# 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 |
|---|---|
| Vicky Chen | 1006672021 |
| Victor Liu | 1007363586 |

# 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.** Your answer ...

$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial Z}$$

$$= \frac{\partial L}{\partial Y} \cdot \frac{\partial}{\partial Z}\left[\sigma_{ReLU}(Z)\right]$$

$$= \frac{\partial L}{\partial Y} \odot H$$

Where $H \in \mathbb{R}^{m \times n}$, and $H_{ij} = \begin{cases} 0, & \text{if } Z_{ij} < 0 \\ 1, & \text{if } Z_{ij} \geq 0 \end{cases}$

### 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.** Your answer ...

$$Z = XW + 1^T b$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial W} = X^T \cdot \frac{\partial L}{\partial Z}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial b} = \frac{\partial L}{\partial Z} \cdot 1^T = \sum_{i=1}^{m} \frac{\partial L}{\partial Z_i}$$

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial X} = \frac{\partial L}{\partial Z} \cdot W^T$$

### 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

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

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.

$$\sigma_i = \frac{e^{s_i - m}}{\sum_{j=1}^{k} e^{s_j - m}}, \quad m = \max_{j \in \{1, \ldots, k\}} s_j$$

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

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

① when $i = j$: $\frac{\partial \sigma_i}{\partial s_j} = \frac{\partial \sigma_i}{\partial s_i}$

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

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

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

② when $i \neq j$: $\frac{\partial \sigma_i}{\partial s_j} = \frac{0 - e^{s_i - m} \cdot e^{s_j - m}}{\left( \sum_{j=1}^{k} e^{s_j - m} \right)^2} = -\sigma_i \sigma_j$

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

### 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.** Your answer ...

$$\frac{\partial L}{\partial \hat{Y}} = \frac{\partial}{\partial \hat{Y}} \left( -\frac{1}{m} \sum_{i=1}^{m} y_i \cdot \ln \hat{y}_i \right)$$

$$= -\frac{1}{m} \sum_{i=1}^{m} y_i \cdot \frac{1}{\hat{y}_i}$$

## 1.2 Two-Layer Fully Connected Networks

### 1.2.1 Implementing Fully Connected Layer

### 1.2.2 Hyperparameter Tuning (in a very small scale)

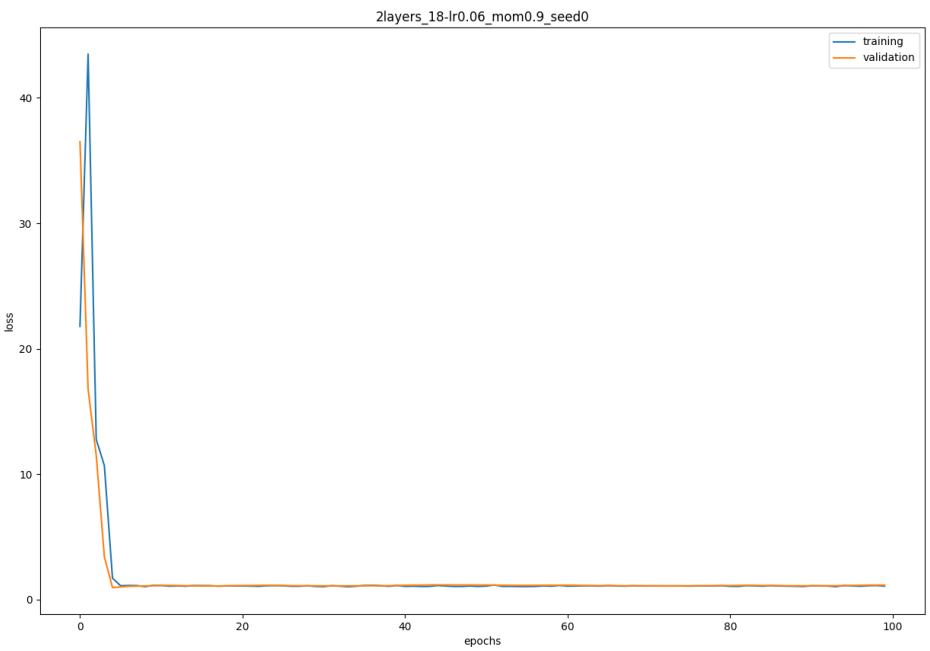
1.2.2.a try at least 3 different combinations of these hyperparameters. Report the results of your exploration, including the values of the parameters you explored and which set of parameters gave the best test error. Provide plots showing the loss versus iterations for your best model and report your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

**Answer.**

- The results of the exploration (including the values of the parameters we explored): ...
- The figure below shows the loss versus iterations for my best model.

Combination 1: lr=0.001, n_out=3 -> Test Loss: 1.6276, Test Accuracy: 0.02
Combination 2: lr=0.06, n_out=18 -> Test Loss: 1.0945, Test Accuracy: 0.38
Combination 3: lr=0.1, n_out=20 -> Test Loss: 1.1484, Test Accuracy: 0.38

