ECE421: Introduction to Machine Learning — Fall 2024

Assignment 3: Feed-forward Neural Network

Due Date: Friday, November 15, 11:59 PM

General Notes

1. Programming assignments can be done in groups of up to 2 students. Students can be in different sections.
2. Only one submission from a group member is required.
3. Group members will receive the same grade.
4. Please post assignment-related questions on Piazza.

Group Members

Name (and Name on Quercus)	UTORid
Edwin Chacko	chackoed
Bala Kannan Murali	muralib1

1 Implementing Feed-Forward Neural Network Using `NumPy`

1.1 Basic Network Layers

1.1.1 ReLU

1.1.1.a Derive the gradient of the downstream loss with respect to the input of the ReLU activation function, Z. You must arrive at a solution in terms of $\partial L/\partial Y$, the gradient of the loss w.r.t. the output of ReLU $Y = \sigma_{\text{ReLU}}(Z)$, and the batched input Z, *i.e.*, where $Z \in \mathbb{R}^{m\times n}$. Include your derivation in your writeup. [**HINT:** you are allowed to use operations like elementwise multiplication and/or division!]

Answer. The ReLU activation function is defined as:

$$Y = \sigma_{\text{ReLU}}(Z) = \max(0, Z)$$

where:

- Y is the output of the ReLU activation.
- Z is the input to the ReLU activation.
- The ReLU activation is applied elementwise.
  Using the chain rule:
  $$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial Z}$$
  Therefore, the derivative of ReLU is:
  $$\frac{\partial Y}{\partial Z_{ij}} = \begin{cases} 1 & \text{if } Z_{ij} > 0 \\ 0 & \text{if } Z_{ij} \leq 0 \end{cases}$$
  This can be written as:
  $$\frac{\partial Y}{\partial Z} = \nVdash(Z > 0)$$
  where $\nVdash(Z > 0)$ is an indicator function.
  Substituting into the chain rule:
  $$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial Y} \odot \nVdash(Z > 0)$$
  Final Expression is given by:
  $$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial Y} \odot \nVdash(Z > 0)$$
  Here:
  - $\frac{\partial L}{\partial Y}$: Downstream gradient.
  - $\nVdash(Z > 0)$: Indicator function acting as a mask where a mask means that if Z is greater than 0, the gradient remains unchanged whereas if Z is less than or equal to 0, then Z contributes nothing to the output Y.
  - $\odot$: Elementwise multiplication

1.1.2 Fully-Connected Layer

1.1.2.a Derive the gradients of the loss $L \in \mathbb{R}$ with respect to weight matrix $W \in \mathbb{R}^{n^{[l]} \times n^{[l+1]}}$, i.e., $\partial L / \partial W$, and with respect to the bias row vector $\mathbf{b} \in \mathbb{R}^{1 \times n^{[l+1]}}$, i.e., $\partial L / \partial \mathbf{b}$, in the fully-connected layer. You will also need to take the gradient of the loss with respect to the input of the layer $\partial L / \partial X$, which will be passed to lower layers, where $X \in \mathbb{R}^{m \times n^{[l]}}$ is the batched input. Again, you must arrive at a solution that uses batched X and Z. Please express your solution in terms of $\partial L / \partial Z$, which you have already obtained in question 1.1.1.a, where $Z = XW + \mathbf{1}^\top \mathbf{b}$ and $\mathbf{1} \in \mathbb{R}^{1 \times m}$ is a row of ones. Note that $\mathbf{1}^\top \mathbf{b}$ is a matrix whose each row is the row vector $\mathbf{b}$. So, we are adding the same bias vector to each sample during the forward pass: this is the mathematical equivalent of numpy broadcasting. Include your derivations in your writeup.

Answer. The gradients of the loss L with respect to the input Z of the ReLU activation:

$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial Y} \odot \nVdash(Z > 0)$$

where:

- $\frac{\partial L}{\partial Y}$: Gradient of the loss with respect to the output Y,
- $\nVdash(Z > 0)$: Element-wise indicator function.

Additionally, the pre-activation output Z is given by:

$$Z = XW + \mathbf{1}^\top b$$

where:

- $X \in \mathbb{R}^{m \times n^{[l]}}$: Batched input,
- $W \in \mathbb{R}^{n^{[l]} \times n^{[l+1]}}$: Weight matrix,
- $b \in \mathbb{R}^{1 \times n^{[l+1]}}$: Bias row vector,
- $\mathbf{1} \in \mathbb{R}^{1 \times m}$: Row vector of ones.

