

Name: Tuguldur Ts. 鐵特德

Professor:孫民

Student ID:109006271

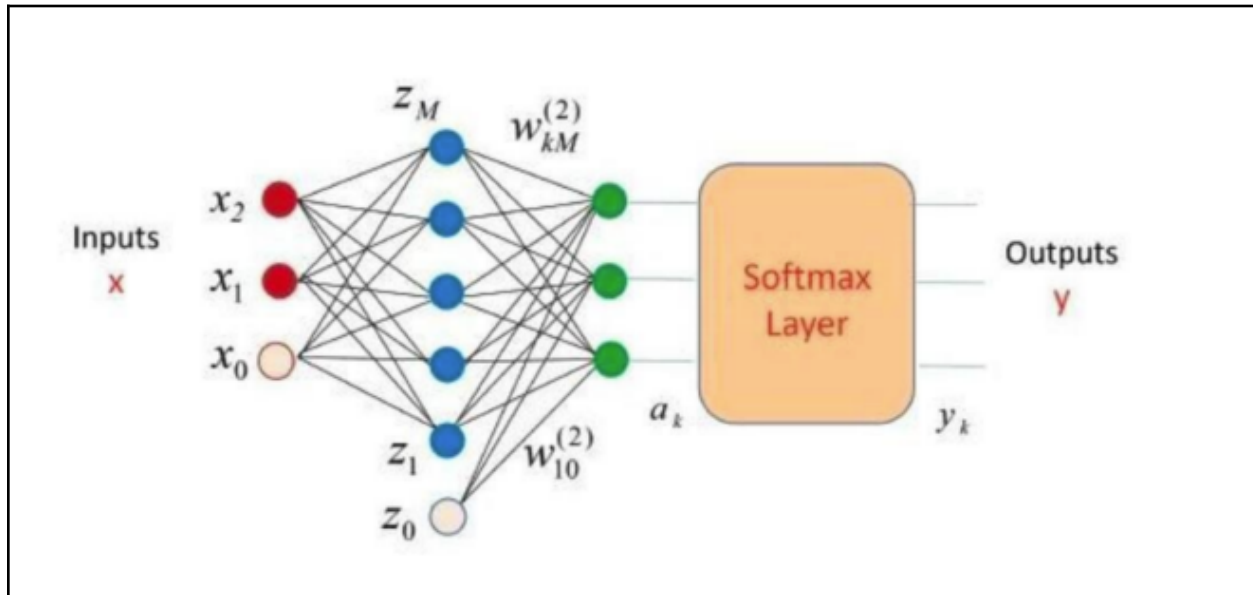
Content: Homework 2 Programming Report

Course: Machine Learning

Homework 2 Programming Report

| | |
|---|-----------|
| Part 0) Introduction..... | 2 |
| Part 1.1) 2 Layer Neural Network Model Architecture Details..... | 3 |
| Part 1.1.1) Data Preprocessing..... | 3 |
| Part 1.1.2) Loss Function Definition..... | 5 |
| Part 1.1.3) Weights and Biases Initialization..... | 7 |
| Part 1.1.4) Forward and Backward Pass Implementation..... | 8 |
| Part 1.1.5) Model Accuracy Evaluation..... | 11 |
| Part 1.2) Test Accuracy..... | 12 |
| Part 1.3) Plotting Loss Curves..... | 14 |
| Part 1.4) Plotting Decision Regions for Training and Testing Data..... | 15 |
| Part 2.1.0) 3 Layer Network Model Architecture Details..... | 17 |
| Part 2.1.1) Relu Function..... | 18 |
| Part 2.1.2) Backward Pass Function..... | 20 |
| Part 2.2) Test Accuracy..... | 20 |
| Part 2.3) Plotting Loss Curves..... | 23 |
| Part 2.4) Plotting Decision Regions for Training and Testing Data..... | 24 |
| Part 3) Comparing 2 and 3-Layer Neural Network Performance..... | 25 |
| Part 4) Try Other Regularization and Comparing Performance..... | 26 |
| Part 5) References..... | 29 |

Part 0) Introduction



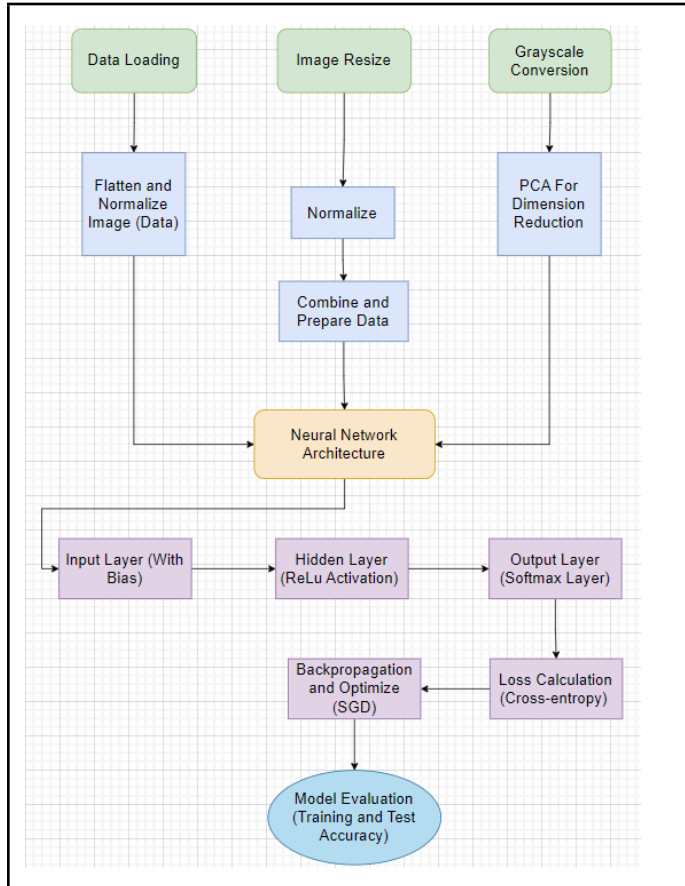
(Figure 0.1 Two-layer neural network with three inputs nodes and output nodes.)

The goal of this homework was to design and implement a two-layer neural network model that successfully works on three types of fruit (Carambula, Lychee, and Pear) and classify them. However, the challenging part of this homework is to not use any machine learning library such as Tensorflow or Pytorch where we have to build a neural network from scratch. The hint given to us was that “implement the backpropagation algorithm”. In this report, I will explain how I built the 2 and 3-Layer Network Models, starting with their model architecture and some important Python coding. Then, will go in comparing their performance as their training/test accuracy, loss, and decision regions, and finally Try Other Regularization and Comparing Performance. An example, a model chart can be seen in the **Figure 0.1**.

(To node: The pictures of the accuracy may not match the ones in the “ipynb” file as they would seem slightly off because I did the part 4 (using the python libraries). Because I had to restart the kernel couple of times and the original data that used in the report is before doing part 4.)

Part 1.1) 2 Layer Neural Network Model Architecture Details

Initially, to develop the two-layer neural network model, I conceptualized the structure through a detailed flowchart, as illustrated in **Figure 1.1.1.0, titled 'Flowchart of 2-Layer NN.'** This visualization aids in understanding the network's architecture and the data flow from input to output, which is fundamental to executing the subsequent steps in the model construction.



(Figure 1.1.1.0 Flowchart of 2-Layer NN)

The workflow was meticulously crafted, progressing sequentially through the following critical stages:

Part 1.1.1) Data Preprocessing

This includes the application of Principal Component Analysis (PCA) to reduce the dimensionality of the input data. It helps facilitate computation and possibly improve the model's capacity for generalization by making it concentrate on the most pivotal characteristics of the

input image data during putative dimensionality reduction. I have created a function “load_image” where it resizes it to **100x100** and also it utilizes the python library “from **PIL** import **Image**” to open the images/datas. In case, the image sizes are inconsistent with each other. After standardizing the images, I convert the images into a **Numpy** array. From the **Numpy** array, I use the “flatten” function to convert the 2D image arrays into 1D format by flattening it. Lastly, append to the list to return where it is used as the target output for training the neural network where the function returns the numpy array and label of the images. The coding snippet can be seen in the **Figure 1.1.1.1 Data Process Snippet Part 1**.

```
with Image.open(image_path) as img:
    img = img.resize((img_height, img_width))
    image_array = np.array(img)
    images.append(image_array.flatten())
    labels.append(idx)
```

(Figure 1.1.1.1 Data Process Snippet Part 1)

After all the images are processed with the “load_image” function, X_train and y_train both save the processed images and their corresponding labels respectively. Now it is ready to do the Principal Component Analysis (PCA). Initializes a PCA transformer to reduce the dimensionality of the image data to 2 principal components. This helps in reducing the complexity and computational cost of training the neural network, while retaining the most important information in the images. Fits the PCA model to the training data and transforms it according to the model. The result, X_train_pca, contains the two principal components for each image.

```
# Apply PCA to reduce to 2 dimensions
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)

#for matrix multi
X_train_pca_with_bias = np.hstack((X_train_pca, np.ones((X_train_pca.shape[0], 1))))
```

Figure 1.1.1.2 Data Process Snippet Part 2)

Thus, I add an bias term because I was getting error when doing matrix multiplication later on in other functions. With the help of numpy hstack function, horizontally stacks the principal components and a column of ones (the bias term). The X_train_pca.shape was (1470,2) before, now it is (1470,3). This can be seen in the **Figure 1.1.1.2 Data Process Snippet Part 2**.

