Assignment 2: *Feedforward Neural Networks*

Date: *Oct 1, 2025* Due : *Oct 15, 2025*

Preface

This is the second series of assignments for the course *Applied Deep Learning*. The exercises are aimed to review fully-connected feedforward neural networks studied in Chapter 2. Below, you can find the information about contents of these exercises, as well as instructions on how to submit them.

General Information The assignments are given in two sections. In the first section, you have written questions that you can answer in words or by derivation. The questions are consistent with the material of Chapter 2 and you do not need any further resources to answer them. The second section includes the programming assignments. For these assignments, you are assumed to have some very basic knowledge of programming in Python. For those who are beginners, a quick introduction to the install and use of Python libraries has been given in a separate file. In the case that a question is unclear or there is any flaws, please contact over Piazza. Also, in case that any particular assumption is required to solve the problem, feel free to consider the assumption and state it in your solution. The total mark of the assignments is **100 points** with total mark of written questions adds to **40 points** and the total mark of the programming assignments adds to **60 points**.

How to Submit A notebook has been provided with basic code that you can complete. The submission is carried out through the Crowdmark. Please submit the answer of each question **separately**, following the **steps below**. Please note that *failure to meet the formatting can lead to mark deduction*.

1. For Written Questions, you can submit handwritten or typed answers as a `.pdf`, `.jpg` or `.png` file.
2. For Programming Questions, the answer should **complete the Python Notebook** shared with this assignment. For *each question*, please print the corresponding part of the notebook as a `.pdf` file and upload it in the corresponding field on Crowdmark. Your uploaded `.pdf` should contain **all figures, diagrams and codes requested** in the question.
3. The completed Notebook, i.e., the `.ipynb` file, including all the codes and the outputs should also be submitted as the attachment to the last item on Crowdmark.

When submitting your notebook, please pay attention to the following points:

1. The file should be named `Lastname_Firstname_Assgn2.ipynb`
2. Please make sure to name the files with the name that is displayed on the Quercus account

The deadline for your submission is on **October 15, 2025 at 11:59 PM**.

- You can delay up to two days. After this extended deadline no submission is accepted.
- For each day of delay, 10% of the mark after grading is deducted.

Please submit your assignment **only through Crowdmark, and not by email.**

1 Written Exercises

Question 1 [20 Points] **(Forward and Backward Pass)** Consider the simple FNN in **Figure. 1.1**. This is the same architecture we used to realize XOR in Chapter 1 with modified activation functions: in the hidden layer, we use *soft-ReLU* activation, i.e., the function $f(\cdot)$ in **Figure. 1.1** is

$$f(z) = \log(1 + e^z),$$

and in the output layer, we use a sigmoid, i.e.,

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

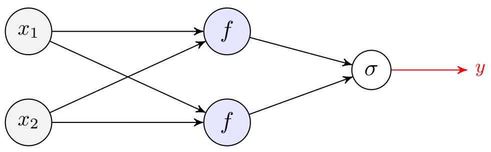

Figure 1.1: Fully-connected FNN with two-dimensional input $\boldsymbol{x} = [x_1, x_2]^\mathsf{T}$.

To train this FNN, we use the *cross-entropy loss function*. We are given with the following data-point $\mathbf{x}$ and label v

$$\mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \text{and} \qquad v = 1.$$

We intend to perform one forward and backward pass by hand. To this end, we assume that all weights and biases are initiated by value 0.1. Answer the following items:

1. Compute all variables in the forward pass. You have to explain the order of your computation using the forward propagation algorithm.
2. Compute the gradient of the loss with respect to all the weights and biases at the given initial values *via backpropagation*.
   Note: You **must** use the backpropagation algorithm.
3. Assume we are training this FNN only on this single data-point. Compute the updated weights and biases for the next iteration of gradient descent algorithm.

Question 2 [10 Points] **(Chain-Rule for Bias)** Assume that scalar $\hat{R} \in \mathbb{R}$ is a function of vector $\mathbf{y} \in \mathbb{R}^K$, i.e., $\hat{R} = \mathcal{L}(\mathbf{y})$, for some $\mathcal{L}(\cdot) : \mathbb{R}^K \mapsto \mathbb{R}$. We already have computed the gradient of $\hat{R}$ with respect to $\mathbf{y}$, i.e., we know the value of $\nabla_\mathbf{y} \hat{R}$. We further know that $\mathbf{y}$ is an affine function of an input $\mathbf{z} \in \mathbb{R}^N$, i.e.,

$$\mathbf{y} = \mathbf{A}\mathbf{z} + \mathbf{b}$$

for some weight matrix $\mathbf{A} \in \mathbb{R}^{K \times N}$ and bias vector $\mathbf{b} \in \mathbb{R}^K$. We want to compute the gradient of $\hat{R}$ with respect to the bias $\mathbf{b}$ from $\nabla_\mathbf{y} \hat{R}$. To this end, answer the following items:

1. Plot the vectorized computation graph of this functional relation, i.e., the computation graph that relates $\mathbf{b}$ to $\hat{R}$.
2. Determine the gradient of $\hat{R}$ with respect to $\mathbf{b}$, i.e., $\nabla_\mathbf{b} \hat{R}$, in terms of the known gradient $\nabla_\mathbf{y} \hat{R}$.
   Hint: *You need to complete the backward pass of the computation graph.*

Question 3 [10 Points] **(Softmax plus Cross-Entropy)** Vector $\mathbf{z} \in \mathbb{R}^N$ has been transformed by *softmax* to $\mathbf{y} \in \mathbb{R}^N$. This means that $\mathbf{y}$ has also N entries and its entry j is computed from $\mathbf{z}$ as

$$y_j = \frac{e^{z_j}}{\Pi(\mathbf{z})}$$

where $\Pi(\mathbf{z})$ is defined as

$$\Pi(\mathbf{z}) = \sum_{i=1}^{N} e^{z_i}.$$

We compute scalar $\hat{R}$ from $\mathbf{y}$ by finding its cross-entropy with vector $\mathbf{p} \in \mathbb{R}^N$ which is defined as

$$\hat{R} = \mathrm{CE}(\mathbf{y}; \mathbf{p}) = -\sum_{j=1}^{N} p_j \log y_j.$$

1. Write $\hat{R}$ explicitly in terms of $\mathbf{z}$.
2. Compute the gradient of $\hat{R}$ with respect to $\mathbf{z}$.
   Hint: *You may use the explicit expression you found in Part 1.*
3. Write down the computed gradient, i.e., $\nabla_{\mathbf{z}} \hat{R}$, in terms of the vectors $\mathbf{y}$ and $\mathbf{p}$.

Good to Know
Due to what you showed in Question 3, many people call *backpropagation* the *error-propagation* algorithm.

2 Programming Exercises

Question 1 [48 Points] **(Implementing Deep FNN)** In this assignment, we implement a deep FNN for a simple classification task. Read the following test to get familiar with required packages and modules.

Required Packages We are going to use three main packages `torch`, `numpy` and `sklearn`. The two former are used for computational tasks while the latter is used to generate some simple dataset. From `sklearn`, we mainly require the datasets and a function that splits them into training and test batches. So, we only import them. We need to use the *neural network* module of PyTorch, i.e., `torch.nn`, frequently. We hence import it separately and call it `nn`. We also import the package `tqdm` for visualization.

```python
import torch
import torch.nn as nn
import numpy as np
import sklearn.datasets as DataSets
from sklearn.model_selection import train_test_split
from tqdm import tqdm
```

Our first step is to load a dataset: we consider a simple dataset for *binary classification*. Our dataset has input vector data-points $\mathbf{x}_i \in \mathbb{R}^N$ which are labeled by binary integers, i.e., $v_i \in \{0, 1\}$. We can load such a dataset using the function `make_classification()` in module `sklearn.datasets`. With this in mind, complete the following items.

1. Write a simple code that loads a binary classification dataset with 10,000 sample data-points of dimension $N = 20$. Save the input data-points in list `X` and their corresponding labels in list `v`.
2. Verify the dimensions of `X` and `v` and print the first data-point along with its label. Specify the data type using the function `type()`.

We next write a function to split the dataset into training and test dataset. We also split the training dataset into mini-batches that can be used to train the model by SGD. To learn how to do it, read the following text:

Data Splitting To split the data we use the `train_test_split()` function that we initially imported from module `sklearn.model_selection`. For splitting into training and test datasets, we should specify `train_size` argument that is a *real number between* 0 *and* 1 that specifies the fraction of total data used for training. For splitting training dataset into mini-batches, we need to specify argument `batch_size`, which is an integer specifying the size of each mini-batch.

3. Write the function `data_splitter()` that gets `X` and `v` as well as `batch_size` and `train_size` and returns training and test datasets after *random shuffling*. The function should also return a list of integers that indicate the beginning index of each mini-batch.
4. Verify the output of the implemented function for two independent instances.

Unlike former practices in the lecture and tutorials, we now use module `torch.nn` to implement our FNN in only few lines. For this, read the following text.

Basic Classes Class `nn.Linear()` realizes a linear layer, `nn.ReLU()` is the ReLU activation function, and `nn.Sigmoid` realizes the sigmoid activation. When we instantiate a class, e.g., `nn.Linear()`, its weights are initiated randomly. Each time we apply a step of SGD, these weights get updated. We can access the weights and other parameters of these classes by looking into their attributes.
We now implement an FNN with following specification:

- It has *two hidden layers*: the first hidden layer has 64 neurons, and the second hidden layer has 32 neurons.
- All hidden neurons are *activated by ReLU*.
- The *output layer* has a single neuron activated by *sigmoid*.