1. Gradient with respect to W

$$\frac{\partial L}{\partial W} = X^\top \frac{\partial L}{\partial Z} = X^\top \left(\frac{\partial L}{\partial Y} \odot \nVdash(Z > 0) \right)$$

2. Gradient with respect to b

$$\frac{\partial L}{\partial b} = \mathbf{1} \frac{\partial L}{\partial Z} = \mathbf{1} \left(\frac{\partial L}{\partial Y} \odot \nVdash(Z > 0) \right)$$

3. Gradient with respect to X

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Z} W^\top = \left(\frac{\partial L}{\partial Y} \odot \nVdash(Z > 0) \right) W^\top$$

1.1.3 Softmax Activation

1.1.3.a For a single training point, derive $\partial\sigma_i/\partial s_j$ for an arbitrary (i, j) pair, *i.e.* derive the Jacobian Matrix

$$\begin{bmatrix} \frac{\partial\sigma_1}{\partial s_1}, & \frac{\partial\sigma_1}{\partial s_2}, & \dots, & \frac{\partial\sigma_1}{\partial s_k} \\ \frac{\partial\sigma_2}{\partial s_1}, & \frac{\partial\sigma_2}{\partial s_2}, & \dots, & \frac{\partial\sigma_2}{\partial s_k} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial\sigma_k}{\partial s_1}, & \frac{\partial\sigma_k}{\partial s_2}, & \dots, & \frac{\partial\sigma_k}{\partial s_k} \end{bmatrix}.$$

Include your derivation in your writeup. You do not need to use batched inputs for this question; an answer for a single training point is acceptable.

Answer. Generally,

$$J_{i,j} = \frac{\partial L}{\partial s_j} = \frac{\partial L}{\partial \sigma_i} \cdot \frac{\partial \sigma_i}{\partial s_j}$$

For $i = j$,

$$\frac{\partial\sigma_i}{\partial s_j} = \frac{\partial\sigma_i}{\partial s_i} = \frac{\partial}{\partial s_i}\frac{e^{s_i-m}}{\sum_{j=1}^k e^{s_j-m}} = \frac{\partial}{\partial s_i}\frac{f(s_i)}{g(s_i)} = \frac{f_{s_i}(s_i)\cdot g(s_i) - g_{s_i}(s_i)\cdot f(s_i)}{g(s_i)^2}$$

$$= \frac{e^{s_i-m}\cdot\sum_{j=1}^k e^{s_j-m} - e^{s_i-m}\cdot e^{s_i-m}}{(\sum_{j=1}^k e^{s_j-m})^2} = \frac{e^{s_i-m}}{(\sum_{j=1}^k e^{s_j-m})} - \frac{e^{s_i-m}\cdot e^{s_i-m}}{(\sum_{j=1}^k e^{s_j-m})^2}$$

$$= \frac{e^{s_i-m}}{(\sum_{j=1}^k e^{s_j-m})} - \frac{(e^{s_i-m})^2}{(\sum_{j=1}^k e^{s_j-m})^2}$$

$$= \frac{e^{s_i-m}}{(\sum_{j=1}^k e^{s_j-m})} - \left(\frac{e^{s_i-m}}{\sum_{j=1}^k e^{s_j-m}}\right)^2$$

$$= \sigma_i - \sigma_i^2 = \sigma_i(1-\sigma_i)$$

For $i \neq j$,

$$\frac{\partial\sigma_i}{\partial s_j} = \frac{\partial}{\partial s_j}\left(\frac{e^{s_i-m}}{\sum_{l=1}^k e^{s_l-m}}\right) = e^{s_i-m}\cdot\frac{\partial}{\partial s_j}\left(\frac{1}{\sum_{l=1}^k e^{s_l-m}}\right) = e^{s_i-m}\cdot\frac{\partial}{\partial s_j}\left(\sum_{l=1}^k e^{s_l-m}\right)^{-1}$$

$$= e^{s_i-m}\cdot\left(\sum_{l=1}^k e^{s_l-m}\right)^{-2}\cdot\frac{\partial}{\partial s_j}\left(\sum_{l=1}^k e^{s_l-m}\right) = -e^{s_i-m}\cdot\left(\sum_{l=1}^k e^{s_l-m}\right)^{-2} e^{s_j-m}$$

$$= -\frac{e^{s_i-m}\cdot e^{s_j-m}}{(\sum_{l=1}^k e^{s_l-m})^2}$$

$$= -\frac{e^{s_i-m}}{\sum_{l=1}^k e^{s_l-m}}\cdot\frac{e^{s_j-m}}{\sum_{l=1}^k e^{s_l-m}}$$

$$= -\sigma_i\cdot\sigma_j$$

Therefore, we have found that,

$$\frac{\partial\sigma_i}{\partial s_j} = \begin{cases}\sigma_i(1-\sigma_i) & \text{if } i = j \\ -\sigma_i\cdot\sigma_j & \text{if } i \neq j\end{cases}$$

1.1.4 Cross-Entropy Loss

1.1.4.a Derive $\partial L/\partial \hat{Y}$ the gradient of the cross-entropy cost with respect to the network's predictions, $\hat{Y}$. You must use batched inputs. Include your derivation in your writeup.
[**HINT:** You are allowed to use operations like elementwise multiplication and/or division!]

Answer. Note that $\ln \hat{Y}$ represents an element-wise natural logarithm.

For a minibatch, we have $Y \in \mathbb{R}^{m\times k}$ and $\hat{Y} \in \mathbb{R}^{m\times k}$ with rows $\mathbf{y}_i \in \mathbb{R}^m$ and $\hat{\mathbf{y}}_i \in \mathbb{R}^m$, respectively.

$$Y = \begin{bmatrix} \text{---} & \mathbf{y}_1^\top & \text{---} \\ \text{---} & \mathbf{y}_2^\top & \text{---} \\ & \vdots & \\ \text{---} & \mathbf{y}_m^\top & \text{---} \end{bmatrix} \qquad \hat{Y} = \begin{bmatrix} \text{---} & \hat{\mathbf{y}}_1^\top & \text{---} \\ \text{---} & \hat{\mathbf{y}}_2^\top & \text{---} \\ & \vdots & \\ \text{---} & \hat{\mathbf{y}}_3^\top & \text{---} \end{bmatrix}$$

Expanding the dot product,

$$L = -\frac{1}{m}\sum_{i=1}^{m} \mathbf{y}_i \cdot \ln \hat{\mathbf{y}}_i = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k} y_{ij} \cdot \ln \hat{y}_{ij}$$

Solving the gradient of each element in $\hat{Y}$

$$\frac{\partial L}{\partial \hat{y}_{\alpha\beta}} = \frac{\partial}{\partial \hat{y}_{\alpha\beta}}\left(-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k} y_{ij} \cdot \ln \hat{y}_{ij}\right) = -\frac{1}{m}\frac{y_{\alpha\beta}}{\hat{y}_{\alpha\beta}}$$

$$J_{\alpha\beta} = \frac{\partial L}{\partial \hat{y}_{\alpha\beta}} = -\frac{1}{m}\frac{y_{\alpha\beta}}{\hat{y}_{\alpha\beta}}$$

Alternatively, we can use matrix calculus, using the operator $\circ$ to denote element-wise multiplication and $\oslash$ to denote element-wise division.

$$\frac{\partial L}{\partial \hat{Y}} = \frac{\partial}{\partial \hat{Y}}\left(-\frac{1}{m} Y \circ \ln \hat{Y}\right) = \frac{\partial L}{\partial \hat{Y}} = -\frac{1}{m} Y \circ \frac{\partial}{\partial \hat{Y}}\left(\ln \hat{Y}\right)$$

$$\frac{\partial L}{\partial \hat{Y}} = -\frac{1}{m} Y \oslash \hat{Y}$$

Therefore, we can write,

$$\frac{\partial L}{\partial \hat{Y}} = -\frac{1}{m} Y \oslash \hat{Y}$$

Where

$$\frac{\partial L}{\partial \hat{Y}}_{ij} = -\frac{1}{m}\frac{y_{ij}}{\hat{y}_{ij}}$$

1.2 Two-Layer Fully Connected Networks

1.2.1 Implementing Fully Connected Layer

1.2.2 Hyperparameter Tuning (in a very small scale)

1.2.2.a try at least 3 different combinations of these hyperparameters. Report the results of your exploration, including the values of the parameters you explored and which set of parameters gave the best test error. Provide plots showing the loss versus iterations for your best model and report your final test error.

