

Лекция 3 (часть 1): Современные техники глубокого обучения

Автор: Сергей Вячеславович Макрушин e-mail: SVMakrushin@fa.ru (<mailto:SVMakrushin@fa.ru>)

Финансовый университет, 2021 г.

При подготовке лекции использованы материалы:

- ...
- V 0.4 04.02.2021
- V 0.5 09.03.2021 (старое название: TCN20_INNp1b_v4)

In [1]:

```
# загружаем стиль для оформления презентации
from IPython.display import HTML
from urllib.request import urlopen
html = urlopen("file:./lec_v2.css")
HTML(html.read().decode('utf-8'))
```

Out[1]:

Вторая весна ИИ и глубокое обучение

- [к оглавлению](#)

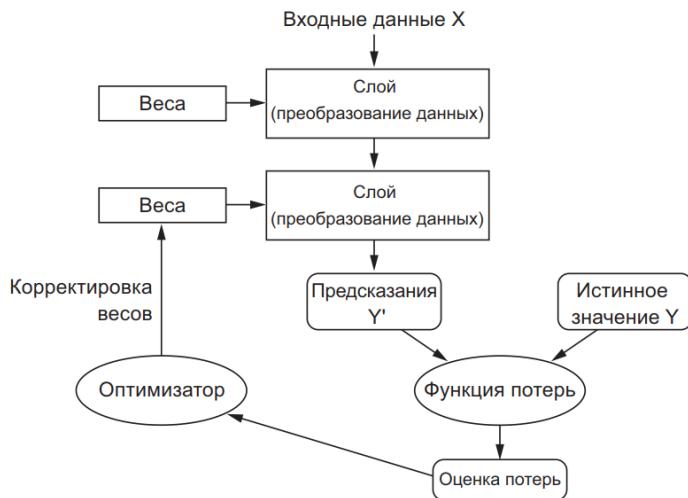
Разделы:

- [Введение](#)
- [улучшенные подходы к регуляризации](#)
- [улучшенные схемы инициализации весов](#)
- [улучшенные функции активации](#)
- [улучшенные схемы оптимизации](#) -
- [к оглавлению](#)

Алгоритмические достижения в области глубокого обучения

Кроме оборудования и данных, до конца 2000-х нам не хватало надежного способа обучения очень глубоких нейронных сетей, как результат:

- нейронные сети оставались очень неглубокими, имеющими один или два слоя представления;
- \Rightarrow они не могли противостоять более совершенным поверхностным методам (методу опорных векторов и случайные леса). Основная **проблема** заключалась в **распространении градиента через глубокие пакеты слоев**. Сигнал обратной связи, используемый для обучения нейронных сетей, затухает по мере увеличения количества слоев.



Принципиальная логика обучения нейронной сети

В районе 2010 г. появились некоторые простые, но важных алгоритмические усовершенствования, позволившие улучшить распространение градиента:

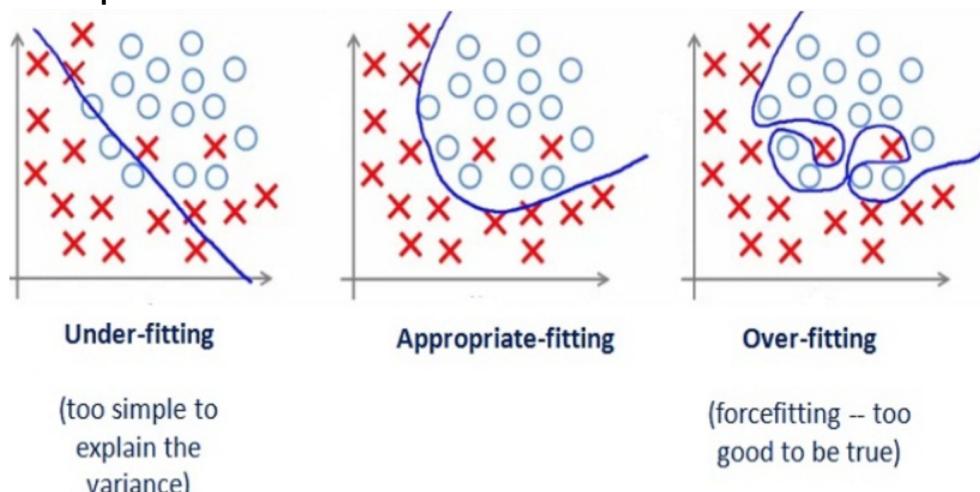
- **улучшенные подходы к регуляризации**
- **улучшенные схемы инициализации весов**
- **улучшенные функции активации**
- **улучшенные схемы оптимизации** (такие как RMSProp и Adam)

В последнее время были открыты еще более совершенные способы распространения градиента, с применением которых появилась возможность обучать с нуля модели с тысячами слоев в глубину. В частности:

- пакетная нормализация
- обходные связи
- отделимые свертки

Улучшенные подходы к регуляризации

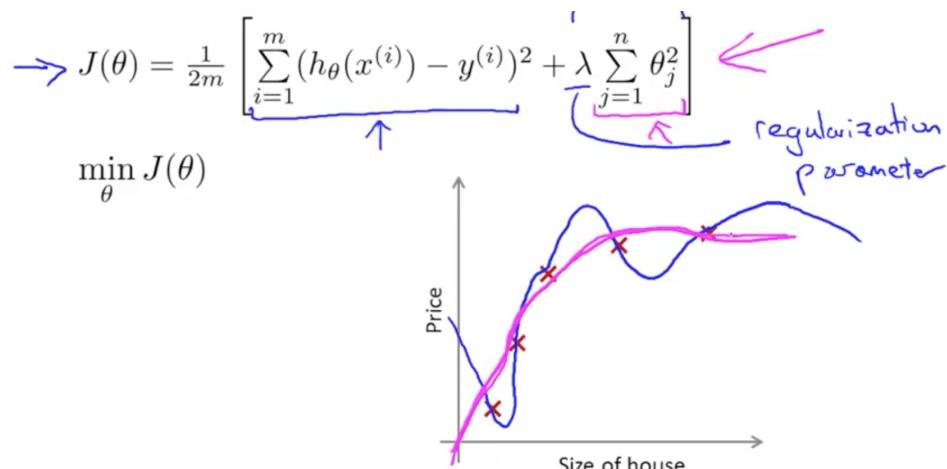
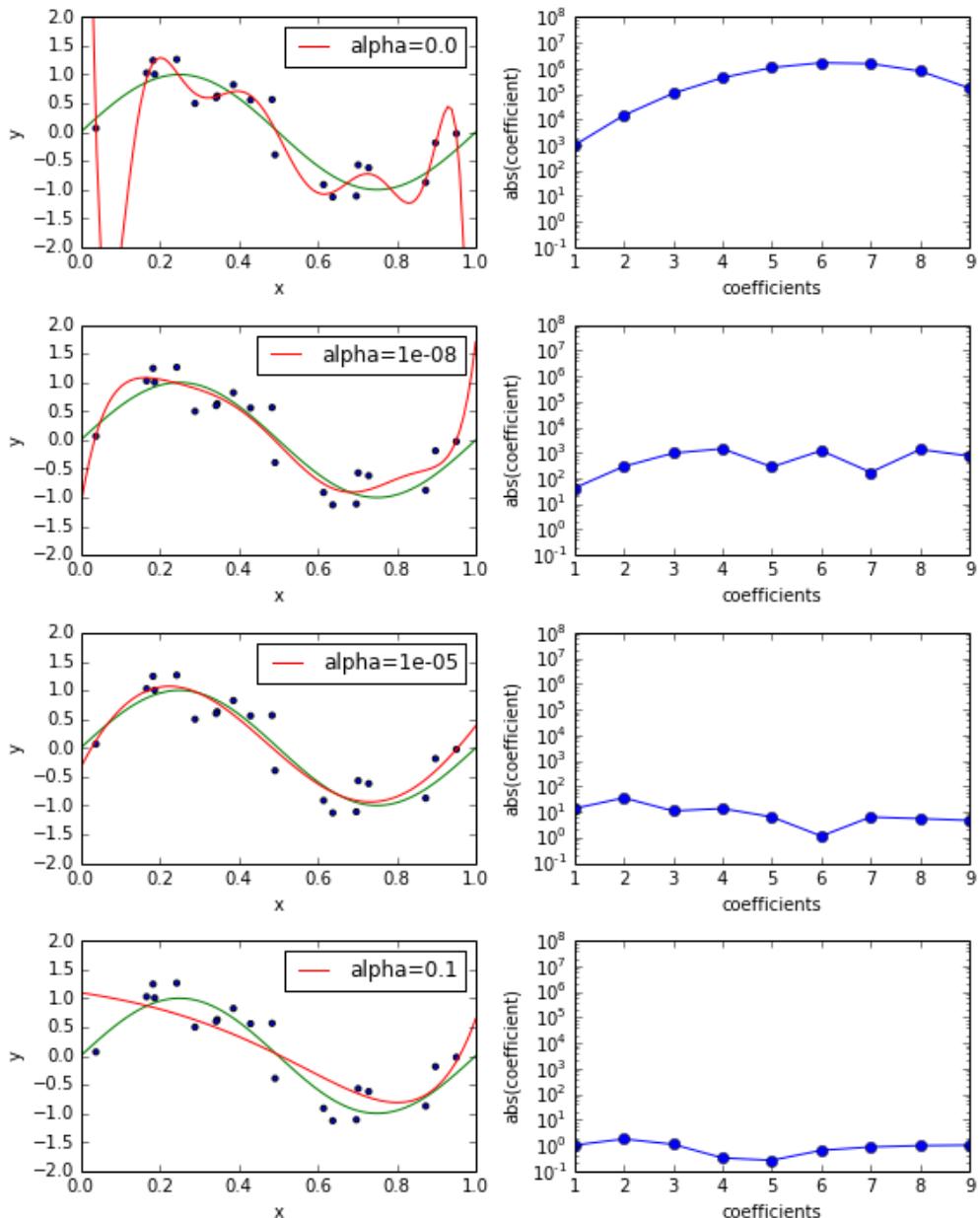
Регуляризация в нейронных сетях



