SGD доработки

Цель занятия

В результате обучения на этой неделе:

- вы узнаете об основных подходах градиентной оптимизации;
- рассмотрите основные техники регуляризации в глубоком обучении;
- научитесь бороться с переобучением нейросети с помощью регуляризации;
- построите модель с регуляризацией в PyTorch.

План занятия

Конспект занятия

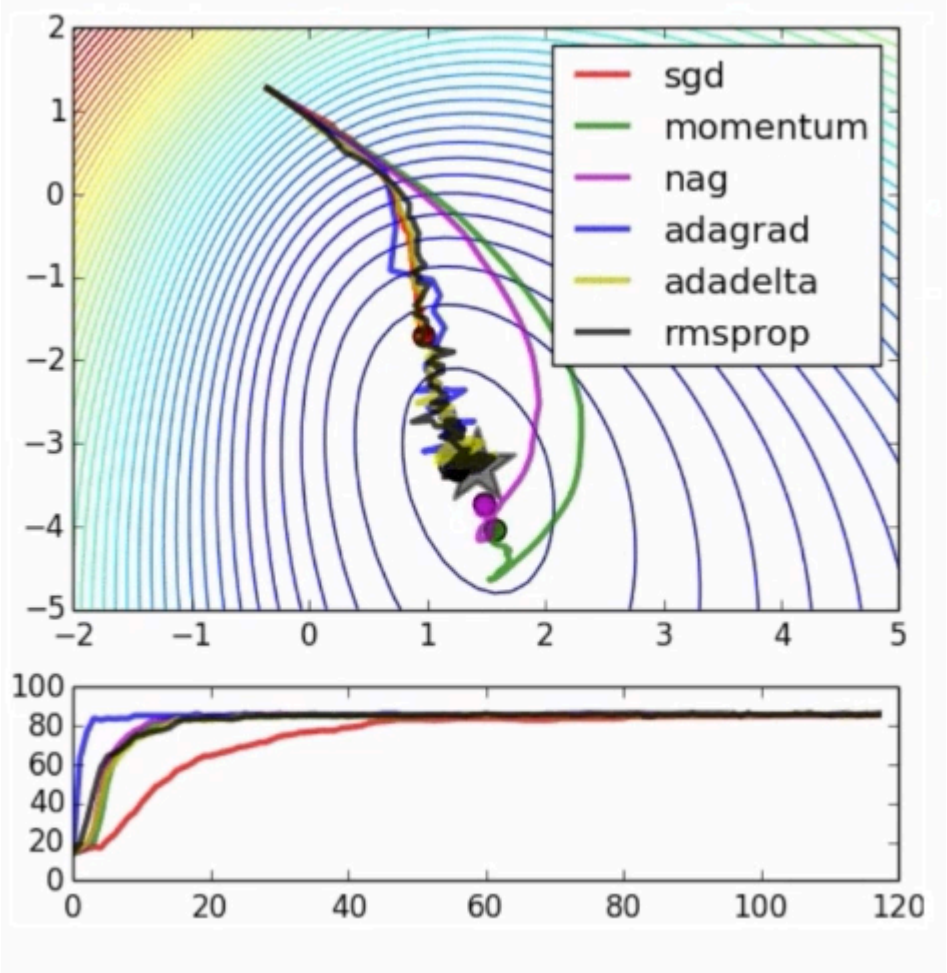
1. SGD доработки

Рассмотрим, как сделать стохастический градиентный спуск (SGD) чуть более устойчивым, как ускорить его сходимость в некоторых случаях и какие идеи могут быть применены, когда мы говорим о градиентной оптимизации.

Доработать градиентный спуск можно несколькими способами:

- Momentum (техника тяжелого шарика);
- AdaGrad;
- Adadelta;
- RMSProp;

- Adam и др.



Многие из этих оптимизаторов базируются на одних и тех же идеях.

Повторим, что такое стохастический градиентный спуск. Пусть у нас есть некоторое множество параметров W, и функция потерь L зависит от параметров. Среднее значение функции потерь:

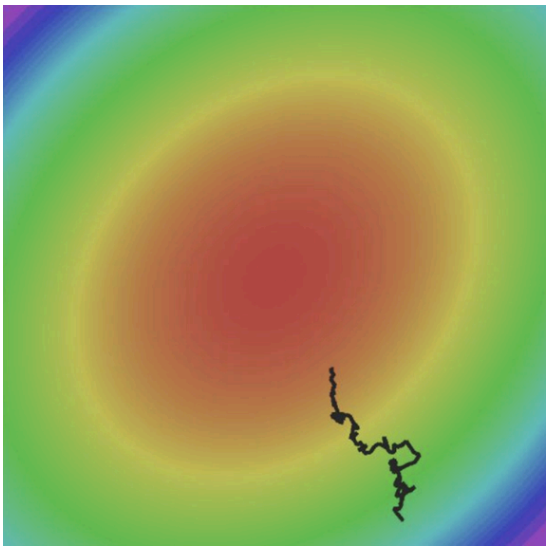
$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i\left(x_i, y_i, W\right)$$

Среднее значение градиента на некоторой подвыборке:

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$

Берем среднее значение градиента и используем классический градиентный шаг.

Чем меньше подвыборка у стохастического градиентного спуска, тем более шумная оценка градиента. Поэтому направление градиента может меняться, флуктуировать от шага к шагу, но приближаться к оптимуму.



Уменьшить флуктуацию можно увеличением подвыборки (батча) для каждого шага. Но увеличивать размер батча не всегда получается — чем больше батч, тем больше потребуется памяти компьютера. Либо придется использовать несколько проходов с одними и теми же параметрами модели, что сильно замедляет ее обучение.

Ускорить SGD можно несколькими способами.

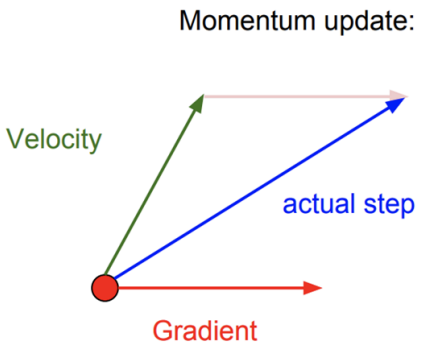
Momentum

Обычный SGD:

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

На каждом шаге будем запоминать, в каком направлении мы уже двигались. То есть у нашего вектора параметров есть некоторая инерция. На каждом шаге будем накапливать некоторый импульс:

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

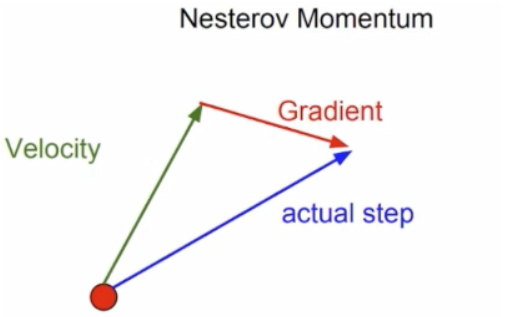


Такой способ позволяет быстро проходить «пологие» места функции потерь, на которых очень маленький градиент и функция потерь минимизируется очень долго.

Nesterov momentum

Momentum часто дорабатывается с помощью идеи Нестерова.

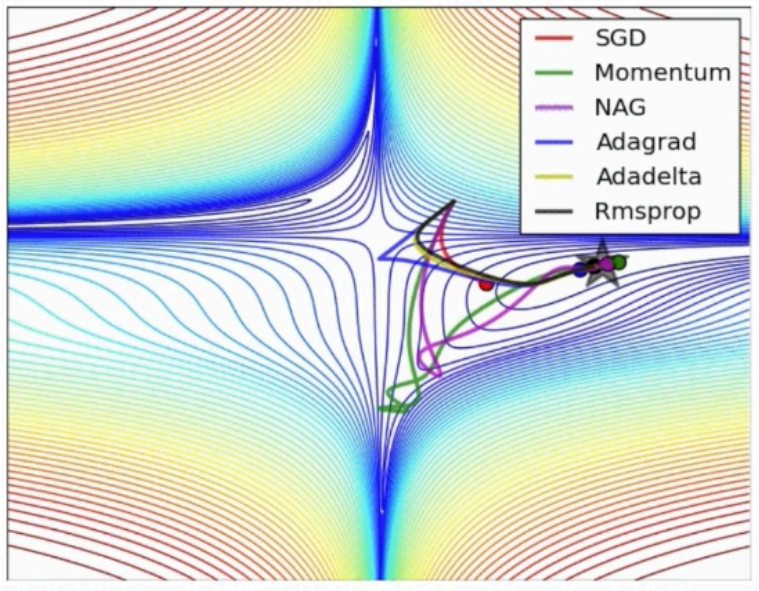
Предположим, что большой импульс уже накоплен, и по инерции вектор параметров очень сильно поменяется. Если в следующем шаге мы сместимся очень далеко, то релевантность значения градиентов в текущей точке очень низкая. Исправить это можно с помощью Nesterov momentum — он позволяет заглядывать немного в будущее.



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

Такой способ позволяет достаточно быстро вносить поправки в накопленный импульс и гораздо более эффективно сходиться. Это связано с тем, что иногда в Momentum значение параметров может по инерции улететь очень далеко от оптимума, а потом долго возвращаться.

На иллюстрации можно посмотреть сравнение разных способов оптимизации:



Зеленая траектория — Momentum. На другой иллюстрации можно увидеть, что он по инерции улетел далеко от оптимума.

RMSProp, AdaGrad, Adadelta

Рассмотрим еще одно семейство оптимизаций для SGD, которое эксплуатирует другую идею: они позволяют подобрать нормировочную константу для каждой из компонент вектора по отдельности.