Answer.

- The results of the exploration (including the values of the parameters we explored):

(lr, n_out)	Training Loss	Training Accuracy	Val Loss	Val Accuracy	Test Loss	Test Accuracy
(0.01, 5)	1.1005	0.296	1.1269	0.2667	1.1096	0.26
(0.01, 10)	0.0786	0.96	0.0875	0.9333	0.4447	0.74
(0.03, 10)	0.0612	0.968	0.0706	0.9333	0.0535	0.98
(0.05, 10)	0.4998	0.584	0.3235	0.8	0.446	0.62
(0.01, 100)	0.0363	0.984	0.0024	1.0	0.1831	0.96
(0.03, 100)	0.042	0.984	0.068	1.0	0.1292	0.96
(0.001, 100)	0.0384	0.992	0.003	1.0	0.1952	0.96

Table 1: Results per hyperparameter test run.

The best model is `lr` $= 0.03$ and `n_out` $= 10$ based on the lowest test loss and thereby highest test accuracy. The larger models with `n_out` $= 100$ get better validation accuracies but do not see gains in test accuracy or loss. This indicates over fitting and so we pick the smaller model. Additionally, this model performs well with 10 times less units, making it preferred for performance – not that it is a significant constraint here.

- Figure 1 below shows the loss versus iterations for my best model.
- The set of parameter that gave the best test error is: `lr` $= 0.03$ and `n_out` $= 10$

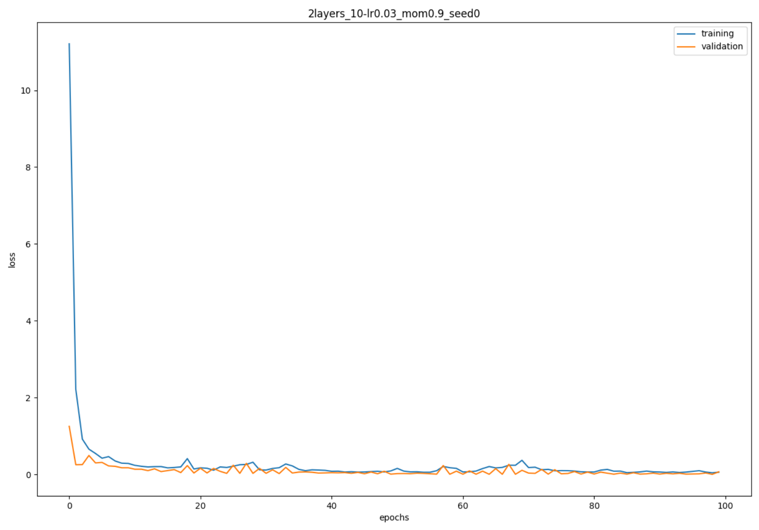

Figure 1: Loss versus iterations for the best model.

2 Implementing Neural Networks Using `PyTorch`

2.1 Understanding `PyTorch`

2.2 Implementing a Multi-Layer Perceptron Model Using `PyTorch`

2.2.a Code for training an MLP on MNIST (you should upload your ipynb notebook and also provide your code for this section in `PA3_qa.pdf`, as a screen shot, or in latex typesetting, etc.).