Проблема переобучения модели

- модель, у которой слишком много свободных параметров, **плохо обобщается**: то есть слишком близко «облизывает» точки из тренировочного множества и в результате недостаточно хорошо предсказывает нужные значения в новых точках.

- В современных нейронных сетях огромное число параметров (даже не самая сложная архитектура может содержать миллионы весов) \Rightarrow надо регуляризовать параметры!



Принцип регуляризации параметров модели

- Наиболее распространенные регуляризаторы:
 - L_2 -регуляризатор: сумма квадратов весов $\lambda \sum_w w^2$
 - L_1 -регуляризатор: сумма модулей весов $\lambda \sum_w |w|$

- В теории нейронных сетей такая регуляризация называется сокращением весов (weight decay), потому что действительно приводит к уменьшению их абсолютных значений.
- в Keras есть возможность для каждого слоя добавить регуляризатор на три вида связей:
 - kernel_regularizer — на матрицу весов слоя;
 - bias_regularizer — на вектор свободных членов;
 - activity_regularizer — на вектор выходов.

Пример L1 и L2 регуляризации в Keras:

```
model.add(Dense(256, input_dim=32,  
kernel_regularizer=regularizers.l1(0.001),  
bias_regularizer=regularizers.l2(0.1),  
activity_regularizer=regularizers.l2(0.01)))
```

Пример использования L1 и L2 регуляризации в PyTorch:

```

import torch
from torch.autograd import Variable
from torch.nn import functional as F


class MLP(torch.nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.linear1 = torch.nn.Linear(128, 32)
        self.linear2 = torch.nn.Linear(32, 16)
        self.linear3 = torch.nn.Linear(16, 2)

    def forward(self, x):
        layer1_out = F.relu(self.linear1(x))
        layer2_out = F.relu(self.linear2(layer1_out))
        out = self.linear3(layer2_out)
        return out, layer1_out, layer2_out

batchsize = 4
lambda1, lambda2 = 0.5, 0.01

model = MLP()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

# usually following code is Looped over all batches
# but let's just do a dummy batch for brevity

inputs = Variable(torch.rand(batchsize, 128))
targets = Variable(torch.ones(batchsize).long())

optimizer.zero_grad()
outputs, layer1_out, layer2_out = model(inputs)
cross_entropy_loss = F.cross_entropy(outputs, targets)

all_linear1_params = torch.cat([x.view(-1) for x in model.linear1.parameters()])
all_linear2_params = torch.cat([x.view(-1) for x in model.linear2.parameters()])
l1_regularization = lambda1 * torch.norm(all_linear1_params, 1)
l2_regularization = lambda2 * torch.norm(all_linear2_params, 2)

loss = cross_entropy_loss + l1_regularization + l2_regularization
loss.backward()
optimizer.step()

```

Регуляризация с помощью ранней остановки (early stopping)

- Отложим часть тренировочного набора (назовем отложенную часть **валидационным множеством**).
- При обучении на тренировочном множестве будем вычислять ошибку и на валидационном множестве
- Можно предположить, что **ошибка на валидационном множестве будет хорошо оценивать ошибку и на новых точках** (тестовом множестве), ведь она взята из данных той же природы, но тех, на которых мы не обучались.

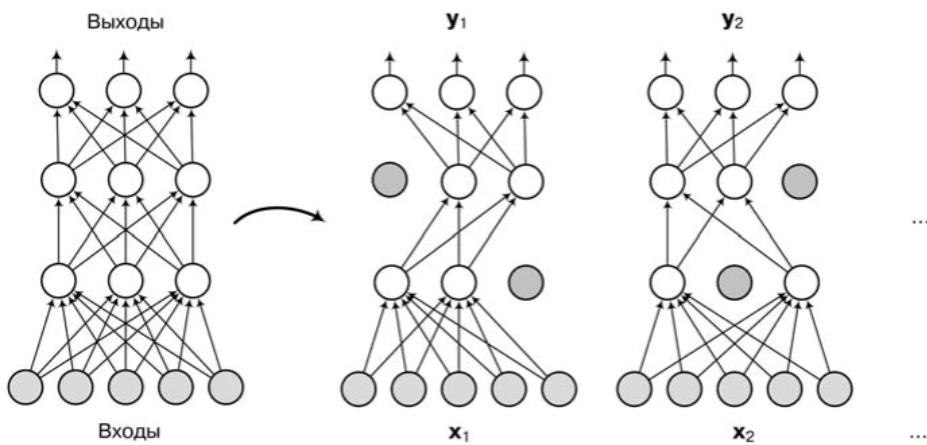
- Остановить обучение нужно будет не тогда, когда сеть прийдет в локальный оптимум для тренировочного множества, а тогда, **когда начнет ухудшаться ошибка на валидационном множестве**.

Регуляризации нейронных сетей с помощью дропаута

- **Регуляризация с помощью дропаута** (dropout regularization) - один из важнейших методов регуляризации нейронных сетей обеспечивший революцию глубокого обучения

Идея метода (очень простая!):

- Для каждого нейрона (кроме самого последнего, выходного слоя) установим некоторую вероятность p , с которой он будет выброшен из сети.
- Алгоритм обучения меняется таким образом:
 - на каждом новом тренировочном примере x мы сначала для каждого **разыгрываем вероятность p** и в зависимости от результата либо используем нейрон как обычно, **либо устанавливаем его выход всегда строго равным нулю** (вероятность этого события $1 - p$).
 - **Дальше все происходит без изменений**; ноль на выходе приводит к тому, что нейрон фактически выпадает из графа вычислений: и прямое вычисление, и обратное распространение градиента останавливаются на этом нейроне и дальше не идут.
 - Для применения обученной сети **используются все нейроны** в конфигурации, которая была до применения дропаута, но **выход каждого нейрона умножается на вероятность p** (с которой нейрон оставляли при обучении).
- Для очень широкого спектра архитектур и приложений замечательно подходит $p = 1/2$



Пример дропаута в трехслойной нейронной сети: на каждом новом тренировочном примере структура сети изменяется

Пример дропаута

- Практика обучения нейронных сетей показывает, что дропаут действительно дает очень серьезные улучшения в качестве обученной модели
- Дропаут — это метод добиться **усреднения огромного числа моделей** (до 2^N возможных моделей, N — число нейронов, которые подвергаются дропауту). Он эквивалентен усреднению всех моделей, которые получались на каждом шаге случайным выбрасыванием отдельных нейронов.

