Machine Learning

The implementation of a multilayered perceptron or MLP on a training and validation datasets THA2train.xlsx and THA2validate.xlsx. The input layer of two neurons, with two hidden layers containing 10 neurons each, and 1 output layer of 2 neurons.

A. MLP

1. Activation and loss function:

Here we considered number of activation functions such as following written in the code part.

```
#Activation functions
def sigmoid( x): # sigmoid function used at the hidden layer and output layer
    return 1 / (1 + np.exp(-x))
def tanh(x):     # hyperbolic tangent
    return np.tanh(x)
def relu(x):     #RELU
    return np.maximum(0, x)
def leaky_relu(x, alpha=0.01):   #Leaky RELU
    return np.where(x > 0, x, alpha * x)
def elu(x, alpha=1.0):    #exponential LU
    return np.where(x > 0, x, alpha * (np.exp(x) - 1))
```

From these we mainly focused on three activation functions

a. Sigmoid :- It is a S curve whose values lies between 0 and 1
b. Tanh or hyperbolic tangent activation function:- tanh is also like sigmoid but better. Range of tanh lies between (-1,1) and it is also S shaped.
c. ReLU(Rectified Linear Unit):- It is haf rectified from bottom f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.

From these for the hidden layers (ϕ(0) and ϕ(1)), we have used the ReLU activation functions, as it would give us good results for binary classification problem. For the output layer (ϕ(2)) use of the sigmoid function. It simplifies the values between 0 and 1, providing non-linearity to the model.

Loss function used is binary cross-entropy. This is the loss functions which is works best for binary classification models – where model takes in an input and has to classify it into two pre-set categories. Hence this best suits our MLP model.

$$CE\ Loss = \frac{1}{n}\sum_{i=1}^{N} -\left(y_i \cdot log(p_i) + (1 - y_i) \cdot log(1 - p_i)\right)$$

2. **Learning Rate, Batch size and Initialization:**

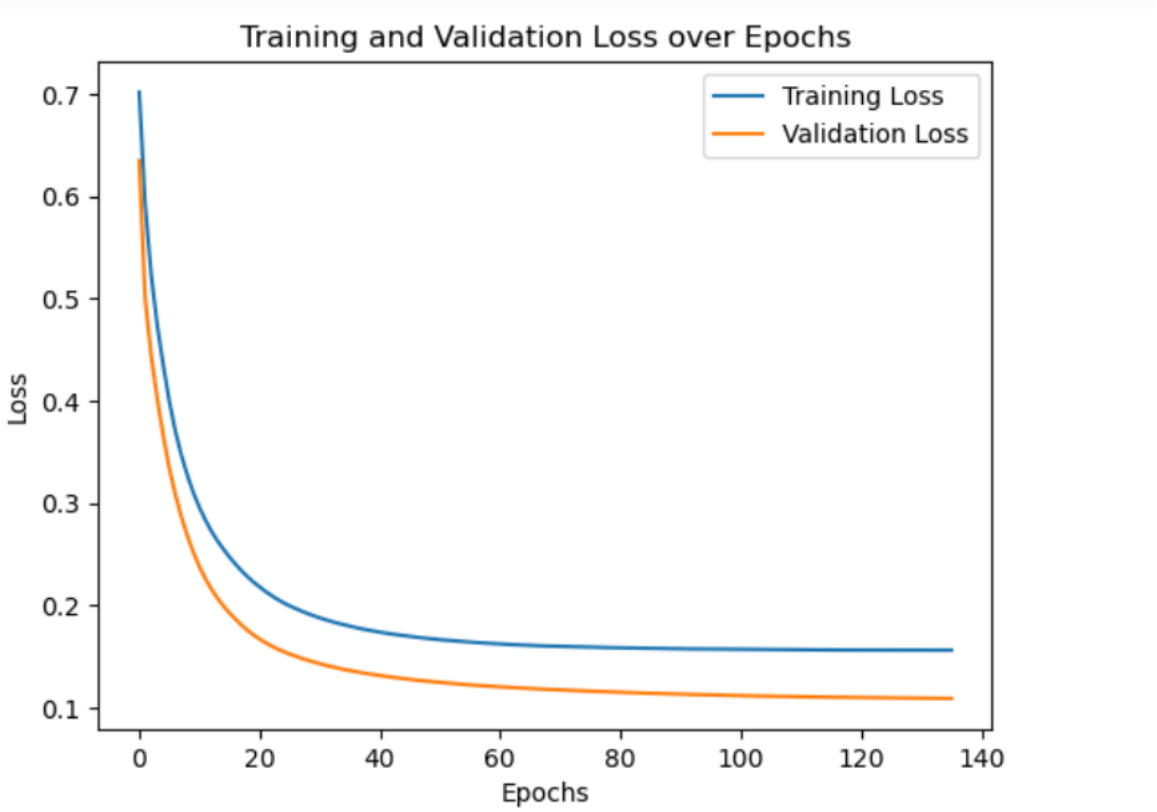
Initially we considered learning rate of 0.0001 which we increased later on to 0.001. This was selected based on cost and accuracy we received.

