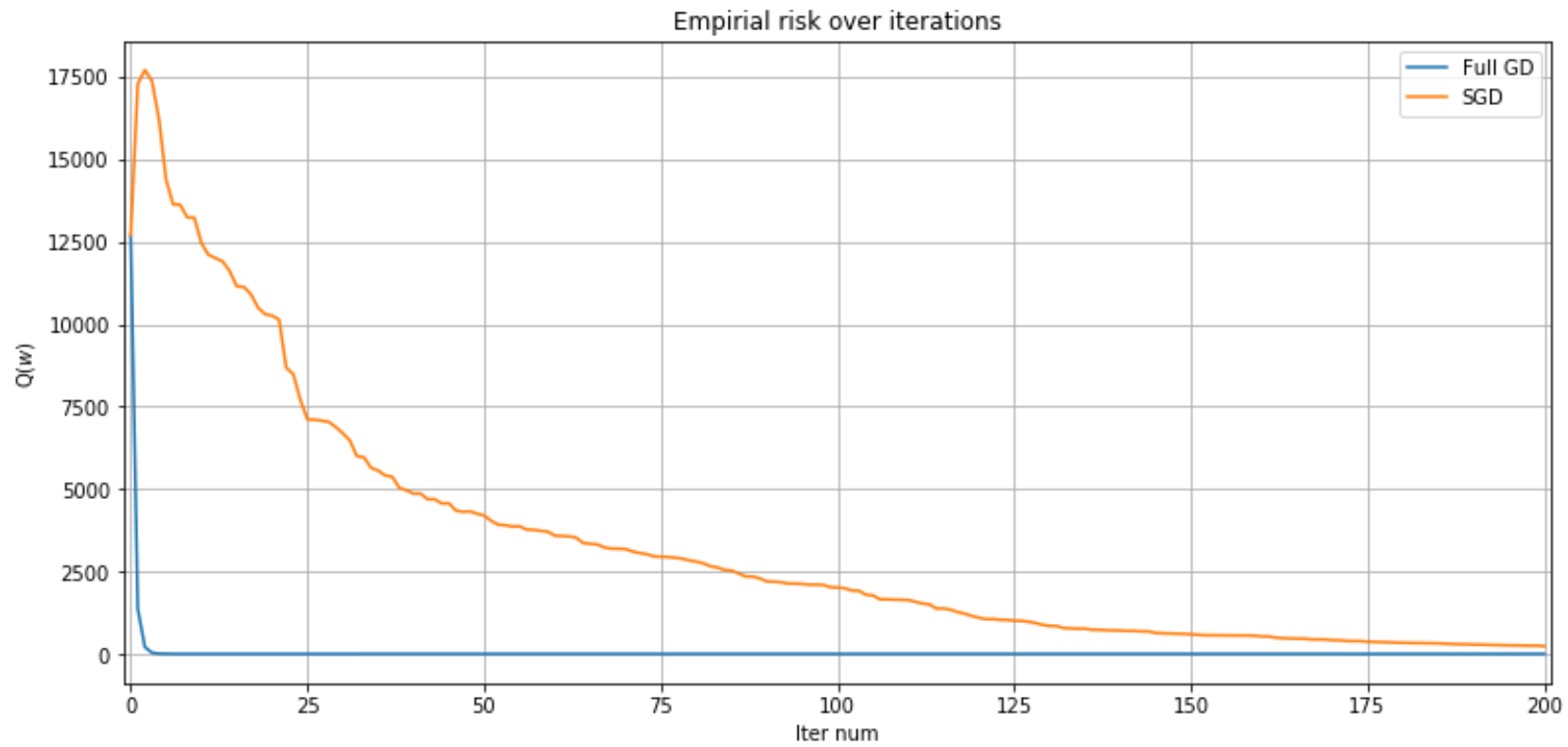


# Стохастический градиентный спуск

$$\eta_t = \frac{0.1}{t^{0.3}}$$



# Стохастический градиентный спуск



# Mini-batch GD

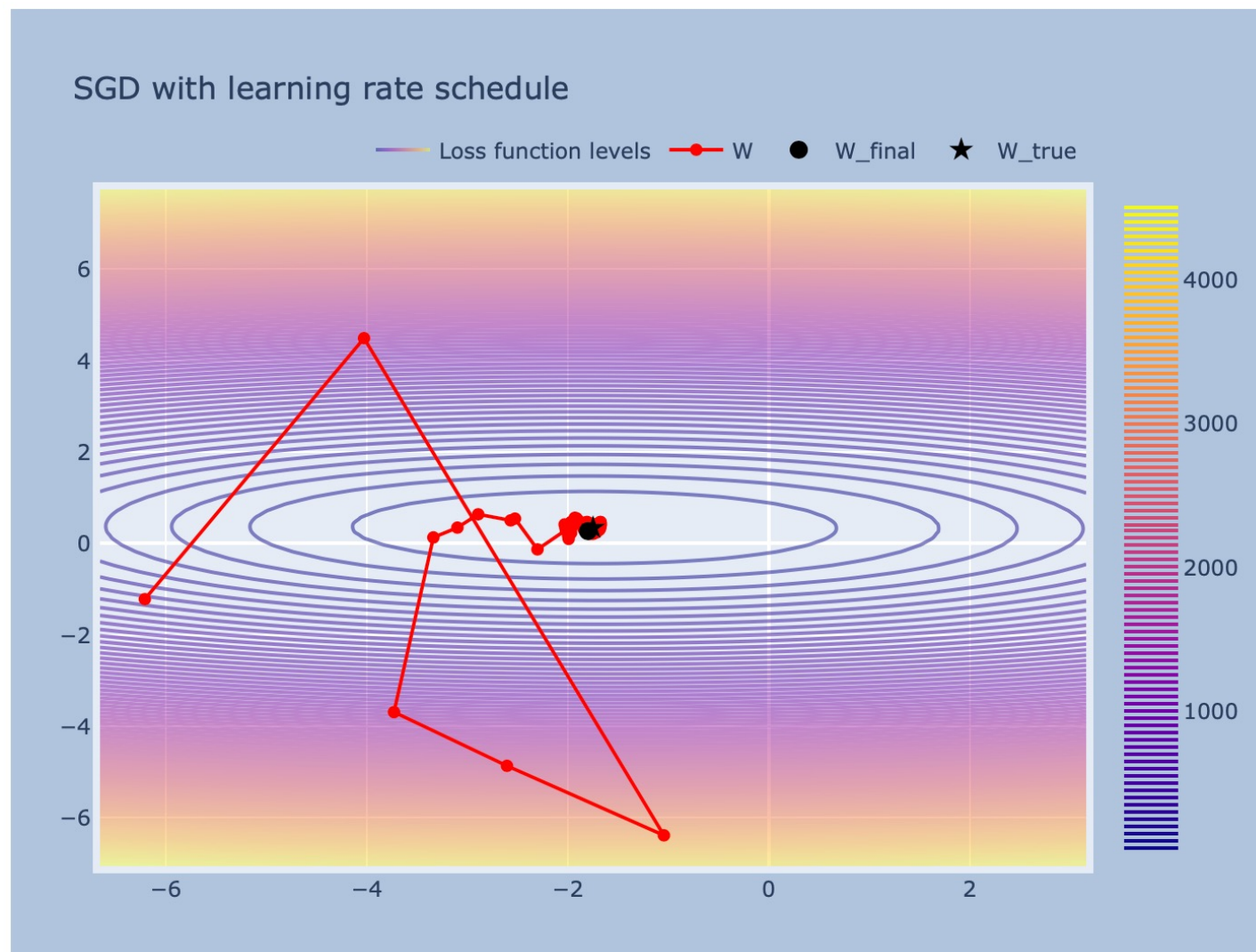
1. Начальное приближение:  $w^0$
2. Повторять, каждый раз выбирая  $m$  случайных объектов  $i_1, \dots, i_m$ :

$$w^t = w^{t-1} - \eta_t \frac{1}{n} \sum_{j=1}^n \nabla L(y_{t,j}, a(x_{t,j}))$$

