Ullrich Köthe
ullrich.koethe@iwr.uni-heidelberg.de

Exercise 3

Deadline: 04.06.2019, 2:00 pm

In this exercise we introduce the pytorch framework, a leading open-source Python library for neural network research, mainly developed by FacebookAI. It supports both CPU- and GPU-based execution. Neural networks (or any other computation) are expressed in terms of computation graphs, which define functional relationships between variables (e.g. Tensors) and allow to calculate the gradients of any nested expression automatically, from within Python. pytorch tutorials and documentation can be found at `http://pytorch.org`.

Regulations

Please create a Jupyter notebook `cnn.ipynb` for your solution and export it into `cnn.html`. Zip both files along with your comments to exercise 2 (cf. task 1) into a single archive with naming convention (sorted alphabetically by last names)

`lastname1-firstname1_lastname2-firstname2_exercise03.zip`

or (if you work in a team of three)

`lastname1-firstname1_lastname2-firstname2_lastname3-firstname3_exercise03.zip`

and upload it to Moodle before the given deadline. We will give zero points if your zip-file does not conform to the naming convention.

1 Comment on your solution to exercise 2

Study the sample solutions on Moodle and use them to comment on your own solutions to exercise 2. Specifically, comment your solutions to the paper-and-pencil tasks (hand-crafted network and proof of linear network behavior) with a colored pen and hand in the file network_commented.pdf. Similarly, copy your solution network.py into network_commented.py and insert comments starting with

```
# Comment:
```

The point of these comments is that you identify your errors and bugs yourselves, so that you learn from your mistakes. In addition, the tutor will have an easier time distinguishing between the initial mistake and consequential errors caused by the first one and will only deduct points for the former. If you fail to hand in comments, the tutor is not required to make this distinction and will deduct points for all errors alike.

2 Introduction (5 Points)

First you need to make yourself familiar with pytorch. The following code (available on Moodle as `intro.py`) defines a simple neural network with 2 hidden layers

```
import numpy as np
import matplotlib.pyplot as plt
import math

import torch
import torch.optim
import torch.functional as F

import torchvision
import torchvision.datasets as dset
import torchvision.transforms as transforms

```

```python
from torch.nn.functional import conv2d, max_pool2d



mb_size = 100 # mini-batch size of 100


trans = transforms.Compose([transforms.ToTensor(),
                            transforms.Normalize((0.5, 0.5, 0.5),
                                                 (0.5, 0.5, 0.5))])


dataset = dset.MNIST("./", download = True,
                     train = True,
                     transform = trans)


dataloader = torch.utils.data.DataLoader(dataset, batch_size=mb_size,
                                         shuffle=True, num_workers=1,
                                         pin_memory=True)



def init_weights(shape):
    # xavier initialization (a good initialization is important!)
    # http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization
    fan_in = shape[0]
    fan_out = shape[1]
    variance = 2.0/(fan_in + fan_out)
    w = torch.randn(size=shape)*np.sqrt(variance)
    w.requires_grad = True
    return w

def rectify(X):
    return torch.max(torch.zeros_like(X), X)


# you can also use torch.nn.functional.softmax on future sheets
def softmax(X):
    c = torch.max(X, dim=1)[0].reshape(mb_size, 1)
    # this avoids a blow up of the exponentials
    # but calculates the same formula
    stabelized = X-c
    exp = torch.exp(stabelized)
    return exp/torch.sum(exp, dim=1).reshape(mb_size, 1)


# this is an example as a reduced version of the pytorch internal RMSprop optimizer
class RMSprop(torch.optim.Optimizer):
    def __init__(self, params, lr=1e-3, alpha=0.9, eps=1e-8):
        defaults = dict(lr=lr, alpha=alpha, eps=eps)
        super(RMSprop, self).__init__(params, defaults)

    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                grad = p.grad.data
                state = self.state[p]

                # State initialization
                if len(state) == 0:
                    state['square_avg'] = torch.zeros_like(p.data)

                square_avg = state['square_avg']
                alpha = group['alpha']

                # update running averages
```

```
                square_avg.mul_(alpha).addcmul_(1 - alpha, grad, grad)
                avg = square_avg.sqrt().add_(group['eps'])

                # gradient update
                p.data.addcdiv_(-group['lr'], grad, avg)


def model(X, w_h, w_h2, w_o, p_drop_input, p_drop_hidden):
    #X = dropout(X, p_drop_input)
    h = rectify(X @ w_h)
    #h = dropout(h, p_drop_hidden)
    h2 = rectify(h @ w_h2)
    #h2 = dropout(h2, p_drop_hidden)
    pre_softmax = h2 @ w_o
    return pre_softmax


w_h = init_weights((784, 625))
w_h2 = init_weights((625, 625))
w_o = init_weights((625, 10))

optimizer = RMSprop([w_h, w_h2, w_o])

# put this into a training loop over 100 epochs
for (_, (X, y)) in enumerate(dataloader, 0):
    optimizer.zero_grad()
    noise_py_x = model(X.reshape(mb_size, 784), w_h, w_h2, w_o, 0.8, 0.7)
    cost = torch.nn.functional.cross_entropy(noise_py_x, y)
    cost.backward()
    print("Loss: {}".format(cost))
    optimizer.step()
```

Task: Install pytorch (best with conda), convert `intro.py` into a Jupyter notebook and run the code.

3 Dropout (5 Points)

We want to use dropout learning for out network. Therefore, implement the function

```
def dropout(X, p_drop=1.):
    ...
```

that sets random elements of X to zero (do *not* use pytorch's existing dropout functionality).

Dropout:

- **If** $0 < p_{\text{drop}} < 1$:
  For every element $x_i \in X$ draw Φ_i randomly from a binomial distribution with $p = p_{\text{drop}}$. Then reassign
  $$x_i \rightarrow \begin{cases} \frac{x_i}{p_{\text{drop}}} & \text{if } \Phi = 1 \\ 0 & \text{if } \Phi = 0 \end{cases}$$
- **Else:**
  Return the unchanged X.

You can now enable the dropout functionality. To this end, remove the comments from the lines

```
X = dropout(X, p_drop_input)
h = dropout(h, p_drop_hidden)
h2 = dropout(h2, p_drop_hidden)
```

and check that your code still runs. **Question:** Explain in a few sentences how the dropout method works and how it reduces overfitting. Why do we need to initialize two models for dropout? Compare the test error with the test error from Section 2

Ullrich Köthe
ullrich.koethe@iwr.uni-heidelberg.de

4 Parametric Relu (10 Points)

Instead of a simple rectify mapping (aka rectified linear unit(Relu)) we want to add a parametric Relu that maps every element x_i of the input X to

$$x_i \to \begin{cases} x_i & x_i > 0 \\ a_i x_i & x_i \leq 0 \end{cases}.$$

A detailed description can be found in the paper **Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification** (see `http://arxiv.org/pdf/1502.01852.pdf`). The crux of this method is the learnable weightvector a that needs to be adjusted during training. Define the function

```
def PRelu(X,a):
    ...
```

that creates a PRelu layer by mapping $X \to \text{PRelu}(X)$.

Incorporate the parameter a into the **params** list and make sure that it is optimized during training.

5 Convolutional layers (20 Points)

In this exercise we want to create a similar neural network to LeNet from Yann LeCun. LeNet was designed for handwritten and machine-printed character recognition. It relies on convolutional layers that transform the input image by convolution with multiple learnable filters. LeNet contains convolutional layers paired with sub sampling layers as displayed in Figure 1. The Subsampling is done by max pooling which reduces an area of the image to one pixel with the maximum value of the area. Both functions are already available in pytorch:

```
from torch.nn.functional import conv2d, max_pool2d

convolutional_layer = rectify(conv2d(previous_layer, weightvector))
subsampleing_layer = max_pool_2d(convolutional_layer, (2, 2)) # reduces window 2
    x2 to 1 pixel
out_layer = dropout(subsample_layer, p_drop_input)
```

5.1 Create a Convolutional network

Now we can design our own convolutional neural network that classifies the handwritten numbers from MNIST.

Implementation task:

- Reshape the input image with:

```
trX = trX.reshape(-1, 1, 28, 28) #trainings data
teX = teX.reshape(-1, 1, 28, 28) #test data
```

- Replace the first hidden layer **h** with 3 convolutional layers (including subsampling and dropout)
- connect the convolutional layers to the vectorized layer **h2** by flattening the input with **torch.reshape**.
- The shape of the weight parameter for **conv2d** determines the number of filters f, the number of input images pic_{in}, and the kernel size $k = (k_x, k_y)$. You can initialize the weights with

```
init_weights((f, pic_in, k_x, k_y))
```

Ullrich Köthe
ullrich.koethe@iwr.uni-heidelberg.de

Make a neural network with

convolutional layer:	first	second	third
f	32	64	128
pic_{in}	1	32	64
k_x	5	5	2
k_y	5	5	2

and add the weightvectors to the **params** list.

- In Section 5.2 you will determine the number of output pixels of the CNN. Use it to adjust the size of the rectifier layer to

```
w_h2 = init_weights((number_of_output_pixel, 625))
```

- Use a softmax output layer with 625 inputs and 10 outputs (as before).

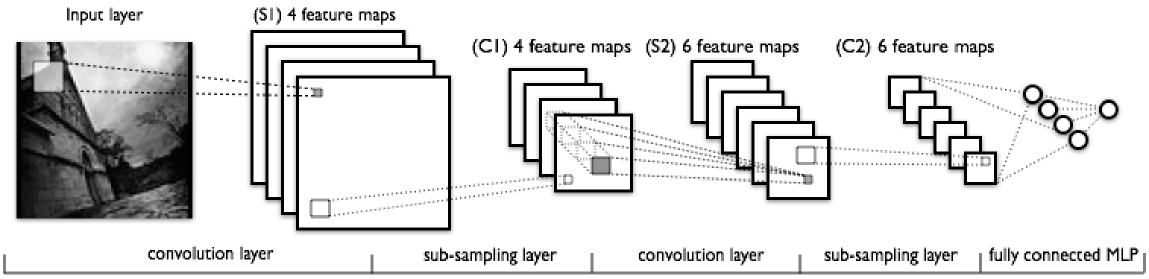

Abbildung 1: Sketch of convolutional neural network similar to LeNet

5.2 Application of Convolutional network

Task:

- draw a sketch of the network(like Figure 1) and note the sizes of the filter images (This will help you to determine how many pixels there are in the last convolution layer).
- after the training plot
  - one image from the test set
  - its convolution with 3 filters of the first convolutional layer
  - the corresponding filter weights(this should be 5 by 5 images).

Finally, choose one of the following tasks:

- add or remove one convolutional layer (you may adjust the number of filters)
- increase the filter size (you may plot some pictures i
- apply a random linear shift to the trainings images. Does this reduce overfitting?
- use unisotropic filters $k_x! = k_y$
- create a network architecture of your choice and see if you can improve on the previous results

and compare the new test error.

Ideally you should create an overview table that lists the test errors from all sections.