The initialized values for the weights were generated randomly from np.random.rand . This is used so that the weights can be randomly generated. Also we initialized all biases to zero which will be updated during the training process.

Initial batch size used for our code is 32, which provides good results with chosen learning rate.

3. **Training:**

After plotting training and validation loss over epochs we get the following curves.



This shows that loss for training as well as validation data is decreasing over epochs till the time it becomes more or less constant.

Stopping criteria used in our code is :

```
# Check for early stopping based on validation loss
if len(train_loss_history)>100 and train_loss > np.mean(train_loss_history[-10:]):
    print("Stopping early as validation loss is not improving.")
    break
```

Here we are checking if current calculated train loss is higher than previous 10 train loss and also if training loss history has already more than 100 elements in it, then model can stop. Since loss is not improving in this case, we are stopping the model to avoid overfitting.

The final accuracy on the validation set and the confusion matrix, these helps give a better view of the model's performance and see how it may skew the results. Our outputs for final accuracy and confusion matrix are:

```
Confusion Matrix:
[[80.  2.]
 [ 2. 80.]]

Final accuracy of the model is: 0.98
```

4. **Implementation**:

We have done implementation from scratch without any inbuilt libraries for multi layer perceptron. However we have used some auxiliary libraries such as numpy, matplotlib, and pandas, which are used for the operations, plots, and data. Few of the resources which we used as guidance are:

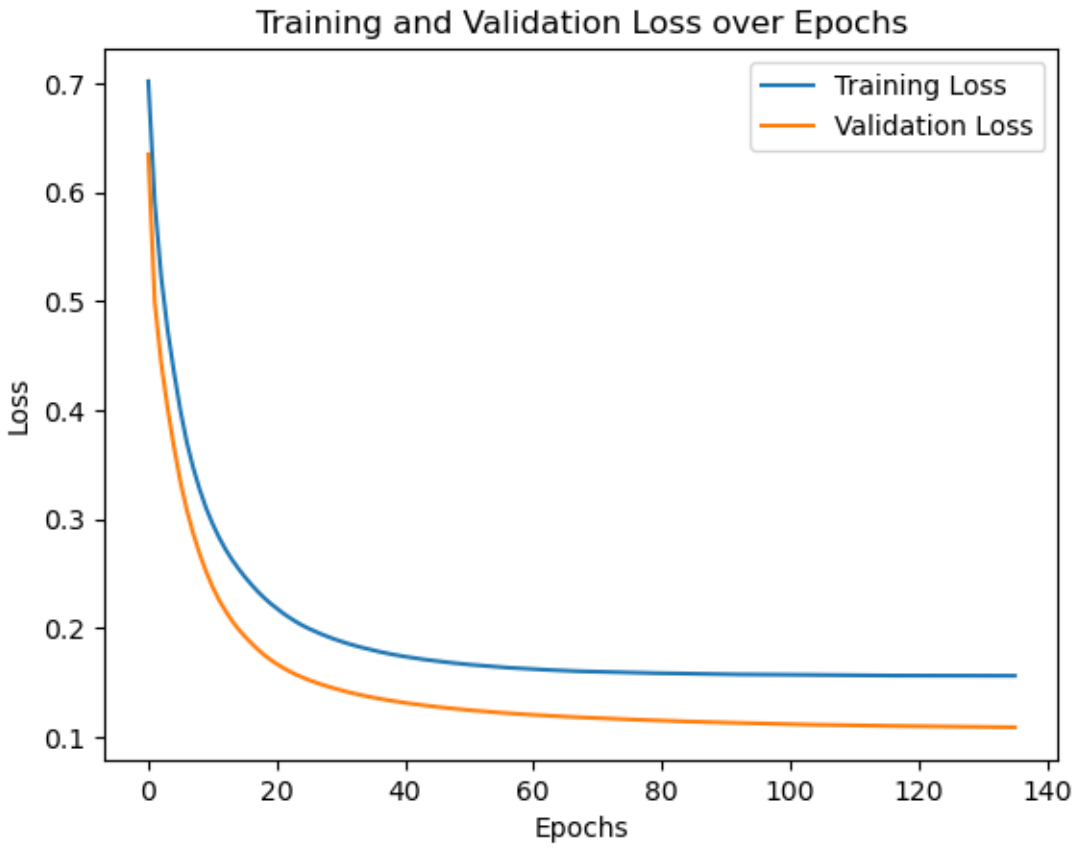
GitHub sources:
https://github.com/mitthoma/MLP-from-scratch/blob/master/MLP.ipynb
https://github.com/MaviccPRP/mlp_from_scratch/blob/master/mlp_np.py

B. Hyperparameters Optimization

1. Parameter Initialization:

As per given in the question, best performing model of part A is with parameters Learning rate – 0.001, Weights – randomly chosen, biases- initialized to zero initially, two hidden layers of 10 neurons each, activation functions used for hidden layers is ReLU and for output layer is Sigmoid.

