Faculty of computer and information science

Assignment #7

Course: *Machine learning*
Date: *December 15th, 2023*

Assignment

In this assignment, you will familiarize yourself with a typical neural network training and evaluation workflow. You will construct a dataset framework for batched training, create your own implementations of common neural network operations, and train your model on the Fashion-MNIST dataset. You will accomplish this in the PyTorch environment. To use PyTorch install the API on your local machine.

First, load the Fashion-MNIST dataset from the Fashion-MNIST GitHub repository. Use the provided **utils.mnist_reader** class to load the **training** data, which should contain 60000 training samples. Then, normalize the training sample values to the range $[0, 1]$ and visualize one random member from each of the 10 classes.

Split the data into two subsets: a training subset and a validation subset, where the latter constitutes of 20 percent random samples from the entire data set. Reshape the sample data in both subsets such that each sample corresponds to a two-dimensional image with a single channel. The dimensions of each subset should be equal to $[N, 1, 28, 28]$, where N denotes the total number of samples in the subset. Convert both the samples and the labels of both subsets to PyTorch Tensors, where you additionally transform the class labels to **one-hot encodings**. Then, define the following Python class and fill out its class methods:

```
class Dataset():

    def __init__(self, samples, labels, batch_size):
        # TODO: Provide implementation
        pass

    def __getitem__(self, index):
        # TODO: Provide implementation
        pass

    def __len__(self):
        # TODO: Provide implementation
        pass

    def shuffle(self):
        # TODO: Provide implementation
        pass
```

The Dataset class will serve batched, randomly shuffled samples during training and validation. Therefore, the **samples** and **labels** should contain the matrix of data samples and their corresponding labels. The **batch_size** denotes the number of samples each call of the **__getitem__** method should provide. Finally, **__getitem__** should return the data batch corresponding to the counter **index**, **__len__** the length of the dataset in terms of batches, and the **shuffle** method should randomly permute the samples in the dataset.

Use the Dataset class to create two datasets: a training dataset using the previous training data subset, and a validation dataset using the validation data subset. Visualize 16 samples from a batch belonging to the training dataset and their corresponding class labels.

- The **__getitem__** function should be deterministic: each method call with the same index on the same dataset permutation should always return the same batch!
- Pay attention to the case where the **batch_size** does not evenly divide the number of samples!

Now, we will implement three basic operations that will serve as the backbone of our classification neural network. These operations are: **two-dimensional convolution**, **two-dimensional max pooling**, and the element-wise **Linear Rectified Unit**.

```
class Conv2d(torch.nn.Module):

    def __init__(self, in_channels, out_channels,
                 kernel_size, stride):
        super().__init__()

        # TODO: Provide implementation

        self.weight = None
        self.bias   = None

        pass

    def forward(self, x):
        # TODO: Provide implementation
        pass

class MaxPool2d(torch.nn.Module):

    def __init__(self, kernel_size, stride):
        super().__init__()

        # TODO: Provide implementation
        pass

    def forward(self, x):
        # TODO: Provide implementation
        pass

class ReLU(torch.nn.Module):

    def __init__(self):
        super().__init__()

        # TODO: Provide implementation
        pass

    def forward(self, x):
        # TODO: Provide implementation
        pass
```

Use the class constructors to do the necessary function orchestration and implement the actual operation of the input sample x in the **forward** method. For more details on the individual functions and their arguments consult the PyTorch documentation: Conv2d, MaxPool2d, ReLU. The convolution and maximal pooling operations should provide no padding to the input. Here is a snippet of how each function should be applied to a batch of samples after you have completed its implementation:

```
x, y = training_dataset[0]
f = ReLU()
result = f(x)
```

Use the dataset implementation from the previous exercises to test your model. Do not use the PyTorch implementations of these functions (**torch.nn.Conv2d**, **torch.nn.MaxPool2d**, **torch.nn.ReLU**) instead of implementing the functions yourselves!

After you have completed each function's implementation make sure that they perform the same as their PyTorch counterparts: **torch.nn.Conv2d**, **torch.nn.MaxPool2d**, and **torch.nn.ReLU**. Pass the same input through the corresponding functions and observe the pixel-wise absolute difference. The maximal difference should be smaller than 10^{-5}. When comparing the convolution operations make sure that both functions contain the same **weight** and **bias** tensors (the **weight** and **bias** member of the convolution class).