Пример использования Dropout в Keras:

```

def create_model():
    # create model
    model = Sequential()
    # Dropout:
    model.add(Dropout(0.2, input_shape=(60,)))
    model.add(Dense(60, kernel_initializer='normal', activation='relu', kernel_constraint=maxnorm(3)))
    model.add(Dense(30, kernel_initializer='normal', activation='relu', kernel_constraint=maxnorm(3)))
    model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
    # Compile model
    sgd = SGD(lr=0.1, momentum=0.9, decay=0.0, nesterov=False)
    model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
    return model

```

Пример использования Dropout в PyTorch:

```

def __init__(self, rnn_type, input_size, node_fdim, hidden_size, depth, dropout):
    super(MPNEncoder, self).__init__()
    self.hidden_size = hidden_size
    self.input_size = input_size
    self.depth = depth
    self.W_o = nn.Sequential(
        nn.Linear(node_fdim + hidden_size, hidden_size),
        nn.ReLU(),
        nn.Dropout(dropout)
    )

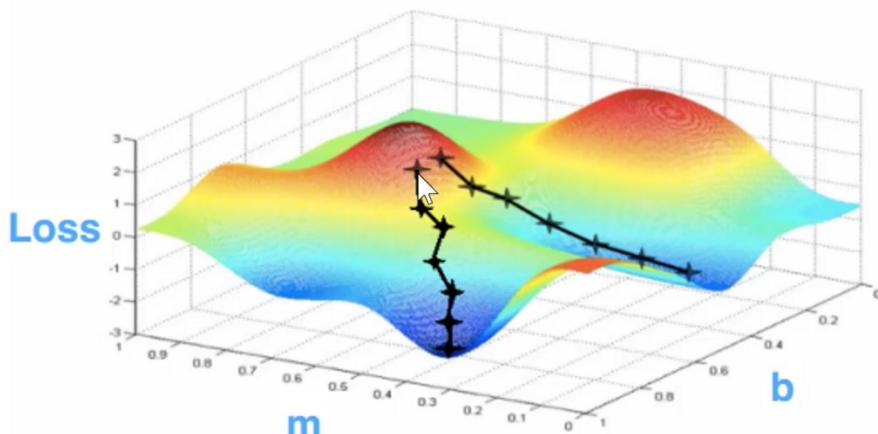
    if rnn_type == 'GRU':
        self.rnn = GRU(input_size, hidden_size, depth)
    elif rnn_type == 'LSTM':
        self.rnn = LSTM(input_size, hidden_size, depth)
    else:
        raise ValueError('unsupported rnn cell type ' + rnn_type)

```

Улучшенные схемы инициализации весов

- Обучение сети — сложная задача оптимизации в пространстве очень высокой размерности, которая фактически решается методами локального поиска.
- Для таких задач один из ключевых вопросов: **где начинать этот локальный поиск?**

$$f(x) = \text{nonlinear function of } x$$



Пример работы градиентного спуска для функции двух переменных

- Качество начального приближения принципиально влияет на получаемые в результате локальные оптимумы.
- Хорошая инициализация весов может позволить нам обучать глубокие сети:
 - лучше (в смысле метрик качества)
 - быстрее (в смысле числа требующихся обновлений весов, т.е. числа итераций, т.е. времени обучения)

Первая идея, которая привела к большим успехам в этом направлении: **предобучение без учителя** (unsupervised pretraining):

- отдельные слои глубокой сети последовательно обучаются без учителя
- затем веса полученных слоев считаются начальным приближением и дообучаются уже на размеченном наборе данных
- сначала основным инструментом для предобучения без учителя в стали так называемые **ограниченные машины Больцмана**
- затем для этого стали использоваться **автокодировщики**

Основные принципы использующиеся при предобучении:

1. Предобучение слоев происходит последовательно, от нижних к верхним.
 - позволяет избежать проблемы затухающих градиентов
 - существенно уменьшает объем вычислений на каждом этапе
2. Предобучение протекает без учителя, то есть без учета имеющихся размеченных данных.
 - это часто позволяет существенно расширить обучающую выборку (например, собрать миллионы изображений из интернета без описания намного проще, чем собрать даже тысячу правильно размеченных изображений).
3. В результате предобучения получается модель, которую затем нужно дообучить на размеченных данных.
 - модели, обученные таким образом, в конечном счете стабильно сходятся к существенно лучшим решениям, чем при случайной инициализации.

Но:

- сейчас на практике предварительное послойное обучение проводится редко, т.к. был найдены более простой и хорошо мотивированный способ инициализации весов, позволяющий существенно ускорить обучение и улучшить качество, его часто называют инициализацией Ксавье (Xavier initialization).

Инициализация Ксавье

Общий вид перцептрана:

$$\hat{y} = f(z) = f(w_0 + w_1 x_1 + \dots + w_n x_n) = f(\mathbf{w}^T \mathbf{x})$$
, где $\mathbf{x} = (1, x_1, \dots, x_n)$ Т.е. $z = \mathbf{w}^T \mathbf{x}$

Т.е. дисперсия $\text{Var}(z)$ не зависит от свободного члена $w_0 b$ и выражается через дисперсии $\mathbf{x}' = (x_1, \dots, x_n)$ и $\mathbf{w}' = (w_1, \dots, w_n)$.

Для $z_i = w_i \cdot x_i$, в предположении о том, что w_i и x_i независимы (что вполне естественно), мы получим дисперсию:

$$\begin{aligned}\text{Var}(z_i) &= \text{Var}(w_i \cdot x_i) = \mathbb{E}[x_i^2 w_i^2] - (\mathbb{E}[x_i w_i])^2 = \mathbb{E}[x_i^2] \mathbb{E}[w_i^2] - \mathbb{E}[w_i]^2 \mathbb{E}[x_i]^2 = (\text{Var}(x_i) + \mathbb{E}[x_i]^2) \\ &= \mathbb{E}[x_i]^2 \text{Var}(w_i) + \mathbb{E}[w_i]^2 \text{Var}(x_i) + \text{Var}(x_i) \text{Var}(w_i)\end{aligned}$$

Если мы используем симметричные функции активации и случайно инициализируем веса со средним значением, равным нулю, то первые два члена последнего выражения оказываются тоже равными нулю, а значит:

$$\text{Var}(z_i) = \text{Var}(x_i) \text{Var}(w_i)$$

Если теперь мы предполагаем, что как x_i , так и w_i инициализируются из одного и того же распределения, причем независимо друг от друга (это сильное предположение, но в данном случае вполне естественное), мы получим:

$$\text{Var}(z) = \text{Var}\left(\sum_{i=1}^{n_{out}} z_i\right) = \sum_{i=1}^{n_{out}} \text{Var}(x_i w_i) = n_{out} \text{Var}(x_i) \text{Var}(w_i),$$

где n_{out} — число нейронов последнего слоя. Другими словами, дисперсия выходов пропорциональна дисперсии входов с коэффициентом $n_{out} \text{Var}(w_i)$.

Ранее стандартным эвристическим способом случайно инициализировать веса новой сети было равномерное распределение следующего вида:

$$w_i \sim U\left[-\frac{1}{\sqrt{n_{out}}}, \frac{1}{\sqrt{n_{out}}}\right]$$

В этом случае получается, что:

$$\text{Var}(w_i) = \frac{1}{12} \left(\frac{1}{\sqrt{n_{out}}} + \frac{1}{\sqrt{n_{out}}} \right)^2 = \frac{1}{3 \cdot n_{out}},$$

тогда: $n_{out} \text{Var}(w_i) = \frac{1}{3}$.

- После нескольких слоев такое преобразование параметров распределения значений между слоями сети фактически **приводит к затуханию сигнала**: дисперсия результата слоя каждый раз уменьшается (фактически делится на 3), а среднее у него было нулевое.
- Аналогичная ситуация повторяется и на шаге обратного распространения ошибки при обучении.
- Если мы используем симметричную функцию активации с единичной производной в окрестности нуля (например, \tanh), то теперь мы получим коэффициент пропорциональности для дисперсии $n_{in} \text{Var}(w_i)$, где n_{in} — число нейронов во входном слое, а не на выходе.

- Идея инициализации Ксавье заключается в том, что для беспрепятственного распространения значений активации и градиента по сети дисперсия в обоих случаях должна быть примерно равна единице.
- Поскольку для неодинаковых размеров слоев невозможно удовлетворить оба условия одновременно, предлагается инициализировать веса очередного слоя сети симметричным распределением с такой дисперсией:

$$\text{Var}(w_i) = \frac{2}{n_{in} + n_{out}}$$

что для равномерной инициализации приводит к следующему распределению:

$$w_i \sim U \left[-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}} \right]$$



Пример использования в PyTorch:

```
conv1 = torch.nn.Conv2d(...)
torch.nn.init.xavier_uniform(conv1.weight)
```

Пример на Keras

In []:

```
from keras.models import Sequential
from keras.layers import Dense

from keras.datasets import mnist

# загрузка исходных данных:
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# правильные ответы заданы в виде цифр, придется перекодировать их в виде векторов:
from keras.utils import np_utils
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
```

Теперь осталось для удобства переведем матрицы X_train и X_test из целочисленных значений на отрезке [0, 255] к вещественным на [0, 1] (нормализовать), а также сделать из квадратных изображений размера 28 × 28 пикселов одномерные векторы длины 784; это значит, что сами тензоры X_train и X_test будут иметь размерность (число примеров) × 784:

In []:

```
X_train = X_train.reshape([-1, 28*28]) / 255.
X_test = X_test.reshape([-1, 28*28]) / 255
```

In []:

```
def create_model(init):
    '''init - текстовый параметр, который интерпретируется как тип инициализации
    (для нашего эксперимента это будут значения uniform и glorot_normal)
    Функция возвращает функция простую полно связную модель с четырьмя промежуточными слоями

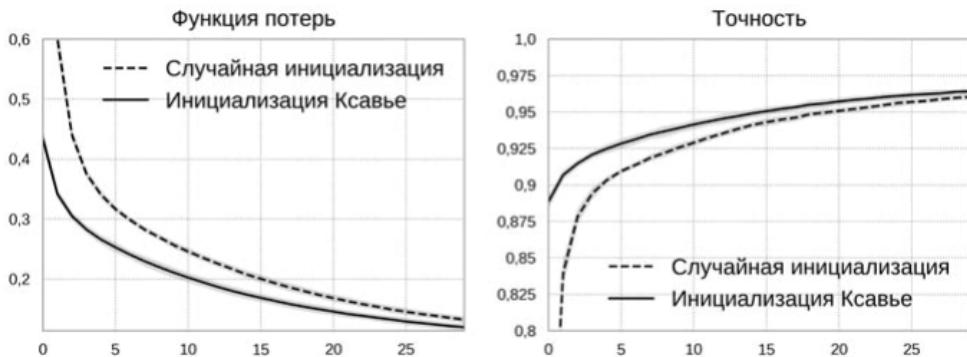
model = Sequential()
model.add(Dense(100, input_shape=(28*28,), init=init, activation='tanh'))
model.add(Dense(100, init=init, activation='tanh'))
model.add(Dense(100, init=init, activation='tanh'))
model.add(Dense(100, init=init, activation='tanh'))
model.add(Dense(10, init=init, activation='softmax'))
return model
```

In []:

```
uniform_model = create_model("uniform")
uniform_model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
uniform_model.fit(x_train, Y_train,
                   batch_size=64, nb_epoch=30, verbose=1, validation_data=(x_test, Y_test))
```

In []:

```
glorot_model = create_model("glorot_normal")
glorot_model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
glorot_model.fit(x_train, Y_train,
                  batch_size=64, nb_epoch=30, verbose=1, validation_data=(x_test, Y_test))
```



Сравнение инициализации Ксавье и случайной инициализации весов

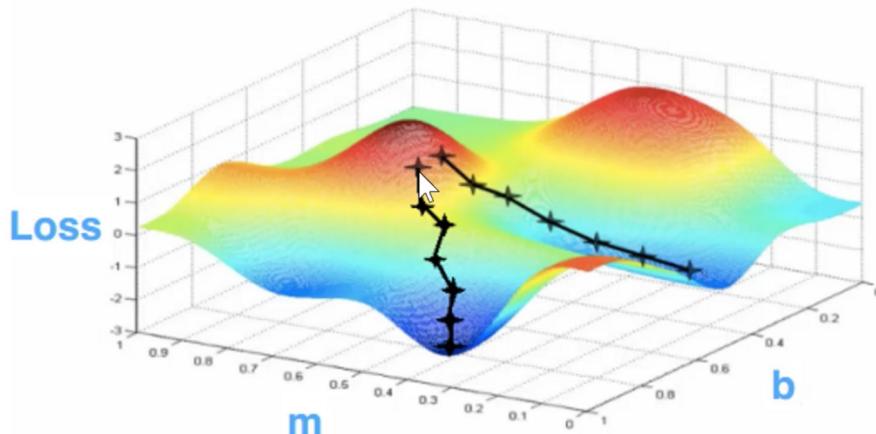
Сравнение инициализация Ксавьев и случайной инициализации весов

- Видно, что при инициализации весов по методу Ксавье модель уже после первой эпохи находит решение с точностью около 90 %, на что модели, чьи веса инициализированы равномерным распределением, требуется около 10 эпох.

Нормализация по мини-батчам

При обучении нейронных сетей один шаг градиентного спуска обычно делается не на одной точке входных данных, а на **мини-батче**, то есть на небольшой коллекции данных, которая обычно выбирается из всего обучающего множества случайно. С точки зрения оптимизации у такого подхода есть сразу несколько преимуществ.

$$f(x) = \text{nonlinear function of } x$$



Пример работы градиентного спуска для функции двух переменных

Стохастический градиентный спуск

- Шаг метода градиентного спуска:

$$\mathbf{w}^{t+1} = \mathbf{w}_t - \gamma \nabla_{\mathbf{w}} E(\mathbf{w}^{t-1}) = \mathbf{w}_t - \gamma \sum_{(\mathbf{x}, \mathbf{y}) \in D} \nabla_{\theta} E(f_L(\mathbf{x}, \mathbf{w}^{t-1}), \mathbf{y})$$

- Выполнение на каждом шаге градиентного спуска суммирования по всем $(\mathbf{x}, \mathbf{y}) \in D$ происходит слишком долго
- Создаем батчи - случайные наборы из фиксированного количества элементов выборки (например M элементов) $D^M, D^M \subset D (|D| = N)$:

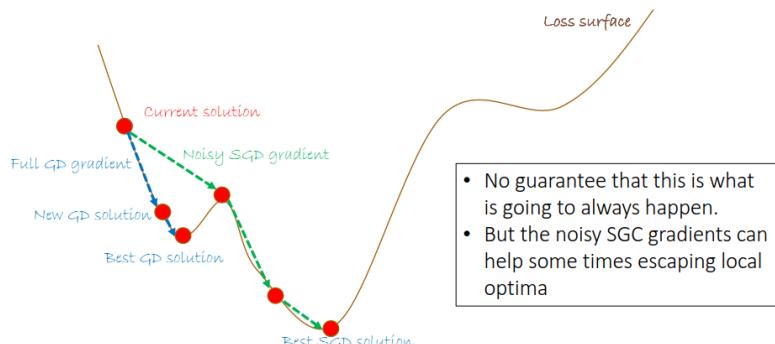
$$\nabla_{\mathbf{w}} E(\mathbf{w}^{t-1}) \approx \frac{N}{M} \sum_{(\mathbf{x}, \mathbf{y}) \in D^M} \nabla_{\theta} E(f_L(\mathbf{x}, \mathbf{w}^{t-1}), \mathbf{y})$$

Это работает, т.к.:

- Т.к. обычно поверхность:
 - не является квадратичной функцией
 - не выпуклая
 - имеет очень высокую размерность
- Обычно наборы данных слишком большие, чтобы вычислять градиенты полностью
- Нет никаких гарантий, что:
 - итоговое решение будет хорошим
 - решение быстро сходится к итоговому решению
 - или, что оно вообще сходится

Положительные свойства использования стохастического градиента:

- работает намного быстрее чем ГС
- на практике точность результата выше чем у ГС
- мини-батчи позволяют работать с наборами данных, которые меняются со временем
- дисперсия градиента возрастает при убывании размера батча ($\sim 1/\sqrt{M}$)



Пример успешной работы стохастического градиентного спуска

- Усреднение градиента по нескольким примерам представляет собой аппроксимацию градиента по всему тренировочному множеству, и чем больше примеров используется в одном мини-батче, тем точнее это приближение.
 - Максимальная точность достигается на шаге сразу на всем тренировочном датасете, но это слишком затратно вычислительно.
- Глубокие нейронные сети подразумевают большое количество последовательных действий с каждым примером. GPU (и многоядерные CPU) позволяет эту длинную последовательность рассчитывать параллельно для большого количества примеров.

