Introduction to PyTorch

Computer Vision
Assignment 1

1 Environment setup

For setting up the Python environment, we will be using miniconda. For the best and easiest experience, we strongly recommend using a Linux distribution (e.g., Ubuntu). However, miniconda is also compatible with Windows and Mac. Please download the latest miniconda version for your OS from the following link https://docs.conda.io/en/latest/miniconda.html.

Once that is done, you can run the following commands from the root of the code directory to install and activate the environment.

```
conda env create -f env.yml
conda activate cv-intro-to-pytorch
```

Throughout the assignment, do not modify the function interfaces or the already written complete code. We have added extra comments to the base code to guide you so make sure to take them into account. The PyTorch documentation available at https://pytorch.org/docs/stable/index.html is your best friend. Once you're done with filling up a certain function, don't forget to remove the `raise NotImplementedError()`. In case you run into any issues, please check the moodle forum and open a new topic if needed.

2 Simple 2D classifier (50 pts.)

The first task is training a simple binary classifier on the 2D data visualized in Figure 1. In the figure, red points are part of cluster 0, and blue – part of cluster 1. For this section, you will work on the code present in the `simple-2D-classifier` directory.

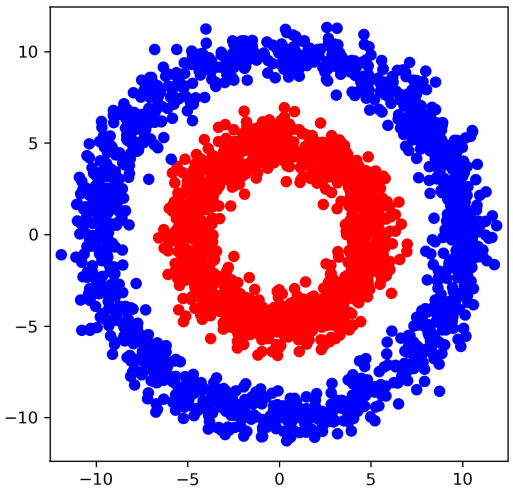

Figure 1: **Training data for 2D classifier.**

2.1 Dataset (10 pts.)

Start by filling in the `Simple2DDataset` class from `lib/dataset.py`.

First, in the class constructor `__init__`, you will need to read the right `npz` file from disk based on the `split` parameter and then save the contents to the associated class members.

Second, in the `__getitem__` method, you will need to recover a single data sample and its annotation based on its index `idx`.

2.2 Linear classifier (10 pts.)

Next step is defining your first network by filling in the `LinearClassifier` class from `lib/networks.py`. Add a single linear layer `nn.Linear` inside the `nn.Sequential` call. The output should be a single value, corresponding to the probability of a given 2D point being part of cluster 1.

2.3 Training loop (15 pts.)

Finally, finish implementing the training loop in the `run_training_epoch` function from `train.py`. There are four important steps during each training step and respecting the order is essential:

1. Clear the gradients from the previous step using `optimizer.zero_grad()`
2. Forward pass of the network to obtain the predictions
3. Compute the loss (binary cross entropy)
4. Backwards pass on the loss using `loss.backward()` to obtain the gradients followed by one step of gradient descent (optimization) using using `optimizer.step()`

Once this is done, you should be able to run the `train.py` script. What accuracy do you achieve with the linear classifier? Is this an expected result? Justify your answer.

2.4 Multi-layer perceptron (10 pts.)

Now let us try a multi-layer classifier with non-linearities by filling in the `MLPClassifier` class from `lib/networks.py`. Implement an MLP with 2 hidden layer and one final linear prediction layer. The hidden layers should have 16 neurons and be followed by a `nn.ReLU` non-linearity.

Switch to the new network by uncommenting L83 in `train.py`. What accuracy does this network obtain? Why are the results better compared to the previous classifier?

2.5 Feature transform (5 pts.)

One common way to improve the performance of a classifier is feature engineering. The easiest type of feature engineering is a coordinate system change. Think of a coordinate system that renders the two classes linearly separable and justify your choice.

Start by filling in the `Simple2DTransformDataset` class from `lib/dataset.py` as you did above. Next, update the `transform` function to change to your proposed coordinate system. Verify the hypothesis by training a linear classifier on the new representation. You will have to uncomment L65 and L75 in `train.py`.

3 Digit classifier (50 pts.)

Next task is handwritten digit classification. Some examples are visualized in Figure 2. The images in this dataset are 28×28 pixels. There are 10 classes – one per digit from 0 to 9. For this section, you will work on the code present in the `mnist-classifier` directory.

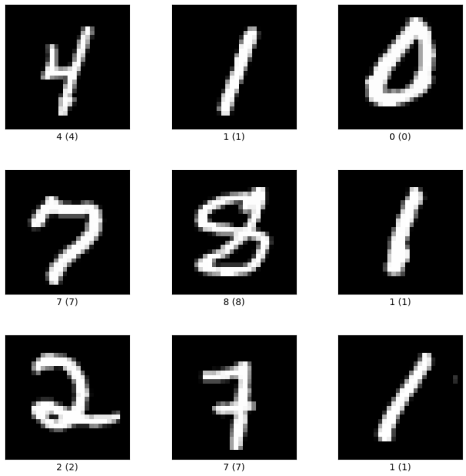

Figure 2: **Training data for digit classifier.** Source: https://www.tensorflow.org/datasets/catalog/mnist.

3.1 Data normalization (5 pts.)

Gray-scale images are generally stored in memory as `uint8` arrays. This means that each pixel can take values between 0 and 255. For better convergence of neural networks, it is generally common practice to normalize the values to the range $[-1, 1]$. Update the `normalize` function in `lib/dataset.py` accordingly.

3.2 Training loop (5 pts.)

As above, finish implementing the training loop in the `run_training_epoch` function from `train.py`. The main difference is the loss which should be cross entropy instead of the binary version.

3.3 Multi-layer perceptron (10 pts.)

Implement a linear classifier by filling in the `MLPClassifier` class from `lib/networks.py`. What performance do you obtain? Next, use an MLP with one hidden layer of dimension 32 followed by ReLU and then the final linear prediction layer. What is the new testing accuracy?

3.4 Convolutional network (10 pts.)

Implement a convolutional network by filling in the `ConvClassifier` class from `lib/networks.py`. The network should be as follows:

1. Convolutional layer `nn.Conv2d` with 3×3 kernel and 8 channels followed by ReLU and 2×2 max pooling `nn.MaxPool2d` with stride 2.
2. Convolutional layer `nn.Conv2d` with 3×3 kernel and 16 channels followed by ReLU and 2×2 max pooling `nn.MaxPool2d` with stride 2.
3. Convolutional layer `nn.Conv2d` with 3×3 kernel and 32 channels followed by ReLU and the already defined adaptive max pooling

Finally, the classifier should be a simple linear prediction layer. What testing accuracy do you obtain with this architecture?

3.5 Comparison of number of parameters (10 pts.)

Compute the number of parameters of the MLP with one hidden layer. Compute the number of parameters of the convolutional network. **You should count both the weights and biases.** Provide detailed explanations and computations.

3.6 Confusion matrix (10 pts.)

The confusion matrix $M_{i,j}$ is a useful tool for understanding the performance of a classifier. It is defined as follows: $M_{i,j}$ is the number of test samples for which the network predicted i, but the ground-truth label was j. Ideally, in the case of 100% accuracy, this will be a diagonal matrix. Based on the `run_validation_epoch` function from `train.py`, update `plot_confusion_matrix.py` to compute the confusion matrix. Provide the plot in the report and comment the results.