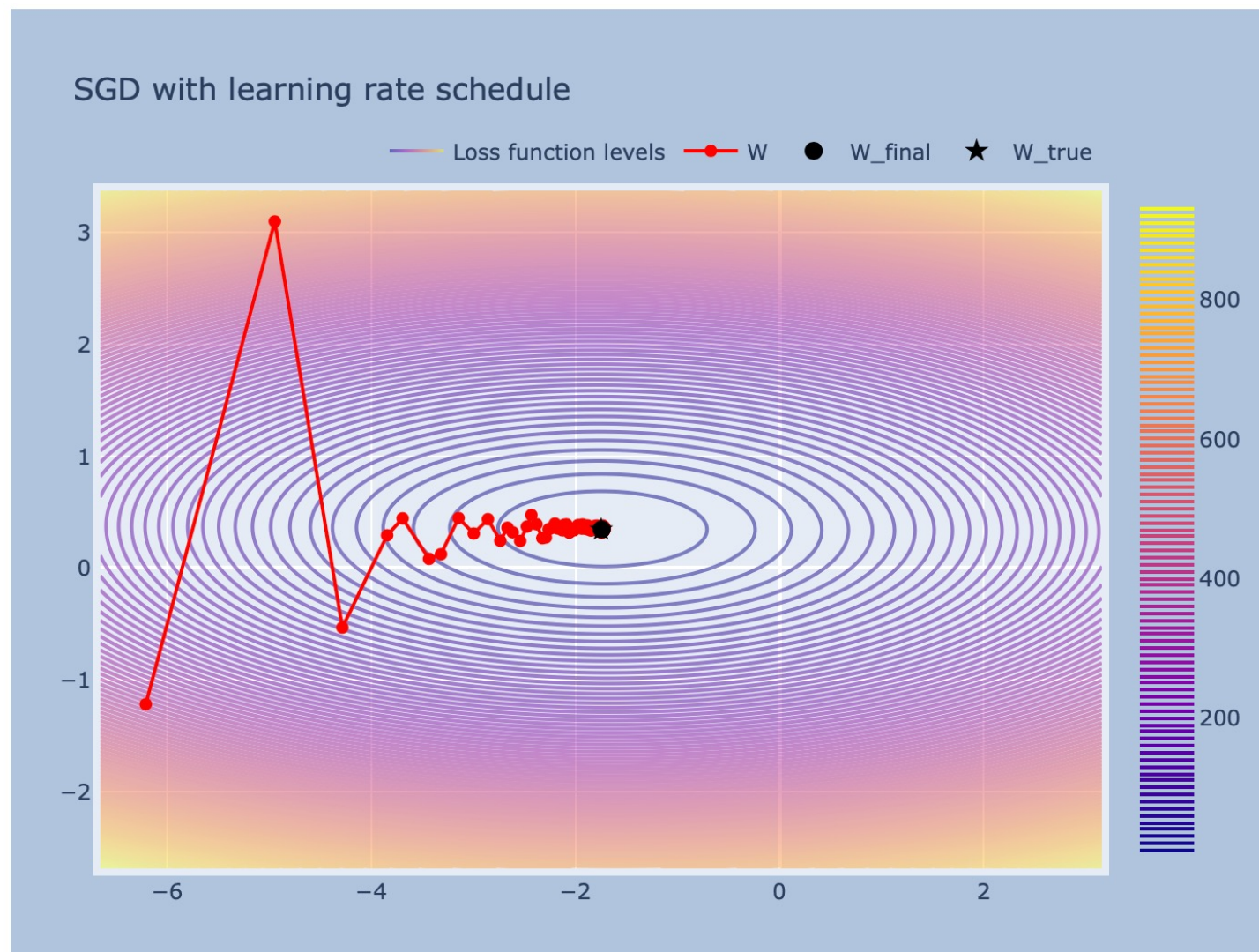
3. Останавливаемся, если ошибка на тестовой выборке перестает убывать

$x_{t,j}$  — объект номер  $j$  из батча, сформированного на шаге  $t$

# Батч размера 1



# Батч размера 10



# Batch size

- Размер пакета — обычно порядка десятков или сотни
- Имеет смысл брать степень двойки
- Возможно, делает оценку градиента более стабильной
- Вычислительно почти так же эффективен, как шаг по градиенту одного объекта — за счёт векторизации

# Batch size

The collected experimental results for the CIFAR-10, CIFAR-100 and ImageNet datasets show that increasing the mini-batch size progressively reduces the range of learning rates that provide stable convergence and acceptable test performance. On the other hand, small mini-batch sizes provide more up-to-date gradient calculations, which yields more stable and reliable training. The best performance has been consistently obtained for mini-batch sizes between  $m = 2$  and  $m = 32$ , which contrasts with recent work advocating the use of mini-batch sizes in the thousands.

# Batch size



**Yann LeCun**

April 27, 2018 · 🌐



Training with large minibatches is bad for your health.

More importantly, it's bad for your test error.

Friends dont let friends use minibatches larger than 32.

Let's face it: the *\*only\** people have switched to minibatch sizes larger than one since 2012 is because GPUs are inefficient for batch sizes smaller than 32. That's a terrible reason. It just means our hardware sucks.

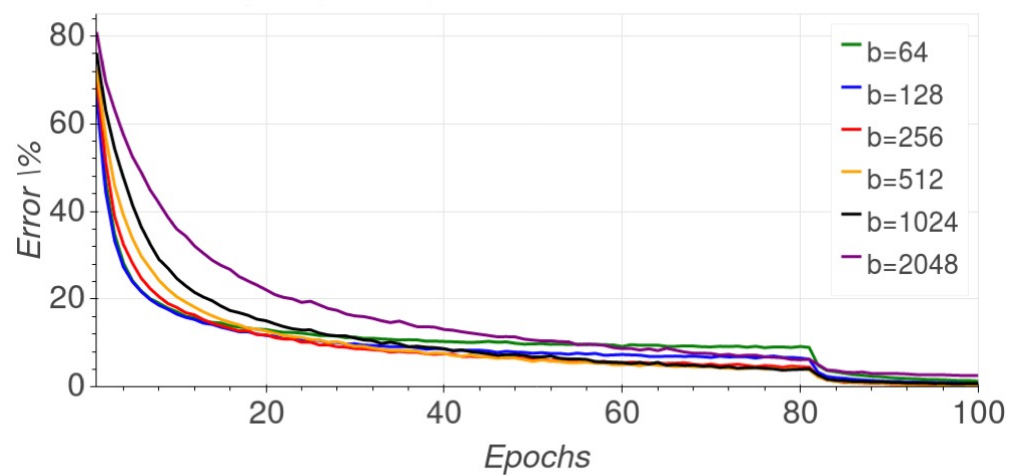
What's worse is that the easiest way to parallelize training is to make the minibatch even larger and distribute it across multiple GPUs and multiple nodes.

Minibatch sizes over 1024 aren't just bad for your health. They cause brain tumors.

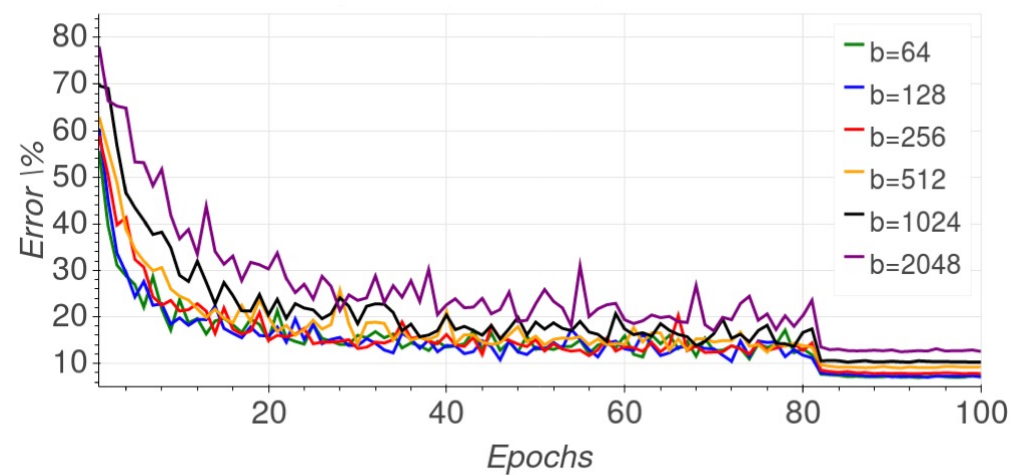
They learn quickly, but the wrong thing.