Для AdaGrad:

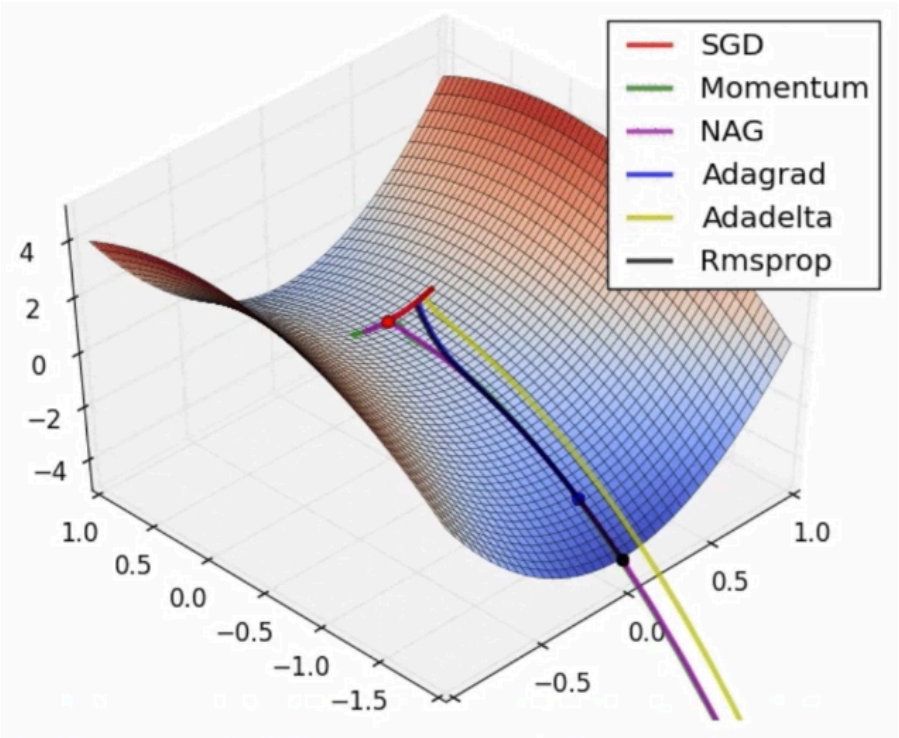
$$\text{cache}_{t+1} = \text{cache}_t + (\nabla f(x_t))^2$$

$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$

Мы предполагаем, что один learning rate на все веса – это мало. Для одних весов нужна очень тонкая настройка, для других — целое плато, и можно оптимизироваться с очень большим learning rate.

Будем опираться на то, какие значения принимает градиент. Если градиент большой, то нужно «ходить» помедленнее, если маленький — брать шаг побольше.

В идее AdaGrad и RMSProp используется значение градиента в качестве нормировочной константы. Этот подход достаточно хорошо работает. Это можно увидеть на следующей иллюстрации, когда мы говорим про седловые точки, в которых есть локальный экстремум:



Здесь точки методов Momentum и Нестеров моментум застревают в локальном экстремуме довольно надолго.

У AdaGrad есть недостаток: cache, на который мы нормируемся, либо постоянно возрастает, либо не убывает. То есть нормировочная константа всегда будет неубывающей, а, значит, нормированный градиент будет постепенно затухать. Починить это можно следующим образом: будем cache постепенно сглаживать.

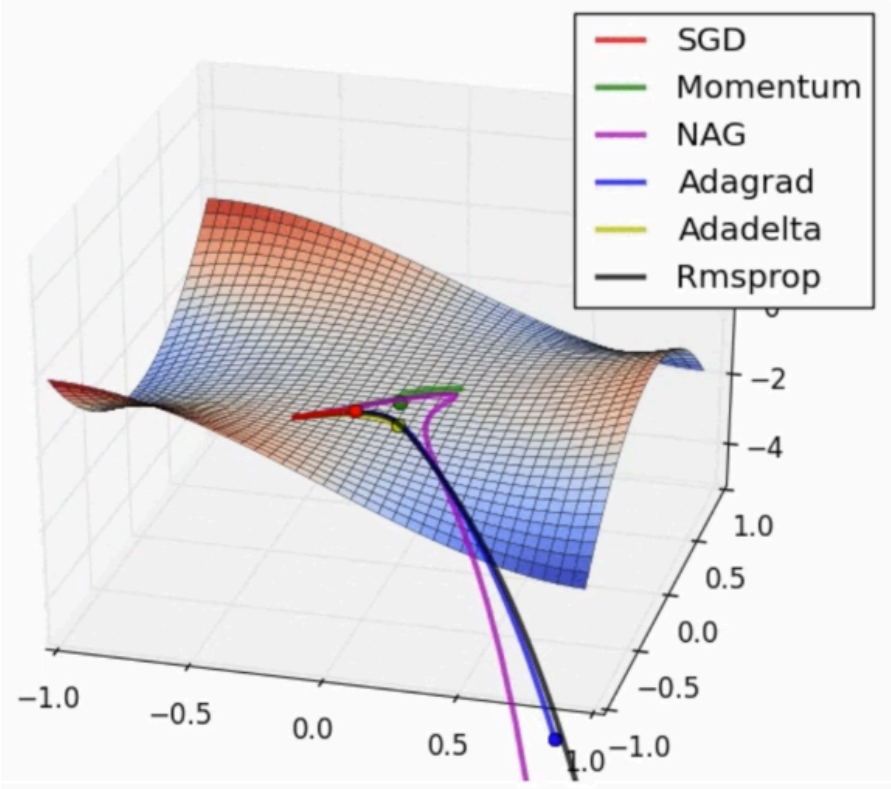
RMSProp — это SGD, где cache постепенно сглаживается с помощью экспоненциальной функции:

$$\text{cache}_{t+1} = \beta \text{cache}_t + (1-\beta)(\nabla f(x_t))^2$$
$$x_{t+1} = x_t - \alpha \frac{\nabla f(x_t)}{\text{cache}_{t+1} + \varepsilon}$$

Подробнее об этом можно почитать в лекции Overview of mini-batch gradient descent.

Объединение подходов

На следующей картинке представлена идея объединения всех приведенных методов:



$$v_{t+1} = \gamma v_t + (1-\gamma)\nabla f(x_t)$$
$$\text{cache}_{t+1} = \beta \text{cache}_t + (1-\beta)(\nabla f(x_t))^2$$
$$x_{t+1} = x_t - \alpha \frac{v_{t+1}}{\text{cache}_{t+1} + \varepsilon}$$

При объединении Momentum и RMSProp мы получаем Adam. О принципах функционирования метода Adam можно почитать на сайте CS231n: Convolutional Neural Networks for Visual Recognition. А по ссылке — увидеть наглядную демонстрацию.

У всех перечисленных методов необходимо правильно подбирать learning rate.

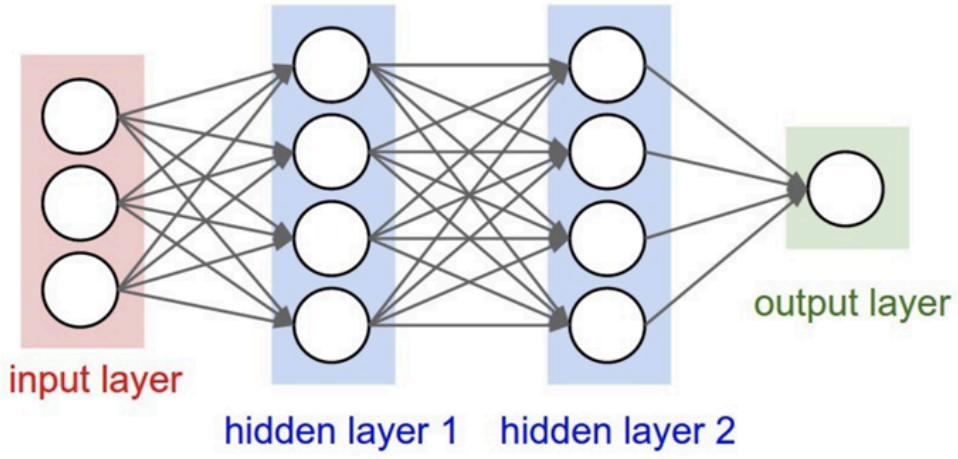
Каждый метод потребляет определенное количество памяти. Cash и накопленный импульс требуют дополнительной памяти, а, значит, эффективный размер батча при обучении будет еще ниже, чем у обычной SGD. В некоторых случаях лучше взять SGD с батчем побольше, чтобы получить более устойчивую сходимость. Этим часто пользуются в задачах компьютерного зрения. Adam — хороший выбор по умолчанию, но не всегда можно использовать только его. Важно следить за learning rate, за качеством модели, чтобы она не начала переобучаться.

2. Регуляризация в DL

Рассмотрим техники регуляризации в глубоком обучении (DL). В первую очередь обратимся к техникам нормировки начальных и промежуточных данных.

Батч-нормализация появилась в 2015 году и позволила достигать больших скоростей обучения нейронных сетей.

Рассмотрим нейронную сеть:



Предположим, что мы можем просматривать любой слой, кроме первого.

Будем считать, что исходные данные первого слоя мы уже отнормировали. Дальше у нас есть некоторые параметры в промежуточных слоях, которые порождают промежуточные представления. В данном примере в третьем слое мы получаем ответ.

На каждом этапе мы хотим посчитать производные, чтобы взять антиградиент и обновить параметры весов.

Когда обновляем параметры второго слоя, мы базируемся на представлении первого слоя. В этом и кроется проблема. Мы опираемся на параметры первого слоя, обновляем параметры второго слоя в прямом проходе, а на обратном проходе считаем градиенты. Градиент первого слоя зависит от градиента второго слоя. Промежуточное значение x первого слоя, соответственно, меняется. Также меняется и распределение x, потому что поменялись параметры. Если используем данные из другого распределения, мы нарушаем предположение iid. Это большая проблема называется covariance shift — меняется последующий слой, а предыдущий ломает предположения, в которых обновился последующий.

Починить это можно следующим образом:

- мы гарантируем каждому слою, что ему на вход придут нормированные данные с нулевым средним и с нулевой дисперсией:

$$h_i = \frac{h_i - \mu_i}{\sqrt{\sigma_i^2}}$$

- изменение данных происходит следующим образом:

$$\mu_i := \alpha \cdot mean_{batch} + (1 - \alpha) \cdot \mu_i,$$

$$\sigma_i^2 := \alpha \cdot variance_{batch} + (1 - \alpha) \cdot \sigma_i^2$$

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Оригинальный алгоритм из статьи 2015-го года:

- посчитали среднее;
- посчитали дисперсию;
- обновили данные;
- посчитали дополнительные линейные преобразования.