Проблема внутреннего сдвига переменных (internal covariance shift) при глубоком обучении:

- Если на очередном шаге градиентного спуска меняются веса одного из первых (нижних) слоев
- \Rightarrow изменяются распределения активаций выходов этого слоя
- \Rightarrow всем последующим слоям надо адаптироваться к новому распределению входных данных

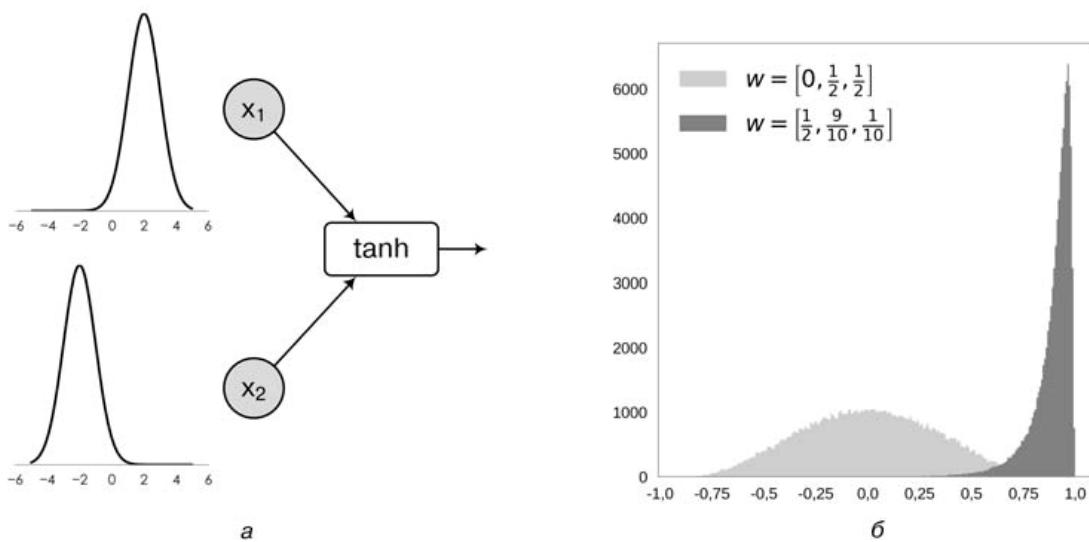
Пример:

Пусть имеется нейрон первого слоя:

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

и его веса меняются со значений $w = (w_0, w_1, w_2) = (0, 1/2, 1/2)$ на значения

$w = (w_0, w_1, w_2) = (1/2, 9/10, 1/10)$.



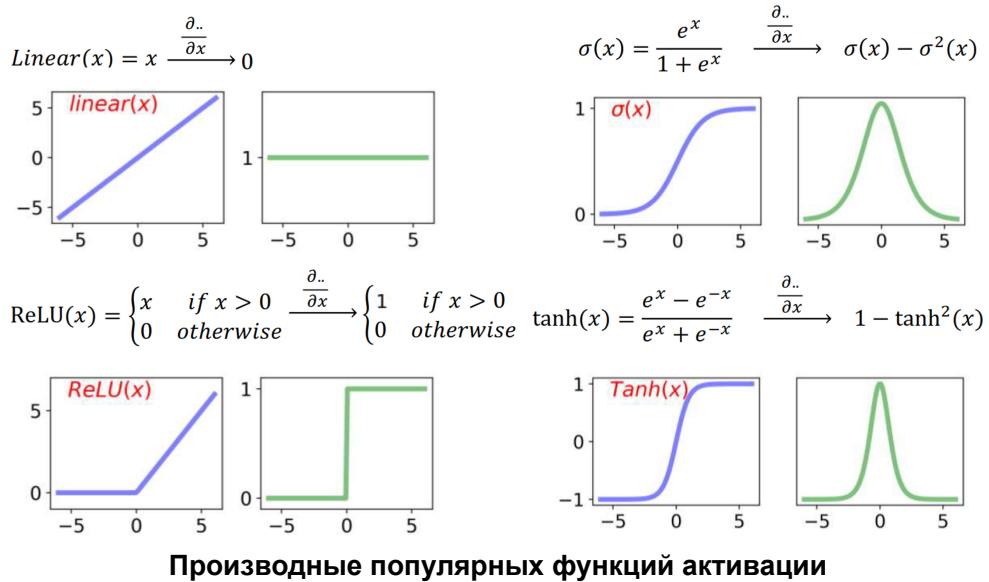
Пример успешной работы стохастического градиентного спуска

- а — структура первого слоя сети и входные распределения
- б — результат для двух разных векторов весов

Ситуация с точки зрения нейронов следующего уровня:

- сначала на вход получали одно распределение (серый график) и обучились на нем
- потом, когда вектор весов сильно сместился и то, чему обучились нейроны второго уровня, стало почти бесполезным: входы теперь берутся из совершенно новой области, и обучаться надо фактически заново

Сопутствующая проблема: "насыщение" функций активации



Производные популярных функций активации

Часто в нейронных сетях используются сигмоидальные функции активации ($f(x) = \sigma(x)$, $f(x) = \tanh(x)$), одной из особенностей которых является "насыщение" значений функций активации:

- когда входы получают большие по модулю значения производная $f'(x)$ быстро стремится к 0
- отрицательное следствие: при близких к 0 производных обратное распространение ошибки очень сильно затухает на этих градиентах
- потенциальное решение: замена функций активации на $ReLU$, но это не единственный и не всегда подходящий способ

В частности, при внутреннем сдвиге переменных высока вероятность насыщения функций активации и возникновения существенных сложностей при обучении глубоких моделей.

- Проблема сдвига переменных не является специфической сугубо для глубоких нейронных сетей.
- В классическом машинном обучении распространена аналогичная проблема:
 - распределение данных в тестовой выборке существенно отличается от распределения данных в обучающей выборке
- Наиболее распространенный метод решения: **нормализация данных**
- Для классических нейронных сетей эта процедура выглядела как «отбелевание» (whitened) входов сети:
 - среднее значение входных данных приводится к нулю
 - матрица ковариаций приводится к единичной матрице

Нормализация входов часто помогает, и прежде чем обучать нейронную сеть, ее желательно делать.

Нормализация входов часто помогает, и прежде чем обучать нейронную сеть, ее желательно делать.

(гипотеза): нормировать входы очередного слоя внутри сети на каждом шаге обучения:

- если не учитывать эту операцию при обучении это приведет к неадекватному изменению весов (например, констант)
- \Rightarrow нужно учитывать нормализацию при градиентном спуске

Прямолинейный подход:

Введем слой нормализации: $\hat{\mathbf{x}} = \text{Norm}(\mathbf{x}, \mathcal{X})$

где \mathbf{x} - текущий обучающий пример, а \mathcal{X} - все примеры из тренировочной выборки (!).

- \Rightarrow для шага градиентного спуска нам необходимо вычислить якобианы для $\frac{\partial \text{Norm}}{\partial \mathbf{x}}$ и $\frac{\partial \text{Norm}}{\partial \mathcal{X}}$,

причем рассчитывать второй якобиан точно придется!

- для операции «отбеливания» (декорреляции) потребуется вычислить матрицу ковариаций:

$$\text{Cov}[x] = \mathbb{E}_{\mathbf{x} \in \mathcal{X}} [\mathbf{x}\mathbf{x}^\top] - \mathbb{E}[\mathbf{x}]\mathbb{E}[\mathbf{x}]^\top$$

затем обратить ее и вычислить из нее квадратный корень, а при градиентном спуске еще и производные такого преобразования.

Полноценную декорреляцию для каждого слоя **сделать за разумное время невозможно**, особенно для больших датасетов, поэтому используются **упрощенные варианты**:

Вместо декорреляции всех входов совместно, нормализуют каждый элемент входного вектора по отдельности:

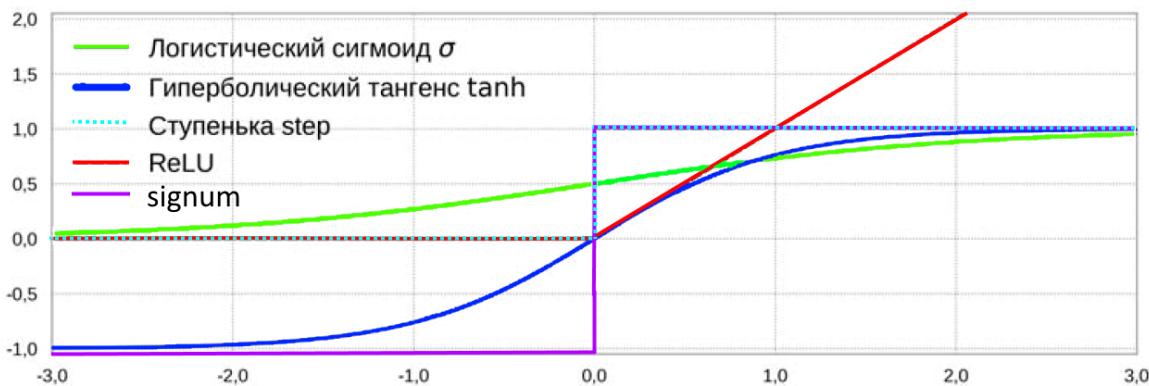
Представим $\mathbf{x} = (x_1, \dots, x_d)$, тогда нормализация компоненты вектора выглядит так:

$$\hat{x}_k = \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\text{Var}(x_k)}}$$

Среднее и дисперсию в формуле хотелось нужно вычислять по всему датасету \mathcal{X} , но это совершенно невозможно вычислительно, поэтому применим очередное упрощение: будем рассчитывать эти величины по текущему мини-батчу. Данный подход называется **нормализацией по мини-батчам**.

