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

Exercise 3

Deadline: 02.06.2022, 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. Zip all files into a single archive `ex03.zip` and upload this file to MaMPF before the given deadline.
Moreover, 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 the invitation code before the submission deadline. 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 as external link on MaMPF and under `https://tinyurl.com/HD-AML-intro-py`) defines a simple neural network with 2 hidden fully-connected layers.

```

import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.optim as optim
from torch.utils.data import DataLoader

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

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

plt.rc("figure", dpi=100)

batch_size = 100

# transform images into normalized tensors
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5,), std=(0.5,))
])

train_dataset = datasets.MNIST(
    "./",
    download=True,
    train=True,
    transform=transform,
)

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

train_dataloader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=1,
    pin_memory=True,
)

test_dataloader = DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    shuffle=False,
    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):
    # Rectified Linear Unit (ReLU)
    return torch.max(torch.zeros_like(x), x)


class RMSprop(optim.Optimizer):
    """
    This is a reduced version of the PyTorch internal RMSprop optimizer
    It serves here as an example
    """
    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'])


# define the neural network
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


# initialize weights

# input shape is (B, 784)
w_h = init_weights((784, 625))
# hidden layer with 625 neurons
w_h2 = init_weights((625, 625))
# hidden layer with 625 neurons
w_o = init_weights((625, 10))
# output shape is (B, 10)

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


n_epochs = 100

train_loss = []
test_loss = []

# put this into a training loop over 100 epochs
for epoch in range(n_epochs + 1):
    train_loss_this_epoch = []
    for idx, batch in enumerate(train_dataloader):
        x, y = batch

        # our model requires flattened input
        x = x.reshape(batch_size, 784)
        # feed input through model
        noise_py_x = model(x, w_h, w_h2, w_o)

        # reset the gradient
        optimizer.zero_grad()

        # the cross-entropy loss function already contains the softmax
        loss = cross_entropy(noise_py_x, y, reduction="mean")

        train_loss_this_epoch.append(float(loss))

        # compute the gradient
        loss.backward()
        # update weights
        optimizer.step()

    train_loss.append(np.mean(train_loss_this_epoch))

    # test periodically
    if epoch % 10 == 0:
        print(f"Epoch: {epoch}")
        print(f"Mean Train Loss: {train_loss[-1]:.2e}")
        test_loss_this_epoch = []

        # no need to compute gradients for validation
        with torch.no_grad():
            for idx, batch in enumerate(test_dataloader):
                x, y = batch
                x = x.reshape(batch_size, 784)
                noise_py_x = model(x, w_h, w_h2, w_o)

                loss = cross_entropy(noise_py_x, y, reduction="mean")
                test_loss_this_epoch.append(float(loss))

        test_loss.append(np.mean(test_loss_this_epoch))

        print(f"Mean Test Loss:  {test_loss[-1]:.2e}")

```

```python
plt.plot(np.arange(n_epochs + 1), train_loss, label="Train")
plt.plot(np.arange(1, n_epochs + 2, 10), test_loss, label="Test")
plt.title("Train and Test Loss over Training")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
```

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

2 Dropout (8 Points)

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

```python
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 Φ_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

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

containing the same fully-connected layers as in function `model()` of task 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

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

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

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 (17 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).

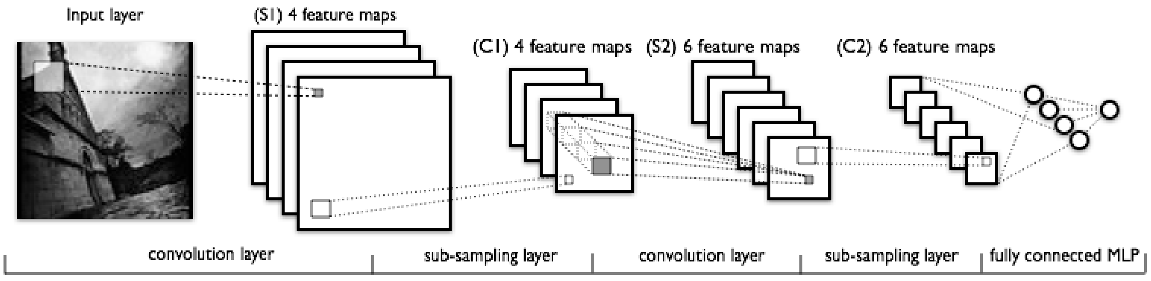

Abbildung 1: Sketch of convolutional neural network similar to LeNet

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.