Complete the following tasks:

5. Write the class `myClassifier()` inherited from class `nn.Module` that implements the above FNN.
6. Write the function `forward()` for the class `myClassifier()` that takes a sample input data-point $\mathbf{x}$ and completes the forward pass. The function should return the output of the FNN.
7. Pass the first data-point in your dataset `X` forward and observe the model's output. Compare this output with the true label in list `v`.

We next implement the backward pass. This can be readily done by the auto-grad feature of PyTorch that you have learned in tutorials. Read the following text for more details.

Backpropagation in PyTorch PyTorch realizes backpropagation directly through its auto-grad functionality. This means that you do ***not*** need to write a separate `backward()` function for your class. You can directly calculate the gradient by applying `.backward()` method on the risk. It is worth knowing that behind the scene, PyTorch sketches the same computation graph that we had in the lecture. Once we apply the method `.backward()`, it computes all partial derivatives on the graph. At this point, it is clear to you that you need to complete the forward pass to be able to apply `.backward()`.
Given the above description, we now practice backward pass by PyTorch.

8. Write a simple script that takes a sample from dataset and passes it forward through the model. Compute the loss between the output and the true label using binary cross-entropy. Apply the method `.backward()` on the computed loss. You may find class `nn.BCELoss()` useful.
9. Print the partial derivative of the loss with respect to the bias of the output neuron **before** and **after** applying `.backward()`. What do you see? You may find the attribute `.bias.grad` of the output layer helpful for accomplishing this task.

The auto-grad feature of the PyTorch helps us easily implement the training loop by the built-in optimizers. To learn this, read the following text:

Optimizer in PyTorch Since in PyTorch each variable has its grad, we can readily realize any optimization algorithm for training in a single line of code. We should specify which optimizer we want. We could access pre-implemented optimizers in the module `torch.optim`. We then pass the parameters of our model to the optimizer and specify the learning rate. If a `model` is instantiated from a class written by `torch.nn` module, like our class `myClassifier()`, its parameters are accessible via the (inherited) method `.parameters()`.

10. Write a code that instantiate the model `myClassifier()` and the Adam optimizer. Pass the model parameters to the optimizer and set the learning rate to 0.0001.

We now have all we need to implement the training loop. We only need to split data, break the training dataset into mini-batches, apply the optimizer in each iteration and test the model at the end of each epoch. There are two last pieces of trick:

- Whenever we apply `.backward()` on a variable, PyTorch updates all gradients on the computation graph. To make sure that PyTorch does not computes a new gradient based on previous computation graph, before each backward pass we make the gradients zero by applying `.zero_grad()` on the optimizer. We see this clearly in the implementation.

- When we start training, we use `.train()` method on our model to tell PyTorch that we are training. This tells PyTorch how to deal with things like *dropout* and *batch-normalization*[1]. Once we start testing, we apply `.eval()` method on our model to tell PyTorch that we are now testing, and hence it can use the parameters computed up this iteration of training for evaluation.

We now complete the training and test loop for 300 epochs and batch size 40. For training, we use Adam optimizer.

11. Complete the code in the attached `.ipynb` file which implements the training loop as the function `training_loop()`. This function gets an instance of the model as input and returns the training and test loss of each epoch in two separate lists.

12. Instantiate `myClassifier()` and run the training loop to train the model. Plot both the training and test risk against the number of epochs. You can do this by completing the `.ipynb` file.

Question 2 [12 Points] **(MNIST Dataset)** In this question, we learn how to load MNIST data-points as a `torch.Tensor`. You can read MNIST from module `torchvision.datasets`. We can further use the module `torchvision.transforms` to convert the data-points to `torch.Tensor`. Read the following description to complete this task.

Loading a Data-Point We can now readily load MNIST (or many other datasets) as

```python
import torchvision.datasets as DS
import torchvision.transforms as transform
mnist = DS.MNIST('./data' , train=True, transform=transform.ToTensor(), download=True)
```

In this code, we indicate that the dataset is saved in folder `'data'` inside our current directory. We indicate that we load the training dataset. We apply the transform `.ToTensor()` to load them as `torch.Tensor`, and finally we let the dataset to be downloaded. The object `mnist` is a collection of 60,000 tuples: the first entry of the tuple is a `torch.Tensor` whose entries are pixels of the image and the second entry is the label.
Given above description, answer the following items.

[1]We have not yet applied these techniques to our model. But we will do that soon.

1. Use the command `len()` to check the length of object `mnist`.
2. Read the first tuple in `mnist` and specify its pixel tensor and label.
3. Use the method `.reshape()` to reshape the pixel tensors into pixel vectors.

PyTorch has modules that can be used to divide a dataset into mini-batches. But, we want to do this ourselves in this task. You can imagine how easy it is: we only need to make a loop.

4. Complete the function `myBatcher` in the `.ipynb` file that gets `batch_size` as input and returns a list of mini-batches of size `batch_size`.
5. Run the function `myBatcher` with `batch_size = 100` and print the labels of the first and third mini-batches.