Недостатки нормализации по мини-батчам:

- Если используется сигмоидная функция активации (например: $f(x) = \sigma(x)$, $f(x) = \tanh(x)$), то после нормализации ее аргумента нелинейность по сути пропадает
 - т.к. подавляющее большинство нормализованных значений будут попадать в область, где сигмоид ведет себя очень похоже на линейную функцию, и функция активации фактически станет линейной



Графики функций активации

Для исправления этого недостатка, слой нормализации должен иметь возможность "настроиться" как тождественная функция. Т.е. при некоторых комбинациях параметров он должен работать как $f(x) = x$.

- для этого введем параметры γ_k и β_k для масштабирования и сдвига нормализованной активации по каждой компоненте:

$$y_k = \gamma_k \hat{x}_k + \beta_k = \gamma_k \frac{x_k - \mathbb{E}[x_k]}{\sqrt{Var(x_k)}} + \beta_k$$

- параметры γ_k и β_k будут обучаться вместе со всеми параметрами ИНС и позволяют восстановить выразительную способность сети в целом
- в частности, при настройке значений $\gamma_k = \sqrt{Var(x_k)}$ и $\beta_k = \mathbb{E}[x_k]$ слой нормализации может обучаться реализовывать тождественную функцию

Резюмируем описание работы слоя нормализации по мини-батчам:

- Слой получает на вход очередной мини-батч $B = \{\mathbf{x}_1, \dots, \mathbf{x}_m\}$
- Вычисляются базовые статистики по мини-батчу:

$$\mu_B = \frac{1}{m} \sum_{i=1}^m \mathbf{x}_i, \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (\mathbf{x}_i - \mu_B)^2$$

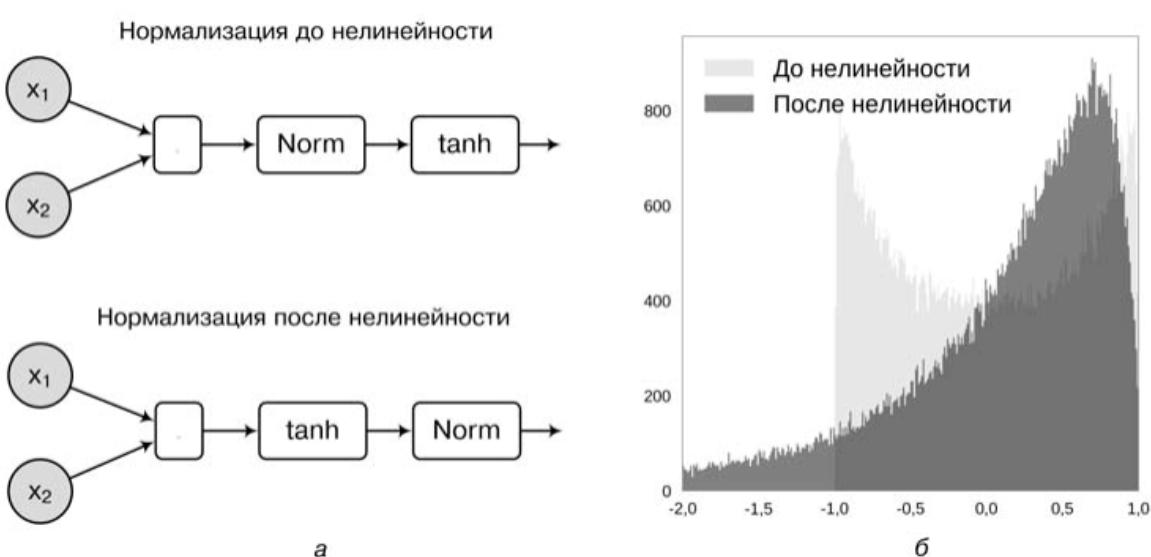
- Нормализует выходы:

$$\hat{x}_k = \gamma_k \frac{x_k - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta_k$$

небольшая положительная константа ϵ необходима для того, чтобы избежать деления на 0

- Рассчитывает результат:

$$y_k = \gamma_k \hat{x}_k + \beta_k$$



Пример нормализации до и после нелинейности

* а — графы вычислений * б — соответствующие результаты сэмплирования

На данный момент нет устоявшегося мнения по вопросу о том, лучше делать ее после очередного слоя или после линейной части слоя, до нелинейной функции активации. Различные варианты дают разный эффект, в частности:

- область значений в варианте нормализации после сигмоидальной нелинейности будет шире
- область значений в варианте нормализации после сигмоидальной нелинейности будет уже, т.к. $tanh$ или σ снова вернут нормализованные результаты на отрезок $[-1, 1]$ или $[0, 1]$ соответственно

Пример использования нормализации по батчам в PyTorch:

```
class network(nn.Module):
    def __init__(self):
        super(network, self).__init__()
        self.linear1 = nn.Linear(in_features=40, out_features=320)
        self.bn1 = nn.BatchNorm1d(num_features=320)
        self.linear2 = nn.Linear(in_features=320, out_features=2)

    def forward(self, input): # Input is a 1D tensor
        y = F.relu(self.bn1(self.linear1(input)))
        y = F.softmax(self.linear2(y), dim=1)
        return y

model = network()
x = torch.randn(10, 40)
output = model(x)
```

Документация по слою нормализации в Keras:

<https://keras.io/layers/normalization/> (<https://keras.io/layers/normalization/>).

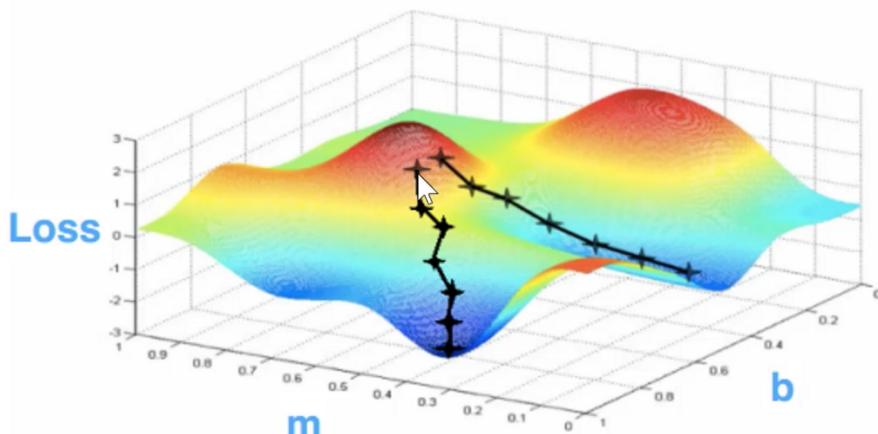
Пример использования слоев нормализации по батчам:

<https://www.programcreek.com/python/example/100588/keras.layers.normalization.BatchNormalization>
[\(<https://www.programcreek.com/python/example/100588/keras.layers.normalization.BatchNormalization>\)](https://www.programcreek.com/python/example/100588/keras.layers.normalization.BatchNormalization)

Усовершенствованные методы градиентного спуска

- **Метод градиентного спуска** - метод нахождения локального экстремума (минимума или максимума) функции с помощью движения вдоль градиента. В нашем случае шаг метода градиентного спуска выглядит следующим образом:
$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}_{t-1}) = \boldsymbol{\theta}_{t-1} - \eta \sum_{(\mathbf{x}, \mathbf{y}) \in D} \nabla_{\boldsymbol{\theta}} E(f_L(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y})$$
- (!) Выполнение на каждом шаге градиентного спуска суммирование по всем $(\mathbf{x}, \mathbf{y}) \in D$ обычно слишком неэффективно
- Для выпуклых функций **задача локальной оптимизации** - найти локальный минимум (максимум) автоматически превращается в **задачу глобальной оптимизации** - найти точку, в которой достигается наименьшее (наибольшее) значение функции, то есть самую низкую (высокую) точку среди всех.
- Оптимизировать веса одного перцептрона - выпуклая задача, но **для большой нейронной сети целевая функция не является выпуклой**.

$$f(x) = \text{nonlinear function of } x$$



Пример работы градиентного спуска для функции двух переменных

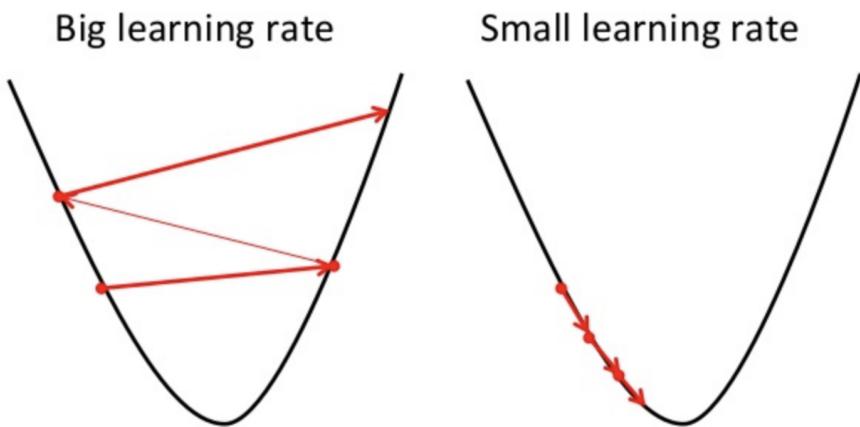
У нейронных сетей функция ошибки может задавать очень сложный ландшафт с огромным числом локальных максимумов и минимумов. Это свойство необходимо для обеспечения выразительности нейронных сетей, позволяющей им решать так много разных задач.