Part 1.1.2) Loss Function Definition

The model utilizes the cross-entropy loss function, which is particularly suitable for classification tasks. The cross-entropy loss measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. The Formula is:

$$L = -\frac{1}{m} \sum_{i=1}^m \sum_{j=1}^n y_{i,j} \log(p_{i,j}) \text{ whereas}$$

- m is the number of samples.
- n is the number of classes.
- y is the true label (one-hot encoded).
- p is the predicted probability.

In the code, there is a small constant “1e-8” to avoid log(0). To explain the “y_true * np.log(y_pred + 1e-8)”, this computes the logarithm of the predicted probabilities which is y_pred. The result is then multiplied element-wise by the true labels y_true. Only the correct class's probability contributes to the sum.

The next function is the Cross-Entropy Derivative which is:

$$\frac{dL}{dp} = p - y \text{ whereas}$$

- p is the predicted probability.
- y is the true label.

The derivative of the cross-entropy loss with respect to the output of the softmax (predictions) is used to update the weights during backpropagation. It's crucial for understanding how to adjust the weights to reduce the loss in iterations. The python coding snippets can be seen in the **Figure**

1.1.2.1 Cross-Entropy Loss Function Snippet down below. To note, this was referenced from online, for more information check the reference at the end of the report.

```
def cross_entropy_loss(y_true, y_pred):  
    """  
    y_true: true labels, one-hot encoded, shape (n_classes, n_samples)  
    y_pred: predicted probabilities, shape (n_classes, n_samples)  
    Add epsilon to avoid log(0)  
    """  
    m = y_true.shape[1]  
    loss = -np.sum(y_true * np.log(y_pred + 1e-8)) / m  
    return loss  
  
def cross_entropy_derivative(y_true, y_pred):  
    """  
    y_true: true labels, one-hot encoded, shape (n_classes, n_samples)  
    y_pred: predicted probabilities, shape (n_classes, n_samples)  
    This is the gradient of the loss w.r.t the output of softmax  
    """  
    return y_pred - y_true
```

(Figure 1.1.2.1 Cross-Entropy Loss Function Snippet)

In this encoding process only one element against a row is equal to 1, and all other columns against that particular row are equal to 0; One row is made against a class and many columns against many samples of that particular class. In this matrix, a '1' is put in the row belonging to the class label of the sample, while all the other elements in the column are set to '0'. The python coding snippet can be seen in the **Figure 1.1.2.2 One-hot encode**.

```
# One-hot encode the labels  
y_train_one_hot = np.zeros((output_size, len(y_train)))  
y_train_one_hot[y_train, np.arange(len(y_train))] = 1
```

(Figure 1.1.2.2 One-hot encode)

This particular encoding method proves very efficient in translating the categorical class labels, to a binary format understood by the neural network where the cross-entropy loss function can

directly compare the probability distribution calculated by the neural network, against the actual class labels. This is useful mainly in forecasting the loss during training and the gradient obtained from the loss when updating the model parameters. In the “ipynb” file, I have added some docstring in the Cross-Entropy Loss Function cell.

Part 1.1.3) Weights and Biases Initialization

Before training, weights and biases are initialized.

- **W1** and **b1** are the weights and biases for the connections between the input layer and the hidden layer.
- **W2** and **b2** are the weights and biases for the connections between the hidden layer and the output layer.

```
# For reproducibility
np.random.seed(42)
W1 = np.random.randn(hidden_size, input_size) * np.sqrt(2. / (input_size + hidden_size))
b1 = np.zeros((hidden_size, 1))
W2 = np.random.randn(output_size, hidden_size) * np.sqrt(2. / (hidden_size + output_size))
b2 = np.zeros((output_size, 1))
```

(Figure 1.1.3.1 Weight and Biases Initialization)

np.random.randn(hidden_size, input_size) generates $\text{hidden_size} \times \text{input_size}$ normally distributed random values. This is scaled by **np.sqrt(2. / (input_size + hidden_size))**, which is a common practice to help in maintaining a variance of 1 in the outputs of the neurons, a concept known as **He** initialization. Also, the same goes for the **W2** which initializes the weights between the hidden layer and the output layer. Thus, **b1** and **b2** are initialized as zero matrices.

The **input_size** and **output_size** is fixed to be 3 because input is two principal components and includes a bias term, and should match the number of classes in a classification task (There are 3 fruits: **Carambula**, **Lychee**, **Pear**). Finally, the **hidden_size** is number of neurons in the hidden layer but after testing in training data, I found that it is best to set it to 20 (will be explained below). The python coding snippet for this part is shown in the **Figure 1.1.3.1 Weight and Biases Initialization**.

Part 1.1.4) Forward and Backward Pass Implementation

The core of the training process involves the forward pass, where inputs are processed through the network layers to produce outputs, and the backward pass, where gradients of the loss function with respect to each weight are calculated. These gradients are essential for updating the weights and minimizing the loss. The methods are explained in the following:

$$1.1.4.1) \textit{sigmoid}(z) = \frac{1}{1+e^{-z}}$$

Often used in binary classification problems and as the activation for the output layer in binary classifiers.

$$1.1.4.2) \textit{ReLu}(z) = \max(0, z)$$

Helps with the vanishing gradient problem and allows for faster computation.

$$1.1.4.3) \textit{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^j}$$

Used for multi-class classification problems to normalize the output of a network to a probability distribution over predicted output classes.

1.1.4.4) Forward Pass:

This is one of the most important parts of the machine learning model because it is where the neural network makes predictions based on the current knowledge of weights and biases. There are also couple steps in this process:

Input to Hidden Layer: The data (like an image turned into numbers) enters the network. It first meets the hidden layer. The network uses the input data, multiplies it by the weights (which are initially set randomly), and adds biases (which help adjust the output along with the weights). The purpose of weights and biases is to fine-tune the input data into something the network can use to make a decision.

$Z1 = W1 * X^T + b1$ where it computes the hidden layer activation Z1 by linear combination of inputs and weight plus the bias.

Activation Function (ReLU): Once the network has the initial processed data (from the multiplication and addition above), it passes this data through the ReLU function. ReLU (Rectified Linear Unit) is a simple mechanism that looks at the data: if a number is negative, it turns it to zero; if it's positive, it keeps it as is. This step introduces non-linearity, helping the network understand complex patterns.

$A1 = ReLu(Z1)$ where it is non-linear transformation of Z1

Hidden Layer to Output: Next, the processed data from the hidden layer moves to the output layer, undergoing another round of multiplication with weights and addition of biases.

$Z2 = W2 * A1 + b2$ where it computes the outer layer input Z2 by linear combinations using the hidden layer outputs.

Activation Function (Softmax): At the output layer, we use the softmax function, which is a special function used when we want to classify data into categories. Softmax takes the numbers from the previous step and converts them into probabilities (numbers between 0 and 1 that add up to 1). Each number represents the probability of the input being in a particular class.

$A2 = softmax(Z2)$ where it computes the probability distribution over classes for classification.

1.1.4.5) Backward Pass:

The backward pass is how the network learns from its mistakes. After guessing, the network looks at its answer and compares with the correct ones and update weight and bias from it. Therefore it can make better guesses in the future. This part of the code also takes couple of steps:

Calculate Error: We start by measuring how wrong the network's predictions were using

a method called the "cross-entropy loss." This step quantifies the error in terms of probability: how far off the network's probability guesses were from the actual outcomes.

