Home work 1: Basic Artificial Neural Networks

Credit https://github.com/yandexdataschool/YSDA_deeplearning17 (https://github.com/yandexdataschool/YSDA_deeplearning17), https://github.com/DmitryUlyanov (https://github.com/DmitryUlyanov)

Зачем это всё нужно?! Зачем понимать как работают нейросети внутри когда уже есть куча библиотек?

- Время от времени Ваши сети не учатся, веса становятся nan-ами, все расходится и разваливается -- это можно починить если понимать бекпроп
- Если Вы не понимаете как работают оптимизаторы, то не сможете правильно выставить гиперапрааметры :) и тоже ничего выучить не выйдет
- https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b (https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b)

The goal of this homework is simple, yet an actual implementation may take some time :). We are going to write an Artificial Neural Network (almost) from scratch. The software design of was heavily inspired by Torch (http://torch.ch) which is the most convenient neural network environment when the work involves defining new layers.

This homework requires sending **"multiple** files, please do not forget to include all the files when sending to TA. The list of files:

- This notebook
- hw1_Modules.ipynb

If you want to read more about backprop this links can be helpfull:

- http://udacity.com/course/deep-learning--ud730 (http://udacity.com/course/deep-learning--ud730)
- http://cs231n.stanford.edu/2016/syllabus.html (http://cs231n.stanford.edu/2016/syllabus.html)
- http://www.deeplearningbook.org (http://www.deeplearningbook.org)

Check Questions

Вопрос 1: Чем нейросети отличаются от линейных моделей а чем похожи?

Линейная модель - частный случай нейросети, состоящей из одного полносвязного слоя.

Вопрос 2: В чем недостатки полносвзяных нейронных сетей какая мотивация к использованию свёрточных?

По сути полносвязная сеть не далеко ушла от линейных моделей. Она не имеет возможности учитывать локальность исследуемых объектов на картинке, то, что они могут быть повернуты на угол, могут принимать различные формы. Сверточные сети справляются с этой задачей, анализируя локальные оласти картинки.

Вопрос 3: Какие слои используются в современных нейронных сетях? Опишите как работает каждый слой и свою интуицию зачем он нужен.

- `DenceLayer` -- линейное преобразование `Wx + b`, можно интерпретировать как преобразования пространства размерности длины стобца `W` в пространство размерности длины строки `W`, очень похож на линейную алгебру и по этому работает так линейная модель
- `SoftMax` -- используется, чтобы привести `score` для разных классов задачи классификации к виду, сожему с вероятностями принадлежности каждому классу. Используется формула, похожая на сигмоиду.
- `Dropout` -- выкидывает слуайные элементы из выходного вектора. Является альтернативой рагуляризации.
- `LeakyReLU` -- нелинейность, домножающая входное значение на некий заданый коэффициент `a` в отрицателной области.
- `ReLU` -- нелинейноть, зануляющая входное значение в отрицательной области. Можно использовать между линейными слоями. Частный случай `LeakyReLU` при `a=1`

Вопрос 4: Может ли нейросеть решать задачу регрессии, какой компонент для этого нужно заменить в нейросети из лекции 1?

softmax заменяем на линейный полносвязный слой, переводящий отображающий вход в одно число.

Вопрос 5: Почему обычные методы оптимизации плохо работают с нейросетями? А какие работают хорошо? Почему они работают хорошо?

Поскольку шаг обычного градиентного спуска не меняется, он в большом количестве случаев может войти в колебания, из которых никогда не выйдет. Более сложные методы борятся с этой проблемой тем или иным образом меняя шаг от итерации к итерации. Это методы моментум, адаград, адам и другие.