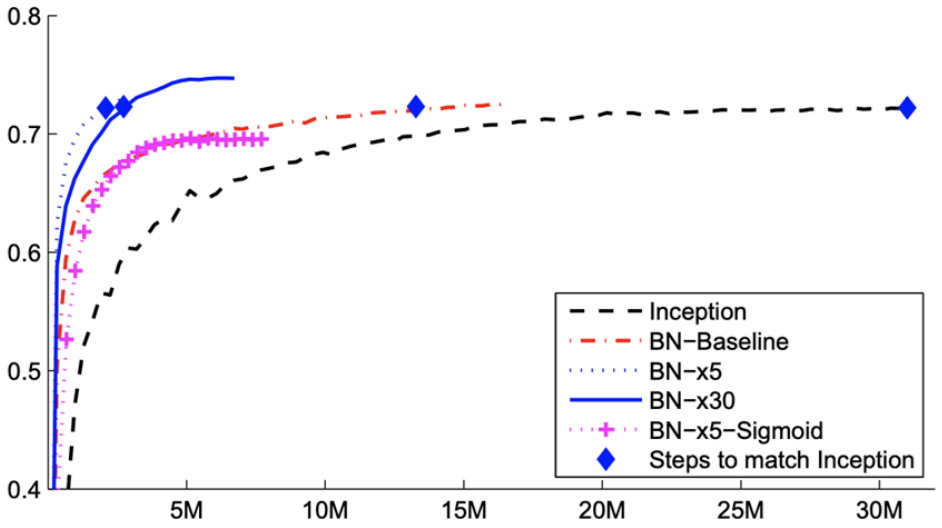
Этот пример показывает, как правильно мыслить при попытке сформировать преобразование и встроить его в нейронную сеть. Логика простая: вычислить среднее и поделить на дисперсию — это хорошо, на вход очередному слою придут данные с фиксированным средним и дисперсией. Эти статистики сохраняются. Но возможно часть информации была выкинута.

Автором статьи была предложена идея: для y появляется параметрическое преобразование, γ и β уже могут обучаться. В этом случае нейронная сеть может самостоятельно выучить эти значения таким образом, чтобы мы могли получить тождественное преобразование. То есть γ и β должны быть такими, чтобы результирующее выражение было тождественно обратному преобразованию. Конечно, мы предполагаем, что данные приходят изначально из одного и того же распределения, поэтому среднее и дисперсия не должны сильно меняться от шага к шагу.

Батчнорм позволил значительно ускорить модели. До введения батчнорм распределение пытались сохранить, используя маленький learning rate от шага к шагу. Параметры немного изменялись, распределение немного сдвигалось, модель постепенно сходилась. При появлении батчнорм learning rate можно увеличить, но при этом модель все равно продолжает сходиться.

Результат — задача классификации на ImagNet:

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%



Удалось сократить число шагов в 10 раз, чтобы добиться того же качества. Благодаря нормировке данных, итоговое качество задачи выросло до 74,8%.

Следует помнить, что нейронные сети — это не просто универсальные модели, которые решат любую задачу. Они умеют достаточно неплохо подбирать веса и строить промежуточное признаковое описание.

Батчнорм обычно ставят либо после каждого слоя, либо после каждого второго слоя.

3. Проблема переобучения

Стоит обратить внимание на большую проблему — переобучение. Батчнорм косвенно помогает бороться с ней.

Проблема переобучения свойственна и решающим деревьям, и линейным моделям, если у них слишком много признаков, и, конечно, нейронным сетям.

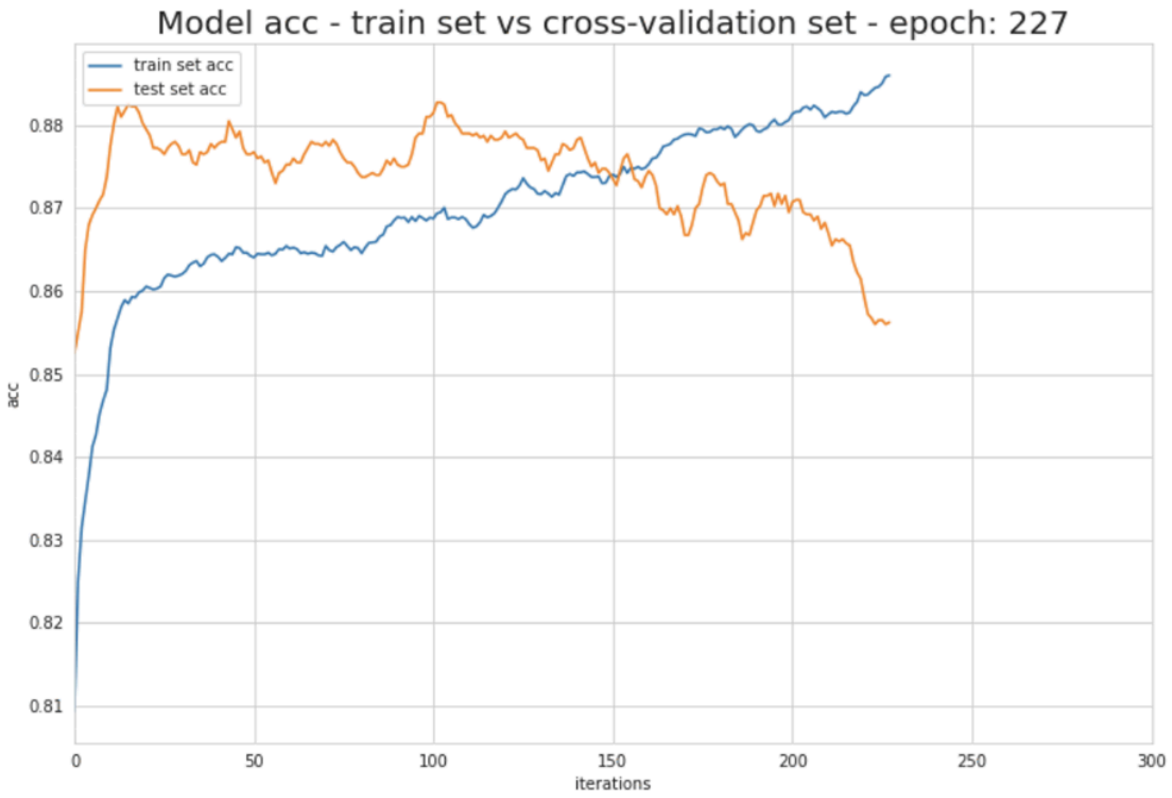
Переобучение — это ситуация, когда модель начинает запоминать специфичные для обучающей выборки зависимости, например, шум данных, вместо того, чтобы обобщать полученные данные.

Бороться с переобучением можно:

- либо ограничением сложности модели;
- либо чисткой данных;
- либо внесением дополнительных ограничений на модель.

Очень важно вовремя детектировать начало процесса переобучения. Для этого нужно строить качественный процесс валидации и следить за качеством модели.

Если качество на обучающей выборке постоянно растет, а на валидационной с некоторого момента начинает падать, это сигнал, что началось переобучение.



Рассмотрим подходы борьбы с переобучением.

Регуляризация

Регуляризация — ограничение на веса модели.

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{i\neq y_i} max\left(0,\ f\left(x_i;\ W\right)_j - f\left(x_i;\ W\right)_{y_i} + 1\right) + \lambda R(W).$$

Мы уже изучили L1 и L2 регуляризации:

- L2 регуляризация $R(W) = ||W||_2^2$
- L1 регуляризация $R(W) = ||W||_1$

- Линейная комбинация: (L1 + L2): $R(W) = \beta||W||_2^2 + ||W||_1$

Норму вектора весов $||W||$ добавляем в функцию потерь L. Теперь функция потерь получает штраф за слишкам большие веса. Тем самым мы заставляем модель сохранять маленькие веса.

Как и с линейными моделями, L1 регуляризация будет зануллять некоторые веса, L2 регуляризация будет пытаться сделать веса более-менее одинаковыми.

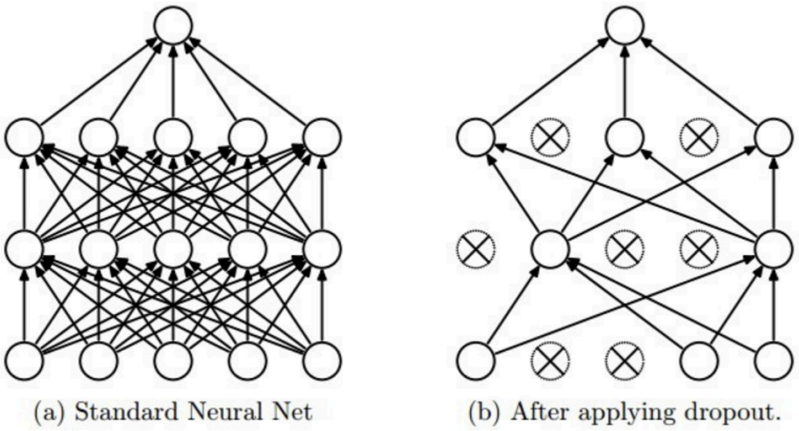
Регуляризацию часто включают прямо внутрь оптимизатора.

Метод регуляризации является общим для всех задач.

Dropout

Метод Dropout регуляризации является специфичным для нейронных сетей. Подробнее можно почитать про этот метод в статье.

Dropout позволяет «выкинуть» некоторые промежуточные представления (нейроны).



(a) Standard Neural Net (b) After applying dropout.

Мы зануляем некоторые промежуточные признаки, чтобы они не обучались и не влияли на предсказание. Таким образом зануляются и градиенты, идущие через эти латентные признаки. Это позволяет не переобучаться на определенные латентные представления и получить чуть более устойчивую модель.

Стоит понимать, что использование Dropout по факту снижает эффективную обобщающую способность модели.