Backpropagate the Error: The core idea here is to take the error and see how each weight and bias contributed to it. By understanding their impact, we can tweak them to reduce the error. To understand, I try to employ the following formulas:

Gradient of loss with respect to weight W and biases b between the output layer and the last hidden layer \Rightarrow

$$\frac{dL}{dW^{[2]}} = \frac{1}{m} A^{[1]T} (A^{[2]} - Y)$$

$$\frac{dL}{dW^{[2]}} = \frac{1}{m} \sum (A^{[2]} - Y)$$

Gradient of loss with respect to the activations from the previous $A^{[1]} \Rightarrow$

$$\frac{dL}{dA^{[1]}} = (A^{[2]} - Y) W^{[2T]}$$

- $A^{[1]}$ - activations from last hidden layer
- $A^{[2]}$ - output predictions after softmax function
- Y - matrix of true labels which is the one-hot encode

Update Weights and Biases: We adjust the weights and biases to reduce the error. This adjustment is done using a method called "gradient descent." Think of it as trying to walk down a hill blindfolded (the hill is the error, and the bottom is the best performance); with each step, you feel the ground to see if you're going downhill and adjust your direction accordingly.

$$W = W - \alpha \frac{dL}{dW}$$

$$b = b - \alpha \frac{dL}{db} \quad (\alpha \text{ is the learning rate})$$

Repeat: This process of making guesses (forward pass), measuring how wrong those guesses were, and learning from the errors (backward pass) repeats over many cycles (epochs). With each cycle, the network gets better at making predictions.

Part 1.1.5) Model Accuracy Evaluation

After all of the functions are done, we are ready to train our model. However, I needed a function to compute the accuracy of the model. Therefore, I created the function `compute_accuracy` function. It is just (number of correct predictions) divided by (total number of predictions). Using `np.argmax`, the function converts both the predicted probabilities (`y_pred`) and actual labels (`y_true`) from one-hot encoded form to class indices. The python code snippet can be seen in the **Figure 1.1.5.1 Compute Accuracy Function Snippet**.

```
def compute_accuracy(y_true, y_pred):  
    """  
    Compute the accuracy of the predictions.  
  
    y_true: true labels, one-hot encoded, shape (n_classes, n_samples)  
    y_pred: predicted probabilities, shape (n_classes, n_samples)  
  
    Returns:  
    accuracy: The proportion of correctly classified samples.  
    """  
    y_true_labels = np.argmax(y_true, axis=0)  
    y_pred_labels = np.argmax(y_pred, axis=0)  
    accuracy = np.mean(y_true_labels == y_pred_labels)  
  
    return accuracy
```

(Figure 1.1.5.1 Compute Accuracy Function Snippet)


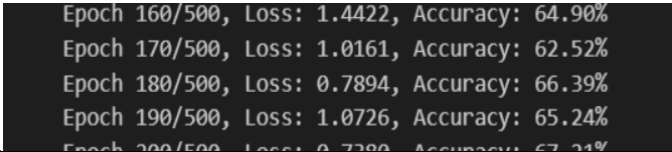
Finally, we can build our training model. It is where the neural network learns from the data by repeatedly applying the forward pass, calculating the loss, performing the backward pass, and updating the weights. Each interval is recorded/printed for testing validation purposes and in the end the final true accuracy is calculated.

Part 1.2) Test Accuracy

For this part of the report, I will focus on the different variable numbers of hidden_size, epochs, and learning rate to get decent results from the model and final test data accuracy (Since input and output size is constant). My table of testing can be seen in the **Table 1.2.1 Testing Training Model** and verifications of Training model accuracy can be seen in the **Table 1.2.2 Training accuracy verifications**.

| Number | hidden_size | epochs | Learning rate | Training Accuracy (%) |
|--------|-------------|--------|---------------|-----------------------|
| 1 | 5 | 1000 | 0.01 | 57.55 |
| 2 | 20 | 500 | 0.001 | 69.46 |
| 3 | 30 | 1500 | 0.001 | 70.61 |

(Table 1.2.1 Testing Training Model)

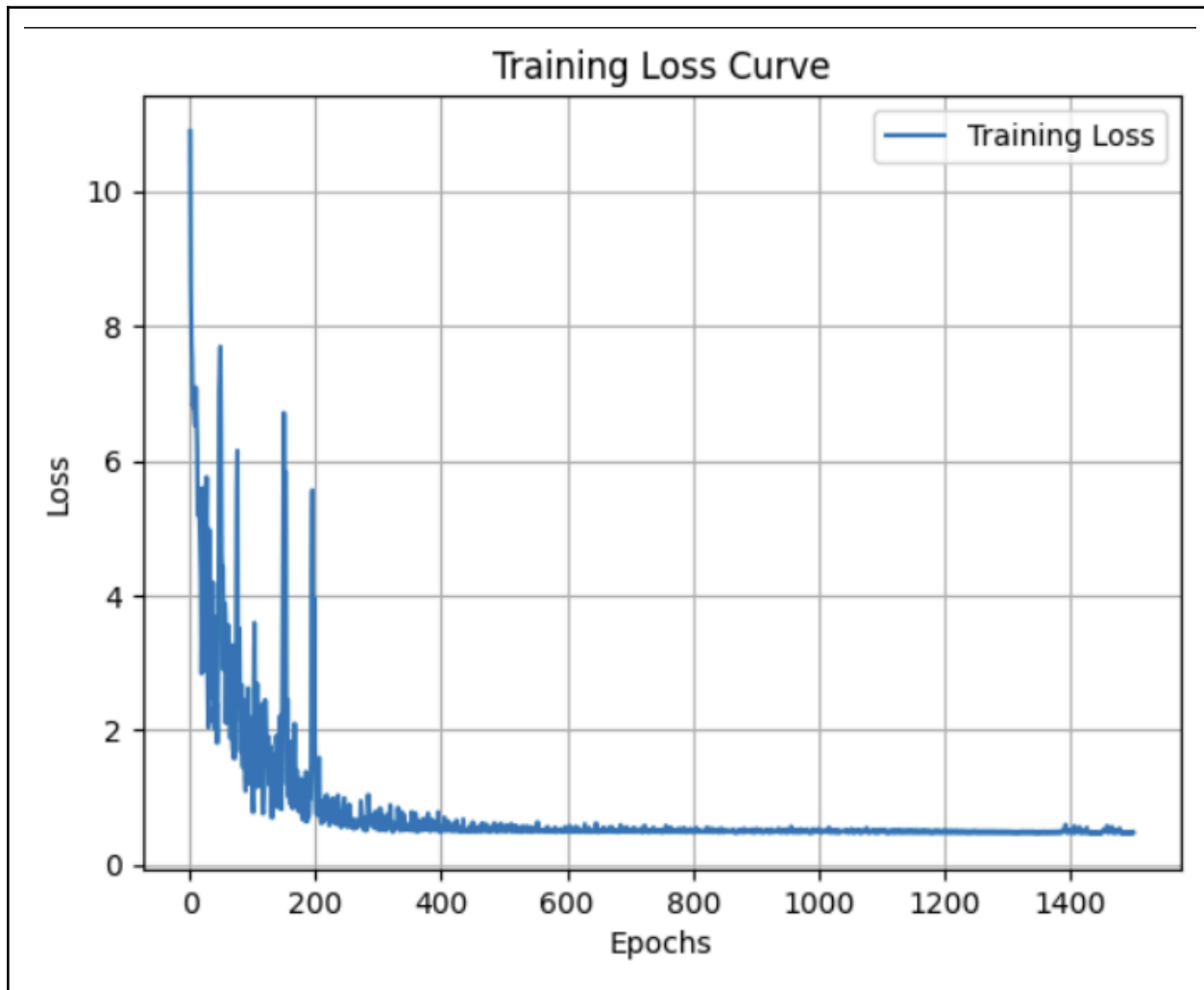
| Number | Picture of Training Accuracy |
|--------|---|
| 1 |  <pre> ... Epoch 1/1000, Loss: 1.2431, Accuracy: 23.33% Epoch 100/1000, Loss: 1.0489, Accuracy: 37.62% Epoch 200/1000, Loss: 0.9899, Accuracy: 48.91% Epoch 300/1000, Loss: 0.9563, Accuracy: 49.18% Epoch 400/1000, Loss: 0.9313, Accuracy: 57.01% Epoch 500/1000, Loss: 0.9103, Accuracy: 57.01% Epoch 600/1000, Loss: 0.8923, Accuracy: 57.01% Epoch 700/1000, Loss: 0.8768, Accuracy: 57.01% Epoch 800/1000, Loss: 0.8633, Accuracy: 57.01% Epoch 900/1000, Loss: 0.8515, Accuracy: 57.01% Epoch 1000/1000, Loss: 0.8411, Accuracy: 57.55% Final Training Accuracy: 57.55% </pre> |
| 2 |  <pre> Epoch 160/500, Loss: 1.4422, Accuracy: 64.90% Epoch 170/500, Loss: 1.0161, Accuracy: 62.52% Epoch 180/500, Loss: 0.7894, Accuracy: 66.39% Epoch 190/500, Loss: 1.0726, Accuracy: 65.24% Epoch 200/500, Loss: 0.7380, Accuracy: 67.31% </pre> |

| | |
|---|---|
| 3 | <pre> Epoch 160/1500, Loss: 0.9579, Accuracy: 71.90% Epoch 170/1500, Loss: 1.4097, Accuracy: 68.10% Epoch 180/1500, Loss: 0.6903, Accuracy: 70.75% Epoch 190/1500, Loss: 0.7279, Accuracy: 71.70% Epoch 200/1500, Loss: 1.1245, Accuracy: 71.02% Epoch 210/1500, Loss: 0.6155, Accuracy: 71.77% Epoch 220/1500, Loss: 0.9889, Accuracy: 66.46% Epoch 230/1500, Loss: 0.6213, Accuracy: 71.77% Epoch 240/1500, Loss: 0.7807, Accuracy: 70.34% ... Epoch 1480/1500, Loss: 0.4735, Accuracy: 78.16% Epoch 1490/1500, Loss: 0.4704, Accuracy: 76.33% Epoch 1500/1500, Loss: 0.4795, Accuracy: 79.93% Final Training Accuracy: 70.61% </pre> |
|---|---|