# Batch size

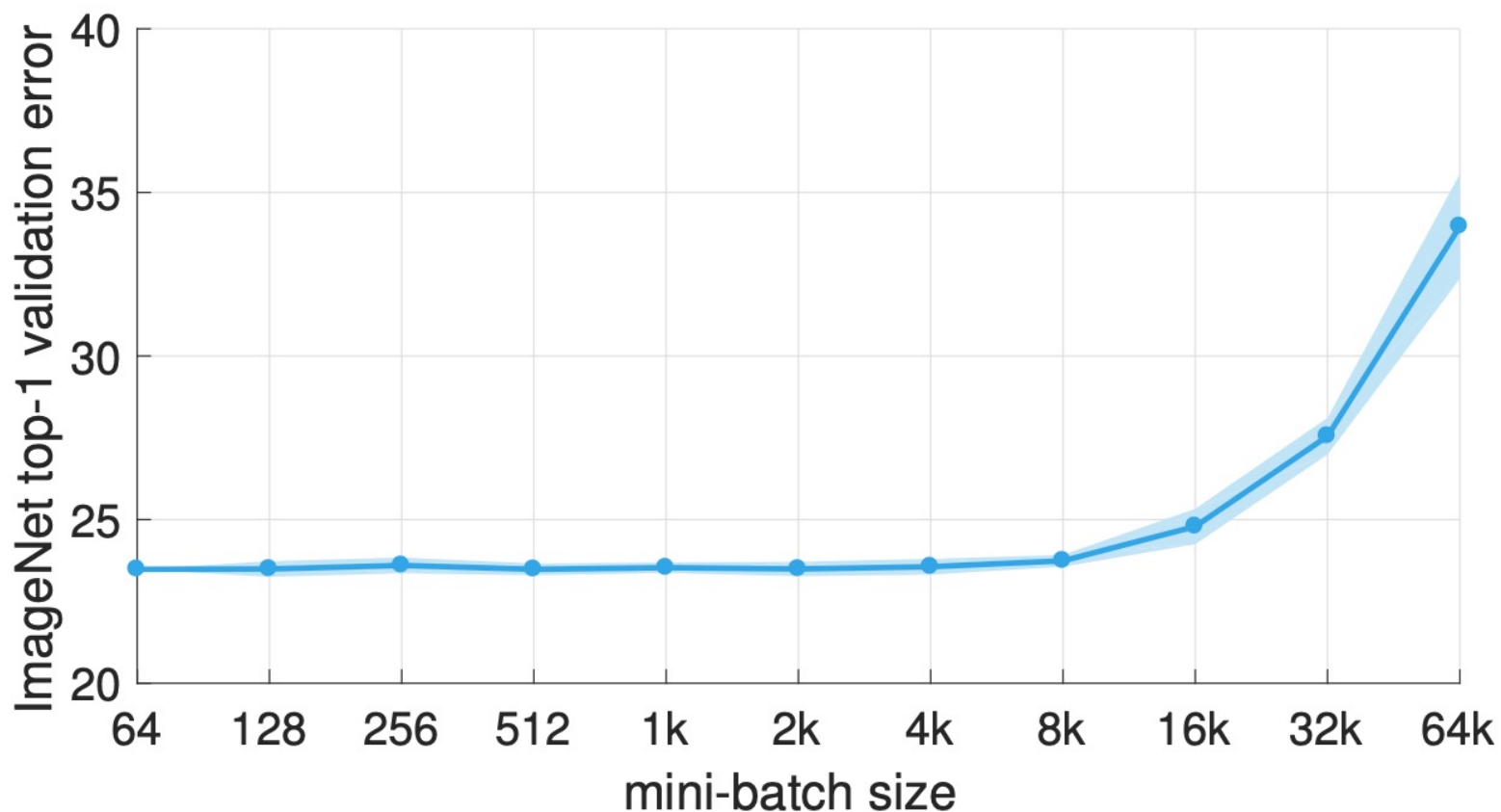


(a) Training error

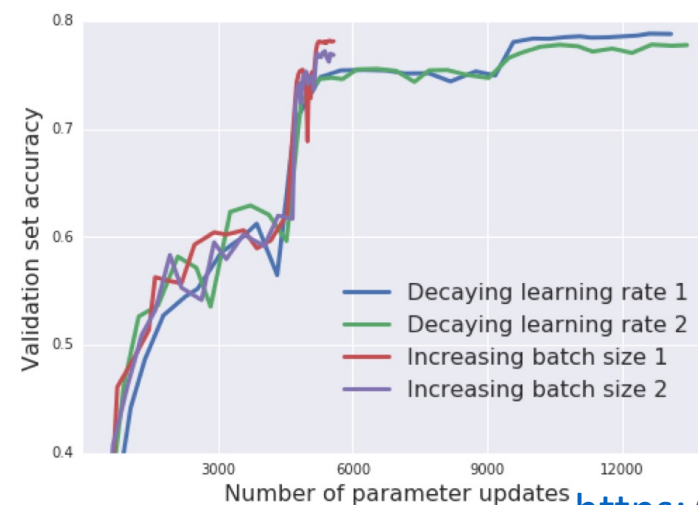
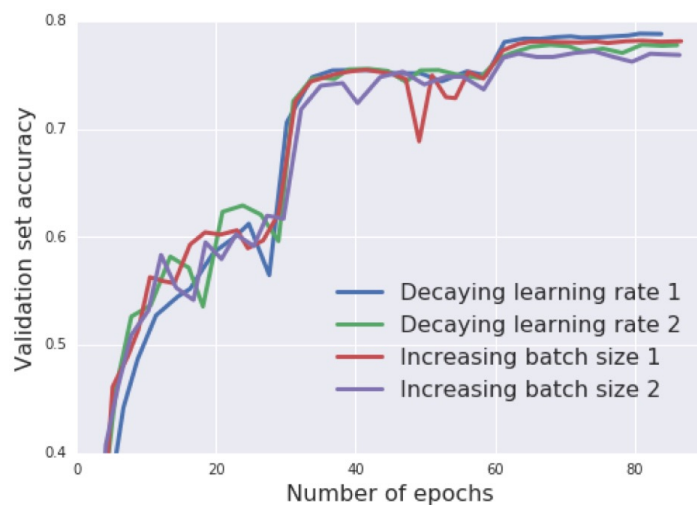
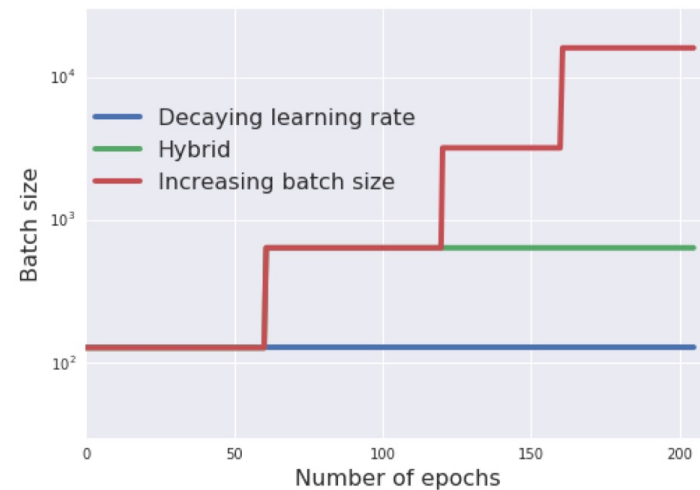
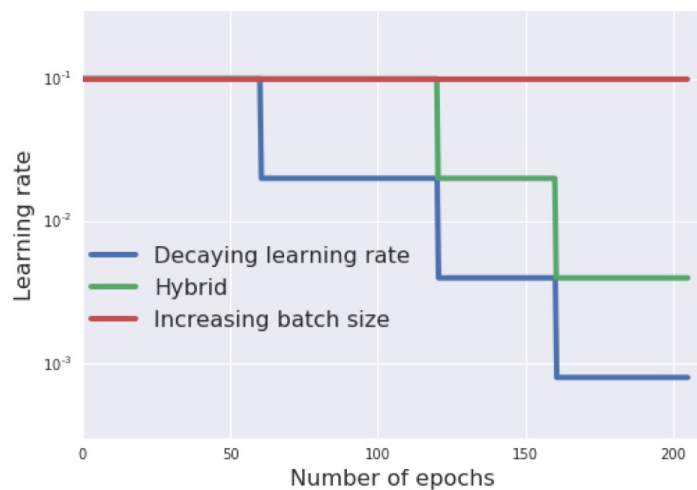


(b) Validation error

# Надо грамотно подбирать формулу для длины шага!



# Batch size

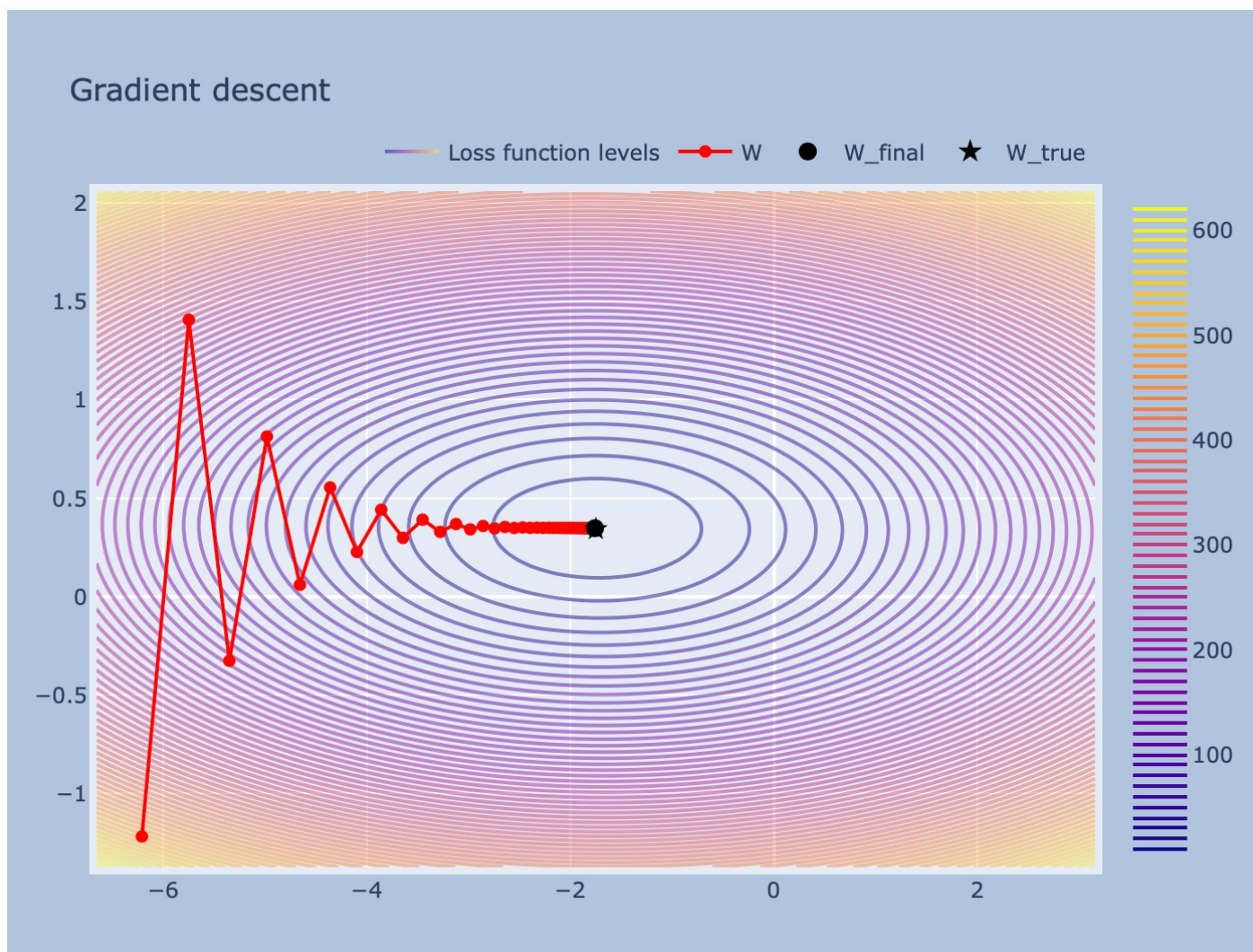


# Batch size

- От большого размера батча может быть польза
- Для этого надо грамотно подбирать длину шага и/или постепенно увеличить размер батча