Градиентный спуск с изменяемой скоростью обучения

Параметр метода градиентного спуска: скорость обучения η показывает, насколько сильно мы сдвигаем параметры в сторону градиента на очередном шаге.

Скорость обучения — это чрезвычайно важный параметр.

- Если она будет слишком большой:
 - алгоритм станет "прыгать" по практически случайным точкам пространства и не попадет в минимум, потому что все время будет его "перепрыгивать"
- Если она будет слишком маленькой:
 - обучение станет гораздо медленнее
 - алгоритм рискует успокоиться и сойтись в первом же локальном минимуме, который скорее всего не окажется самым лучшим.



Последствия неверного выбора шага градиентного спуска

Простейшая стратегия управления скоростью обучения:

- сначала скорость обучения должна быть большой: это позволяет быстрее прийти в правильную область пространства поиска

- затем скорость обучения должна быть маленькой: это позволяет детально исследовать окрестности точки минимума и в итоге попасть в нее

Реализация в виде линейного затухания:

$$\eta = \eta_0 \left(1 - \frac{t}{T}\right)$$

Реализация в виде экспоненциального затухания:

$$\eta = \eta_0 e^{-\frac{t}{T}}$$

где t — это время прошедшее с начала обучения время (число мини-батчей или число эпох обучения), а T — параметр, определяющий, как быстро будет уменьшаться η .

- правильный подбор η_0 и T позволяет существенно улучшить градиентный спуск
- как правильно подобрать η_0 и T ? Если правильно подобрать параметры η_0 и T , такая стратегия будет почти наверняка работать лучше, чем градиентный спуск с постоянной скоростью, а если повезет, то и вообще работать будет хорошо.

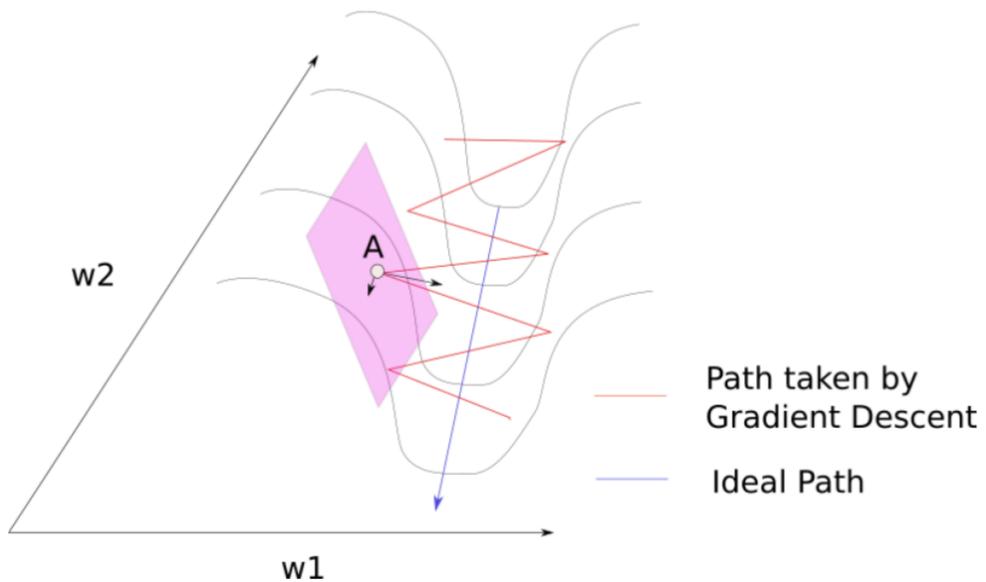
Адаптивные методы градиентного спуска

- Замедление обучения с фиксированными параметрами никак не учитывает характеристики оптимизируемой функции
- Адаптивные методы градиентного спуска** меняют параметры ГС в зависимости от результатов взаимодействия с оптимизируемой функцией

Пример:

В местах, где поверхность функции больше наклонена в одном измерении, чем в другом, обычный стохастический градиентный спуск может вести себя некорректно. Например, если минимум находится в сильно вытянутом «овраге»:

- шаг градиентного спуска будет направлен от одной близкой и крутой стенки этого оврага к другой
- когда точка попадет на другую стенку, градиент станет направлен в противоположную сторону
- т.е. в процессе оптимизации текущее решение будет "прыгать" между стенками оврага, но к общему минимуму будет продвигаться очень медленно



Пример неэффективной работы не адаптивного градиентного спуска

В этой ситуации могут помочь адаптивные методы градиентного спуска:



Пример работы градиентного спуска для функции двух переменных

Метод импульсов

Идея метода импульс (momentum): точка не просто подчиняется правилам градиентного спуска, а подчиняется законам механики, в первую очередь имеет инерцию, т.е.:

- у точки есть скорость
- положение точки на следующем шаге определяется ее текущим положением и скоростью
- ускорение (скорость изменения скорости) определяется величиной градиента
- на каждом шаге пересчитывается как положение точки, так и ее скорость

Т.е. у точки, которая спускается по поверхности, появляется импульс, она движется по инерции и стремится этот импульс сохранить (отсюда и название метода).

Формальная запись метода импульсов:

$$\begin{aligned}\boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - u_t \\ u_t &= \gamma u_{t-1} + \eta \nabla_{\theta} E(\boldsymbol{\theta}_{t-1})\end{aligned}$$

здесь:

- u_t - скорость точки в момент времени t
- γ - параметр метода моментов:
 - параметр определяет, какую часть прошлого градиента мы хотим сохранить на текущем шаге
 - $\gamma < 1$
- Теперь, когда точка "катится с горки", и все больше ускоряется в том направлении, в котором были направлены сразу несколько предыдущих градиентов, но будет двигаться достаточно медленно в тех направлениях, где градиент все время меняется.
- \Rightarrow Метод импульсов помогает ускорить градиентный спуск в нужном направлении и уменьшает его колебания.

Метод Нестерова

Метод Нестерова улучшает метод импульсов. При расчетах на шаге t для получения $\boldsymbol{\theta}_t$:

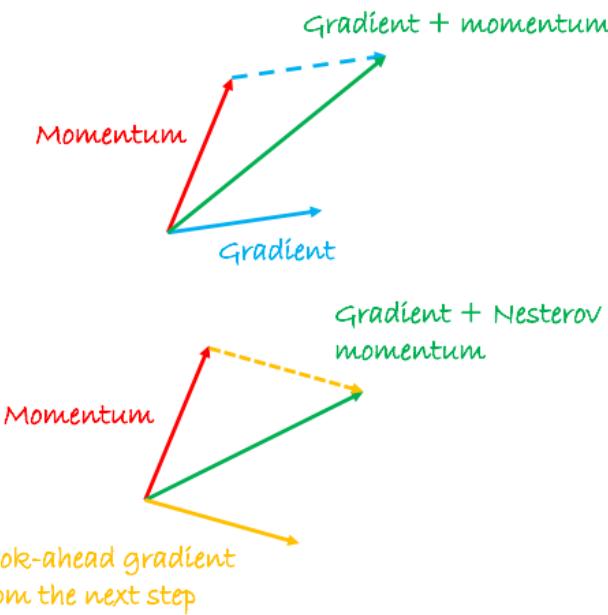
- вместо расчета градиента значения функции ошибки в точке $\boldsymbol{\theta}_{t-1}$ (как было в методе моментов)
- метод Нестерова рассчитывает градиента значения функции ошибки в точке $\boldsymbol{\theta}_{t-1} - \gamma u_{t-1}$

Т.е. вместо:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - u_t = \boldsymbol{\theta}_{t-1} - \gamma u_{t-1} - \eta \nabla_{\theta} E(\boldsymbol{\theta}_{t-1})$$

рассматриваем:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - u_t = \boldsymbol{\theta}_{t-1} - \gamma u_{t-1} - \eta \nabla_{\theta} E(\boldsymbol{\theta}_{t-1} - \gamma u_{t-1})$$



