Exercise Sheet 3: Multi Layer Perceptrons (MLPs)

Due on 10.05.2024, 10:00

Pingchuan Ma (p.ma@lmu.de)

General Information and **Important Notes** see previous sheets.

The goal of the following exercise is to get you familiarized with Multilayer Perceptrons (MLPs) for classification taks, which is a fundamental part of many different networks. In the next exercises, you will see how we can inverse the classification process and turn it for visualization purposes. But in this sheet, we will focus on a simple classification task. For the first three tasks, there are code snippet provided. ***Hint:*** *You are free to modify the code, but they have to show that the missing parts are implemented by yourself and showcase your understanding of the task.*

Task 1: Backpropagation and Simple Training **(4P)**

In this exercise you will create your own implementation of a Multi Layer Perceptron and train it with the backpropagation algorithm. The network should consist out of alternating linear layers $f^{(k)}$ and ReLU activation functions $g^{(k)}$, whereas the last linear layer is followed by a softmax activation function p. An averaged Cross-Entropy loss $\mathcal{L} = \frac{1}{N}\sum_n H\left(p\left(x_n\right), y_n\right)$ should be used as cost function. In summary we have the following layers:

- **Linear**: $f_j\left(x_i\right) = \sum_i w_{ji}x_i + b_j$
- **ReLU**: $g_j\left(f_j\right) = \max\left(0, f_j\right)$
- **Softmax**: $p_j\left(f_j\right) = \frac{\exp(f_j+C)}{\sum_k \exp(f_k+C)}$ ($C = -\max\left(\mathbf{f}\right)$ is for numerical stabilization, but you can also try setting it to $C = 0$)
- **Cross-Entropy**: $H\left(p, y\right) = -\sum_j y_j \log\left(p_j\right)$ (y is a one-hot class vector)

Use the chainrule of calculus to obtain respective gradients for **Linear** and **ReLU** (example $\frac{\partial \mathcal{L}}{\partial w_{ji}^{(0)}}$), and update parameters by applying gradient descent. (we already did it for **Softmax** and **Cross-Entropy** in the last exercise)

The attached *task1.py* contains predefined classes and training setup. Your task is to

1. fill in code for the forward, backward pass, and update for **Linear** and **ReLU** (1P),

2. implement missing parts in the training routine (1P),
3. and Plot loss curves on training and test set (1P).
4. Increase numbers of hidden layers by changing **hidden_channels=[ ]**, which by default is set to **[30, 30]** and report what you observe on the training/test loss curves (related to a term we mentioned in the lecture). (1P)

In case you are not able to solve this task, you can also make use of pytorch layers as in the next exercise. However this will result in getting less points!

Task 2: Data Preparation and Visualization (4P)

Now let us move on to a harder dataset than the toy moon dataset we had above. When working with new data it is important to first look at its basic characteristics. One popular tool for this task is Principal Component Analysis (PCA). In the case of high dimensional image data (e.g. a tiny 28×28 pixel image is already 784 dimensional when reshaped to a vector) the PCA identifies the directions with the highest variance (i.e. principal components) in this high dimensional space. The idea is simple: The principal components of our data (e.g. 784-dimensional vectors), along which the position of individual data points (e.g. image vectors) varies the most, are likely the most informative part of our data. In this task, you will work with images from the popular MNIST dataset, prepare the training and the test sets, visualize their largest principal components, and use those to plot a two-dimensional distribution of the images. We provide you with a starter code for this task which you find in file *task2.py*.

1. Using the function **load_data**, load the train split of MNIST and plot 10 examples of digits with their label as title. Also report the following statistics about the train set: **min, max, mean, shape, dtype**. Here, you can imagine the train set as a large data array and report the statistics on that array. (1P)
2. Convert all images into plain vectors and process them to be centered around 0 in the range of **[-1, 1]**. In the end you should have two arrays of images and labels. (1P)
3. Now run the provided **do_pca** on the converted data in order to obtain a matrix of sorted eigenvectors that represent the principal components of the train set. Reshape the 10 most important principal components to the shape of **[28, 28]** in order to plot them as images. **Explain what you are**

seeing. What would you expect the principal components to look like, if the problem was easy? (1P)

4. Project the MNIST vectors of the train set onto the two most important principal components (associated with two largest eigenvalues). Use the dot product for the projection into the 2D feature space spanned by the two principal components and plot the resulting points in a scatter (use the **scatter** provided by **matplotlib** for this) plot. To get a better overview you can also choose a subset of the points. Color each dot corresponding to its class. Interpret the plot. What can it tell us about the MNIST dataset? Can you make a statement regarding the difficulty of MNIST digit classification problem? (1P)

Task 3: Defining, Training and Evaluating an MLP **(8P)**

Now you can rely on pytorch and let it take care of the gradients and update of the weights. Make yourself familiar with the pytorch API by looking at the following guide [1].

We will use MLPs to get to know the core parts of **pytorch** for neural network training, namely its **nn** library for defining layers of neural networks and its automatic backpropagation functionalities. All code accompanying this task can be found in the file *task3.py*.
Here is the great chance to switch to a GPU for training, either on CIP pool or your own GPU, otherwise it will take 28 **mins per epoch to train on CPU.**

1. Using all skeleton functions provided for this task, build a 5 layer neural network, which accepts MNIST vectors (those from *task 2*) as input and outputs for each MNIST vector a classification vector containing 10 values, one for each MNIST class. Each hidden layer of the network has 100 hidden units (output feature dimension). All hidden layers should use ReLU activations. (3P)

2. Implement a function, which can report the accuracy of a batch of predictions in percentage. (1P)

3. Train the network for at least 5 epochs and validate the classifier on the test split of the data after each epoch. Do so using the ADAM optimizer from

[1] https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html

pytorch.optim package and set the learning rate to 1×10^{-4} . Use the **CrossEntropyLoss** provided by pytorch as loss function and then update the network weights using backpropagation. (4P)

Task 4: Visualizing MLP Features **(4P)**

Now that we have a trained network let us take a look at its intermediate or hidden layers and the corresponding feature representation.

1. For all MNIST images of the test split, or if you have limited computational resource for a subset of at least 1000 images, store the activation features from after each ReLU call. You should have 4 arrays of shape **[10000, 100]**. (1P)
2. For each of the arrays repeat the steps from *task 2* to find the principal components of the resulting vectors and project the features onto the two most important principal components. (1P)
3. Make a scatter plot of the resulting projected points for the features from the first, second, third and fourth layer. What can you observe? How well are the individual digit classes separated in the scatter plot? (You can use the PCA implementation from *task 2* for this. Add the respective code for this task to the file *task3.py*.) (1P)
4. Compare the plots to the scatter plot from *task 2*. What has changed, what is similar? Do you think that the MLP is a suitable method for MNIST digit classification? Provide an explanation. (1P)

Task 5: Reading Materials on Normalization Layers **(0P)**

Some further reading materials on Normalization techniques if you are interested in:

1. Batch Normalization
2. Different Normalization Layers in Deep Learning
3. Why do some models use different kinds of normalization layers?