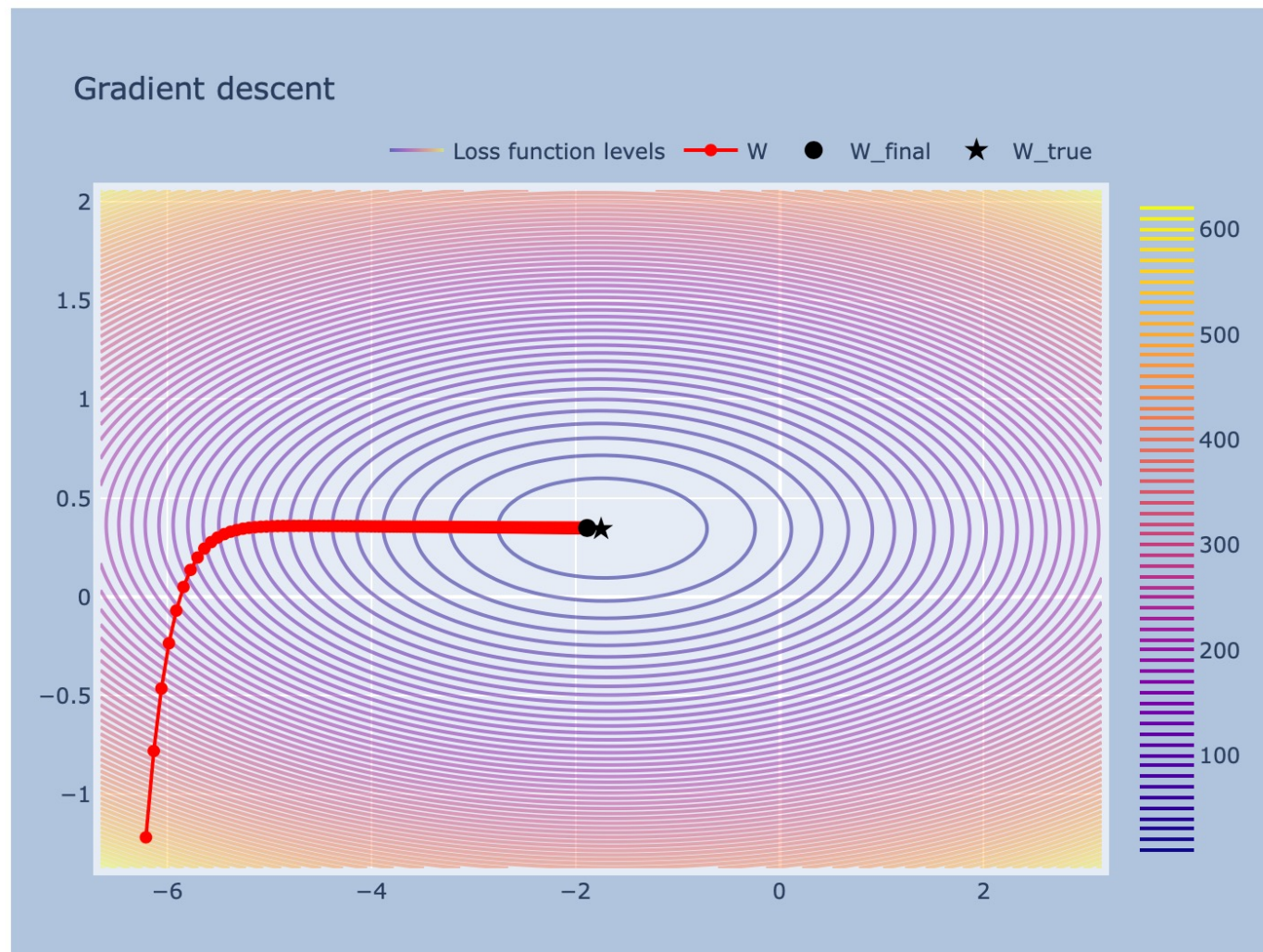
# Модификации градиентного спуска

# Проблемы





# Проблемы



# Проблемы

- Если у функции «вытянуты» линии уровня, то градиентный спуск требует аккуратного выбора длины шага и будет долго сходиться



# Momentum

$$h_t = \alpha h_{t-1} + \eta_t \nabla Q(w^{t-1})$$

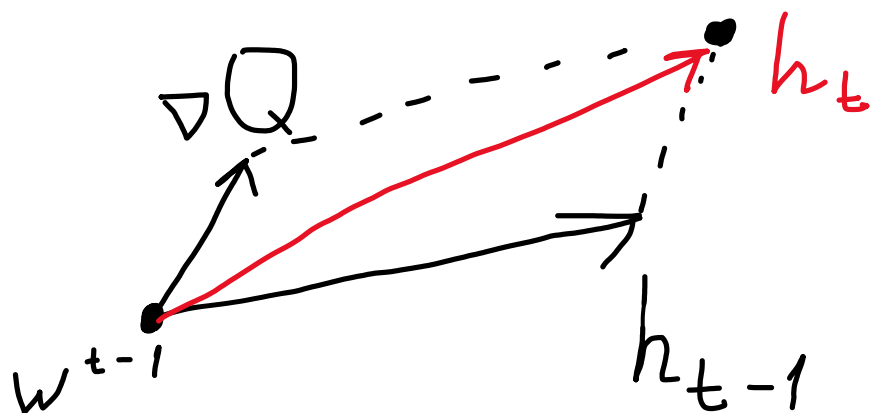
$$w^t = w^{t-1} - h_t$$

- $h_t$  — «инерция», усреднённое направление движения
- $\alpha$  — параметр затухания
- Как будто шарик, который катится в сторону минимума, очень тяжёлый

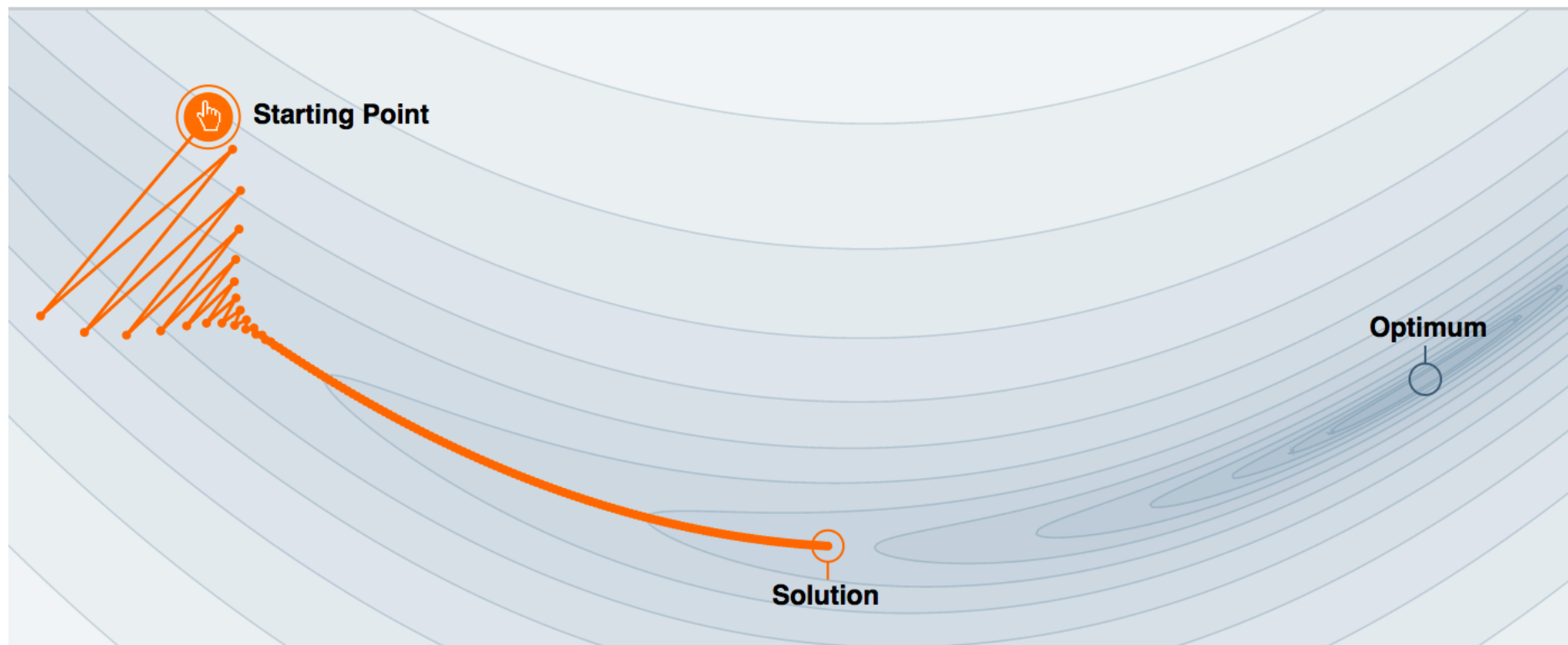
# Momentum

$$h_t = \alpha h_{t-1} + \eta_t \nabla Q(w^{t-1})$$

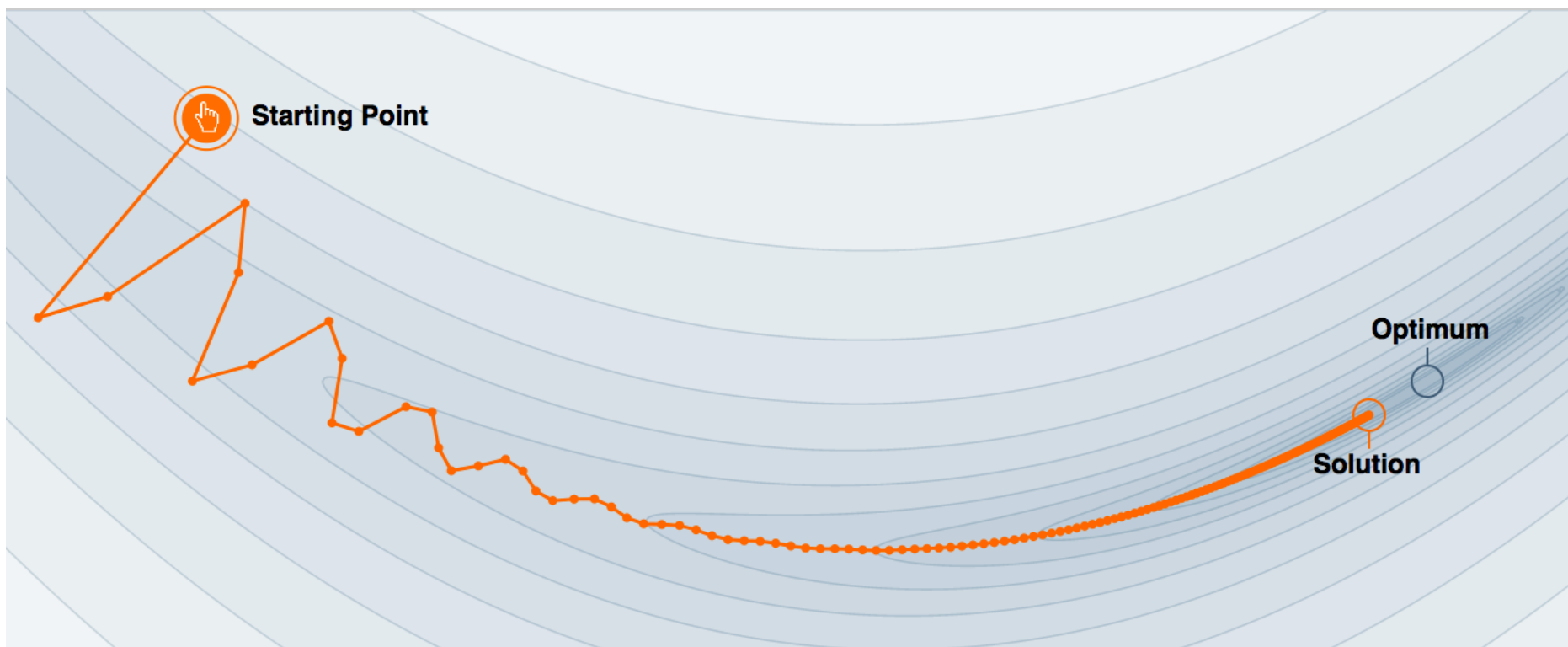
$$w^t = w^{t-1} - h_t$$



# Без инерции



# С инерцией

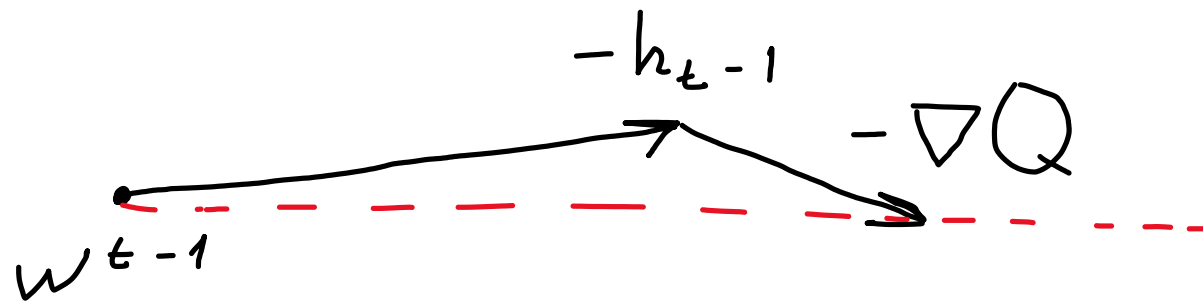
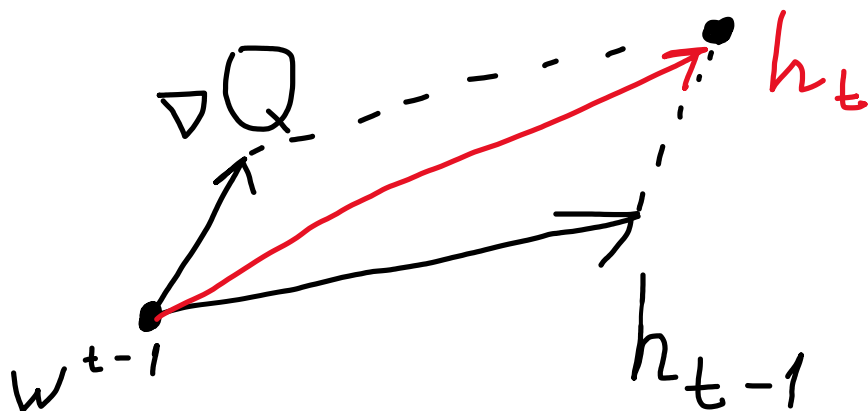


# Nesterov Momentum

$$h_t = \alpha h_{t-1} + \eta_t \nabla Q(w^{t-1} - \alpha h_{t-1})$$

$$w^t = w^{t-1} - h_t$$

- $w^{t-1} - \alpha h_{t-1}$  — неплохая оценка того, куда мы попадём на следующем шаге



# Проблема с разреженными данными

- Пример: модель над категориальными признаками
- Используем one-hot кодирование

	1		0
	0		0
	1		0
	1		0
	1		0
	0		0
	0		1

↑  
популярная категория

↑  
редкая категория

- Делаем стохастический градиентный спуск
- Через 7 шагов мы сделаем 4 обновления веса популярной категории и 1 обновление веса редкой категории

← тут уже медленные шаги

# Проблема с разреженными данными

- По разным параметрам мы движемся с разной скоростью
- Будет здорово это учитывать — иначе мы обучим разные параметры с разным качеством

# Проблема с разными масштабами

- Допустим, признаки имеют разный масштаб — от единиц до миллионов
- Тогда странно шагать по каждому параметру с одинаковой скоростью



# AdaGrad

$$G_j^t = G_j^{t-1} + (\nabla Q(w^{t-1}))_j^2$$

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

- По каждому параметру своя скорость
- $\eta_t$  можно зафиксировать
- Длина шага может убывать слишком быстро и привести к ранней остановке

# RMSProp

$$G_j^t = \alpha G_j^{t-1} + (1 - \alpha) (\nabla Q(w^{t-1}))_j^2$$

$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{G_j^t + \epsilon}} g_{tj}$$

- $\alpha$  можно взять около 0.9
- Скорость зависит только от недавних шагов

# Adam

$$m_j^t = \frac{\beta_1 m_j^{t-1} + (1 - \beta_1)(\nabla Q(w^{t-1}))_j}{1 - \beta_1^t}$$

$$v_j^t = \frac{\beta_2 v_j^{t-1} + (1 - \beta_2)(\nabla Q(w^{t-1}))_j^2}{1 - \beta_2^t}$$

$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{v_j^t} + \epsilon} m_j^t$$

- Рекомендации:  $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

# Adam

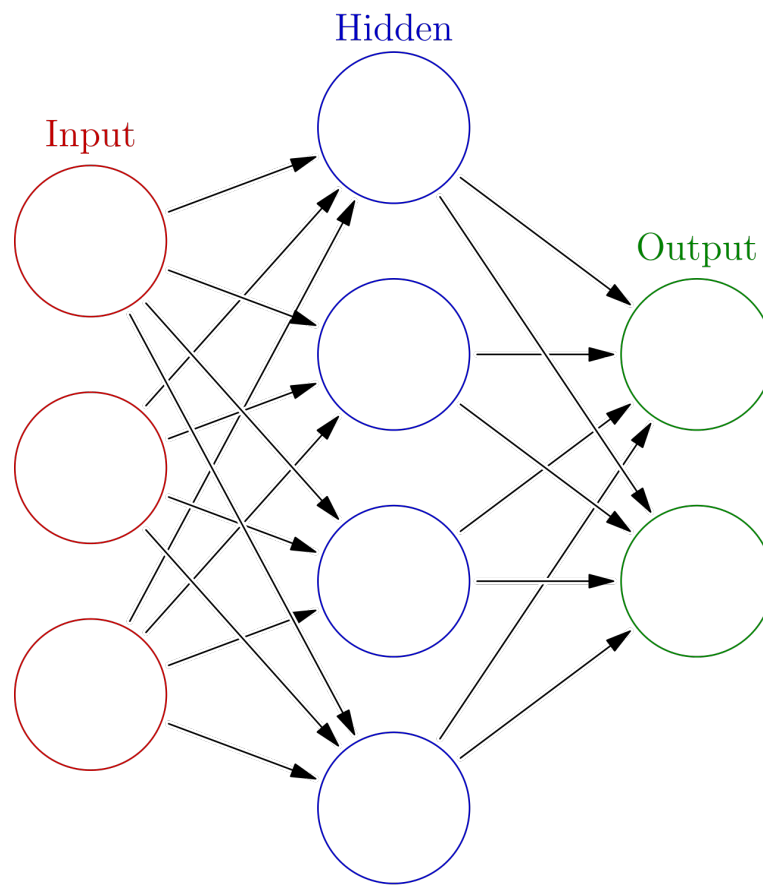
- Совмещает в себе идеи из метода инерции и RMSProp

# Dropout

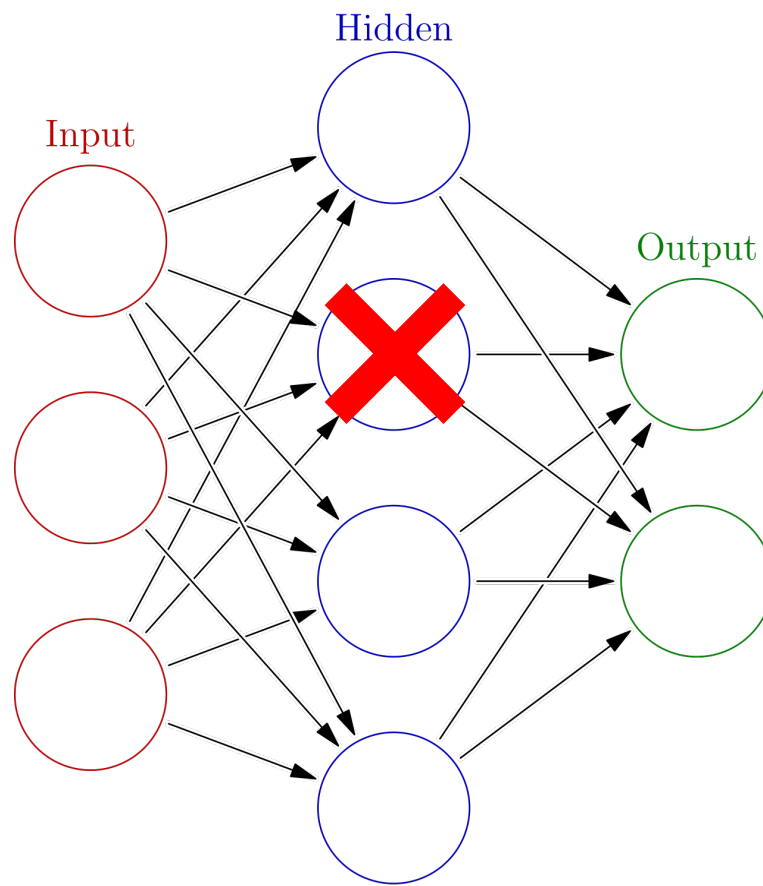
# Борьба с переобучением

- Сокращение числа параметров (свёрточные слои помогают с этим)
- Регуляризация
- Можно как-то ещё мешать модели подгоняться под обучающую выборку

# Dropout

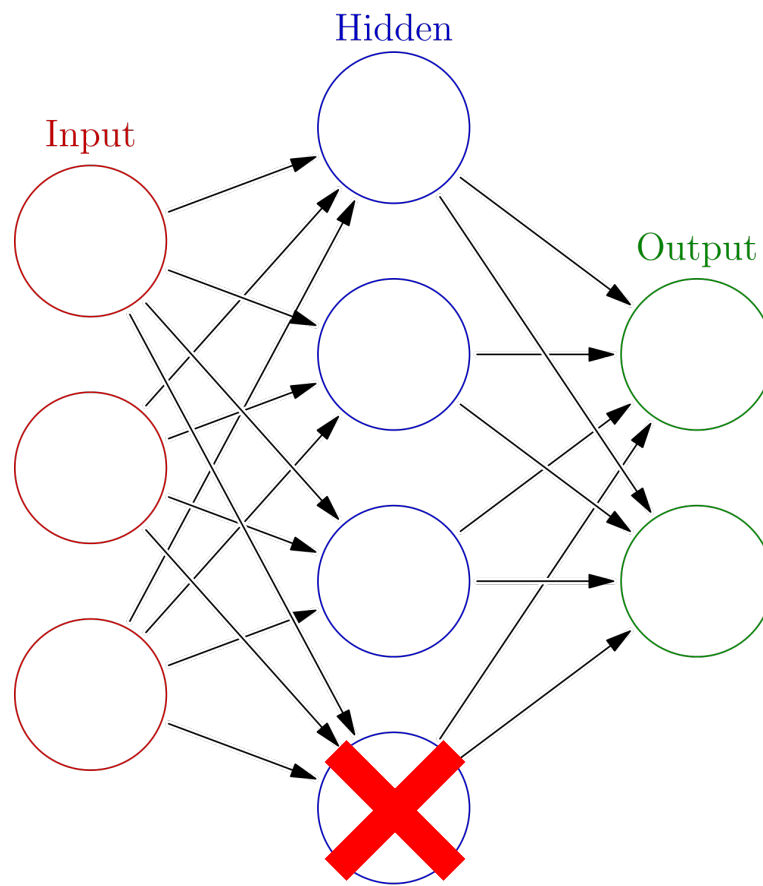


# Dropout

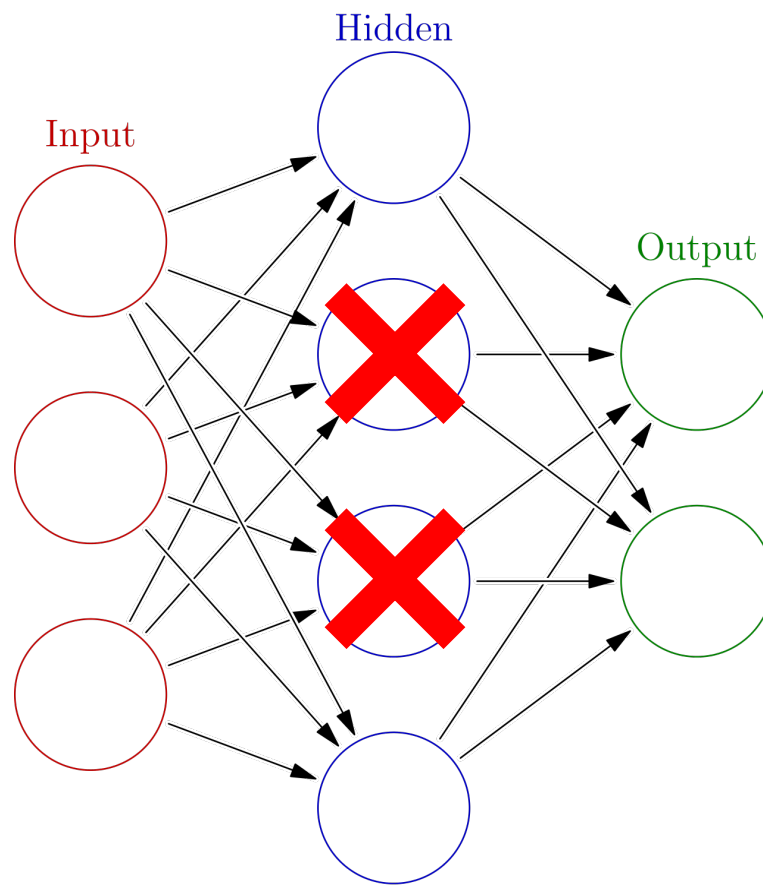




# Dropout



# Dropout



# Dropout

- Можно определить как слой  $d(x)$
- Параметров нет, единственный гиперпараметр —  $p$  (вероятность удаления нейрона)
- На этапе обучения:

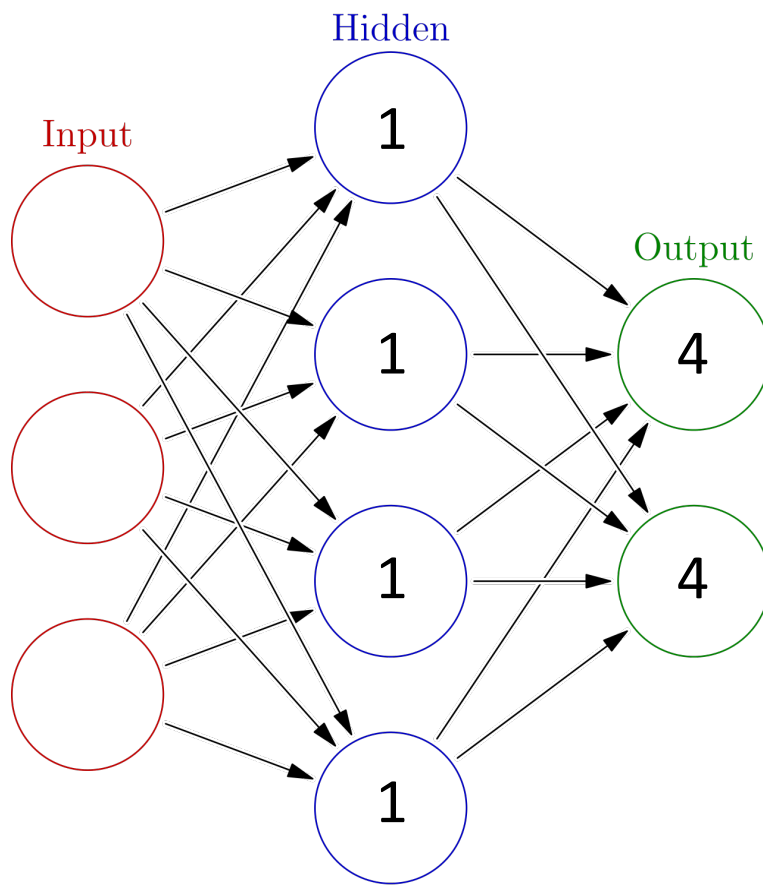
$$d(x) = \frac{1}{1-p} m \circ x$$

( $m$  — вектор того же размера, что и  $x$ , элемент берется из распределения  $\text{Ber}(p)$ )

- Деление на  $p$  — для сохранения суммарного масштаба выходов

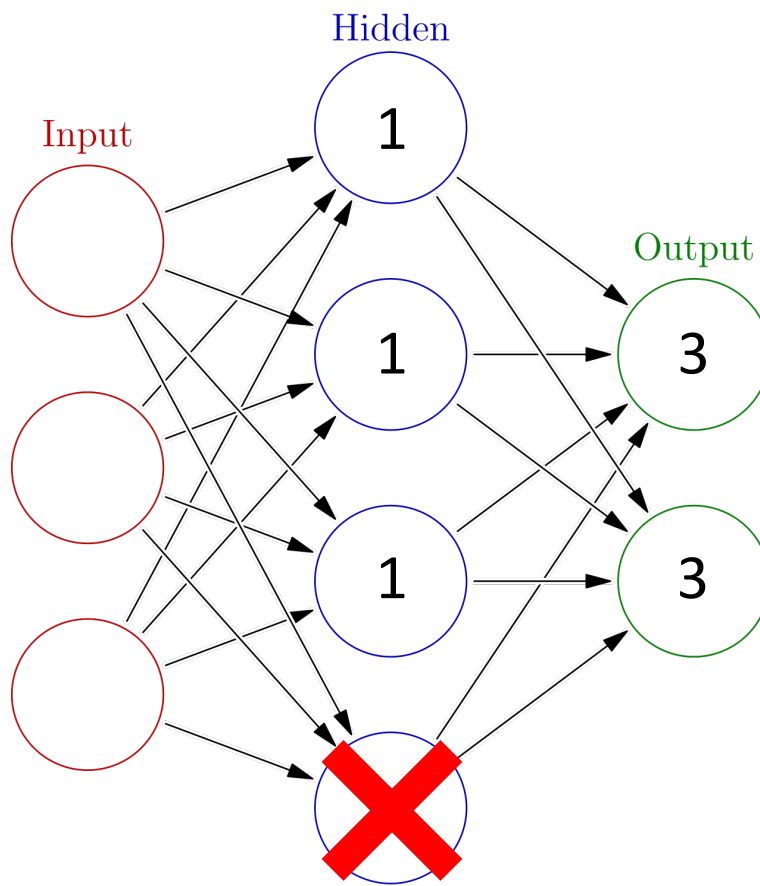
# Dropout

Пусть все веса единичные



# Dropout

Пусть все веса единичные



Надо компенсировать снижение масштаба суммы выходов!

# Dropout

- На этапе обучения:

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

- На этапе применения:

$$d(x) = x$$

В оригинальной статье нет нормировки на этапе обучения, но есть домножение на  $p$  на этапе применения

Вариант на слайде — inverted dropout (чуть меньше операций во время применения сети)

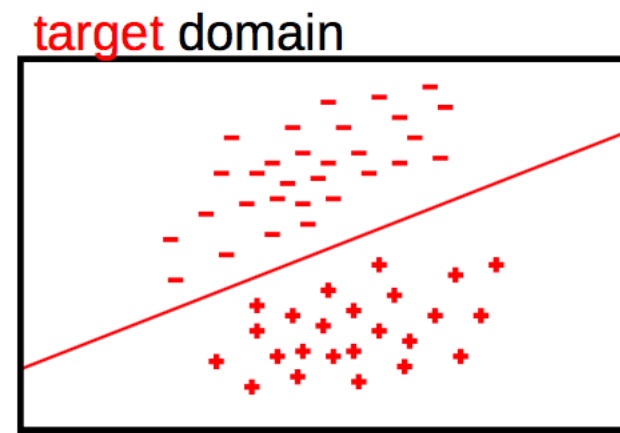
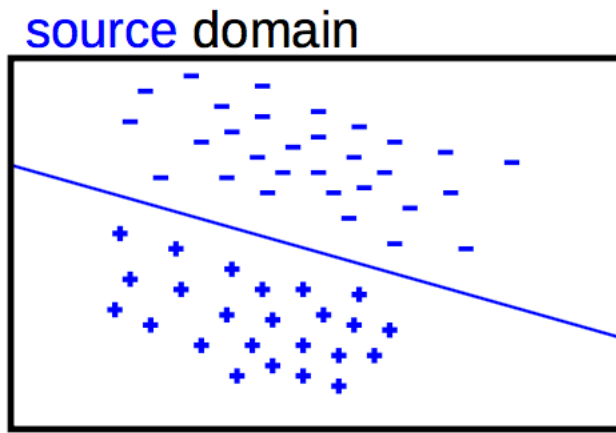
# Dropout

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

Нормализации



# Covariate shift



# Covariate shift

- В классическом машинном обучении — изменение распределения данных
- Много методов решения

# Domain adaptation

- Объекты по-разному распределены на обучении и на контроле
- Идея: взвешивать объекты при обучении

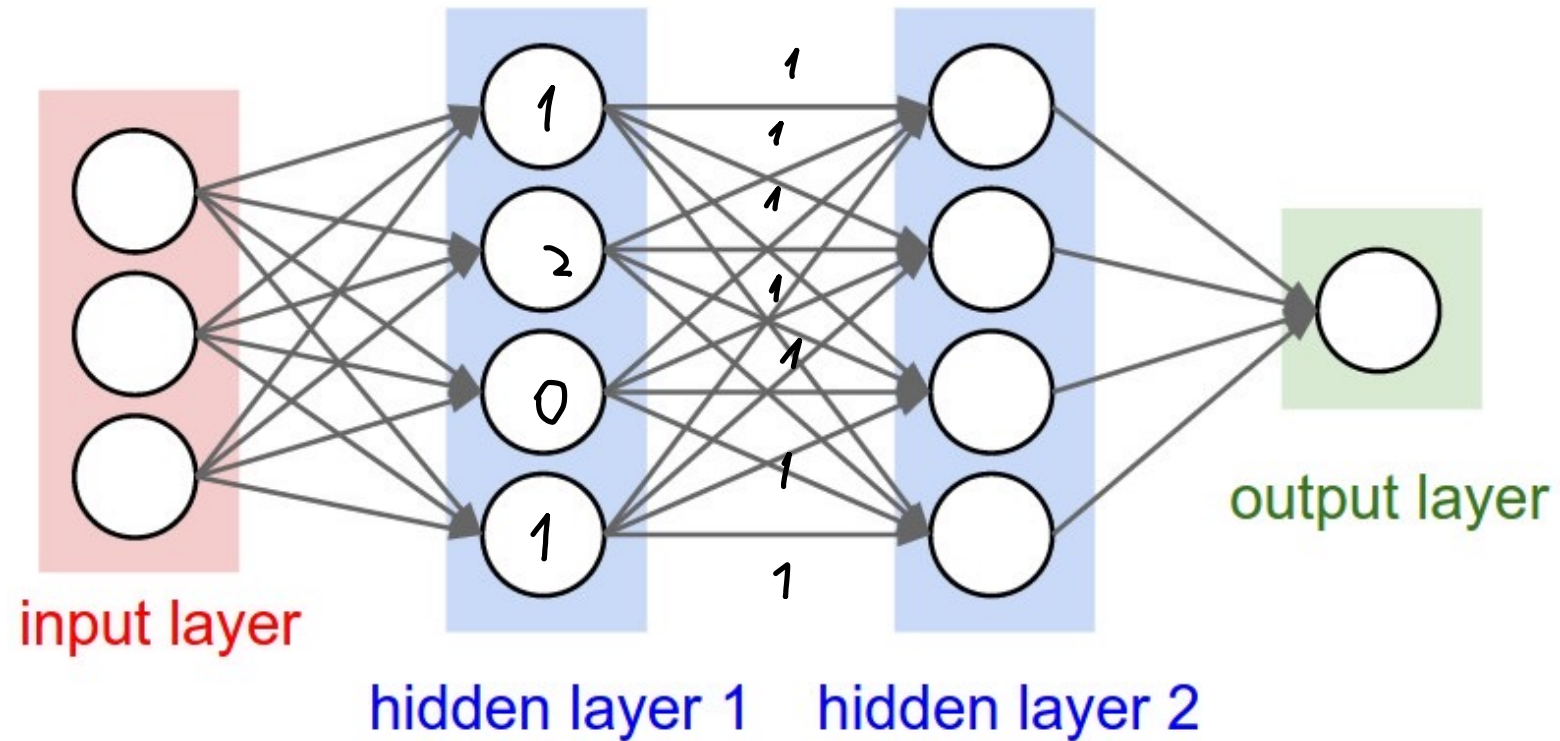
$$\sum_{i=1}^{\ell} s_i (a(x_i) - y_i)^2 \rightarrow \min$$