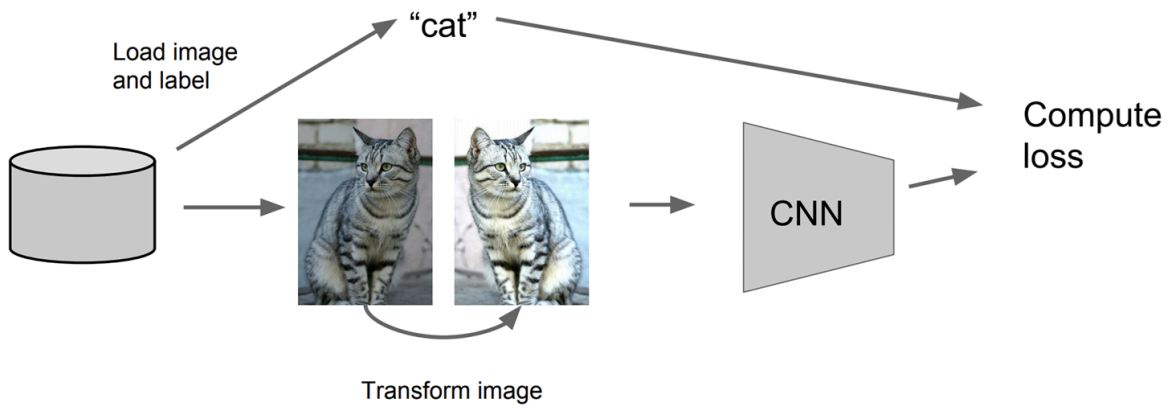
Dropout также требует небольшой перенормировки данных на этапе инференса.

4. Аугментация и итоги

Аугментация — третий тип регуляризации. Аугментация данных — это, по сути, внесение дополнительных примеров, на которых будет обучаться модель. Эти примеры получены из уже имеющейся обучающей выборки с помощью некоторых преобразований, к которым наша модель должна быть инвариантна (не чувствительна).

Пример. Есть фото кота. Мы все равно увидим кота, если:

- отразить эту фотографию вдоль вертикальной оси;
- немного повысить или понизить яркость;
- добавить мелкодисперсный шум.



Ко всем преобразованиям модель не должна быть чувствительна. Это позволит нам задать некоторые преобразования над данными. Во-первых, получить большую обучающую выборку. Во-вторых, сделать модель более устойчивой к таким преобразованиям.

Но стоит помнить, что чем сложнее обучающая выборка, тем сложнее будет модель. Поэтому, если добавить слишком много преобразований, особенно не подходящих, это сильно замедлит модель.

Аугментации, применяемые к данным, должны быть релевантны решаемой задаче.

Если решаем задачу распознавания текста по фотографии, то отображать его относительно вертикальной или горизонтальной оси — бессмысленное преобразование.

Итоги:

- Регуляризация бывает различных типов.
- Регуляризация — это попытка внести дополнительные экспертные предположения в решаемую задачу. Это можно делать:
    - на уровне оптимизируемого функционала;
    - структуры модели;
    - самих данных, используя аугментацию.
- Стоит помнить, что нейронные сети сами по себе не решают все задачи. Это класс моделей, который позволяет решать широкий спектр задач благодаря тому, что нейронные сети способны самостоятельно выучить информативное признаковое представление на основе примеров.
- Нейронные сети — это комбинация линейных и нелинейных преобразований. Функции активации обладают своими определенными свойствами. Их следует выбирать подходящими данному случаю.
- При решении оптимизационной задачи важно помнить, что градиентный спуск — это очень важная часть работы модели. Нужно следить за качеством модели и правильно подбирать learning rate. И, конечно, следить, чтобы модель не переобучилась.

5. PyTorch: модель с регуляризацией

Затухающие градиенты

Затухающий градиент — явление слабого изменения большого параметра из-за маленьких входных данных градиента (learning rate, коэффициент SGD и т. д.).

Посмотрим, из-за чего случаются затухающие градиенты.

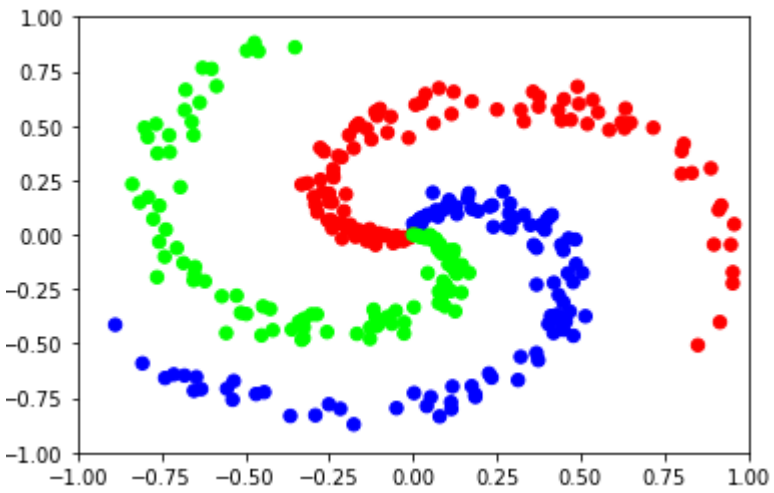
```
import numpy as np
import torch
print('PyTorch version: {}'.format(torch.__version__))
from torch import nn


import matplotlib.pyplot as plt
%matplotlib inline
```

Зададим данные искусственно. Сгенерируем простой двумерный датасет для задачи классификации. Выборка линейно неразделима, поэтому обратимся к нейронным сетям:

```
#generate random data -- not linearly separable
np.random.seed(0)
N = 100 # number of points per class
D = 2 # dimensionality
K = 3 # number of classes
X = np.zeros((N*K,D))
num_train_examples = X.shape[0]
y = np.zeros(N*K, dtype='uint8')
for j in range(K):
  ix = range(N*j,N*(j+1))
```

```
    r = np.linspace(0.0,1,N) # radius
    t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
    X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
    y[ix] = j
fig = plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.brg)
plt.xlim([-1,1])
plt.ylim([-1,1])
```

```
(-1.0, 1.0)
```



На изображении три класса. Мы можем разделить их при помощи какого-нибудь хитрого ядра в SVM, но попробуем решить эту задачу классификации с помощью нейронной сети.

Воспользуемся простой нейронной сетью с двумя скрытыми слоями. В качестве функций активации будем использовать `Sigmoid` и `ReLU`:

```
N_FEATURES_SMALL = 2
N_CLASSES_SMALL = 3
```

```
NUM_EPOCH = 50000
```

Для удобства ниже реализована функция, конструирующая модель. Функция активации и размер скрытого представления выступают в роли гиперпараметров:

```
def create_model(activation, hid_size=50, num_features=N_FEATURES_SMALL,
n_out=N_CLASSES_SMALL):
    model = nn.Sequential()
    model.add_module('l1', nn.Linear(num_features, hid_size))
    model.add_module('activation1', activation())
    model.add_module('l2', nn.Linear(hid_size, hid_size))
    model.add_module('activation2', activation())
    model.add_module('l3', nn.Linear(hid_size, n_out))

    return model
```

Как помним, активации нужны, чтобы выход данных слоя был нелинейным, и нейронка не была линейным преобразованием входных данных.

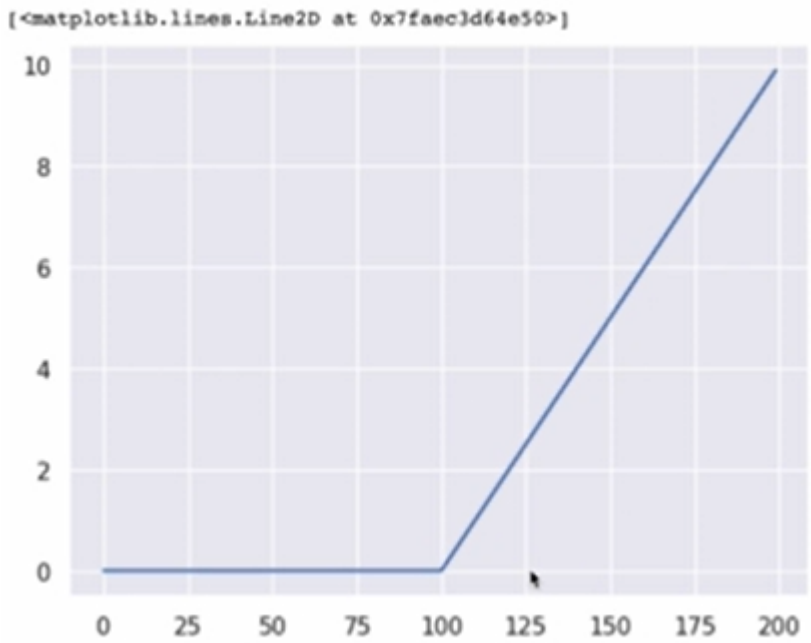
```
model_relu = create_model(nn.ReLU)
```

```
model_relu
```

```
    Sequential(
      (l1): Linear(in_features=2, out_features=50, bias=True)
      (activation1): ReLU()
      (l2): Linear(in_features=50, out_features=50, bias=True)
      (activation2): ReLU()
      (l3): Linear(in_features=50, out_features=3, bias=True)
    )
>>>
```

```
x = np.arange(-10, 10, .1)
relu = lambda x: np.maximum(x, 0)
plt.plot(relu(x))
```

>>>



Следующая функция обучает модель и логгирует полезную информацию:

```
def train_model(model, X_train, y_train):
    loss_function = nn.CrossEntropyLoss()
    opt = torch.optim.SGD(model.parameters(), lr=0.1)



    loss_history, var_layer_1, var_layer_2, var_layer_3 = [], [], [], []
```

```
for epoch_num in range(NUM_EPOCH):
    opt.zero_grad()

    y_predicted = model(X_train)
    loss = loss_function(y_predicted, y_train)
    reg = 0.01 * (torch.norm(model[0].weight) \
                + torch.norm(model[2].weight) \
                + torch.norm(model[4].weight))
    loss += reg
    loss.backward()
    opt.step()

    loss_history.append(loss.item()) # Always use .item() to store scalars in logs!


        var_layer_1.append(torch.sum(torch.abs(model[0].weight.grad)).item()/model[0].w
        eight.shape[0])

        var_layer_2.append(torch.sum(torch.abs(model[2].weight.grad)).item()/model[2].w
        eight.shape[0])

        var_layer_3.append(torch.sum(torch.abs(model[4].weight.grad)).item()/model[4].w
        eight.shape[0])

return model, loss_history, var_layer_1, var_layer_2, var_layer_3
```

```
X_torch = torch.FloatTensor(X)
```

```
y_torch = torch.LongTensor(y)
```

```
model_relu = create_model(nn.ReLU)
model_sigmoid = create_model(nn.Sigmoid)
```