(Table 1.2.2 Training Accuracy Verifications)

Despite my best efforts of making the algorithm to work and coding it in python, I could only achieve high of 70.61 in training. Notably, a neural network configuration with 5 hidden neurons, trained over 1000 epochs at a learning rate of 0.01, resulted in a training accuracy of 57.55%. By increasing the hidden size to 30 and extending the training to 1500 epochs, while keeping the learning rate at 0.001, the accuracy improved to 70.61%. This suggests that an optimal hidden size may be around 20 with a learning rate of 0.001. Although increasing the number of epochs leads to better outcomes, further improvements in the algorithm are necessary for the long run or to have an accuracy average higher than 70%.

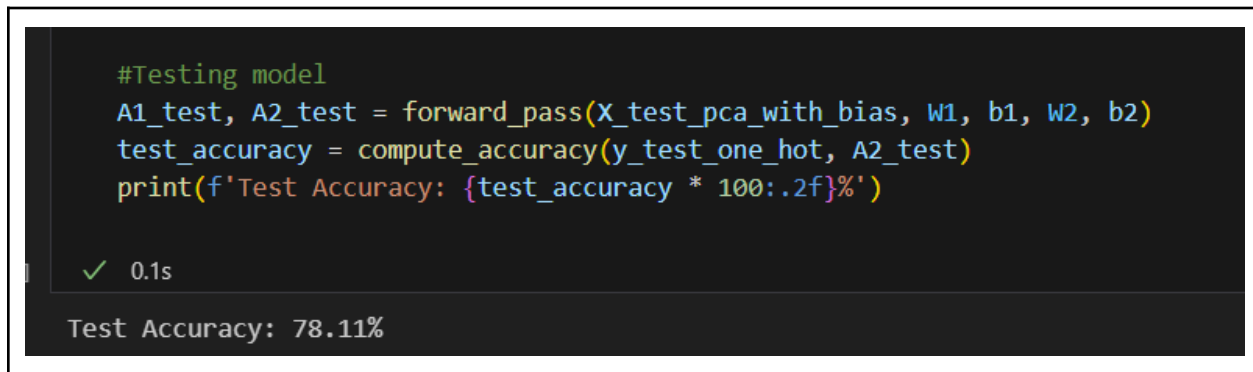
Part 1.3) Plotting Loss Curves



(Figure 1.3.1 Training Loss Curve)

The training loss graph shows that the loss initially drops steeply, illustrating learning effectiveness, but early oscillations could be caused by high learning rates or weight initialization. The loss decreases and then flattens at a very low level after 400 epochs (**Figure 1.3.1 Training Loss Curve**). Any further increase in epochs will not add new information to the loss function, and it might be near the limit of the model's abilities to learn new information from the dataset or at the edge of overfitting. So maybe the perfect number of epochs for this model that I have created might be around 600 to 800 where 1500 is not optimal.

For testing the final test data, I use the same method as previously mentioned in the Part 1.1.1) Data Preprocessing. The result is 78.11% as shown in the below **Figure 1.3.2 Test Data Accuracy**. The test accuracy was notably higher than the training accuracy of 70%. This suggests that the model is generalizing well to new data and capturing new patterns of the dataset well.



```
#Testing model
A1_test, A2_test = forward_pass(X_test_pca_with_bias, w1, b1, w2, b2)
test_accuracy = compute_accuracy(y_test_one_hot, A2_test)
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')
```

✓ 0.1s

Test Accuracy: 78.11%

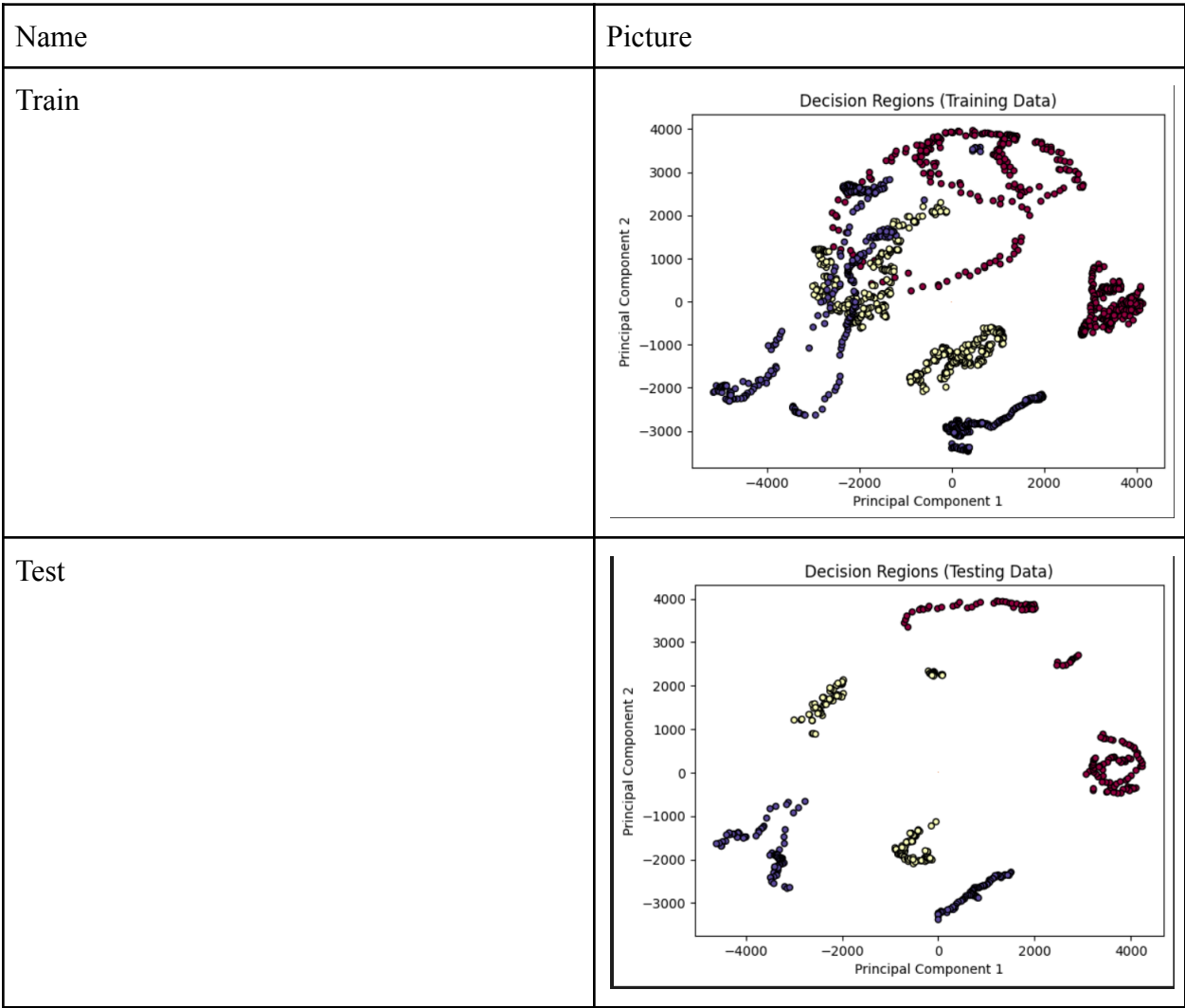
(Figure 1.3.2 Test Data Accuracy)

Part 1.4) Plotting Decision Regions for Training and Testing Data

Training Data Plot: The training data also recognized the proximity of clusters where the data points are grouped together as belonging to the same class especially pointed out with the blue and pink classes. The yellow class seems to have a little interconnection with blue, which could mean that probably the model is sometimes not very sure whether an image belongs to yellow or blue class. The decision boundaries for each of the classes seem more twisted and smooth, which signifies that the model has been learning intricate patterns from the training data; this could be a norm if the training data is representative of the test data; otherwise, it might be overfitting.