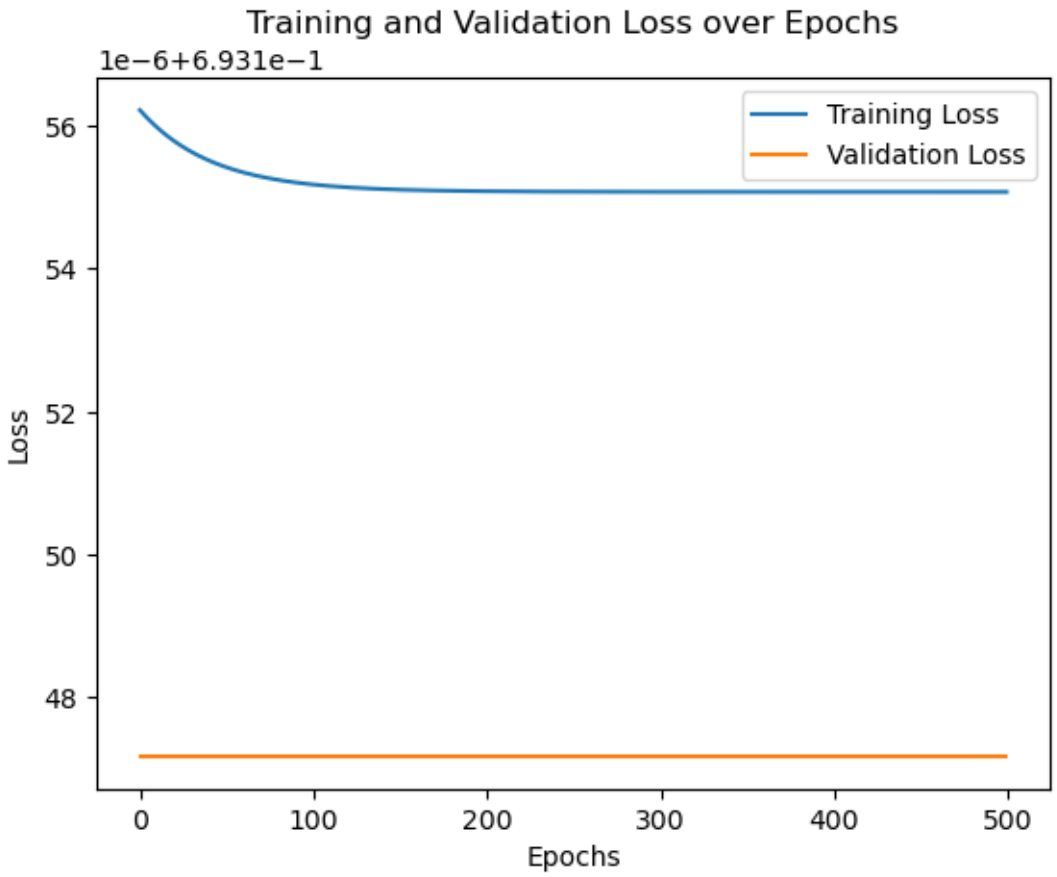
```
Confusion Matrix:
[[80.  2.]
 [ 2. 80.]]
```




```
Final accuracy of the model is: 0.98
```

Now when we initialize all weights and biases to zero and retrain it by keeping all other settings same we get results as follows:

```
Confusion Matrix:
[[ 0. 82.]
 [ 0. 82.]]
```




```
Final accuracy of the model is: 0.50
```

From this output we can see that , model did not perform well when weights were initialized to zero. Also accuracy reported is very low and confusion matrix returned by program shows that model is not behaving as expected.