Обучим нашу модель на ReLU:

```
%%time
out_relu = train_model(model_relu, X_torch, y_torch)
```

>>> CPU times: user 13.3 s, sys: 298 ms, total: 13.6 s

Wall time: 13 s

На Sigmoid:

```
%%time
out_sigmoid = train_model(model_sigmoid, X_torch, y_torch)
```

>>> CPU times: user 18.2 s, sys: 277 ms, total: 18.5 s

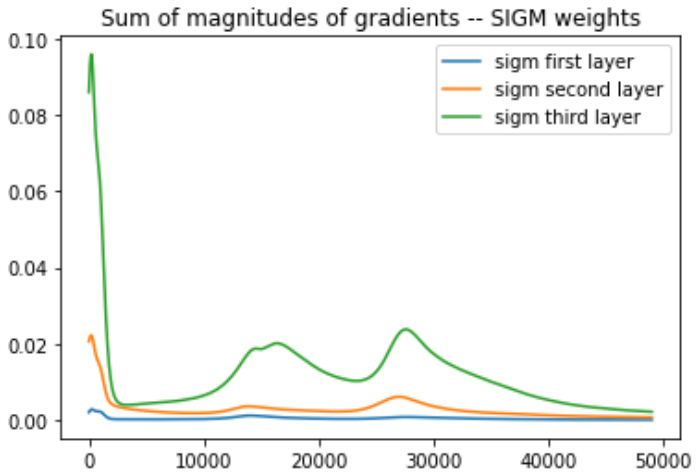
Wall time: 18.2 s

Рассмотрим среднее значение градиента на каждом из слоев. Можно заметить, что чем «дальше» от функции потерь слой, тем меньший градиент получают его параметры:

```
plt.plot(np.array(out_sigmoid[2][1000:]), label="sigm first layer")
plt.plot(np.array(out_sigmoid[3][1000:]), label="sigm second layer")
```

```
plt.plot(np.array(out_sigmoid[4][1000:]), label="sigm third layer")
plt.title('Sum of magnitudes of gradients -- SIGM weights')
plt.legend()
```

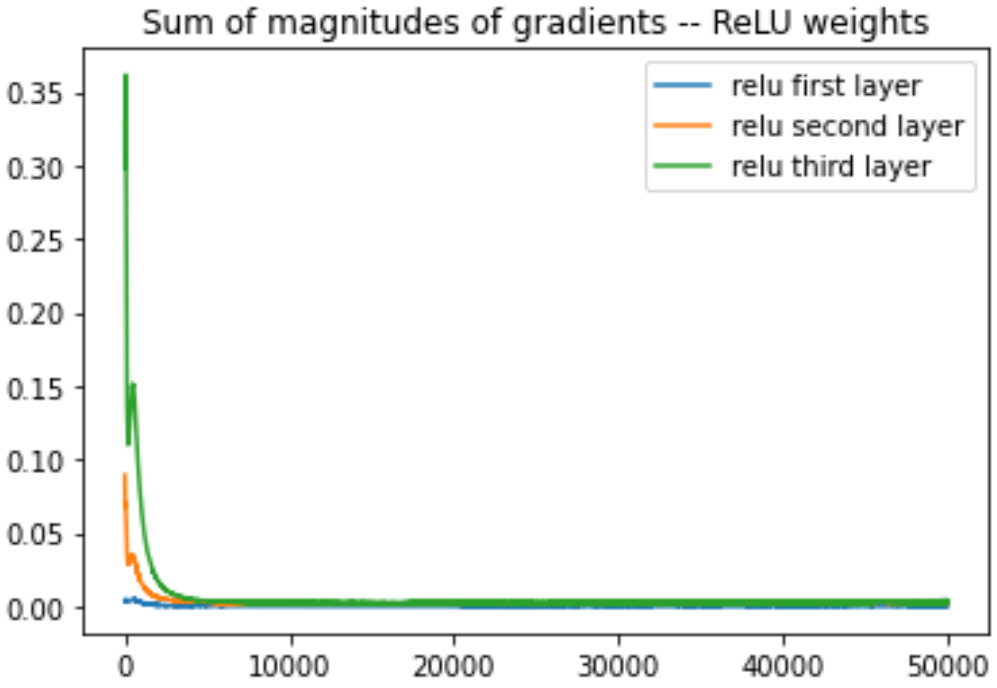
```
<matplotlib.legend.Legend at 0x15dbb5280>
```



Рассмотрим аналогичный график для функции активации ReLU:

```
plt.plot(np.array(out_relu[2]), label="relu first layer")
plt.plot(np.array(out_relu[3]), label="relu second layer")
plt.plot(np.array(out_relu[4]), label="relu third layer")
plt.title('Sum of magnitudes of gradients -- ReLU weights')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x15dc54880>
```



И объединим все результаты на едином графике:

```
# Overlaying the two plots to compare
plt.plot(np.array(out_sigmoid[2][5:]), label="sigm first layer")
plt.plot(np.array(out_sigmoid[3][5:]), label="sigm second layer")
# plt.plot(np.array(out_sigmoid[4]), label="sigm third layer")

plt.plot(np.array(out_relu[2][5:]), label="relu first layer")
plt.plot(np.array(out_relu[3][5:]), label="relu second layer")
# plt.plot(np.array(out_relu[4]), label="relu third layer")

plt.title('Sum of magnitudes of gradients -- hidden layer neurons')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x15dc22520>
```

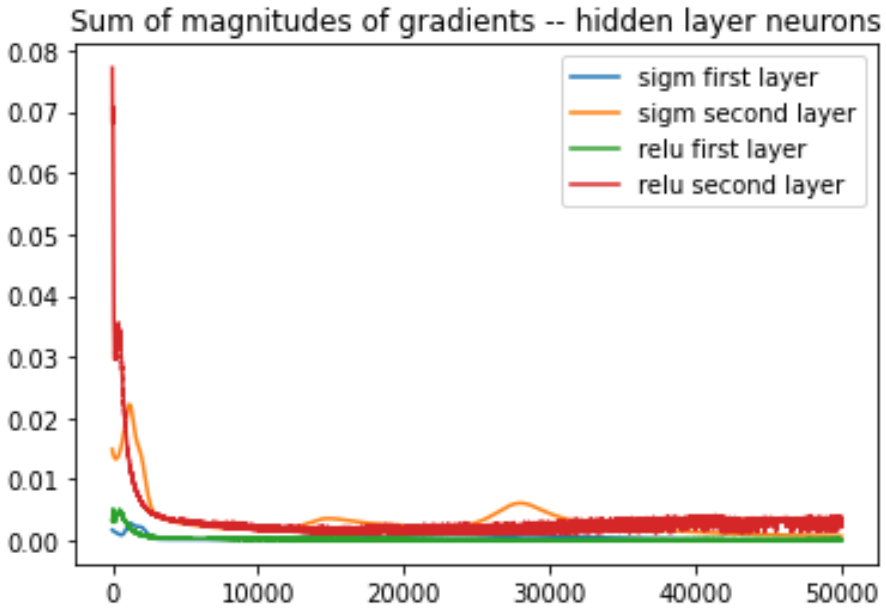
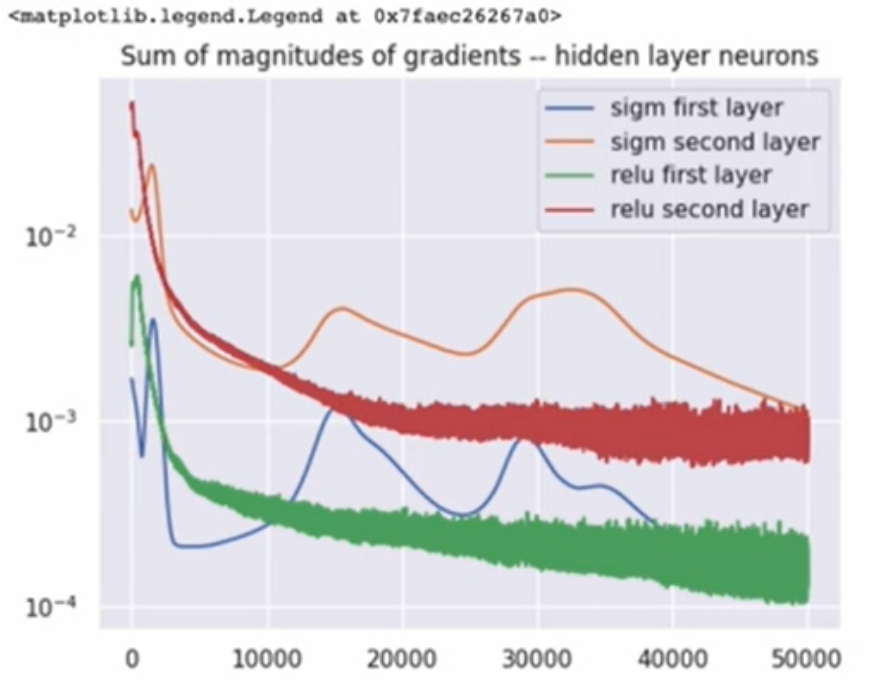


График не очень хорошо читается. Добавим функцию semilogy, которая берет логарифмический масштаб по оси *y*:

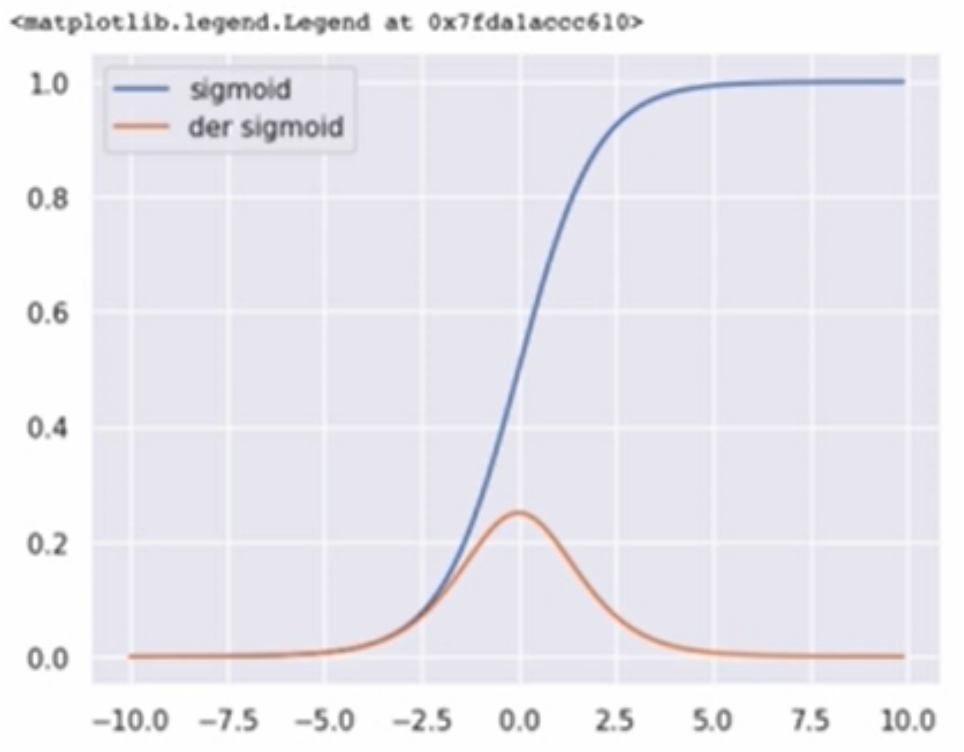
```
# Overlaying the two plots to compare
plt.semilogy(np.array(out_sigmoid[2][5:]), label="sigm first layer")
plt.semilogy(np.array(out_sigmoid[3][5:]), label="sigm second layer")
# plt.plot(np.array(out_sigmoid[4]), label="sigm third layer")

plt.semilogy(np.array(out_relu[2][5:]), label="relu first layer")
plt.semilogy(np.array(out_relu[3][5:]), label="relu second layer")
# plt.plot(np.array(out_relu[4]), label="relu third layer")

plt.title('Sum of magnitudes of gradients -- hidden layer neurons')
plt.legend()
```



Подробнее о том, что это за функция:

```
def sigmoid(x):
        return l./(1+np.exp(-x))


x = np.arange(-10, 10, 0.1)
```

```
plt.plot(x, sigmoid(x), label='sigmoid')
plt.plot(x, sigmoid(x)*(1-sigmoid(x)), label='der sigmoid') # производная от сигмоиды
plt.legend()
```



Производная от сигмоиды в своем максимальном значении сильно меньше 1. Если рассматривать ReLU, то его производная до определенного значения 0, потом скачок и постоянное значение производной (1 или больше) — то есть градиент как минимум не изменится. Для сигмоиды на выходе получаем ¼ умноженное на количество слоев, что является маленьким числом. Градиенты затухнут и не дойдут до начальных слоев.

Попробуйте изменить гиперпараметры и структуру сети. Можете ответить на следующие вопросы:

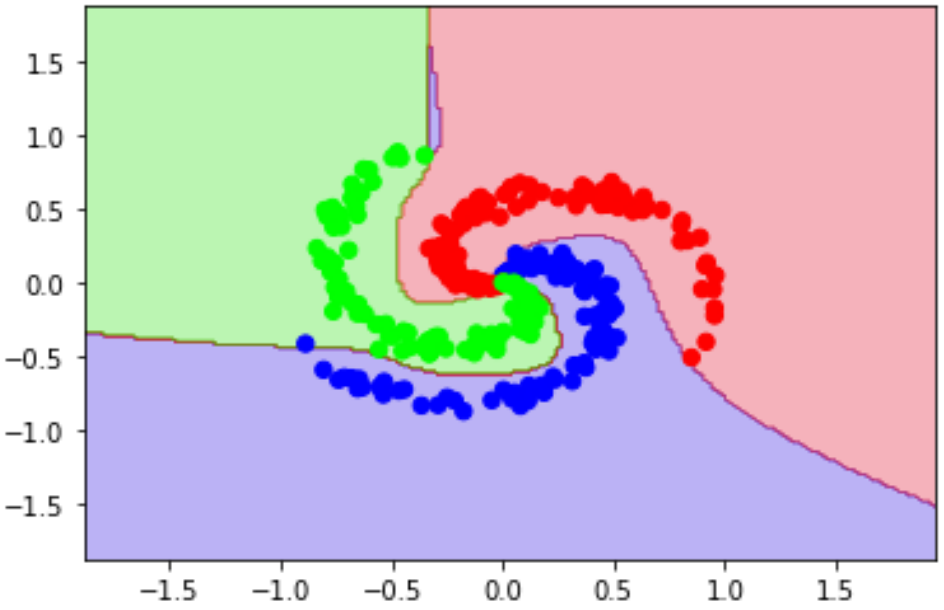
- Насколько быстро сеть будет обучаться, если заменить `ReLU` на `LeakyReLU`? На `Tanh`?
- Как изменится поведение сети, если изменить коэффициент регуляризации?
- Как изменится поведение сети, если изменить `learning rate`?

Также стоит обратить внимание на итоговое качество решения задачи. Для `Sigmoid` может понадобиться большее число итераций или значение `learning rate`.

```
# plot the classifiers- SIGMOID
h = 0.02
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

X_draw = torch.FloatTensor(np.c_[xx.ravel(), yy.ravel()])
```

```
with torch.no_grad():
    Z = model_sigmoid(X_draw).numpy()
Z = np.argmax(Z, axis=1)
Z = Z.reshape(xx.shape)
fig = plt.figure()
plt.contourf(xx, yy, Z, cmap=plt.cm.brg, alpha=0.3)
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.brg)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
```

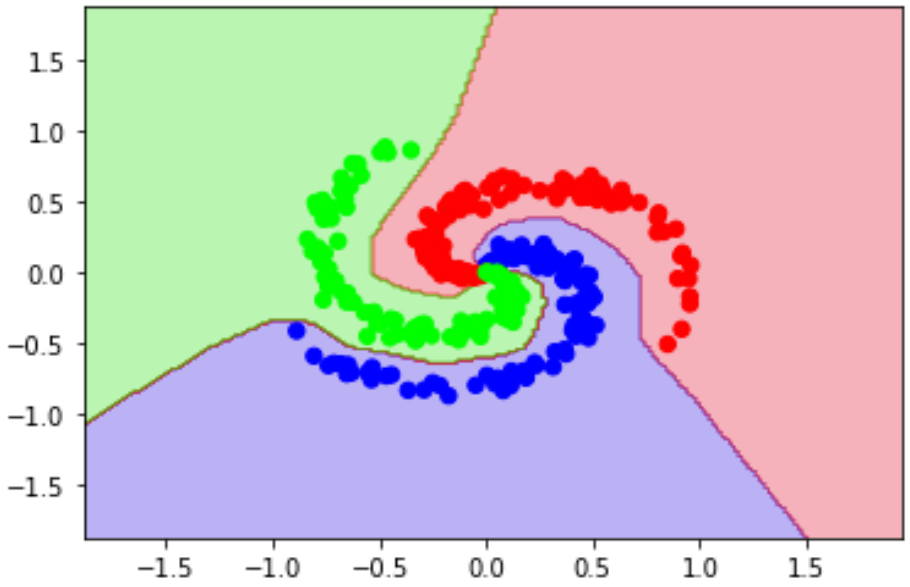
```
(-1.8712034092398278, 1.8687965907601756)
```



На модели с ReLU:

```
with torch.no_grad():
    Z = model_relu(X_draw).numpy()
Z = np.argmax(Z, axis=1)
Z = Z.reshape(xx.shape)
fig = plt.figure()
plt.contourf(xx, yy, Z, cmap=plt.cm.brg, alpha=0.3)
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.brg)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
```

```
(-1.8712034092398278, 1.8687965907601756)
```



И для модели сигмоиды и для модели ReLU получились примерно одинаковые картинки, поскольку мы задали не очень глубокую нейронную сеть и легкую задачу.

Обзор методов регуляризации в нейронных сетях

В процессе регуляризации к функции потерь мы добавляем некоторое слагаемое, которое накладывает ограничения на веса.

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{i\neq j} max\left(0,\ f\left(x_i,\ W\right)_j - f\left(x_i,\ W\right)_{y_i} + 1\right) + \lambda R(W)$$

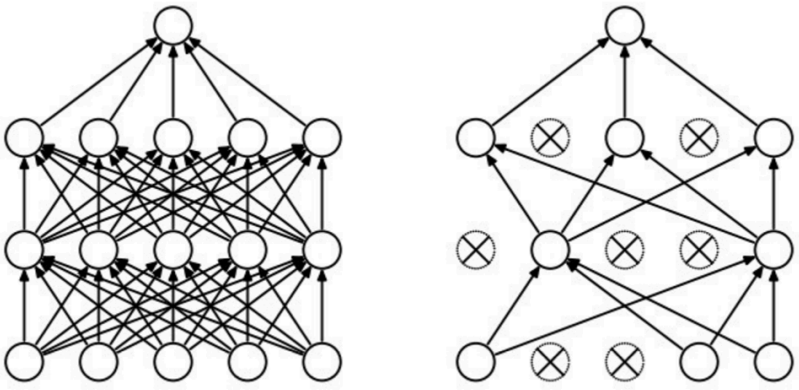
Типы регуляризаций:

- L2 регуляризация $R(W) = ||W||_2^2$
- L1 регуляризация $R(W) = ||W||_1$
- Elastic Net (L1 + L2) $R(W) = \beta||W||_2^2 + ||W||_1$

L2 регуляризация не позволяет получать большие по амплитуде веса. L1 регуляризация создает более разреженную структуру весов.

Существует также техника Dropout.

Dropout действителен только на процессе обучения. Он выключает некоторые нейроны в нейронной сетке и делает так, что наша сетка обучается на меньшем количестве нейронов. Для каждого нового подхода в Dropout это новая сетка с другими выключенными нейронами. Мы обучаем несколько основных моделей, то есть, грубо говоря, — ансамбль.



(a) Standard Neural Net (b) After applying dropout.

У Dropout есть параметр вероятности — с какой вероятностью будут включены или выключены те или иные нейроны. Каждый нейрон учится работать один, он становится более приспособленным к разным поднаборам других нейронов. В режиме обучения с применением Dropout каждый нейрон включается с вероятностью p для веса w. В тестовом режиме уже Dropout не используется, и вклад нейрона будет pw.