Testing Data Plot: Loosely packed spread out clusters area observed in the testing data as compared to the tightly packed and populated clusters on the training data. This spread can tell about potential of the model for generalization for new unseen data. It is also noteworthy that the clusters are still disjointed further asserting that learning and generalization is well occurring. The discrimination between classes in the testing data is fairly clean, but there are some samples from different given classes which lie closer to a different cluster, which is also seen in real life

performances where all data points are not equally distinct from each other. The graphs can be seen in **Figure 1.4.1 Decision Regions for Training and Testing Data.**



(Figure 1.4.1 Decision Regions for Training and Testing Data)

While the training plot reveals much compact clusters which might be caused by model training on this data, the testing plot still provides neat separations, meaning that the model has not simply over-fitted the training data but, has learnt to generalize well. The only overlap between the two clusters, the yellow and blue ones, is that they have slightly more entries in the training set than in the testing set, this might be an avenue that could be optimized, might be through adjusting the hyperparameters or by increasing the complexity of the model if the current capabilities allow for it. These plots are essential to represent the results of the experiments and,

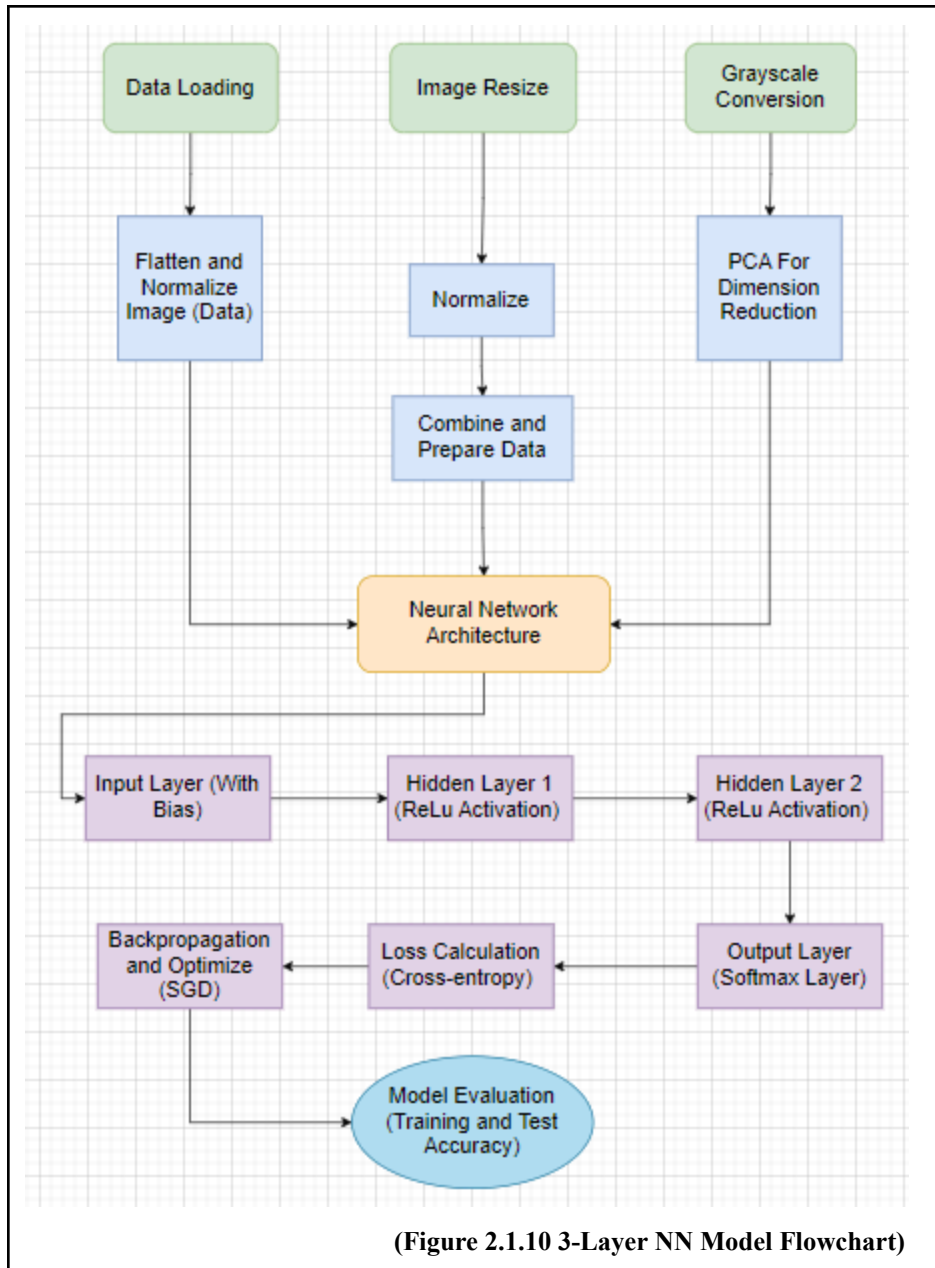
particularly, to demonstrate how the dimensionality reduction is performed by PCA, and how the neural network categorizes the data.

Part 2.1.0) 3 Layer Network Model Architecture Details

The Part 2 of this report will focus on the 3 Layer Neural Network Model. Since the 2-Layer model and the 3-Layer model is similar in some ways, I will only focus on the distinct parts about the 3-Layer Neural Network Model. As we can see in the **Figure 2.1.10 3-Layer NN Model Flowchart** below, the architecture is similar. Except for the part where we add another layer to the model whereas it is labeled as “**Hidden Layer 2**”.

The parts that will skip for the sake of the report being too repetitive and they are Data Preprocessing, Initialize weights and biases (just add another W_3 and b_3), one-hot encoding, softmax function, forward_pass (adjust to have W_3 and b_3), training process (but will have detailed explanation about the accuracy), cross entropy loss and cross entropy derivative functions, and computing accuracy function as they are mentioned above in the **Part 1** of this report.

Since it was 3-layered Network I had to modify the **ReLU**, **backward_pass**, **Plotting Decision Regions** for Training and Testing Data functions. The main reason for using a three-layer neural network is to improve its ability to capture and encode more elaborate structure and hierarchy to the data it is given. Introducing more and deeper layers may help capture more details and subtle features of input data which in turn can benefit on the performance on unseen data. This architecture, therefore is designed, to be moderately complex in order to accommodate more complicated classification problems with increased dimensionality of input data.



Part 2.1.1) Relu Function

Instead of the using the same **ReLU** function as we used in **2-Layer NN (Part 1.1.4)**, I have decided to use a better function called **Leaky ReLU** function. Because when I was testing out the model on the training data, it had come to my attention that no matter what I do or implement I could not get the model to achieve above 33.33% accuracy. After reading online sources, I found out that “The traditional ReLU activation function outputs zero for any negative

input and its derivative is zero for negative inputs”. Where this can lead to an “dying ReLu” problem. The Leaky ReLu solves and often preferred in deeper networks due to its ability to maintain active neurons through non-zero gradients for negative values.

| Name | Coding Snippet |
|--------------------------|--|
| Leaky ReLu | <pre>def leaky_relu_3(z, alpha=0.01): return np.where(z > 0, z, alpha * z)</pre> |
| Derivative of Leaky ReLu | <pre>def leaky_relu_derivative_3(a, alpha=0.01): return np.where(a > 0, 1, alpha)</pre> |

(Table 2.1.1.1 ReLu implementation Snippet)

As seen the above Table 2.1.1.1 ReLu implementation Snippet, α is a small coefficient that gives a small negative slope when z is less than zero. The formulas that I used can be seen in the **Table 2.1.1.1 ReLu implementation Snippet**. However, I won't go into detail about the Leaky ReLu (More information can be found from the link in the References).

| Name | Formula |
|--------------------------|--|
| Leaky ReLu | $\phi(x) = \begin{cases} \alpha x & x \leq 0 \\ x & x > 0 \end{cases}$ |
| Derivative of Leaky ReLu | $\phi'(x) = \begin{cases} \alpha & x \leq 0 \\ 1 & x > 0 \end{cases}$ |

(Table 2.1.1.2 ReLu implementation Snippet)

Part 2.1.2) Backward Pass Function

Firstly, I tried the same method as the 2-Layer Neural Network method (mentioned in **Part 1.1.4** of the report) with the added implementation of the Leaky **ReLU** function. However, the highest training accuracy that I could get was in the 60% range. Therefore, I searched for another method or a way to make the algorithm perform better because 3-layer Neural Network perform worse than the 2-Layer Neural Network is unacceptable.

