Machine Learning Methods

Exercise 4

30.12.2024

1 General Description

In this exercise, you will experiment with Multi-Layer Perceptron (MLP) and Convolutional Neural Network (CNN) models. We will start by examining MLPs on a dataset of europe countries similar to the one from Ex. 3. We will then move on to a real dataset where you will classify images to 2 classes.

You are expected to hand in a report, no longer than 8 pages, describing your experiments and answers to the questions in the exercise. The submission guidelines are described in Sec. 9, please read them carefully.

Note 1: You must explain your results in each question! Unless specified otherwise, an answer without explanation will not be awarded marks.
Note 2: When you are asked to plot something, we expect that plot to appear in the report.

2 Seeding

You should seed your code as in the previous exercises. Specifically, we will need to see both numpy and pytorch as follows:
(i) `np.random.seed(42)`
(ii) `torch.manual_seed(42)`
NOTE: It doesn't matter which seed you just that your results are reproducible!

3 Provided Code Helpers

We provided you with some helper functions as in the previous exercises. The helper functions are not mandatory, but they can greatly assist you. They are found in `helpers.py`.

4 Resources

In the CNN part of the exercise, you will work with ResNet18 architecture, this model is bigger than the models you have worked with so far and is painfully slow on a CPU. Therefore you will need to work with a GPU, if you don't have one, you can use Google Colab. Google offers free use of GPU on their servers, for this you'll have to run your code as a Jupyter Notebook (.ipynb file), therefore, you may submit a Jupyter Notebook as your code. Remember to choose the GPU runtime under the "Runtime" → "Change runtime type" tab. If you are efficient with resources, **you will not need more than the free tier of Colab**, but no promises. Our experiments are very lightweight. If you are having trouble with slow runtimes (hours) your implementation is probably very inefficient. More about Google Colab in 7.1
You don't need a gpu for the MLP part.

5 Data

5.1 European Countries

The dataset used for our exploration consists of three distinct files: *train.csv, validation.csv, test.csv.* Each file is structured as a table, with three columns: longitude (`long`), latitude (`lat`) and `country`. The samples in our dataset are represented as 2D coordinates of cities in Europe from different countries, with the corresponding labels indicating the country. For instance, if we consider a 2D city coordinate such as $(48.866667, 2.333333)$ – representative of Paris – its label would be *'France'*. To make it easier, we already encoded the `country` column into integers.
We will use this spatial dataset for exploring MLPs, by classifying the cities into countries based on their geographical coordinates.

6 Multi-Layer Perceptrons

As you saw in class, MLPs are simply a sequence of alternating linear and non-linear functions. This can be easily implemented using the `pytorch` [1] library. In this section you will first learn to optimize a simple MLP. Then, you will ablate some of its algorithmic choices.

6.1 Optimization of an MLP

Since this is (probably) the first time you are training a neural network, lets take it step-by-step. In this section you will implement and train a small neural network, tweaking the important parameters of neural networks. This section is sort of a tutorial for training a neural network. In the next sections we are expecting you to apply the conclusion from here for training the rest of the models in the exercise.

6.1.1 Task

We will now optimize an MLP's training, so you will understand the importance of each hyper-parameter.
First you will implement a training pipeline from scratch. You are given a skeleton code, read the comments and complete the missing parts before answering the questions.
A few Notes:

1. Read about pytorch Dataset here or any other source online.

2. For clarification, in this assignment, when asked to implement a model with 6 layers, it includes the input layer but not the output layer (therefore, you'll initialize 7 *nn.Linear* instances in total

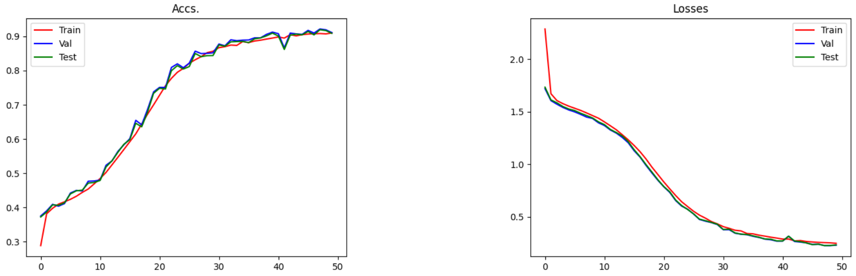
3. The activation function is ReLU.

4. Batch Norm should be before the activation

5. model.train() and model.eval() are necessary for some operations in pytorch (e.g. BatchNorm). It's not connected to back propagation, some layers act differently in train time and in test time. We encourage you to read about BatchNorm to understand why.

6. By default the loss functions in pytorch has *size_average* field set to False, meaning the loss is averaged over the batch. A different option would have been to sum the loss over the batch.

We already set some default hyper-parameters inside the given code. In each question you will alter one of them and look at the results.
Hint: The code plots several plots after each experiment. They might be very useful for analyzing your results, here is an example of an expected plot using the defaults parameters (used with *torch.seed(0)*)



(a) Accuracy

(b) Loss

Figure 1: Expected output behavior

6.1.2 Questions

1. **Learning Rate.** Train the network with learning rates of: $1., 0.01, 0.001, 0.00001$. Plot the validation loss of each epoch, for each one of the learning rates on the same graph. I.e. 4 lines on the same plot, x axis - epochs, y axis - loss. What does a too high learning rate do? What about a too low one? Explain.

2. **Epochs.** Train the network for 100 epochs. Plot in a single graph the loss over the train, test, and validation. How does the number of epochs affect the training? What happens after too many epochs? How does too little epochs affect? Explain.

3. **Batch Norm.** Add a batch norm (`nn.BatchNorm1d`) after each hidden layer. Compare the results of the regular and modified model as before. How does the batch normalization layer contribute the training? Explain.

4. **Batch Size.** Train the network with batch sizes of: $1, 16, 128, 1024$. Change the epochs number according to the table below. How did this choice affect the
(i) accuracy (val): Plot one graph with four curves of validation accuracy vs. epoch.
(ii) speed (train): In each case, mention the number of iterations needed per epoch
(iii) stability (train)? (i.e. how "noisy" is the loss over the epochs). Plot one graph with four curves of the training loss vs. **batch** (not vs. epoch like before).
Explain Your answers.

Batch Size	1	16	128	1024
Epochs	1	10	50	50

Table 1: Batch Size and Epochs Choices

6.2 Evaluating MLPs Performance

In the previous section you learned how to optimize a Neural Network (NN) and tweak its parameters. In this section you will train several MLPs and analyze their algorithmic choices (we will define the questions precisely).

Train 6 classifiers for the following combinations of depth (number of hidden layers) and width (number of neurons in each hidden layer):

Depth	1	2	6	10	6	6	6
Width	16	16	16	16	8	32	64

Table 2: Depth and Width Choices

6.2.1 Questions

1. Choose the model with the best validation accuracy. Plot its training, validation and test losses through the training iterations. Using the visualization helper, plot the prediction space of the model. Use the test points for the visualization. Did this model generalize well? Explain.

2. Do the same for the model with the worst validation accuracy. Did this model over-fit or under-fit the dataset? Explain.

3. **Depth of Network.** Using only the MLPs of width 16, plot the training, validation and test accuracy of the models vs. number of hidden layers. (x axis - number of hidden layers, y axis - accuracy). How does the number of layers affect the MLP? What are the advantages and disadvantages of increasing the number of layers in a neural network? Explain.

4. **Width of Network.** Using only the MLPs of depth 6, plot the training, validation and test accuracy of the models vs. number of neuron in each hidden layer. (x axis - number of neurons, y axis - accuracy). How does the number of neurons affect the MLP? Explain.

5. **Monitoring Gradients.** Train another model with 100 hidden layers each with 4 neurons. This time, keep the magnitude of the gradients for each layer, through the training steps. The magnitude of of the gradients is defined by: $grad\_magnitude = ||grad||_2^2$. Train this network for 10 epochs. For the layers: $0, 30, 60, 90, 95, 99$ plot the mean gradients magnitude through the training epochs (average the magnitudes of each layer in every epoch, i.e., divide by the number of batches in the epoch).
   Do you see any problem of vanishing or exploding gradients? If so, what could we modify in the network, other than its depth, to solve this problem? Explain.

6. **(Bonus up to 5 pts.)** Implement your suggested modification to Q5. Did it solve the problem? Explain.

7. **(Bonus 2.5 pts) Implicit Representation.** When applying deep learning on some low dimensional data with a sequential pattern (e.g. time, location, etc.) it is common to first create an implicit representation of the data. This simply means passing the data through a series of sines and cosines. Sines and cosines are functions which may take a NN several layers to implement by itself, but are very useful. By doing this process ourselves we can leave the network to focus on more complex patterns which we can't recognize and wish it to learn by itself. Implement an implicit representation pre-processing to the data, passing the input through 10 sine function $sin(\alpha \cdot x)$: where $\alpha \in \{0.1, 0.2, ..., 1.\}$. Train a NN with depth 6 and width 16 on top of these representations. Compare this model to a similar architecture (depth, width) on the standard form of the data. Which model is better? What was this model able to learn, that the other

model had trouble with? Explain. (Hint: plot the classification spaces of both models).

Notice that the `plot_decision_boundaries` function has been updated to include an implementation of the needed implicit representation (which you can adapt for the datasets as well).

7 Convolutional Neural Networks

In this section of the exercise, we will explore Convolutional Neural Networks (CNNs), a powerful class of deep learning models specifically designed for image-related tasks. Our primary goal here is to apply CNNs to a realistic real-world problem: detecting deepfake generated images. We'll be working with a labeled dataset containing both real human faces and deepfake-generated ones.

By the end of this exercise, you will gain practical experience in using CNNs for binary classification tasks, distinguishing between authentic and deepfake images. Additionally, you will have an opportunity to explore different training strategies, including training from scratch, linear probing, and fine-tuning.

7.1 Google Colab

For this part of the exercise you will work with bigger models then the ones you've worked with thus far. For this reason, you will need a GPU for fast training and inference. Since you probably don't have one, you can use Google Colab. A few notes about Google Colab, Google offers free use of GPUs for a limited amount of time, after the time is over your runtime will terminate (all files created or uploaded during your runtime will be deleted, you'll still have access to your notebook) and you'll have to wait a while, might take up to 12 hours. To avoid using all of your runtime we encourage you to first run all experiments with a small chunk of your dataset, and see it run on the CPU, before moving to the GPU.
Uploading the dataset directly to your runtime might fail, or take a while which is very inconvenient since restarting your kernel will remove your data and you'll have to repeat this process all over again. If this happens, you can upload the dataset to your drive as a zip file, mount your drive to your runtime and unzip it from your notebook. To mount your drive simple run this in a cell
>> *from google.colab import drive*
>> *drive.mount('/content/drive')*

For those of you new to Google Colab, the different cells can also execute terminal command (bash), therefore, you can unzip your files directly in your code. ChatGPT is very useful if you are stuck using Google Colab.

7.2 Cuda

Cuda allows for fast operation on gpu, there's much to say about cuda but for this exercise it's enough you read about how to perform a training loop on a gpu by moving the model and the batch to 'cuda'.

7.3 Data

For this task, we have a labeled dataset comprising both real human faces and deepfake-generated ones. This dataset is divided into training, validation, and test sets. Each set contains real and fake images. You can download the dataset from this link.

7.4 Task

Your task is to evaluate the following baselines:

1. **XGBoost:** Use the XGBoost classifier with default parameters.

2. **Training from Scratch:** Train a ResNet18 classifier on the training data. The output layer will have a dimension of 1, implementing logistic regression. All the ResNet18 weights will be trained from scratch in this part.

3. **Linear Probing:** Train a frozen ResNet18 model pretrained on ImageNet classification. We'll add a final linear layer on top of this frozen backbone, projecting the data into a size 1 dimension. Only this final layer will be optimized.

4. **(Bonus 2.5 pts) Linear Probing based on sklearn:** In this variation of the linear probing approach, we use a pre-trained ResNet18 model on the ImageNet classification dataset to extract feature representations from the training set. This process results in an array of size (N, d), where N represents the number of training samples, and d denotes the feature dimension. Once we obtain these feature representations, we train a Logistic Regression model using the sklearn library. Specifically, we use the `LogisticRegression` class from `sklearn.linear_model` module. This class provides an efficient implementation of logistic regression with default parameters.

5. **Fine-tuning:** Fine-tune a ResNet18 model pretrained on ImageNet classification. Similar to the linear probing approach, we'll add a final linear layer for binary classification. However, in this case, we will optimize the entire pretrained model.

For all the above baselines, we'll experiment with the Adam optimizer and with different learning rates, [1e-1, 1e-2, 1e-3, 1e-4, 1e-5]. The training will be conducted with a batch size of 32 samples for 1 epoch. The loss function of this task would be binary cross-entropy (you may find torch.nn.BCEWithLogitsLoss useful for this).

For the XGBoost classifier, employ the `xgboost` library with default parameters.

7.5 Model Implementation

We provide you with a skeleton code for the required CNN models and evaluation protocols in `cnn.py`, and skeleton code for the XGBoost classifier in `xg.py`. This code will serve as a starting point for your experiments, allowing you to focus on training and evaluation.

7.6 Questions

1. What are your two best models from each baseline and your worst one in terms of test accuracy? What learning rates did you use for the PyTorch CNN baselines? Explain why these are the trends you see.

2. Visualize 5 samples correctly classified by your best baseline but misclassified by your worst-performing baselines.

8 Ethics

We will not tolerate any form of cheating or code sharing. You may only use the `tqdm`, `numpy`, `pandas`, `mathplotlib`, `xgboost`, `pytorch`, `torchvision` and `sklearn` libraries.

9 Submission Guidelines

You should submit your report, code and README for the exercise as $ex4\_\{YOUR\_ID\}.zip$ file. **Other formats (e.g.** ***.rar*****) will not be accepted!**.

The README file should include your name, cse username and ID.

Reports should be in PDF format and be no longer than 8 pages in length. They should include any analysis and questions that were raised during the exercise. Please answer the questions in Sec. 6.2, 6.1,7 in sequential order. **You should submit your code (including your modified helpers file), Report PDF and README files alone without any additional files.**

9.1 Submission

Please submit a zip file named $ex4\_\{YOURID\}.zip$ that includes the following:

1. A README file with your name, cse username and ID.

2. The *.py*, *.ipynb* files with your filled code.

3. A PDF report.

References

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.