# Exercise 3

**Deadline: 26.05.2021, 4:00 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 into a single archive.

Additionally, please set your Anzeigename/display name and Name in Uebungsgruppen/name in tutorials in MAMPF to your real name, which should be identical to your name in `muesli` and make sure you join the submission of your team via code before or during the submission. Check out `https://mampf.blog/handing-in-homework-assignments` for instructions.

## 1 Introduction (5 Points)

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

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

import torch
import torch.optim

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

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,))])


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

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


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

```

```python
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=mb_size,
                                          shuffle=True, num_workers=1,
                                          pin_memory=True)


def init_weights(shape):
    # Kaiming He initialization (a good initialization is important)
    # https://arxiv.org/abs/1502.01852
    std = np.sqrt(2. / shape[0])
    w = torch.randn(size=shape) * std
    w.requires_grad = True
    return w


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


# 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=0.5, 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_(grad, grad, value=1-alpha)
                avg = square_avg.sqrt().add_(group['eps'])

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


def model(X, w_h, w_h2, w_o):
    h = rectify(X @ w_h)
    h2 = rectify(h @ w_h2)
    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 i in range(101):
    print("Epoch: {}".format(i+1))
    avg_train_loss = 0.
    for (j, (X, y)) in enumerate(dataloader):
        noise_py_x = model(X.reshape(mb_size, 784), w_h, w_h2, w_o)
        optimizer.zero_grad()
        # the cross-entropy loss function already contains the softmax
        cost = torch.nn.functional.cross_entropy(noise_py_x, y, reduction="mean")
```

```
            avg_train_loss += cost
            cost.backward()
            optimizer.step()

        if i % 10 == 0:
            print("Average Train Loss: {}".format(avg_train_loss / (j + 1)))

            # no need to calculate gradients for validation
            with torch.no_grad():
                avg_test_loss = 0.
                for (k, (X, y)) in enumerate(test_loader):
                    noise_py_x = model(X.reshape(mb_size, 784), w_h, w_h2, w_o)
                    cost = torch.nn.functional.cross_entropy(noise_py_x, y, reduction="
                        mean")
                    avg_test_loss += cost

                print("Average Test Loss: {}".format(avg_test_loss / (k + 1)))
```

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

## 2 Dropout (5 Points)

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

```
def dropout(X, p_drop=0.5):
    ...
```

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 $\Phi_i$ randomly from a binomial distribution with $p = p_{\text{drop}}$. Then reassign
  $$x_i \to \begin{cases} 0 & \text{if } \Phi = 1 \\ \frac{x_i}{1-p_{\text{drop}}} & \text{if } \Phi = 0 \end{cases}$$
- **Else:**
  Return the unchanged X.

You can now enable the dropout functionality. To this end, implement a new model

```
def dropout_model(X, w_h, w_h2, w_o, p_drop_input, p_drop_hidden):
    ...
```

containing the same fully-connected layers as in Section 1, but now with three dropout steps. Dropout is applied to the *input* of each layer.

**Task:** Explain in a few sentences how the dropout method works and how it reduces overfitting. Train the model using dropout and report train and test errors. Why do we need a different model configuration for evaluating the test loss? Compare the test error with the test error from Section 1.

## 3 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.

**Task:** Compare the results with the previous models.

## 4 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 via 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))
# reduces (2,2) window to 1 pixel
subsampling_layer = max_pool_2d(convolutional_layer, (2, 2))
out_layer = dropout(subsample_layer, p_drop_input)
```

### 4.1 Create a Convolutional network

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

**Implementation task**:

- Make sure that the input image has the correct shape:

  ```
  trainX = trainX.reshape(-1, 1, 28, 28) #training data
  testX = testX.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))
  ```

  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 weight vectors to the **params** list.

- In Section 4.2 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 pre-softmax output layer with 625 inputs and 10 outputs (as before).

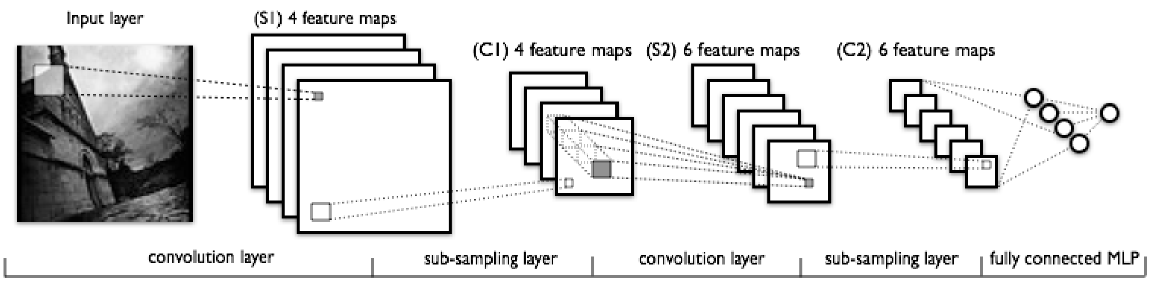

Abbildung 1: Sketch of convolutional neural network similar to LeNet

## 4.2 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).
- Train the model. Then, plot:
  - one image from the test set
  - its convolution with 3 filters of the first convolutional layer
  - the corresponding filter weights (these 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.