After exploring the web for another method or a way to make the algorithm perform better I came across the “regularization_rate” variable. Regularization is a technique used to reduce the model's overfitting on the training data. It works by adding a penalty term to the loss function used to train the model. The most common forms are L1 and L2 regularization. Adds a penalty equivalent to the square of the magnitude of the coefficients. In the function, this is applied as $(\text{regularization_rate} * W_{x_3} / m)$. This term is added to the gradients of the weights, effectively reducing the weight values during the training, which can lead to a more generalized model that performs better on unseen data. The python coding snippet for the backward pass can be seen in the below **Figure 2.1.2.1 Backward Pass for 3NN Coding Snippet**.

```
def backward_pass_3(X, y, A1_3, A2_3, A3_3, W1_3, b1_3, W2_3, b2_3, W3_3, b3_3, learning_rate=0.001, regularization_rate=0.001):
    m = X.shape[0]

    dz3_3 = A3_3 - y
    dw3_3 = (np.dot(dz3_3, A2_3.T) / m) + (regularization_rate * W3_3 / m)
    db3_3 = np.sum(dz3_3, axis=1, keepdims=True) / m

    dA2_3 = np.dot(W3_3.T, dz3_3)
    dz2_3 = dA2_3 * leaky_relu_derivative_3(A2_3)
    dw2_3 = (np.dot(dz2_3, A1_3.T) / m) + (regularization_rate * W2_3 / m)
    db2_3 = np.sum(dz2_3, axis=1, keepdims=True) / m
```

(Figure 2.1.2.1 Backward Pass for 3NN Coding Snippet)

Part 2.2) Test Accuracy

For test accuracy, the hidden_layer 1 and 2 did not have much impact as I tested out multiple combinations and found that hidden_size 1 at 20 and hidden_size 2 at 10 and learning rate at 0.0001 is at best. Because when increasing the hidden_size 1 or 2, it just leads to extra computational problem for the model. Thus, if the learning rate is higher than 0.0001, the final

training accuracy is not optimal. With the help of the Leaky ReLu and updated backward_pass functions, I was able to achieve Final training accuracy of 75.99 as seen in the Figure 2.2.1 Final Training Accuracy.

```
Epoch 1/1500, Loss: 12.3177, Accuracy: 33.13%
Epoch 10/1500, Loss: 6.8575, Accuracy: 61.90%
Epoch 20/1500, Loss: 5.9502, Accuracy: 66.80%
Epoch 30/1500, Loss: 6.6517, Accuracy: 63.67%
Epoch 40/1500, Loss: 6.1381, Accuracy: 65.92%
Epoch 50/1500, Loss: 5.5496, Accuracy: 69.25%
Epoch 60/1500, Loss: 5.5240, Accuracy: 65.37%
Epoch 70/1500, Loss: 5.5750, Accuracy: 65.17%
Epoch 80/1500, Loss: 6.1419, Accuracy: 63.61%
Epoch 90/1500, Loss: 4.4534, Accuracy: 65.92%
Epoch 100/1500, Loss: 5.4933, Accuracy: 63.61%
Epoch 110/1500, Loss: 2.2373, Accuracy: 76.60%
Epoch 120/1500, Loss: 2.7516, Accuracy: 72.52%
Epoch 130/1500, Loss: 1.9304, Accuracy: 75.51%
Epoch 140/1500, Loss: 3.8241, Accuracy: 65.71%
Epoch 150/1500, Loss: 2.9416, Accuracy: 73.13%
Epoch 160/1500, Loss: 1.6295, Accuracy: 73.81%
Epoch 170/1500, Loss: 2.9217, Accuracy: 69.52%
Epoch 180/1500, Loss: 2.2955, Accuracy: 68.71%
Epoch 190/1500, Loss: 1.4224, Accuracy: 72.86%
Epoch 200/1500, Loss: 1.7603, Accuracy: 75.31%
Epoch 210/1500, Loss: 2.5714, Accuracy: 69.73%
Epoch 220/1500, Loss: 1.1147, Accuracy: 76.80%
Epoch 230/1500, Loss: 2.1261, Accuracy: 74.22%
Epoch 240/1500, Loss: 1.3650, Accuracy: 72.72%
...
Epoch 1480/1500, Loss: 0.9527, Accuracy: 73.88%
Epoch 1490/1500, Loss: 0.8825, Accuracy: 74.97%
Epoch 1500/1500, Loss: 0.9660, Accuracy: 73.33%
Final Training Accuracy: 75.99%
```

(Figure 2.2.1 Final Training Accuracy)

If we were to look closely at the first epoch which is 1/1500, the accuracy is low whereas the loss is high. Then, we can also see that at 10/1500 epoch, the model learns extremely well where it jumps from 33.13% accuracy to 61.90% accuracy.

Lastly, for the testing the model on the test data, the model relatively performs well on a high of 81.93% compared to the 2-Layer Neural Network which was 70% (The method for computing accuracy is same as 2-Layer NN). The **Figure 2.2.2 Test accuracy verification snippet** can be seen below.

```
Testing 3 Layer NN

# Define paths to the test data
test_data_path = "Data_test"

# Load the test images
X_test_3, y_test_3 = load_images(test_data_path, fruit_classes)

# Same as above but with 3-Layer NN

X_test_pca_3 = pca_3.transform(X_test_3)

X_test_pca_with_bias_3 = np.hstack((X_test_pca_3, np.ones((X_test_pca_3.shape[0], 1))))

y_test_one_hot_3 = np.zeros((output_size_3, len(y_test_3)))
y_test_one_hot_3[y_test_3, np.arange(len(y_test_3))] = 1

#Final Test Accuracy
A1_test, A2_test, A3_test = forward_pass_3(X_test_pca_with_bias_3, W1_test, W2_test, W3_test)
test_accuracy = compute_accuracy(y_test_one_hot_3, A3_test)
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')

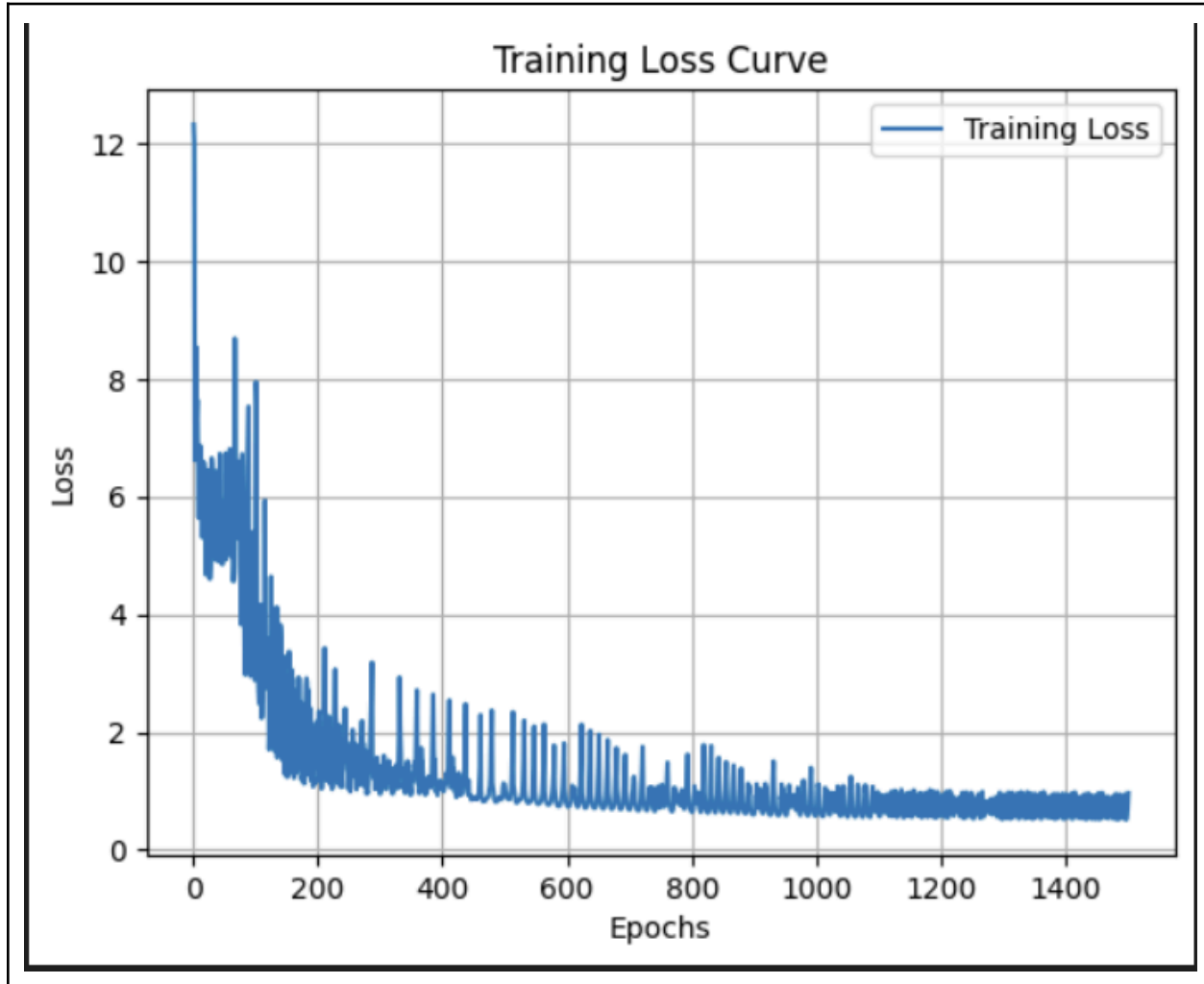
[18] ✓ 0.1s

... Test Accuracy: 81.93%
```