- Большие веса будем ставить объектам, которые похожи на объекты из тестовой выборки

# Internal covariate shift

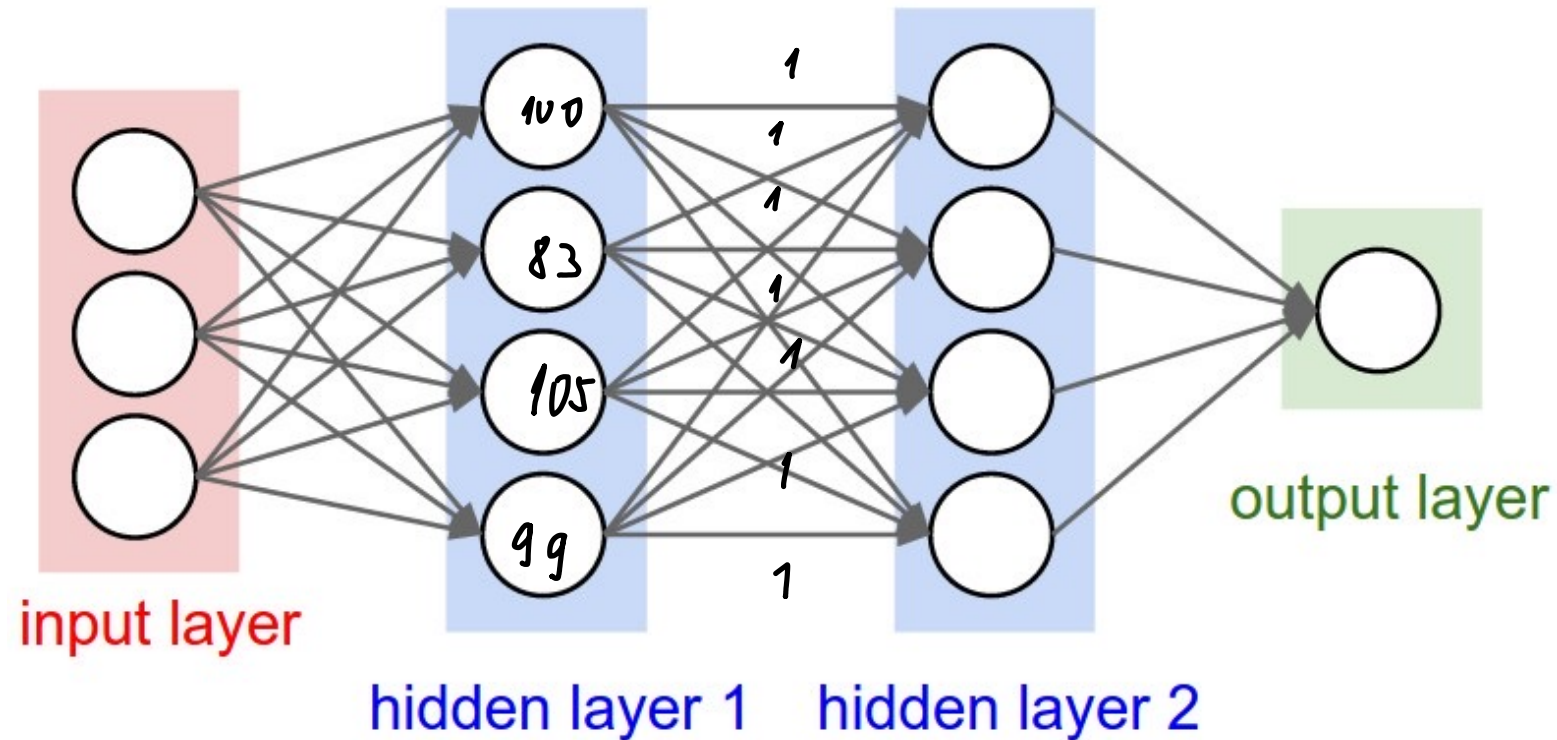
- В нейронной сети каждый слой обучается на выходах предыдущих слоёв
- Если слой в начале сильно меняется, то все следующие слои надо переделывать

# Internal covariate shift



# Internal covariate shift

Допустим, веса первого слоя сильно поменялись после градиентного шага



# Internal covariate shift

- Идея: преобразовывать выходы слоёв так, чтобы они гарантированно имели фиксированное распределение


# Batch Normalization

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

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

$$\sigma_B^2 = \frac{1}{n} \sum_{j=1}^n (x_{B,j} - \mu_B)^2$$

покоординатно



$x_{B,j}$  —  $j$ -й объект в батче  $B$



# Batch Normalization

- Отмасштабируем все выходы:

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

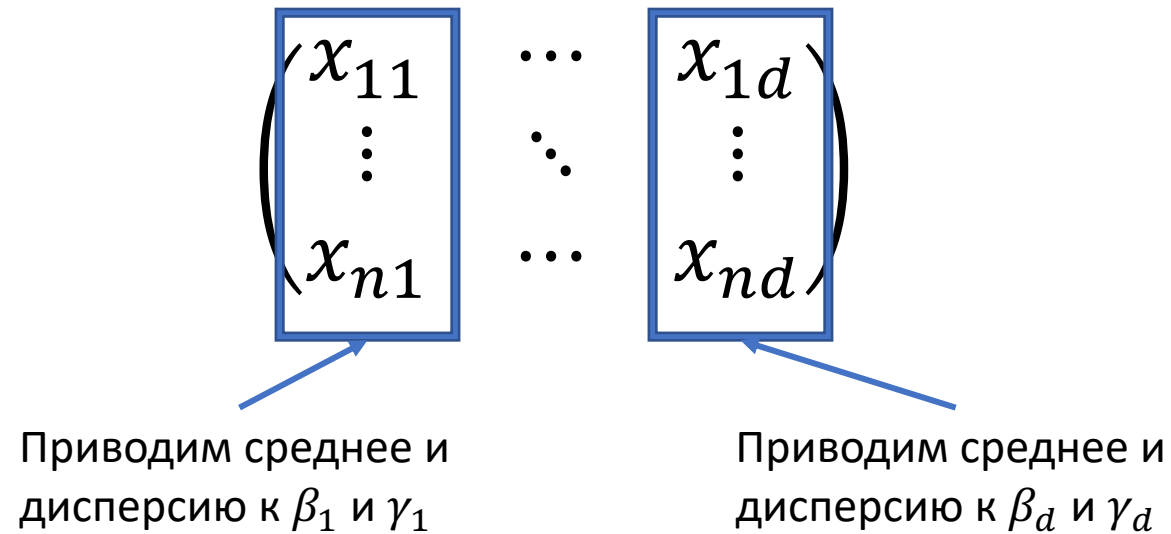
- Зададим нужные нам среднее и дисперсию:

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



обучаемые параметры (векторы, размерность  
равна размерности входных векторов)

# Batch Normalization



- $n$  — размер батча
- $d$  — размерность входного вектора

# Batch Normalization

Важно: после BatchNorm среднее и дисперсия каждого выхода зависят только от параметров нормализации, но не от параметров прошлых слоёв!

# Batch Normalization

Во время применения нейронной сети:

- Те же самые формулы, но вместо  $\mu_B$  и  $\sigma_B^2$  используем их средние значения по всем батчам

# Batch Normalization

- Обычно вставляется между полносвязным/свёрточным слоем и нелинейностью
- Позволяет увеличить длину шага в градиентном спуске
- Не факт, что действительно устраняет covariance shift