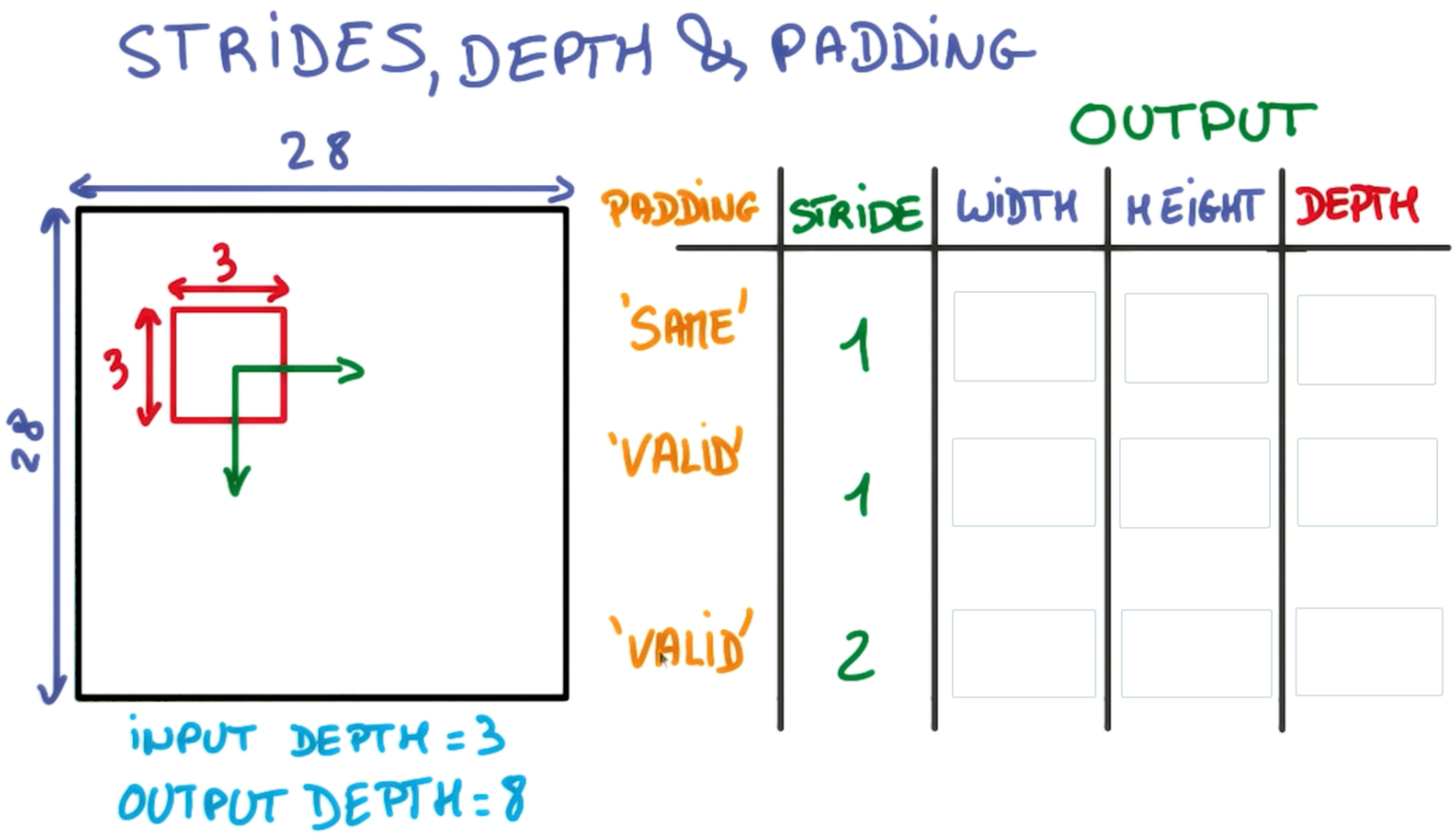
Вопрос 6: Для чего нужен backprop, чем это лучше/хуже чем считать градиенты без него? Почему backprop эффективно считается на GPU?

Без backprop было бы сложно составлять нейросеть как композицию различных слоев, не очень задумываясь о том, как производная считается в целом (т.е. пришлось бы для каждой сети думать, какие получаются частные производные по каждому параметру). backprop помагает абстрагироваться от этого, считая только некоторые проиводные для каждого слова и используя chain-rule. backprop лучше работает на GPU т.к. в этой задаче чаще всего все, что нужно делать это перемножать матрицы, а это очень легко параллелится. При этом в GPU ядер во много раз больше, чем в обычном процессоре.

Вопрос 7: Почему для нейросетей не используют кросс валидацию, что вместо неё? Можно-ли ее использовать?

Кросс-валидация работала бы слишком долго, сети и так обучаются неделями. Поэтому используют специлальные слои - Dropout

Вопрос 8: Небольшой quiz который поможет разобраться со свертками https://www.youtube.com/watch?v=DDRa5ASNdq4 (https://www.youtube.com/watch?v=DDRa5ASNdq4)



<Ответ-Картинка :)>

Политика списывания. Вы можете обсудить решение с одногрупниками, так интереснее и веселее :) Не шарьте друг-другу код, в этом случаи вы ничему не научитесь -- "мыши плакали кололись но продолжали жрать кактус".

Теперь формально. Разница между списыванием и помощью товарища иногда едва различима. Мы искренне надеемся, что при любых сложностях вы можете обратиться к семинаристам и с их подсказками **самостоятельно** справиться с заданием. При зафиксированных случаях списывания (одинаковый код, одинаковые ошибки), баллы за задание будут обнулены всем участникам инцидента.

In [1]:

```
%matplotlib inline
from time import time, sleep
import numpy as np
import matplotlib.pyplot as plt
from IPython import display
from sklearn.metrics import accuracy_score
```

!!! Не забывайте делать градиент чекинг, как в семинаре !!!

Framework

Implement everything in `Modules.ipynb`. Read all the comments thoughtfully to ease the pain. Please try not to change the prototypes.

Do not forget, that each module should return AND store `output` and `gradInput`.

The typical assumption is that `module.backward` is always executed after `module.forward`, so `output` is stored, this would be useful for `SoftMax`.

In [2]:

```
"""
    ----------------------------------------
    -- Tech note
    ----------------------------------------
    Inspired by torch I would use

    np.multiply, np.add, np.divide, np.subtract instead of *,+,/,-
    for better memory handling

    Suppose you allocated a variable

        a = np.zeros(...)

    So, instead of

        a = b + c  # will be reallocated, GC needed to free

    I would go for:

        np.add(b,c,out = a) # puts result in `a`

    But it is completely up to you.
"""
%run hw1_Modules.ipynb
```

Optimizer is implemented for you.

In [3]:

```
def sgd_momentum(x, dx, config, state):
    """
        This is a very ugly implementation of sgd with momentum
        just to show an example how to store old grad in state.

        config:
            - momentum
            - learning_rate
        state:
            - old_grad
    """

    # x and dx have complex structure, old dx will be stored in a simpler one
    state.setdefault('old_grad', {})

    i = 0
    for cur_layer_x, cur_layer_dx in zip(x,dx):
        for cur_x, cur_dx in zip(cur_layer_x,cur_layer_dx):
            cur_old_grad = state['old_grad'].setdefault(i, np.zeros_like(cur_dx))
            np.add(config['momentum'] * cur_old_grad, config['learning_rate'] * c
            state['old_grad'][i] = cur_old_grad
            cur_x -= cur_old_grad
            i += 1
```

In [4]:

```
t = 0
def adam(x, dx, config, state):
    global t
    # x and dx have complex structure, old dx will be stored in a simpler one
    state.setdefault('old_grad', {})
    t += 1
    i = 0
    for cur_layer_x, cur_layer_dx in zip(x,dx):
        for cur_x, cur_dx in zip(cur_layer_x,cur_layer_dx):
            cur_old_grad = state['old_grad'].setdefault(i, np.zeros_like(cur_dx))
            config['m'][i] = config['beta_1'] * config['m'].setdefault(i, 0) + (1
            config['v'][i] = config['beta_2'] * config['v'].setdefault(i, 0) + (1
            mb = config['m'][i] / (1 - config['beta_1'] ** t)
            vb = config['v'][i] / (1 - config['beta_2'] ** t)
            state['old_grad'][i] = cur_old_grad
            cur_x += -config['learning_rate'] * mb / (np.sqrt(vb) + 1e-7)
            i += 1
```

Toy example

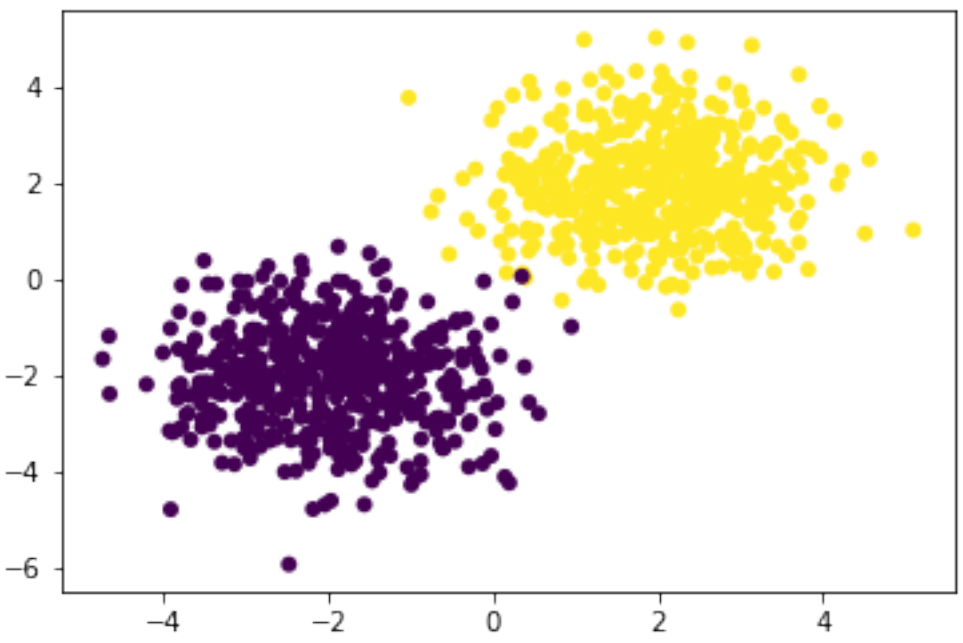
Use this example to debug your code, start with logistic regression and then test other layers. You do not need to change anything here. This code is provided for you to test the layers. Also it is easy to use this code in MNIST task.

In [5]:

```
# Generate some data
N = 500

X1 = np.random.randn(N,2) + np.array([2,2])
X2 = np.random.randn(N,2) + np.array([-2,-2])

Y = np.concatenate([np.ones(N),np.zeros(N)])[:,None]
Y = np.hstack([Y, 1-Y])

X = np.vstack([X1,X2])
plt.scatter(X[:,0],X[:,1], c = Y[:,0], edgecolors= 'none')
```

Out[5]:

```
<matplotlib.collections.PathCollection at 0x106a03dd8>
```



Define a **logistic regression** for debugging.

In [6]:

```
'''
net = Sequential()
net.add(Linear(2, 2))
net.add(SoftMax())

criterion = ClassNLLCriterion()

print(net)
'''
#Test something like that then

net = Sequential()
net.add(Linear(2, 4))
net.add(ReLU())
net.add(Linear(4, 2))
net.add(SoftMax())

criterion = ClassNLLCriterion()

print(net)
```

```
Linear 2 -> 4
ReLU
Linear 4 -> 2
SoftMax
```

Start with batch_size = 1000 to make sure every step lowers the loss, then try stochastic version.

In [7]:

```
# batch generator
def get_batches(dataset, batch_size):
    X, Y = dataset
    n_samples = X.shape[0]

    # Shuffle at the start of epoch
    indices = np.arange(n_samples)
    np.random.shuffle(indices)

    for start in range(0, n_samples, batch_size):
        end = min(start + batch_size, n_samples)

        batch_idx = indices[start:end]

        yield X[batch_idx], Y[batch_idx]
Y = np.array([int(x) for x in Y[:,0]])
```

Train

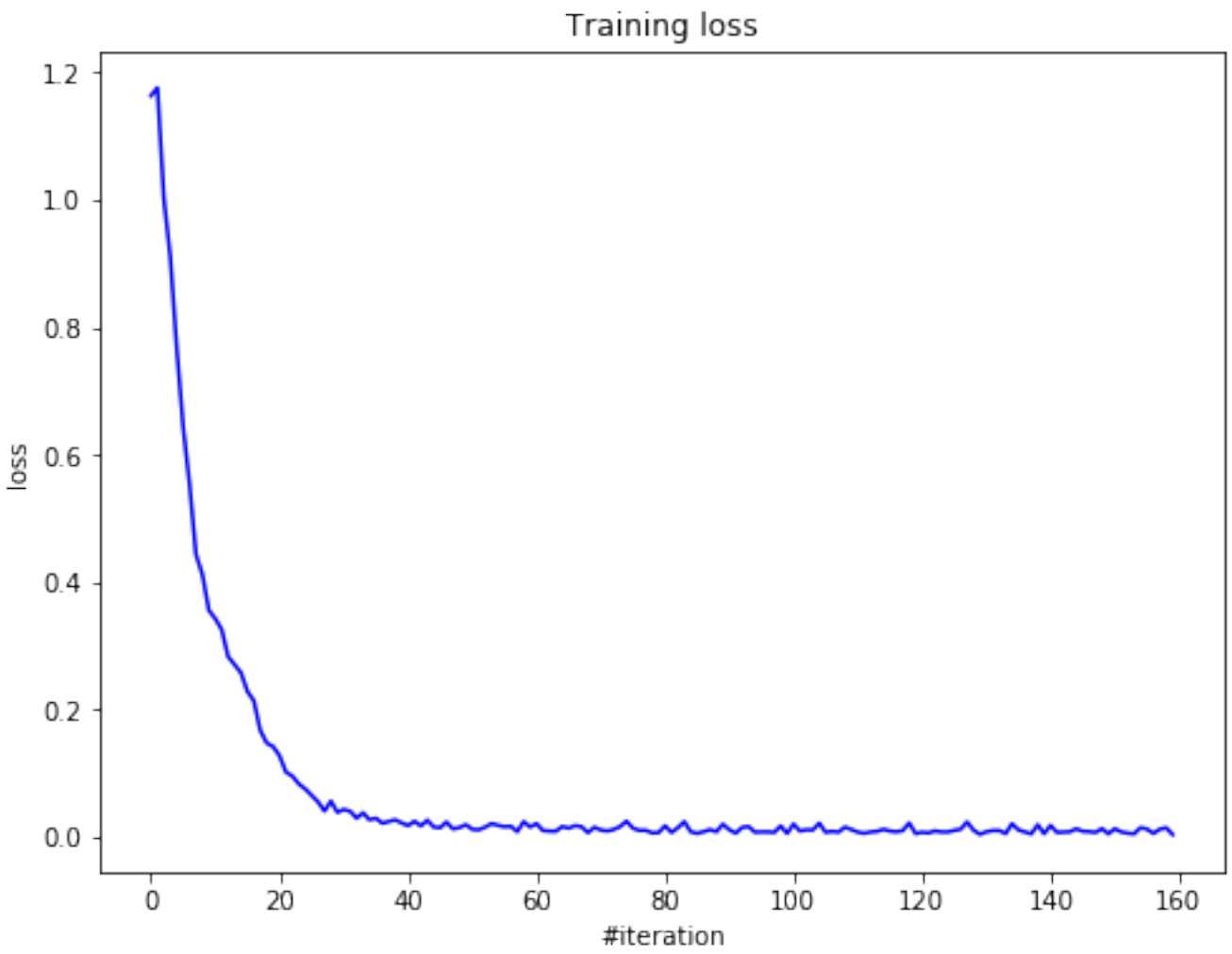
Basic training loop. Examine it.

In [8]:

```
def train(net, train_X, train_Y, n_epoch=20, batch_size=128, learning_rate=0.01,
    loss_history = []

    # Iptimizer params
    optimizer_config = {'learning_rate' : learning_rate, 'momentum': momentum}
    '''
    optimizer_config = {
        'learning_rate' : 1e-1, 'beta_1': 0.9, 'beta_2': 0.9,
        'm': {},
        'v': {}
    }
    '''
    optimizer_state = {}

    for i in range(n_epoch):
        for x_batch, y_batch in get_batches((train_X, train_Y), batch_size):
            net.zeroGradParameters()
            # Forward
            predictions = net.forward(x_batch)
            loss = criterion.forward(predictions, y_batch)

            # Backward
            dp = criterion.backward(predictions, y_batch)
            net.backward(x_batch, dp)

            # Update weights
            sgd_momentum(
                net.getParameters(),
                net.getGradParameters(),
                optimizer_config,
                optimizer_state
            )
            loss_history.append(loss)

        # Visualize
        display.clear_output(wait=True)
        plt.figure(figsize=(8, 6))

        plt.title("Training loss")
        plt.xlabel("#iteration")
        plt.ylabel("loss")
        plt.plot(loss_history, 'b')
        plt.show()

        print('Current loss: {}'.format(loss))
```

In [9]:

```
train(net, X, Y)
```




```
Current loss: 0.0036105068230960183
```

Digit classification

We are using MNIST (http://yann.lecun.com/exdb/mnist/) as our dataset. Lets start with cool visualization (http://scs.ryerson.ca/~aharley/vis/). The most beautiful demo is the second one, if you are not familiar with convolutions you can return to it in several lectures.