Визуализация двух вариантов метода импульсов

Это целесообразно, т.к.:

- согласно методу моментов γu_{t-1} уже точно будет использовано на этом шаге
- изменившийся градиент разумно считать уже в той точке, куда мы приедем после применения момента предыдущего шага

Метод Adagrad

В рассмотренных выше адаптивных методах

- шаг обновления зависел только от текущего градиента и глобальных параметров
- шаг обновления не учитывал историю обновлений для каждого отдельного параметра

Возможный вариант:

- некоторые веса уже близки к своим локальным минимумам \Rightarrow по этим координатам нужно двигаться медленно
- другие веса еще находятся "на крутом склоне" \Rightarrow их можно менять гораздо быстрее

В методах, которые мы обсуждали до сих пор, шаг обновления зависел только от текущего градиента и глобального параметра скорости обучения, но никак не учитывал историю обновлений для каждого отдельного параметра. Однако вполне может так сложиться, что некоторые веса уже близки к своим локальным минимумам, и по этим координатам нужно двигаться медленно и осторожно, а другие веса еще на середине соответствующего склона, и их можно менять гораздо быстрее. К тому же, само по себе регулирование скорости обучения может оказаться довольно затратным делом. Для того чтобы иметь возможность адаптировать скорость обучения для разных параметров автоматически, были созданы адаптивные методы оптимизации.

Идея метода Adagrad:

- шаг изменения должен быть меньше у параметров, которые в большей степени варьируются в данных
- шаг изменения должен быть больше у тех, которые менее изменчивы в разных примерах

Обозначим через $g_{t,i}$ градиент функции ошибки по параметру θ_i на шаге t :

$$g_{t,i} = \nabla_{\theta_i} E(\theta_t)$$

Тогда обновление параметра θ_i на очередном шаге градиентного спуска можно записать так:

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

где G_t - диагональная матрица, каждый элемент которой - сумма квадратов градиентов соответствующего параметра за предыдущие шаги:

$$G_{t,ii} = G_{t-1,ii} + g_{t,i}^2$$

а ϵ - сглаживающий параметр, позволяющий избежать деления на ноль.

В вектороном виде выражения можно записать (произведение выполняется покомпонентно):

$$\boldsymbol{\theta}_t = -\frac{\eta}{\sqrt{G_{t-1} + \epsilon}} \odot \boldsymbol{g}_{t-1}$$

- + Один из плюсов Adagrad является снятие необходимости ручной настройки скорости обучения η т.к. диагональные элементы G по сути являются индивидуальными скоростями обучения для каждого компонента θ .
- - Т.к. слагаемое g^2 всегда положительно $\Rightarrow G$ постоянно увеличивается \Rightarrow скорость оптимизации (обучения) может уменьшаться слишком быстро, что плохо сказывается на обучении глубоких ИНС
- - Глобальную скорость обучения в Adagrad нужно выбирать вручную

Adadelta - несложная, но эффективная модификация алгоритма Adagrad, основная цель которой состоит в том, чтобы попытаться исправить два указанных выше недостатка.

Первая идея: избавиться от накаплиения суммы квадратов градиентов по всей истории обучения

- (I вариант) будем считать суммы квадратов градиентов по некоторому окну
- (II вариант) будем считать суммы квадратов градиентов по всей истории, но с экспоненциально затухающими весами

Для каждого оптимизируемого параметра введем свой метапараметр ρ_i , тогда экспоненциальное затухание можно записать так:

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

при этом, как и в методе моментов, метапараметр $\rho_i < 1$

- Экспоненциальное среднее, в отличие от суммы, будет убывать только тогда, когда убывающими станут градиенты.
- Т.е. уменьшение скорости обучения будет происходить только в тот момент, когда изменение целевой функции замедляется, для более тонкой настройки вокруг локального минимума.

Вторая идея: исправление размерности в шаге алгоритма

- В Adagrad значения обновлений $\Delta\theta$ зависят от отношений градиентов, то есть величина обновлений является безразмерной.
- Правильные "единицы измерений" получаются только в методах второго порядка. В частности в методе Ньютона второго порядка обновление параметров:

$$\Delta\theta \sim H^{-1} \nabla_\theta f \sim \frac{\frac{\partial f}{\partial \theta}}{\frac{\partial^2 f}{\partial \theta^2}} \sim \text{размерность}\theta$$

- Чтобы привести масштабы этих величин в соответствие, достаточно домножить обновление из Adagrad на еще один новый сомножитель: еще одно экспоненциальное среднее, но теперь уже от квадратов обновлений параметров, а не от градиента.
- Поскольку настоящее среднее квадратов обновлений нам неизвестно, то чтобы его узнать, нам нужно как раз сначала выполнить текущий шаг алгоритма, — оно аппроксимируется предыдущими шагами:

$$\mathbb{E}[\Delta\theta^2]_t = \rho\mathbb{E}[\Delta\theta^2]_{t-1} + (1 - \rho)\Delta\theta^2$$

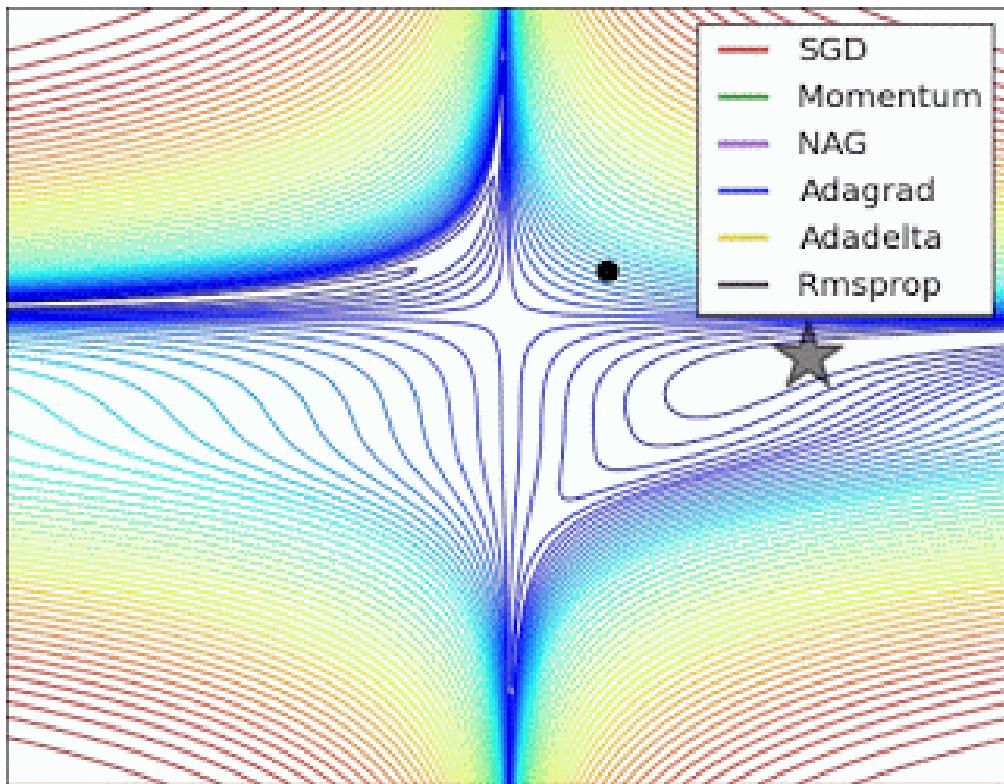
С помощью полученного значения получаем поправочных коэффициент:

$$\mathbf{u}_t = -\frac{\sqrt{\mathbb{E}[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{G_{t-1} + \epsilon}} \odot \mathbf{g}_{t-1}$$

Существует близкий аналог Adadelta алгоритм RMSprop оба метода основаны на классической идее применения инерции, только RMSprop использует ее для оптимизации метапараметра скорости обучения.

Основная разница между RMSprop и Adadelta состоит в том, что RMSprop не делает вторую поправку с изменением единиц и хранением истории самих обновлений, а просто использует корень из среднего от квадратов (вот он где, RMS) от градиентов:

$$\mathbf{u}_t = -\frac{\eta}{\sqrt{G_{t-1} + \epsilon}} \odot \mathbf{g}_{t-1}$$



Пример работы различных вариантов градиентного спуска для функции двух переменных

Методы оптимизации, доступные в Pytorch: [\(https://pytorch.org/docs/stable/optim.html\)](https://pytorch.org/docs/stable/optim.html)

Спасибо за внимание!

Технический раздел:

next **Q**: qs line

next **A**: an line

next **Note**: an line

next **Def**: df line

next **Ex**: ex line

next **+** pl line

next **-** mn line

next **±** plmn line

next **⇒** hn line

In []: