The multilayer perceptron

Deep learning for media technology (TNM112), lab 1

Abstract

The first lab is focused on getting familiar with the multi-layer perceptron (MLP). We will both look at training MLPs for simple 2 dimensional datasets in Keras, as well as implementing our own MLP with matrix operations in numpy.

1 Introduction

The multi layer perceptron (MLP) is simple in construction, but there are many aspects to take into consideration in relation to their use. This goes for both architectural choices (number of layers, layer size, activation function, etc.) and training (optimization algorithm, loss function, initialization, etc.). In this lab, we will look closer at these different aspects when applied to different simple 2D datasets.

For more information on how to report your results, please refer to the LaTeX lab template. It provides information on how to structure the lab report, as well as how results should be reported (both in lab report and provided code).

2 Method

One layer of the MLP can be illustrated as in Fig. 1, mapping from K output activations $h_i^{(l-1)}$ in layer $l-1$ to L activations $h_i^{(l)}$ in layer l. The feed-forward of information between the layers can be described as

$$h^{(l)} = \sigma\left(W^{(l)} h^{(l-1)} + b^{(l)}\right), \tag{1}$$

where $h^{(l)}$ is a $(L \times 1)$ vector with the activations of layer l, $W^{(l)}$ is a $(L \times K)$ matrix with the learnable weights $w_{i,j}$ in layer l, $h^{(l-1)}$ is a $(K \times 1)$ vector with the activations of layer $l-1$, and $b^{(l)}$ is a $(L \times 1)$ vector with the biases of layer l.

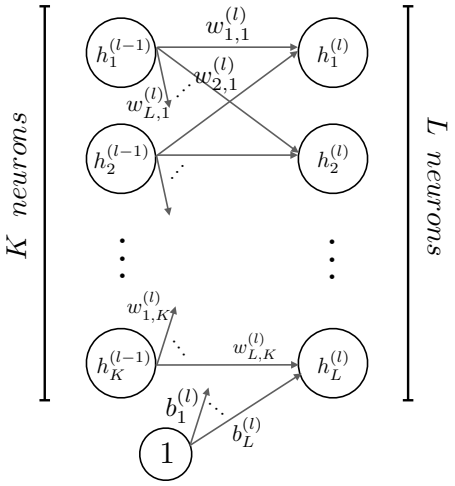

Figure 1: One layer in the MLP.

We will look both at implementing the MLP layers using Eq. 1, as well as exploring different design choices and optimization strategies in Keras[1]. Different design choices include:

1. The number of layers.
2. The number of neurons in each layer.
3. The activation function.

Choices in terms of the optimization include:

1. Optimizer
   (a) SGD without and with momentum.
   (b) Learning rate.
2. Batch size for SGD.
3. Number of training epochs for SGD.
4. Loss function.

[1] `https://keras.io/`

5. Initialization algorithm (the starting point of the optimization).

We refer to these choices (both in relation to architecture and optimization) as *hyper parameters*, i.e. parameters that can be tweaked but which are not part of the actual optimization procedure.

Notes

Data: In deep learning, it is important to use both a training set, a validation set, and a test set for measuring performance. The training set is used for the optimization with SGD. The validation set is used for monitoring the performance of different choices of hyper parameters. The test set is used to measure the final generalization performance on data that has not been seen during the model development. Although the lab code provides all these three sets of data, we will mostly use the training and test sets. This is mainly because we generate new datasets each time we start the training, so overfitting to the test set by finding the best combination of hyper parameters is not an issue, but you should keep in mind that usually we should do the hyper parameter tweaking by monitoring the performance on the validation set.

Performance measurements: You will observe that the results will vary slightly, even if you run training with exactly the same settings (dataset, model, and optimization). This is due to the stochastic nature of the optimization (random initialization point and random shuffling of datapoints for each training epoch). In our case, the dataset is also drawn randomly, which also impacts the final results. For reporting results in deep learning, you should optimally run multiple trainings with the same settings, and report the average performance. However, we will not consider this in the first lab, for efficiency reasons, but you should be aware that results will show some fluctuations.

3 Datasets

We will use simple datasets in 2 dimensions. Look through the `data_generator.py` code to understand how the datasets are generated and plotted. Datasets are generated using normal distributions with a predefined standard deviation, and with a selected number of classes. The '`linear`' dataset produces datapoints that can be linearly separated, see Fig. 2, while the '`polar`' dataset is defined in polar coordinates and then transformed to cartesian coordinates, see Fig. 3.

4 Assignments

You will test the results of the different tasks using the provided Jupyter notebook, `lab01.ipynb`. In addition to this, there are three python scripts, `data_generator.py`, `keras_mlp.py`, and `mlp.py`. For this lab, you will not need to do changes in the `data_generator.py` and `keras_mlp.py`. You will work in the Jupyter notebook, and do some implementations in `mlp.py`.

Task 1 – Keras MLP

In this task, we will train an MLP in Keras, and explore different datasets and hyper parameters to understand the interplay between dataset, network architecture and optimization settings. Look through the `keras_mlp.py` code to understand how the model is setup and trained. The class `KerasMLP` provides a wrapper around Keras[2]. We will not go into the details of using Keras, but it will be helpful to understand how to setup and train a model, not least for the upcoming labs where we will be using Keras more extensively.

Task 1.1

Use a linear dataset with 512 training points and 2 classes (similar to Fig. 2 (left)). Specify a network without hidden layers. This means that we will have a single layer, mapping directly from the input (2 dimensions) to the output (2 dimensions, since we have $K = 2$ classes). As the layer is the last layer, we will use the `softmax` activation function. This only performs a normalization so that the output layer can be thought of as a probability distribution, $\sum_{i=1}^{K} h_i = 1$, where h_i are the

[2]Keras itself can also be thought of as a wrapper around Tensorflow. The `KerasMLP` class is perhaps a bit superfluous in this sense, but it will make it convenient for our exploration and for comparing to our own implementation of an MLP in Task 2.

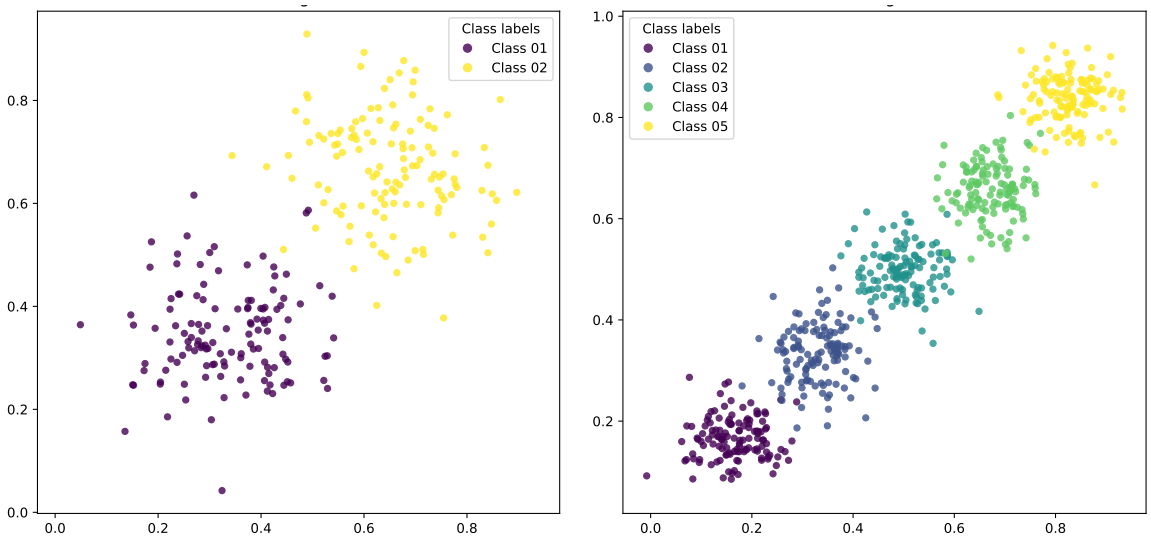

Figure 2: Synthetic 2D datasets with linear separation and different number of classes.

activations of the last layer and K is the number of neurons in this layer. For deciding which class the network predicts, the index of the output neuron with the maximum activation is used. You can look in the `sample_input_space` function in `data_generator.py` to see how classification is performed:

```
yp_test = model.feedforward(self.x_test)
yp_test = np.argmax(yp_test,1)
```

Here, the `argmax` function is used to find the index of the neuron with the maximum activation, which is the predicted class. Make sure you understand how this classification is performed.

Train the MLP with the same settings for 4 epochs using SGD with a learning rate of 1.0, and compare two different batch sizes:

- SGD with batch size 512
- SGD with batch size 16

Answer the following questions:

Q1 Which model is best at separating the two classes (provide the accuracy on the test set)? Elaborate on why there is a difference between the results, by comparing not only accuracy but also the behavior of the optimization[3].

Q2 Why is it possible to do the classification without non-linear activation function (there's only a softmax activation)?

Task 1.2

Use a polar dataset with 512 training points and 2 classes (similar to Fig. 3 (left)). Specify a network with one hidden layer with 5 neurons. Train for 20 epochs with learning rate 1.0 and batch size 16. Train with the same settings, and change only the activation function:

- Using 'linear' activation function
- Using 'sigmoid' activation function
- Using 'ReLU' activation function

Answer the following questions:

Q1 Why does linear activation not work?

Q2 On average, what is the best classification accuracy that can be achieved with a linear activation function on this dataset?

[3]Hint: how many iterations of SGD are performed in the two cases?

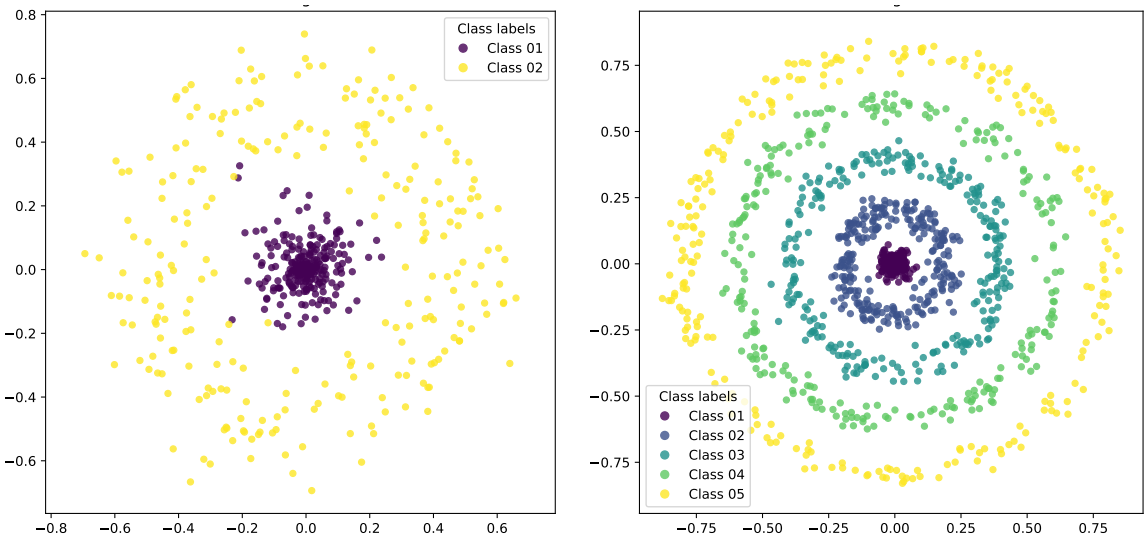

Figure 3: Synthetic 2D datasets with radial separation and different number of classes.

Q3 Can you find an explanation for the difference comparing sigmoid and ReLU activation[4]?

Task 1.3

Use a polar dataset with 512 training points and 5 classes, and sigma $\sigma = 0.05$ (it will be similar to Fig. 3 (right), but with more noise). Specify a network with 10 hidden layers with 50 neurons each and ReLU activation. Try different combinations of hyper parameters to get the best classification accuracy. Examples could be:

- Change mean and stddev of normal initialization.
- Change learning rate and add some momentum in SGD.
- Check the Keras documentation for 'ExponentialDecay', which can be used to specify decaying learning rate for SGD.
- You can also tweak batch size and number of epochs.

You will need to test some different combinations to get the optimization to converge. In doing so, you will hopefully gain some understanding for what changes are critical for successful optimization. To make this process a bit faster, you can start with a limited number of epochs (say 20) and monitor if the optimization improves the performance at all. When you find combinations that seem to work, you can train for more epochs to see if this improves the model.

In this process, make notes on what combinations of hyper parameters works and does not work. Based on you your study, elaborate on the following questions:

Q1 What combination worked best, and what was your best classification accuracy (on the test set)?

Q2 Can you find any patterns in what combinations of hyper parameters work and doesn't work?

Task 1.4

Continue from the previous task, but change initialization to '`glorot_normal`' and optimizer to '`keras.optimizers.Adam()`'. Does this perform better compared to your results in Task 1.3?

[4]Hint: how does the vanishing gradient problem relate to the different activation functions?

Task 2 – Implementing an MLP

From Task 1, you should hopefully have gained some understanding for how different hyper parameters impact the optimization on different datasets. In this task, we will implement our own MLP using matrix operations in numpy.

Implement the `activation`, `setup_model`, `feedforward`, and `evaluate` functions in `mlp.py`. Run the code provided in the last cell of the Jupyter notebook and compare the results to the results generated by the Keras model in the previous cell (compare number of weights, classification loss/accuracy, and the plots with decision boundaries). Test this for different specifications of the model in the previous cell (number of hidden layers, layer width, and activation function).

The instructions in the code in `mlp.py` should be sufficient to understand what you are supposed to add in terms of code. However, here are some pointers to facilitate the task:

- Make sure you understand the matrix operations in Eq. 1.
- It could also be helpful to draw the full diagram/connection scheme for an MLP.
- The hidden layers are the layers which are not directly connected to the output layer.
- Have a look at the `get_weights` function in `keras_mlp.py` to understand how weights and biases are arranged in lists when these are extracted from the Keras model.
- For implementing the `evaluate` function, it could be helpful to look at the code snippet above, under Task 1.1.

In the lab report, you should discuss around how an MLP is implemented in code with the aid of a matrix operation library. Go through all the steps you have implemented and explain how you have solved the problems in code. You should also provide the relevant code as supplementary material when submitting your lab report.

Task 3 – Dissecting the MLP

In the final task, we will use a minimal example of an MLP to understand how the decision boundaries

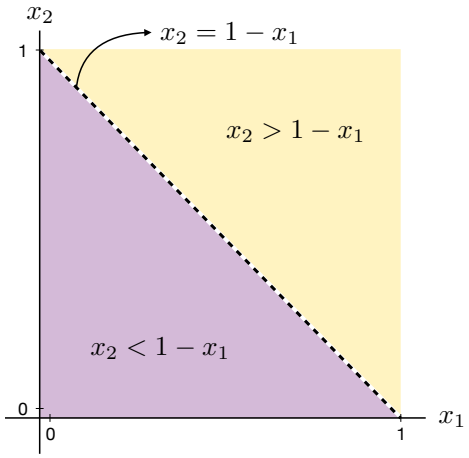

Figure 4: Decision boundary, where the purple area is predicted as class 1 and the yellow area is predicted as class 2.

are drawn in a classifier. Use the linear dataset with 2 classes ($K = 2$). Change the specification of the model parameters. Instead of extracting the weights from the Keras model, you should manually specify a weight matrix (2×2) and a bias vector (2×1), i.e. no hidden layers. Remember that each layer is supposed to be an item in a list, which means that you need to specify the weight matrix and the bias vector in separate 1-item lists. As we only have one layer, it will use the 'softmax' activation function (it will not make a difference if you specify other activation functions, since the last layer always uses the softmax).

Solve the following problems and answer the corresponding questions:

- Manually derive the weights (2×2) and biases (2×1) to specify a model that draws a decision boundary at $x_2 = 1 - x_1$, as depicted in Fig. 4. What is the simplest possible solution? How many additional solutions are possible?
- How can you, in the simplest way, change the weights/biases to switch the predicted class labels? That is, so that the predicted label of the lower vs. upper region in Fig. 4 switches.
- By writing out the output of the MLP (that is, specifying how each output y_i is a combination of input features x_i, weights, and biases, according to Eq. 1), motivate why your choice

of weights and biases creates a decision boundary at $x_2 = 1 - x_1$. Can you find a general formula for specifying which combinations of weights and biases will generate the decision boundary?

5 Conclusion

In this lab, you should have gained knowledge about how MLPs are constructed and trained. Your lab report should show with sufficient clarity that you have gained this knowledge, and that you can discuss around MLPs in a way that shows your understanding.