In [10]:

```
import os
from sklearn.datasets import fetch_mldata

# Fetch MNIST dataset and create a local copy.
if os.path.exists('mnist.npz'):
    with np.load('mnist.npz', 'r') as data:
        X = data['X'] / 255
        Y = data['Y']
        test_X = data['Xtest'] / 255
        test_Y = data['Ytest']
```

One-hot encode the labels first.

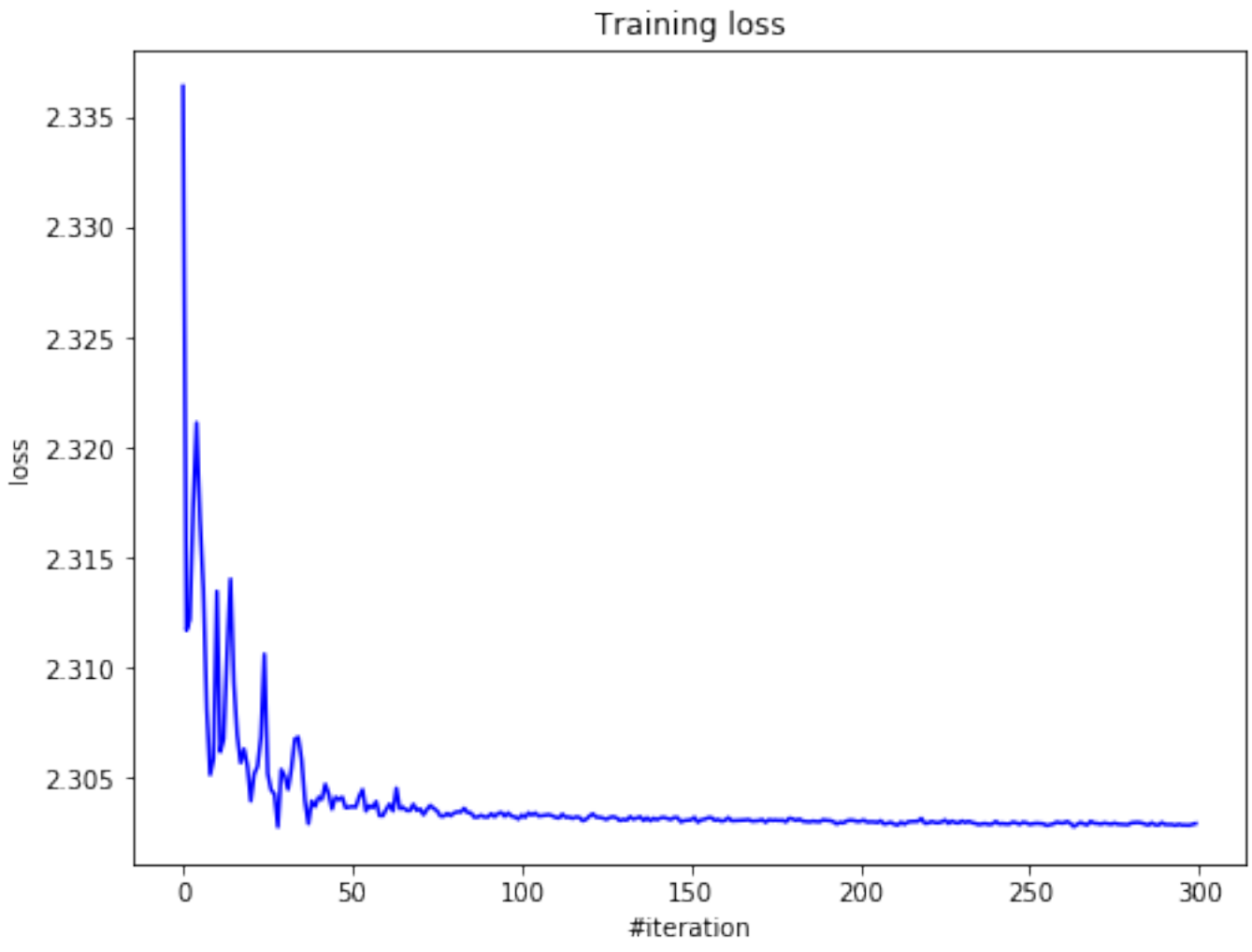
In [22]:

```
net = Sequential()
net.add(Linear(784, 10))
net.add(SoftMax())

criterion = ClassNLLCriterion()

print(net)
```

```
Linear 784 -> 10
SoftMax
```

In [23]:

```
train(net, X, Y, batch_size=1000, n_epoch=5, learning_rate=0.01)
```




```
Current loss: 2.3029437188484168
```

In [24]:

```
predictions = np.argmax(net.forward(test_X), axis=1)
```

In [25]:

```
predictions
```

Out[25]:

```
array([2, 8, 2, ..., 4, 8, 7])
```

In [26]:

```
accuracy_score(predictions, test_Y)
```

Out[26]:

```
0.09900000000000005
```

- **Compare** `ReLU`, `ELU` activation functions. You would better pick the best optimizer params for each of them, but it is overkill for now. Use an architecture of your choice for the comparison.