The set of parameters that gave the best error is: lr=0.075 , n_out=18



# 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.**

```
unique_labels = set([label for _, label in training_data])
print("unique labels in trainign data:", unique_labels) # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
print("num of classes:", len(unique_labels)) # gives 10 classes

# Multi-Layer Perceptron Model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 64) # or 128
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(64, 10)  # output layer for 10 classes

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # flatten the image
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

epochs = 10
batch_size = 10
learning_rate = 0.01

model = MLP().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()

# load training and validation data
training_loader = torch.utils.data.DataLoader(training_data, batch_size=batch_size, shuffle=True)
validation_loader = torch.utils.data.DataLoader(validation_data, batch_size=batch_size, shuffle=False) # Don't shuffle validation data

training_losses = []
validation_losses = []
training_accuracies = []
validation_accuracies = []

model.train()  # put model in training mode
for epoch in range(epochs):
    training_loss = []
    correct = 0 # count the num of correct prediction in current epoch

    for images, labels in tqdm(training_loader, unit="batch"):
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        training_loss.append(loss.item()) # append the batch loss to training loss list
        _, pred = outputs.max(1) # get the index of the predicted class (with the highest probability)
        result = pred.eq(labels) # this returns tensor([True, False, ...])
        correct += result.sum().item() # sums up number of True's, and convert to scalar

    training_acc = correct / len(training_data)
    training_losses.append(sum(training_loss) / len(training_loader))
    training_accuracies.append(training_acc)
```

```
    print(f"Finished Epoch {epoch + 1}")
    print(f"Training Loss: {training_losses[-1]}, Training Accuracy: {training_acc}")

    with torch.no_grad():
        model.eval()  # put model in eval mode
        val_loss = []
        num_correct = 0
        for images, labels in validation_loader:
            images, labels = images.to(device), labels.to(device)
            out = model(images)
            loss = criterion(out, labels)
            val_loss.append(loss.item())
            _, predi = out.max(1)
            num_correct += predi.eq(labels).sum().item()

    val_acc = num_correct / len(validation_data)
    validation_losses.append(sum(val_loss) / len(validation_loader))
    validation_accuracies.append(val_acc)

    print(f"Validation Loss: {validation_losses[-1]}, Validation Accuracy: {val_acc}")
```

```
# epoch_indices = range(1, epochs + 1)
plt.figure(figsize=(6, 5))

# plot loss
plt.plot(training_losses, label='Training Loss')
plt.plot(validation_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')

# plot accuracy
plt.figure(figsize=(6, 5))
plt.plot(training_accuracies, label='Training Accuracy')
plt.plot(validation_accuracies, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')

plt.show()
```

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.**

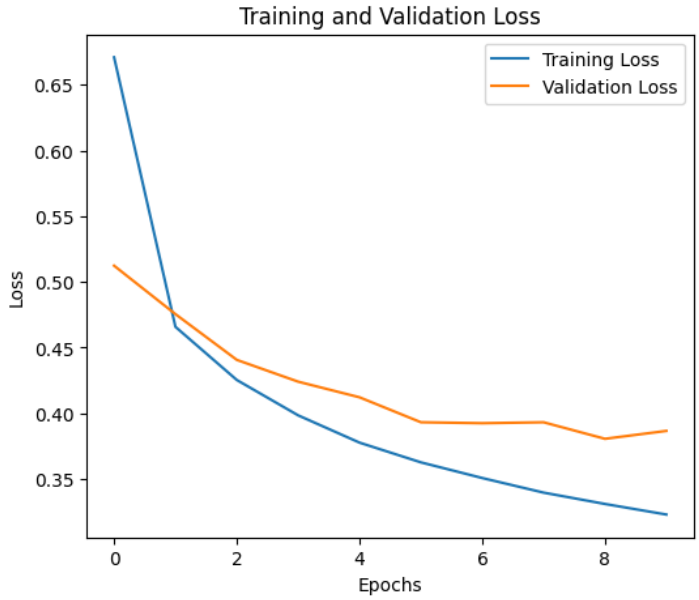



Figure 2: Training loss and validation loss for each epoch.

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.**

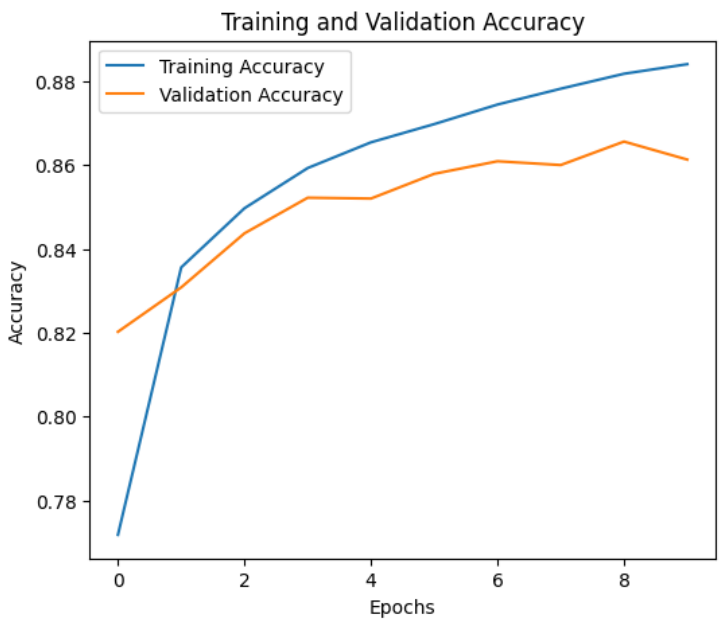



Figure 3: Training and validation accuracy, showing that it is at least 82% for validation by the end of training.

3.a How much time did you spend on each part of this assignment? (5 mark)

1.1.1, 1.1.2, 1.1.4 each took about 30 min, 1.1.3 took about an hour.
1.2 took about 90min
Section 2 took about 2hrs

3.b Any additional feedback? How would you like to modify this assignment to improve it? (5 mark)

Section 2 was a bit challenging for those who have little to no experience with pytorch