(Figure 2.2.2 Test accuracy verification snippet)

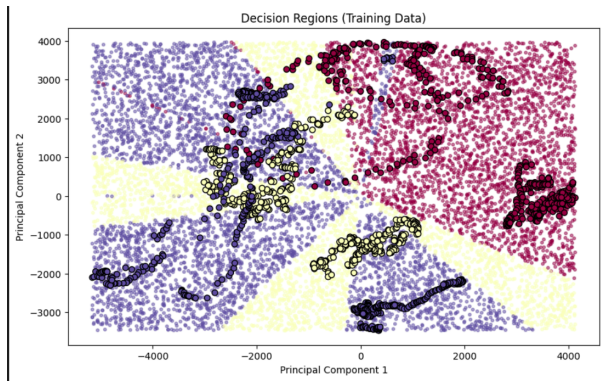
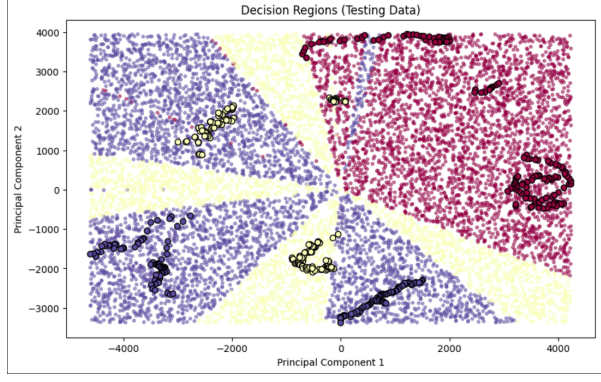
Part 2.3) Plotting Loss Curves

From the Figure 2.3.1 Loss Curve For 3-Layer NN, we see that early sharp spikes and quick drops are typical in neural network training, particularly when starting with random weights. We can tell that the model is making large corrections to its weight and bias from this. As the epochs increase beyond approximately 400, the loss begins to stabilize and reaches a plateau. So initializing the epochs to 1500 may not be a good idea.



(Figure 2.3.1 Loss Curve For 3-Layer NN)

Part 2.4) Plotting Decision Regions for Training and Testing Data

| Name | Picture |
|-------|--|
| Train |  |
| Test |  |

(Figure 2.4.1 Decision Regions for 3NN Training and Testing Data)

Training Data Plot: The training data shows tightly grouped clusters of data points, which are the actual labeled instances used during model training. These clusters have clear boundaries between them, with very distinct separation areas colored differently to show how the model has learned to classify these regions. There is noticeable density in the distribution of points, indicating robust learning from dense regions of the data. Some minor overlaps or misclassifications can be observed (especially around the edges of clusters).

Testing Data Plot: The testing data decision regions show how the model performs on unseen data. The regions are still distinct, but the data points (representing test instances) are more dispersed compared to the training data. This dispersion reflects the model's ability to generalize beyond the exact scenarios it was trained on. There appears to be slightly more overlap between different classes in the testing data, especially between clusters close to each other. The graphs can be seen in **Figure 2.4.1 Decision Regions for 3NN Training and Testing Data.**

Part 3) Comparing 2 and 3-Layer Neural Network Performance

Based on the training and testing datasets, there are significant differences in the methods used by the 2-Layer and 3-Layer Neural Networks, as evident from the performance metrics concluded above. From the architecture analysis of the models, the 2-Layer Neural Network model succeeded in obtaining a training accuracy of 70.61 % and a testing accuracy of 78.11% (Attached is the Figure 3.1 Table of Performance). This demonstrates a somewhat good model that is not overly complex as testified by the higher accuracy of the model in the new unseen data in the test set as compared to the training set. This kind of situation enables the inference that although the model might be imperfect in modeling the training data or a portion of it (underfitting), it optimizes the aspect of generalizing better on the unseen data while avoiding overfitting to the training samples.

| Data Name | 2-Layer NN Accuracy % | 3- Layer NN Accuracy % |
|------------|-----------------------|------------------------|
| Train Data | 70.61 | 75.99 |
| Test Data | 78.11 | 81.93 |

(Figure 3.1 Table of Performance)

On the other hand, the 3-Layer Neural Network show higher accuracy rate in training phase is 75.99% and testing phase is 81.93% . The enhanced training accuracy compared to the 2-Layer model indicates that an extra layer enhances the model's ability to analyze the components of the training data set more thoroughly and avoid the underfitting problem. Most especially, the raise in the testing accuracy implies that the model does not only learn more

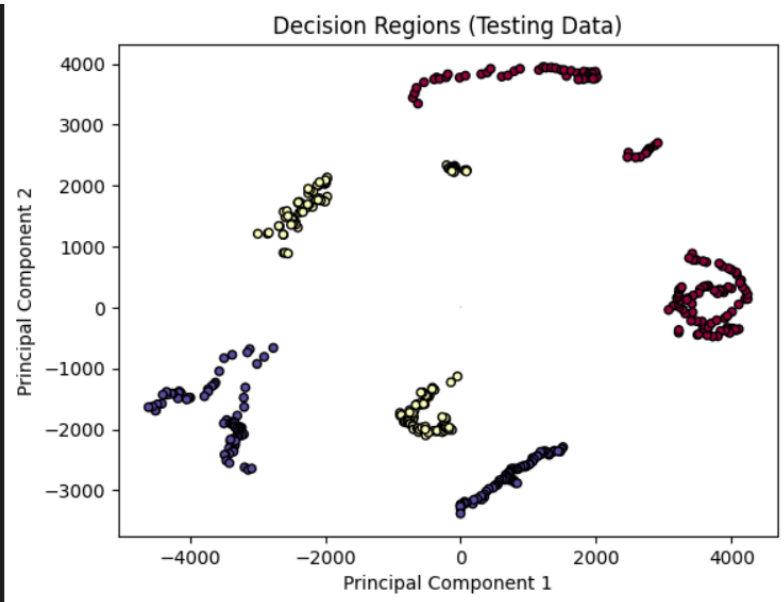
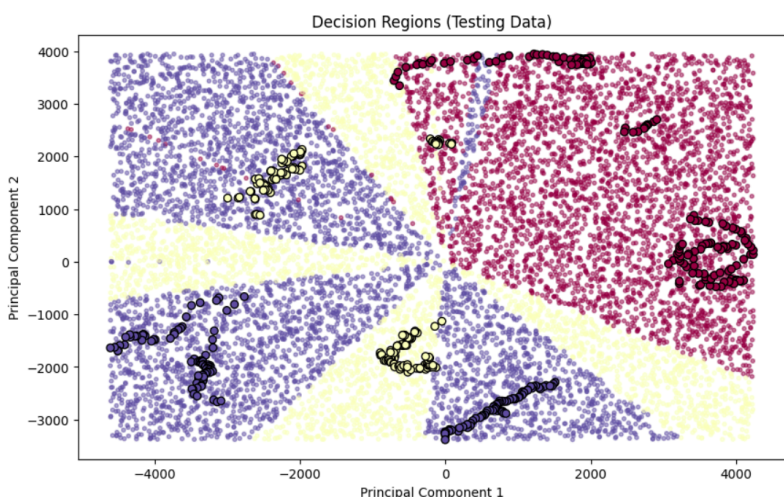
optimally, but also tests better on new data. This improvement in performance underlines the merit of extending the depth of NNs as required in complicated classification problems so as to allow for the identification of more complex structures within the data and as a result improving real application and predictive reliability.