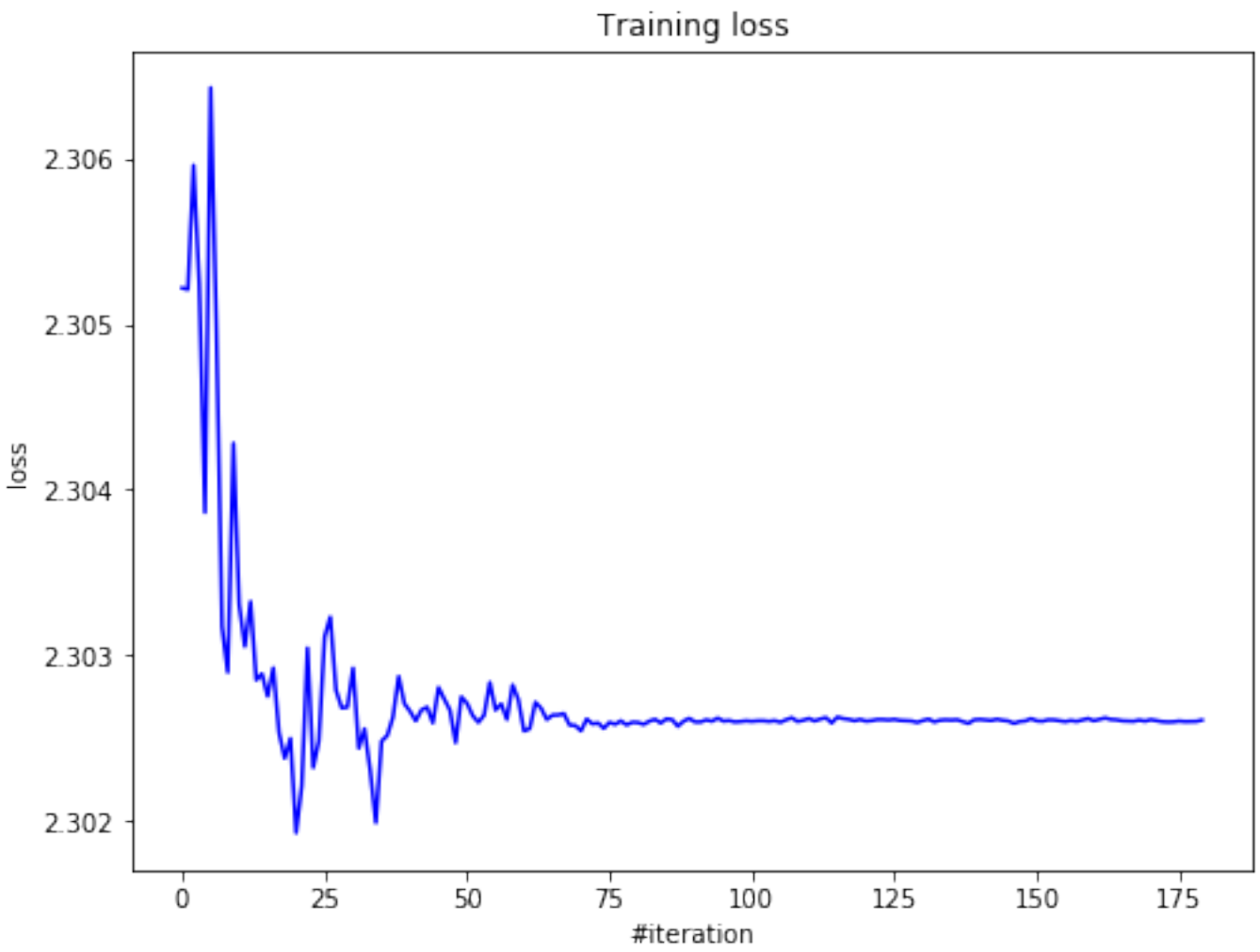
In [17]:

```
criterion = ClassNLLCriterion()

net1 = Sequential()
net1.add(Linear(784, 500))
net1.add(ReLU())
net1.add(Linear(500, 300))
net1.add(ReLU())
net1.add(Linear(300, 100))
net1.add(ReLU())
net1.add(Linear(100, 10))
net1.add(SoftMax())
print(net1)

net2 = Sequential()
net2.add(Linear(784, 500))
net2.add(LeakyReLU())
net2.add(Linear(500, 300))
net2.add(LeakyReLU())
net2.add(Linear(300, 100))
net2.add(LeakyReLU())
net2.add(Linear(100, 10))
net2.add(SoftMax())

print(net2)
```

```
Linear 784 -> 500
ReLU
Linear 500 -> 300
ReLU
Linear 300 -> 100
ReLU
Linear 100 -> 10
SoftMax

Linear 784 -> 500
LeakyReLU
Linear 500 -> 300
LeakyReLU
Linear 300 -> 100
LeakyReLU
Linear 100 -> 10
SoftMax
```

In [18]:

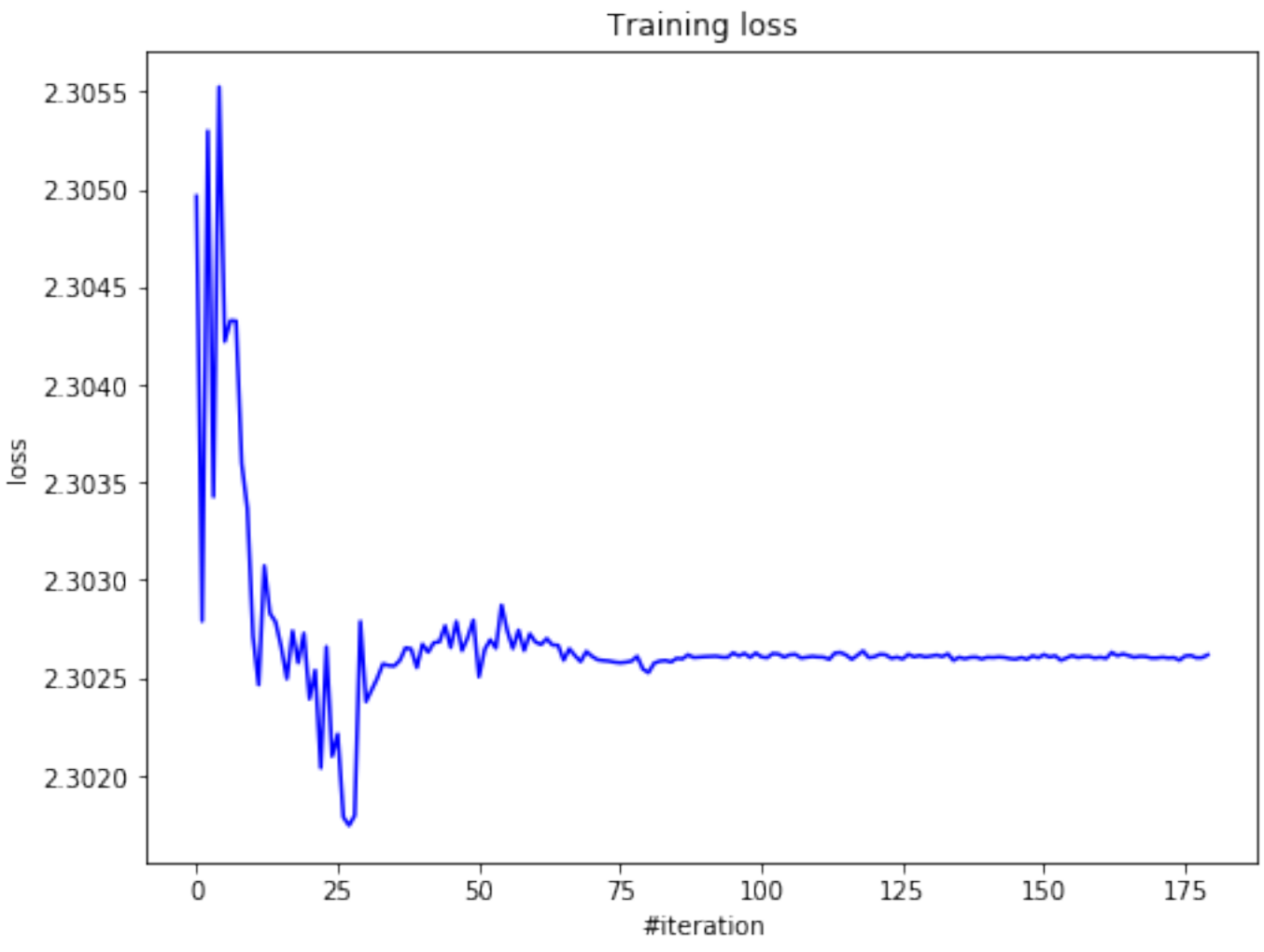
```
train(net1, X, Y, batch_size=1000, n_epoch=3)
```




```
Current loss: 2.3026066891597363
```

In [19]:

```
train(net2, X, Y, batch_size=1000, n_epoch=3)
```




```
Current loss: 2.302617815973572
```

Finally, use all your knowledge to build a super cool model on this dataset, do not forget to split dataset into train and validation. Use **dropout** to prevent overfitting, play with **learning rate decay**. You can use **data augmentation** such as rotations, translations to boost your score. Use your knowledge and imagination to train a model.

In [20]:

```
# Your code goes here. ################################################
```

Print here your accuracy. It should be around 90%.

In [21]:

```
# Your answer goes here. ################################################
```

Я не нашел, где ошибка. На toy example эта штука работала, а на загруженом датасете вне зависимости от топологии сети (можно брать хоть один полносвязный линейный слой, хоть 10 с различными нелинейностями и dropout'ами, все равно выходит примерно то же самое) получается одино и то же значение функции потерь в районе 2.30. При этом качество получается низким. Сравнивать ReLU с другими нелинейностями в таком случае бесполезно. Результаты получаются такими же. "Use all your knowledge to build a super cool model" тоже не выходит.