В PyTorch немного другая парадигма. Один нейрон отключается с вероятностью p. Оставшиеся веса умножаются на значение $\frac{1}{1-p}$, а при тесте никакого умножения на вероятности нет.

Посмотрим, как это выглядит в коде. Создадим нейросетку в режиме train:

```
m = nn.Dropout(p=0.2)
```

```
input = torch.randn(2, 4)
output = m(input)
```

```
input, output
```

```
(tensor([[-0.8059,  0.3559, -1.4146, -1.4363],
         [ 1.7925, -0.3982,  1.3144, -1.3099]]),
 tensor([[-0.0000,  0.0000, -1.7683, -1.7953],
         [ 2.2406, -0.4977,  1.6430, -1.6373]]))
>>>
```

```
input/0.8
```

```
tensor([[-1.0073,  0.4449, -1.7683, -1.7953],
        [ 2.2406, -0.4977,  1.6430, -1.6373]])
>>>
```

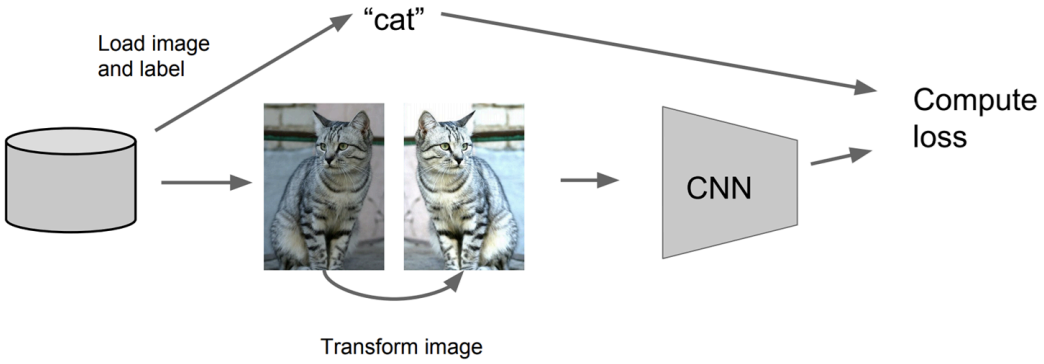
В режиме теста:

```
with torch.no_grad():
        m.eval()
        output = m(input)
```

```
output
```

Output будет такой же, как Input на train.

Рассмотрим еще аугментацию. Она широко используется для обработки изображений:



У Torch много возможностей для преобразования картинок transforms. Библиотека TorchTransforms будет работать, если нужно преобразовать сами данные, но не таргеты, например, в задаче классификации изображения.

В задаче сегментации и данные, и таргеты должны быть трансформированы одинаково, то есть при повороте данных маска тоже должна быть повернута. В этом случае TorchTransforms не работает, он будет отдельно рандомно поворачивать и данные, и маску. Здесь пригодится библиотека albumentations.

Работа с текстами и последовательностями

Переходим к работе с данными, обладающими некоторой внутренней структурой. Введем определение для последовательностей, в частности, текстов.

- **Последовательность** представляет собой набор значений, на которых задано некоторое отношение порядка. Значения могут быть как дискретными (например, в ДНК), так и принимать значения из непрерывного интервала (например, история цен на бирже). Играет роль и порядок значений — их перестановка приведет к потере информации. Если присутствует упорядоченность по времени, то стоит помнить, что нельзя ее нарушать.
- **Тексты** являются частным случаем последовательностей и являются последовательностями значений из дискретного и конечного алфавита. Ввиду существования грамматики они обладают достаточно строгой внутренней структурой.

Для иллюстрации работы с текстами обратимся к задаче классификации отзывов. Для этого воспользуемся широко известным датасетом SST2. Для каждого отзыва о фильме доступна метка класса: 1 для положительных отзывов и 0 для негативных.

Пример:

```
from collections import Counter


import numpy as np
import pandas as pd


from nltk.tokenize import WordPunctTokenizer


import torch
from torch import nn
from torch.nn import functional as F
from torch.optim.lr_scheduler import StepLR, ReduceLROnPlateau


from sklearn.metrics import accuracy_score


import matplotlib
import matplotlib.pyplot as plt
matplotlib.rcParams.update({'font.size': 16})
from IPython import display
%matplotlib inline
```

```
df = pd.read_csv(
```

```
'https://github.com/clairett/pytorch-sentiment-classification/raw/master/data/SST2/train.tsv',
    delimiter='\t',
    header=None
)
```

```
df.values[2]
```

>>> array(["they presume their audience wo n't sit still for a sociology lesson , however entertainingly presented , so they trot out the conventional science fiction elements of bug eyed monsters and futuristic women in skimpy clothes", 0], dtype=object)

```
df
```

	0	1
0	a stirring , funny and finally transporting re...	1
1	apparently reassembled from the cutting room f...	0
2	they presume their audience wo n't sit still f...	0
3	this is a visually stunning rumination on love...	1
4	jonathan parker 's bartleby should have been t...	1
...	...	...
6915	painful , horrifying and oppressively tragic ,...	1
6916	take care is nicely performed by a quintet of ...	0
6917	the script covers huge , heavy topics in a bla...	0
6918	a seriously bad film with seriously warped log...	0
6919	a deliciously nonsensical comedy about a city ...	1

6920 rows × 2 columns

>>>

```
df[1].unique()
```

>>> array([1, 0])

```
df[1].sum()
```

>>> 3610

```
df.shape
```

>>> (6920, 2)

```
texts = df[0].values[:5000]
labels = df[1].values[:5000]
```

```
texts_test = df[0].values[5000:]
labels_test = df[1].values[5000:]
```

Текст отличается от привычных нам данных сразу несколькими свойствами:

1. Представляет собой последовательность токенов из конечного алфавита, т. е. все элементы последовательности принимают дискретные значения.
2. Может быть переменной длины.

Остановимся подробнее на втором свойстве.

До текущего момента используемые нейронные сети работали лишь с данными фиксированной размерности. Для работы с текстами нужно либо предложить метод

представления текста в виде вектора фиксированной размерности, либо адаптировать структуру нейронной сети для работы с последовательностями. Сегодня мы остановимся на первом подходе.

В первую очередь текст нужно превратить в вектор, то есть получить эмбеддинг. **Токен** — минимальный и неделимый элемент текстовой последовательности. В зависимости от выбора эксперта в качестве токенов могут выступать как символы, так и морфемы, слова или даже группы слов.

Для начала построим словарь из всех возможных токенов. Он будет отображать токен и его индекс. Воспользуемся collections.Counter, чтобы также оценить частоту встречаемости слов в наборе данных.

Модель всегда работает в связке с токенайзером. Будем токенизировать модели вручную с помощью WordPunctTokenizer.

```
tokenizer = WordPunctTokenizer()
```

```
tokenizer.tokenize('This is another text, I am really sure!')
```

>>> ['This', 'is', 'another', 'text', ',', 'I', 'am', 'really', 'sure', '!']

```
WordPunctTokenize(“She said ‘hello’.”)
```

>>>[‘She’, ‘said’, “‘”, ‘hello’, “’.”]

Посмотрим, сколько раз встречаются токены, составим их список. Каждому токену присвоим свой индекс:

```
counter = Counter()

for text in texts:
```

```
    counter.update(tokenizer.tokenize(text.lower()))

tokens = set([token for token, count in counter.items() if count > 5])

token_to_idx = {token: idx for idx, token in enumerate(tokens)}
```

Размер нашего словаря:

```
len(tokens)
```

>>> 1839

Теперь представим каждый токен в виде вектора с помощью one-hot кодирования. Каждому токену в словаре был сопоставлен уникальный индекс. Тогда можно поставить в соответствие токену вектор размера словаря, где единственное ненулевое значение стоит на соответствующей токену позиции.

Например, слово «самолет» сопоставлено индексу 0, а размерность словаря — 5. Данному токену соответствует вектор `[1, 0, 0, 0, 0]`. Слово «обед» сопоставлено индексу 4, поэтому ему соответствует вектор `[0, 0, 0, 0, 1]`.

Для представления текста в виде вектора фиксированной размерности нужно учесть все входящие в него токены, игнорируя их порядок. Этот подход называется **мешком слов** или же **Bag of Words, BoW**. Конечно, он теряет большое количество информации, но все еще может быть использован в некоторых задачах (с небольшими усовершенствованиями).

```
def text_to_bow(text):
    bow = np.zeros(len(token_to_idx))

    for token in tokenizer.tokenize(text):
```

```
    if token in tokens:
        bow[token_to_idx[token]] += 1
return bow
```

```
text_to_bow('cats are perfect').sum()
```

>>> 2.0

У этого текста два ненулевых элемента:

```
len(text_to_bow('cats are perfect'))
```

>>> 1839

Длина вектора равна размерности словаря:

```
texts[22]
```

>>> "a wretched movie that reduces the second world war to one man 's quest to find an old flame"

```
Counter(labels)
```

>>> Counter({1: 2607, 0: 2393})

Затокенизируем все наши слова:

```
X_train_bow = np.stack(list(map(text_to_bow, texts)))
```

```
X_test_bow = np.stack(list(map(text_to_bow, texts_test)))
```

```
X_train_bow
```

```
>>> array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

```
len(X_train_bow)
```

>>>5000

Длина равна количеству всех имеющихся текстов в train.

Количество нулевых элементов:

```
sum(X_train_bow[0])
```

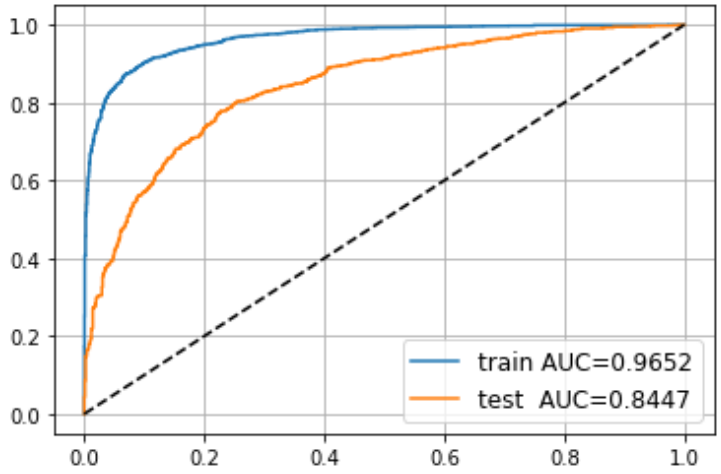
>>>14.0

Построим простую логистическую регрессию из `sklearn` в качестве baseline:

```
from sklearn.linear_model import LogisticRegression
bow_model = LogisticRegression(max_iter=1500).fit(X_train_bow, labels)
```

```
from sklearn.metrics import roc_auc_score, roc_curve, accuracy_score


for name, X, y, model in [
    ('train', X_train_bow, labels, bow_model),
    ('test ', X_test_bow, labels_test, bow_model)
]:
    proba = model.predict_proba(X)[:, 1]
    auc = roc_auc_score(y, proba)
    plt.plot(*roc_curve(y, proba)[:2], label='%s AUC=%.4f' % (name, auc))


plt.plot([0, 1], [0, 1], '--', color='black',)
plt.legend(fontsize='large')
plt.grid()


print('train accuracy', accuracy_score(model.predict(X_train_bow), labels))
print('test accuracy', accuracy_score(model.predict(X_test_bow), labels_test))
```

```
train accuracy 0.9012
test accuracy 0.7739583333333333
```



```
X_train_bow.shape, X_test_bow.shape
```

>>> ((5000, 1839), (1920, 1839))

```
NUM_FEATURES = len(tokens)
```

Теперь можно построить простую нейронную сеть для классификации данных отзывов:

```
model = nn.Sequential()
model.add_module('dropout', nn.Dropout(p=0.8))
model.add_module('l1', nn.Linear(NUM_FEATURES, 16))
model.add_module('relu', nn.ReLU())
model.add_module('dropout', nn.Dropout())
model.add_module('l2', nn.Linear(16, 2))
```

```
loss_function = nn.CrossEntropyLoss()
```

```
opt = torch.optim.Adam(model.parameters(), lr=3e-4)
# lr_scheduler = ReduceLROnPlateau(opt)
```

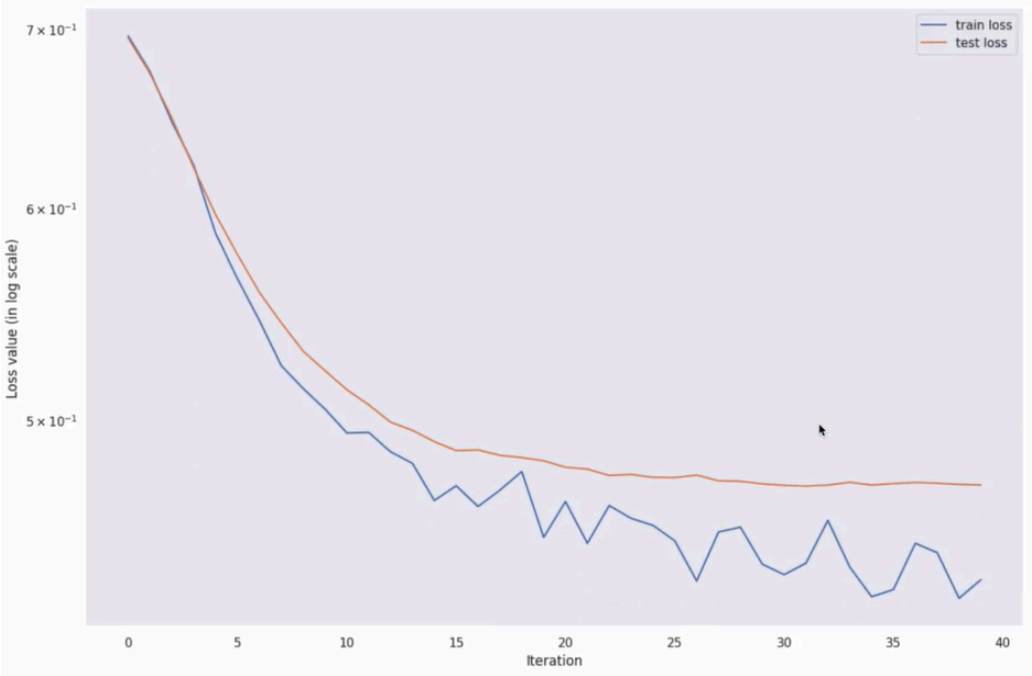
```
X_train_bow_torch = torch.FloatTensor(X_train_bow)
X_test_bow_torch = torch.FloatTensor(X_test_bow)
```

```
y_train_torch = torch.LongTensor(labels)
y_test_torch = torch.LongTensor(labels_test)
```

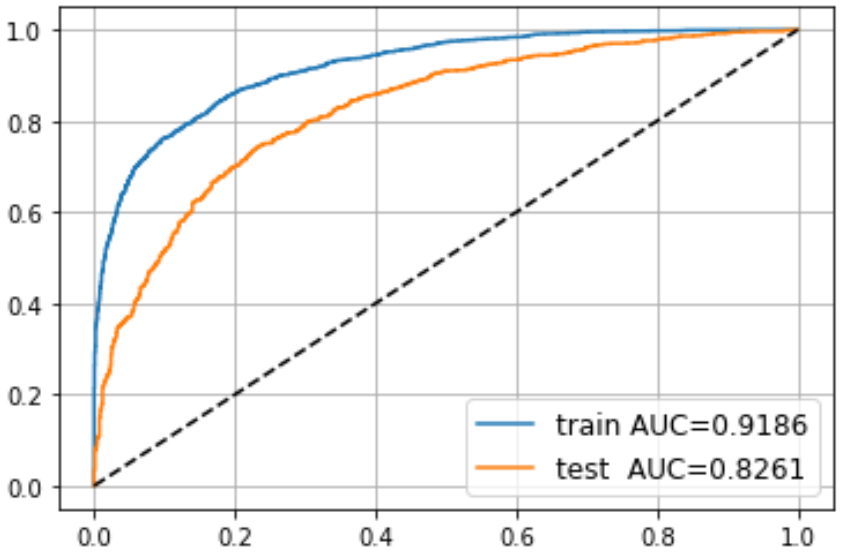
```
n_iterations = 2000
batch_size = 256
train_loss_history = []
train_acc_history = []
plot_history = []
test_plot_history = []
```

```
for _i in range(n_iterations):
    model.train()
    opt.zero_grad()

    ix = np.random.randint(0, len(X_train_bow_torch), batch_size)
    x_batch = X_train_bow_torch[ix]
    y_batch = y_train_torch[ix]

    y_predicted = model(x_batch)
    loss = loss_function(y_predicted, y_batch)
    loss.backward()
    opt.step()

    train_loss_history.append(loss.data.numpy())
    train_acc_history.append(
```

```
        accuracy_score(
            y_batch.detach().numpy(),
            y_predicted.detach().numpy().argmax(axis=1)
        )
    )

    if (_i+25) % 50==0:
        model.eval()
        plot_history.append(np.mean(train_loss_history[-10:]))
        test_plot_history.append(loss_function(model(X_test_bow_torch),
y_test_torch).item())

        display.clear_output(True)
        plt.figure(figsize=(15, 10))
        plt.plot(plot_history, label='train loss')
        plt.plot(test_plot_history, label='test loss')
        plt.yscale('log')
        plt.grid()
        plt.xlabel('Iteration')
        plt.ylabel('Loss value (in log scale)')
        plt.legend()
        plt.show()
```




```
from sklearn.metrics import roc_auc_score, roc_curve


for name, X, y, model in [
    ('train', X_train_bow_torch, labels, model),
    ('test ', X_test_bow_torch, labels_test, model)
]:
    model.eval()
    proba = model(X).detach().cpu().numpy()[:, 1]
    auc = roc_auc_score(y, proba)
    plt.plot(*roc_curve(y, proba)[:2], label='%s AUC=%.4f' % (name, auc))


plt.plot([0, 1], [0, 1], '--', color='black',)
plt.legend(fontsize='large')
plt.grid()
```




```
train_predictions = torch.max(model(X_train_bow_torch).detach().cpu(),
dim=1)[1].numpy()

test_predictions = torch.max(model(X_test_bow_torch).detach().cpu(), dim=1)[1].numpy()


print('train accuracy', accuracy_score(train_predictions, labels))

print('test accuracy', accuracy_score(test_predictions, labels_test))
```

```
>>> train accuracy 0.8374

    test accuracy 0.7578125
```

Видим, что сеть достаточно сильно переобучается. Попробуем воспользоваться изученными техниками регуляризации, например, Dropout.

Обращаем внимание, что Dropout можно использовать как для промежуточных представлений, так и для исходного признакового описания данных.

Выводы

- Различные функции активации обладают различными свойствами. На практике важно о них помнить. Отличное решение по умолчанию — ReLU.
- Затухающие градиенты — существенная проблема для нейронных сетей. В целом, она свойственна любым нейронным сетям, но больше всего проявляет себя в глубоких. Методы частичного решения этой проблемы будут рассмотрены позже.
- Важно вносить ограничения на веса, структуру модели или данные, если они обусловлены решаемой задачей или экспертными знаниями.
- Dropout позволяет получить более устойчивую модель.
- Аугментации данных должны быть осмысленными.
- Dropout (в некотором смысле) эквивалентен ансаблированию моделей.

Дополнительные материалы для самостоятельного изучения

1. Lecture 7: Training Neural Networks, Part 2
2. Neural Networks for Machine Learning. Overview of mini-batch gradient descent
3. Behavior around a saddle point: gif
4. Neural Networks Part 3: Learning and Evaluation. CS231n: Convolutional Neural Networks for Visual Recognition
5. Shotgun Debugging. LTFN 7: A Quick Look at TensorFlow Optimizers
6. Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
7. Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting
8. Радослав Нейчев. Занятие 3. Методы регуляризации в DL. Введение в глубокое обучение
9. TRANSFORMING AND AUGMENTING IMAGES. PyTorch

10. Deeply Moving: Deep Learning for Sentiment Analysis. Sentiment Analysis
11. Table of Contents: class collections.Counter
12. Сравнение разных способов оптимизации: иллюстрация
13. Значение градиента в качестве нормировочной константы в идее AdaGrad и RMSProp