- If the convolution operation is too slow try implementing it using the **torch.nn.Unfold** and **torch.nn.Fold** functions.
- Be careful to accurately compute the shape of the output data in the convolution and max pooling operations.
- You can inspect the **torch.nn.Conv2d**, **torch.nn.MaxPool2d**, and **torch.nn.ReLU** implementations for hints as to how to implement the functions yourselves.

For this exercise, use the classes you constructed in the previous exercises.

We will briefly examine how the ordering of operations inside the neural network affects the output. We will look at the specific case of MaxPool2d and ReLU commutativity. Create a PyTorch tensor with random elements in the range $[-1, 1]$ of dimensions $[1, 100, 100]$. Pass the tensor through two different transformations *individually*:

- Function one: **MaxPool2d(kernel_size = 3, stride = 2)** followed by **ReLU**
- Function two: **ReLU** followed by **MaxPool2d(kernel_size = 3, stride = 2)**

Compare the outputs of the two sets. What did you observe? Explain the obtained results mathematically and think about the implications of this observation. Is the same true if we replace the maximal pooling operation with a convolution?

For this exercise, use the classes you constructed in the previous exercises.

Now, we will define a neural network using the custom functions from previous exercises. This network will produce a vector of 10 elements for each sample, predicting the class it belongs to. Our neural network will consist of the following basic blocks:

- **Conv2d** followed by **MaxPool2d** followed by **ReLU**

There should be two such blocks in your network, one following the other. The kernel size of both the convolution and maximal pooling layers in both blocks should be equal to 3. The output of the second block should be a tensor with the following dimensions: $[B, 64, 4, 4]$, where B is the batch size and 64 is the number of channels. You have to determine the appropriate remaining hyperparameters for the blocks to produce an output of such shape.

Finally, your model should end with a single convolution layer with a **kernel_size** of 4 which will produce 10 output channels. The output of this layer should therefore be a tensor with a dimension equal to $[B, 10, 1, 1]$. Apply a flatten operation (**torch.nn.Flatten**) on the last layer's output to generate the output of size: $[B, 10]$.

Create the same neural network architecture, but use the built-in PyTorch classes: **torch.nn.Conv2d**, **torch.nn.MaxPool2d**, **torch.nn.ReLU**.

- Use the **torch.nn.Sequential** module to store the functions into one operation!

For this exercise, use the models you constructed in the previous exercises.
Lets put it all together! Using the Stochastic Gradient Descent algorithm and the Cross-Entropy loss function define a function **fit**, which takes in two parameters: the **model** you wish to fit and the **number of epochs**, denoting the number of times you will iterate over the training and validation datasets.

```
def fit(model, number_of_epochs):
    # TODO: Provide implementation

    return best_model, training_losses,
           validation_losses
```

Train both models from the previous exercise (your custom model and the PyTorch equivalent) for 10 epochs. The fit function should return the model that best performed on the validation dataset during training and a list of 10 training and validation losses, one for each epoch.

Visualize the training and validation losses for all 10 epochs and both models. Is there a difference in performance? Account for the various potential sources of these differences.

- You can use the **torch.optim.SGD** and **torch.nn.CrossEntropyLoss** classes for the Stochastic Gradient Descent and the Cross-Entropy loss function.

- Be careful that you select the best model at the end of each epoch and not when you iterate through all the epochs.

BONUS
For this exercise, use the fit function and dataset class you constructed in the previous exercises.

Create a custom neural network model using PyTorch functions and fit the model to the training data. Optimize the neural network hyperparameters (number of layers, kernel size, etc...) to get the best results on the validation dataset.
Evaluate your prediction on the test dataset using the **performance.plot_stats** function (included in the assignment materials) which takes as input your model's predictions and the corresponding correct labels (both in **one-hot encoding** format and **numpy ndarrays**). The output of the function is the image titled **stats.png**, containing the **class confusion matrix**, and the **macro recall** and **macro precision** values for each class.

Explain the results you observe. What is the most challenging class to classify? Why?