Оформление дз: заполните форму тут

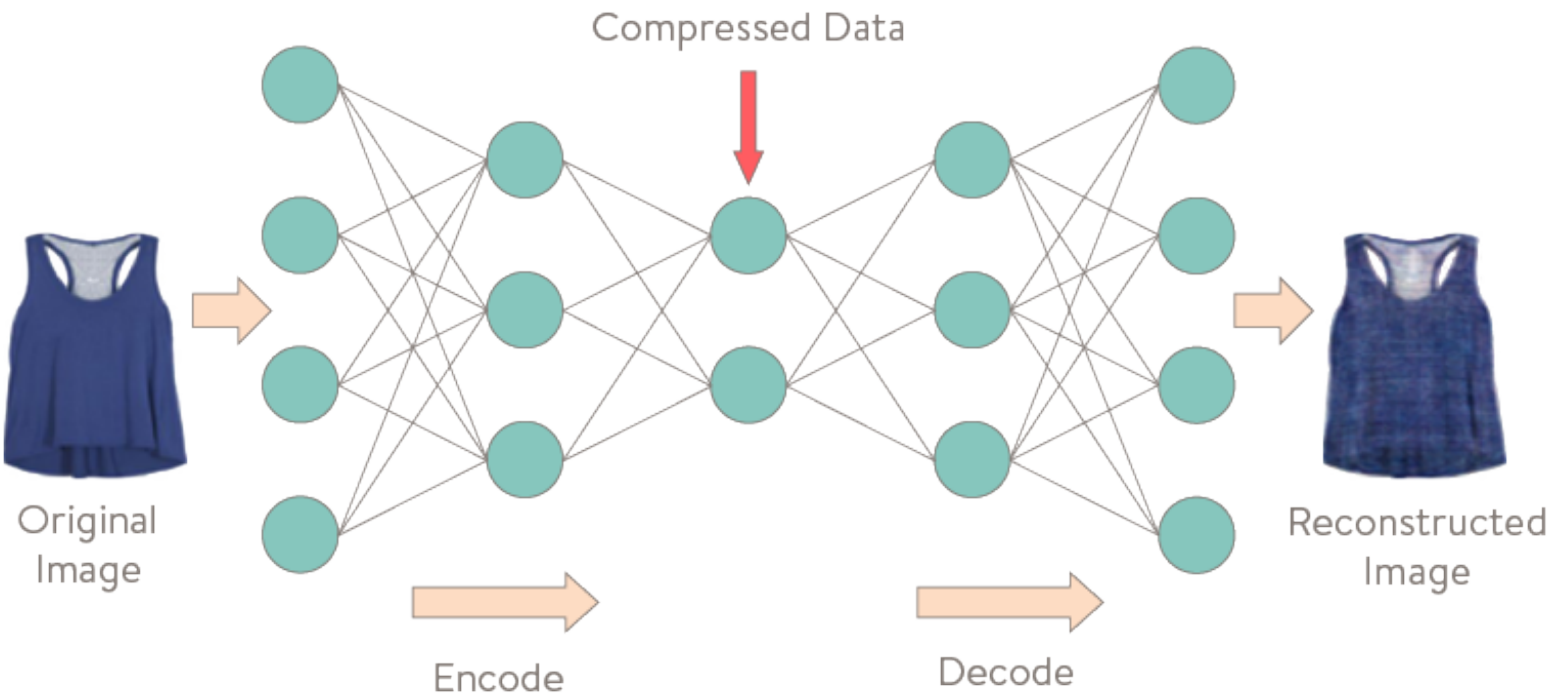
Оформление дз: заполните форму тут https://goo.gl/forms/Jx3OR5Ijjg2MwfKs2 (https://goo.gl/forms/Jx3OR5Ijjg2MwfKs2)

PS: Напоминаем, что дедлайны жесткие, прием дз заканчивается ровно в дедлайн

Bonus Part: Autoencoder

This part is **OPTIONAL**, you may not do it. It will not be scored, but it is easy and interesting.

Now we are going to build a cool model, named autoencoder. The aim is simple: **encode** the data to a lower dimentional representation. Why? Well, if we can **decode** this representation back to original data with "small" reconstuction loss then we can store only compressed representation saving memory. But the most important thing is -- we can reuse trained autoencoder for classification.



Picture from this site (http://multithreaded.stitchfix.com/blog/2015/09/17/deep-style/).

Now implement an autoencoder:

Build it such that dimetionality inside autoencoder changes like that:

$$784\ (\text{data}) -> 512 -> 256 -> 128 -> 30 -> 128 -> 256 -> 512 -> 784$$

Use **MSECriterion** to score the reconstruction.

You may train it for 9 epochs with batch size = 256, initial lr = 0.1 droping by a factor of 2 every 3 epochs. The reconstruction loss should be about 6.0 and visual quality decent already. Do not spend time on changing architecture, they are more or less the same.

In [ ]:

```
# Your code goes here. ################################################
```

Some time ago NNs were a lot poorer and people were struggling to learn deep models. To train a classification net people were training autoencoder first (to train autoencoder people were pretraining single

layers with RBM (https://en.wikipedia.org/wiki/Restricted_Boltzmann_machine)), then substituting the decoder part with classification layer (yeah, they were struggling with training autoencoders a lot, and complex techniques were used at that dark times). We are going to this now, fast and easy.

In [ ]:

```
# Extract inner representation for train and validation,
# you should get (n_samples, 30) matrices
# Your code goes here. ################################################

# Now build a logistic regression or small classification net
cnet = Sequential()
cnet.add(Linear(30, 2))
cnet.add(SoftMax())

# Learn the weights
# Your code goes here. ################################################

# Now chop off decoder part
# (you may need to implement `remove` method for Sequential container)
# Your code goes here. ################################################

# And add learned layers ontop.
autoenc.add(cnet[0])
autoenc.add(cnet[1])

# Now optimize whole model
# Your code goes here. ################################################
```

- What do you think, does it make sense to build real-world classifiers this way ? Did it work better for you than a straightforward one? Looks like it was not the same ~8 years ago, what has changed beside computational power?

Run PCA with 30 components on the *train set*, plot original image, autoencoder and PCA reconstructions side by side for 10 samples from *validation set*. Probably you need to use the following snippet to make aoutpencoder examples look comparible.

In [ ]:

```
# np.clip(prediction,0,1)
#
# Your code goes here. ################################################
```