Answer.

```
images = [training_data[i][0] for i in range(9)]
plt.imshow(
            torchvision.utils.make_grid(torch.stack(images),
            nrow=3,
            padding=5
            ).numpy().transpose((1, 2, 0)))
### YOUR CODE HERE ###
import numpy as np
import torch
import torchvision
from torch import nn
from torch.utils.data import DataLoader
from tqdm import tqdm
import matplotlib.pyplot as plt

class CustomDataset(torchvision.datasets.FashionMNIST):
    """Reshapes image to single dimensional vector of dim 784"""
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def __len__(self):
        return super().__len__()
    def __getitem__(self, index):
        img, target = super().__getitem__(index)
        return img.reshape(-1), target

class MLP(nn.Module):
    def __init__(self, in_dim):
        super().__init__()
        self.l1 = nn.Linear(in_dim, 256)
        self.l2 = nn.Linear(256, 10)

        self.relu = nn.ReLU()

    def forward(self, x):
        l1 = self.relu(self.l1(x))
        l2 = self.l2(l1)
        return l2

def load_data():
    # Defining transformation
    transform = torchvision.transforms.ToTensor()

    # Use custom dataset to load
    training_data = CustomDataset(
        root="data",
        train=True,
        download=True,
        transform=transform,
    )

    validation_data = CustomDataset(
        root="data",
        train=False,
        download=True,
        transform=transform,
    )

    return training_data, validation_data

def plot_metrics(training_losses, training_accs, val_losses, val_accs):
    epochs = range(1, len(training_losses) + 1)

    # Plot training and validation losses
    plt.figure(figsize=(8, 6))
    plt.plot(epochs, training_losses, label="Training Loss")
    plt.plot(epochs, val_losses, label="Validation Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Loss vs. Epochs")
    plt.legend()
    plt.savefig("loss_vs_epochs.png", dpi=300)
    plt.show()

    # Plot training and validation accuracies
    plt.figure(figsize=(8, 6))
    plt.plot(epochs, training_accs, label="Training Accuracy")
    plt.plot(epochs, val_accs, label="Validation Accuracy")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.title("Accuracy vs. Epochs")
    plt.legend()
    plt.savefig("accuracy_vs_epochs.png", dpi=300)
    plt.show()



def validate(model, criterion, val_loader, num_val_pts, device):
    # Validate on val set
    with torch.no_grad():
        model.eval() # Put model in eval mode
        num_correct = 0
        for x, y in val_loader:
            x, y = x.float().to(device), y.long().to(device)
            pred = model(x)

            loss = criterion(pred, y)

            _, pred_indices = torch.max(pred, dim=1)
            num_correct += torch.sum(pred_indices == y).item()
        acc = num_correct / num_val_pts
        print("Validation Accuracy:", round(acc, 4))

        model.train() # Put model back in train mode
    return loss.item(), acc

def main():
    # Define device and training params
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print("Using device:", device)
    epochs = 20
    batch_size = 128
    learning_rate = 0.001

    # Load datasets and create dataloaders
    train, val = load_data()
    num_val_pts = len(val)
    num_train_pts = len(train)

    train_loader = DataLoader(train, batch_size, shuffle=True, num_workers=1)
    val_loader = DataLoader(val, batch_size, num_workers=1)

    # Create model on device
    model = MLP(784).to(device)

    # Define optimizer and loss
    optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
    criterion = torch.nn.CrossEntropyLoss()
    # Put model in training mode
    model.train()

    # Run training loop epoch times
    training_losses = []
    training_accs = []
    val_losses = []
    val_accs = []
    for epoch in tqdm(range(epochs), desc="Training␣Progress"):
        tqdm.write(f"Epoch␣{epoch␣+␣1}/{epochs}")
        training_loss = []
        num_correct = 0
        # Training loop
        for x, y in tqdm(train_loader, unit="batch"):
            x, y = x.float().to(device), y.long().to(device)
            optimizer.zero_grad()
            pred = model(x)

            loss = criterion(pred, y)
            loss.backward()
            optimizer.step()

            _, pred_indices = torch.max(pred, dim=1)
            num_correct += torch.sum(pred_indices == y).item()
            training_loss.append(loss.item())

        # Compute training and val acc and loss
        train_loss = np.mean(training_loss)
        train_acc = num_correct / num_train_pts
        val_loss, val_acc = validate(model, criterion, val_loader, num_val_pts, device

        # Store training and val acc and loss
        training_losses.append(train_loss)
        training_accs.append(train_acc)
        val_losses.append(val_loss)
        val_accs.append(val_acc)

        print(f"Epoch␣{epoch+1}.␣training␣loss:␣{train_loss:2f},␣training␣acc:␣{train_a
)

    plot_metrics(training_losses, training_accs, val_losses, val_accs)
    torch.save(model.state_dict(), "model_smaller_AdamW.pth")


if __name__ == "__main__":
    main()
```

2.2.b A plot of the training loss and validation loss for each epoch of training after trainnig for at least 8 epochs.

Answer.