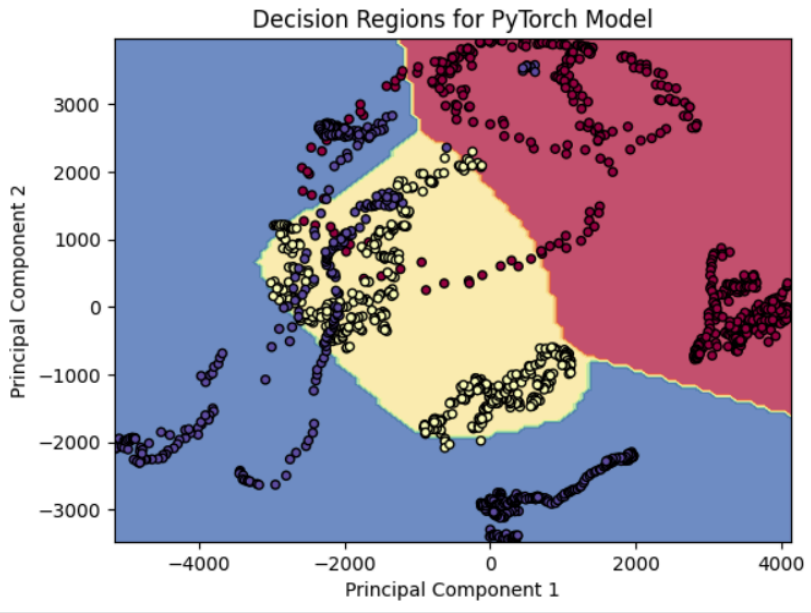
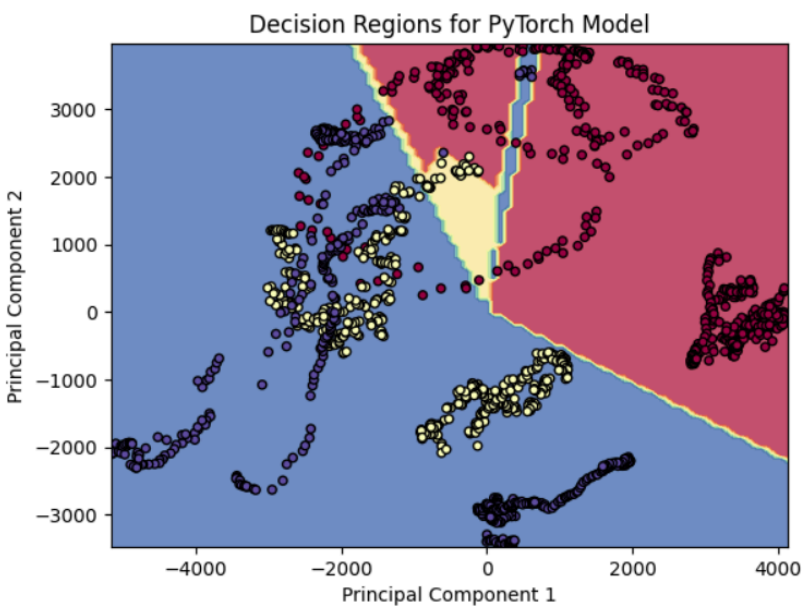
Overall, it is also noted that the observations made from the decision region plots supplement the quantitative results and ensure that the models are effective or could be further improved in specific directions. The separation and clarity realised in 3-Layer model's decision regions especially contribute to overall accuracy, which rises in both training and testing phases, underlining well-optimised model that is not oversophisticated and yet does not lag behind.

Part 4) Try Other Regularization and Comparing Performance

Due to the fact that I implemented my own regularization (L2) in the 3-Layer Neural Network in the previous parts, I tried employing the dropout technique on both testing data of the 2 and 3-Layer Neural Network. In the below Table 4.1, we can see the results and performance.

Now if we look at the table below where it is 2 and 3-Layer Neural Network without using the machine learning library and the dropout technique decision regions vs the 2 and 3-Layer Neural Network with the machine learning library and the dropout technique decision regions (**Table 4.1 Final Performance Table 1**). For extra information, I have added their test data accuracy.

| Name | Test accuracy | Decision Regions |
|------------|---------------|---|
| 2-Layer NN | 70.61 |  |
| 3-Layer NN | 81.93 |  |

| | | |
|-------------------------|--------|--|
| 2-Layer NN with dropout | 80.72 |  <p>The plot shows three distinct decision regions: a blue region on the left, a yellow region in the center, and a red region on the right. Data points are represented by small circles, with black outlines for training data and colored fills for test data. The axes are labeled 'Principal Component 1' (x-axis, ranging from -4000 to 4000) and 'Principal Component 2' (y-axis, ranging from -3000 to 3000). The regions are well-separated, indicating good generalization.</p> |
| 3-Layer NN with dropout | 84.94% |  <p>The plot shows three distinct decision regions: a blue region on the left, a yellow region in the center, and a red region on the right. Data points are represented by small circles, with black outlines for training data and colored fills for test data. The axes are labeled 'Principal Component 1' (x-axis, ranging from -4000 to 4000) and 'Principal Component 2' (y-axis, ranging from -3000 to 3000). The regions are well-separated, indicating good generalization.</p> |

(Table 4.1 Final Performance Table 1)

Comparing the effect of dropout as a regularization technique on the decision regions of 2-layer and 3-layer of neural networks demonstrates an enhanced accuracy and improved models' ability for generalization. When adding dropout to the 2-layer neural network the test accuracy has been improved from 70. 61% to 80. 72%. The decision regions seem more clearly separated, which leads implicitly to a better generalization to the unseen data. Likewise in the

case of 3 layer neural network, the impact of dropout has led to the enhancement of test accuracy from 80. 72% to 84. 94%. When focusing on the decision regions, one can observe that these regions are more divided and there is less overlapping between the classes, meaning that the overfitting has been effectively reduced.

In the decision regions of the testing data, it can also be observed that the improved separation between the classes is most evident after applying dropout. For instance, in the 3-layer neural network the regions are totally different and expand to more different parts of the feature space proving more general view of the difference in features between the classes. This broader separation aids in the reduction of the misclassification rates especially at the regions where the model without dropout had high confusion. Also, the dropout seems to ratify the stabilization of the learning process as evidenced by the continuity changes in different regions which in effect provides greater reliability of the prediction made in the entire feature space.

In conclusion, dropout has given a positive impact in both models by not allowing overfitting, which from the plots, there are better and clearer decision boundaries. It has helped also to determine more proper parameters in the networks, the features that are not formed only by the noise in the training data. Therefore, if dropout is guided, the accuracy increases and in addition, they made it possible to predict the results' generalization in an actual situation.

Part 5) References

Cross entropy loss function:

- <https://medium.com/@vergotten/binary-cross-entropy-mathematical-insights-and-python-implementation-31e5a4df78f3>
- <https://www.datacamp.com/tutorial/the-cross-entropy-loss-function-in-machine-learning>

Weight and Biases Initialization

- <https://www.analyticsvidhya.com/blog/2021/05/how-to-initialize-weights-in-neural-networks/>
- <https://www.geeksforgeeks.org/kaiming-initialization-in-deep-learning/>

Backpropagation:

- <https://www.geeksforgeeks.org/backpropagation-in-neural-network/>

- <https://www.askpython.com/python/examples/backpropagation-in-python>
- <https://www.youtube.com/watch?v=7qYtIveJ6hU>
- <https://towardsdatascience.com/coding-neural-network-forward-propagation-and-backpropagation-ccf8cf369f76>

Leaky ReLu:

- <https://medium.com/@sakeshpusuluri/activation-functions-and-weight-initialization-in-deep-learning-ebc326e62a5c>

Regularization:

- <https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261>
- <https://neuraspikes.com/blog/l2-regularization-with-python/>

Plotting Decision Regions:

- <https://hackernoon.com/how-to-plot-a-decision-boundary-for-machine-learning-algorithms-in-python-3o1n3w07>
- <https://stackoverflow.com/questions/22294241/plotting-a-decision-boundary-separating-2-classes-using-matplotlibs-pyplot>

Dropout technique:

- <https://python-course.eu/machine-learning/dropout-neural-networks-in-python.php>
- <https://pypi.org/project/torch/>
- <https://pytorch.org/docs/stable/library.html>
- <https://www.youtube.com/watch?v=lcI8ukTUEbo>
- <https://www.youtube.com/watch?v=XmLY117DbbA>