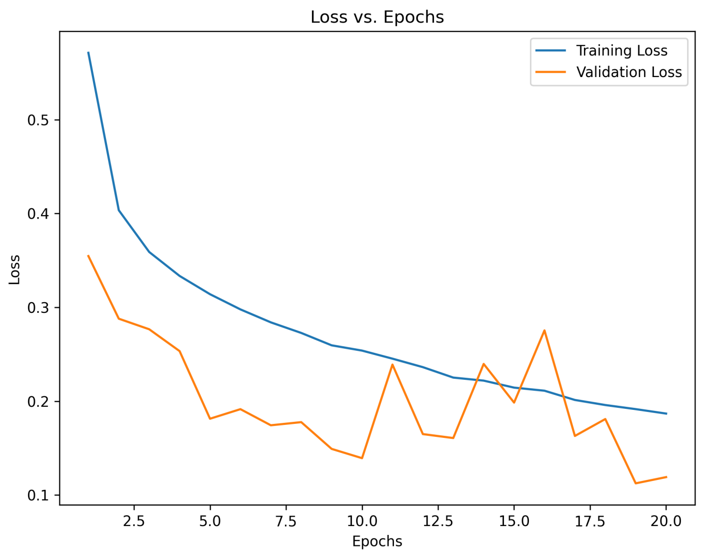

Figure 2: Training loss and validation loss for each epoch.

2.2.c A plot of the training and validation accuracy, showing that it is at least 82% for validation by the end of training.

Answer.

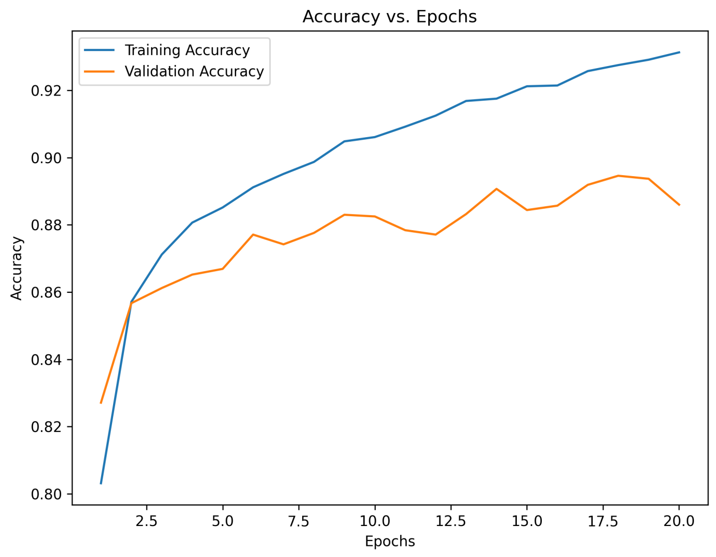

Figure 3: Training and validation accuracy, showing that it is at least 82% for validation by the end of training.

3 Discussion

Please answer the following short questions so we can improve future assignments.

3.a How much time did you spend on each part of this assignment?

Answer. I spent roughly 5 hours on section 1.
1.1 took around 3 hours and 1.2.1 took 40 mins. In 1.1, 1.1.1 took around 40 mins, 1.1.2 took around 50 mins, 1.1.3 took 1 hr, 1.1.4 took less than 30 mins. 1.1.3 took a lot of time particularly with figuring out the conversion between the derived loss and implementing it in `NumPy`. This does not include writing up the derivations in latex. Each of the derivations were fairly simple to come to. Implementing these in the code was an increasingly challenging affair. Writing the derivations in latex (for 1.1.3 and 1.1.4) took around 40 mins. 1.2.2 and adding the plots and creating the table in LaTeX took around an hour.

Section 2 took around 3 hours in total. Bulk of the time was spent experimenting with different model architectures. The plotting took minimal time. The `NumPy` tutorial was brief and easy to follow.

3.b Any additional feedback? How would you like to modify this assignment to improve it?

Answer. This was a good assignment that helped me learn more about neural networks both theoretically and practically. I especially liked the use of `NumPy` and torch. I felt like the assignment could have been improved by making some tasks into smaller more modular ones, such as the layers. I enjoyed the hyperparameter tuning portion, but it might be interesting to explore something equivalent to keras tuner, just for demonstration. The use of visual animations could have helped in understanding the content better but this has its limitations. The MNIST had no structured code and this was good and helped challenge me.