After observing the results carefully, we see that model stopped learning altogether. Hence cost was never improved and hence accuracy is low. Also the graph gives straight lines for training loss and validation loss which means these losses are not improving over epochs. Whereas if we see performance of best model of part A, cost improved over number of epochs, which can be seen with help of graph. Confusion matrix also shows good results as 80% as true positive and 80% as true negative values. Also the final accuracy reported is high as 0.98

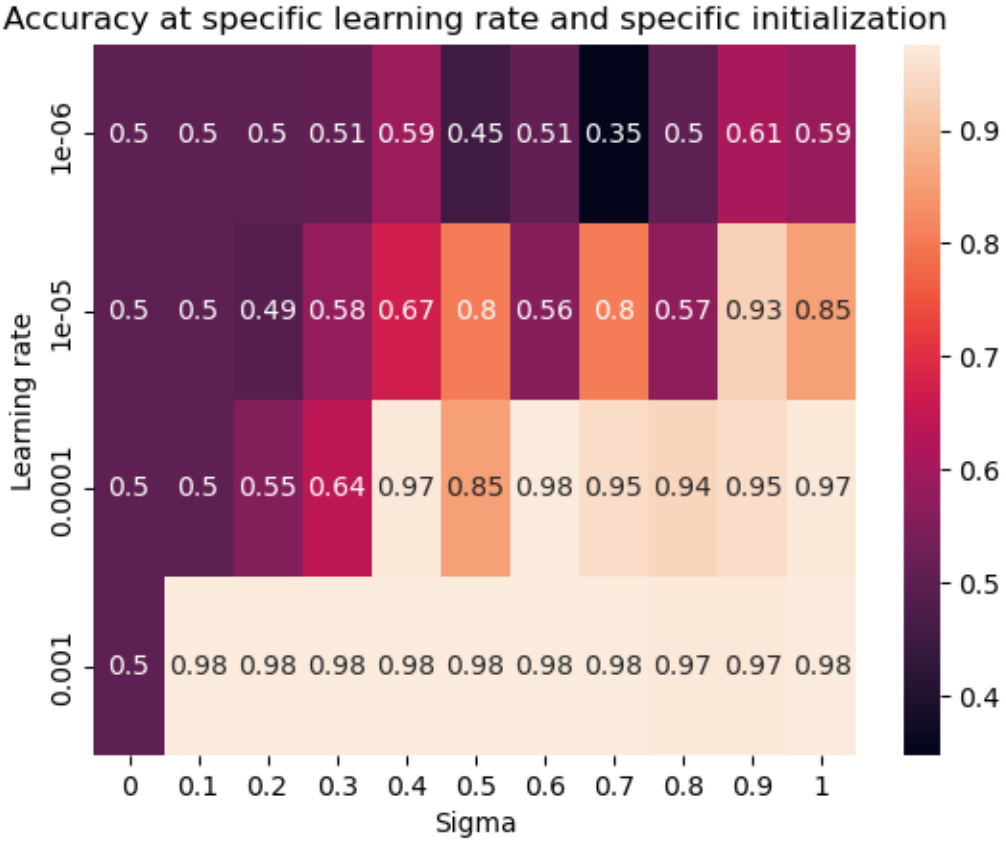
2. Learning Rate vs Parameter Initialization:

Here we took the best performing model of part A is with parameters Learning rate – 0.001, Weights – randomly chosen, biases- initialized to zero initially, two hidden layers of 10 neurons each, activation functions used for hidden layers is ReLU and for output layer is Sigmoid. Now only changes we made in this model were:

- Initialize connection weights and biases with sampling from a normal distribution N(0,σ2). σ2 can take values from {0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1}
- And take learning rate from 0.00001 and gradually increasing it. We have considered learning rate till 0.001.

  Following heatmap was achieved on the validation set after each training. Here x-axis represents σ2 and the y-axis represents the learning rate and colors represents accuracy obtained on the validation set after each training. So each colored cell basically represents accuracy for particular learning rate and specific initialization.

  We can clearly see from the heatmap that when sigma value is 0 accuracy obtained is almost the same which is 0.5. The worst accuracy obtained is for learning rate 0.00001 and sigma 0.7. We can observe that as the learning rate is increased so does the accuracy of the model. Accuracy of the model is highest when learning rate is 0.001 and sigma value lies between 0.1 to 1.



3. **Visualizing the output of each hidden neuron from after applying the activation function from the two hidden layer three times from the worst and the best performing models from Subproblem B.2 in terms of classification accuracy.**

From the subproblem B.2 we know that the worst performing model has the worst accuracy obtained is for learning rate 0.00001 and sigma 0.4. The beast performing model has the highest accuracy when learning rate is 0.001 and sigma value lies anywhere between 0.1 to 1.
Using the function **compute_hidden_outputs_ordered** we calculate the activations of hidden layers in a neural network up to two hidden layers. It computes these activations using the ReLU activation function, storing the results in a Data Frame. This is helpful for understanding how data transforms as it moves through the network during the forward pass.

```
def compute_hidden_outputs_ordered(self, X_val, layer_id):
    # Forward pass up to the specified hidden layer
    if layer_id == 1:
        activation = relu(X_val.dot(self.weights1) + self.bias1)
    elif layer_id == 2:
        activation1 = relu(X_val.dot(self.weights1) + self.bias1)
        activation = relu(activation1.dot(self.weights2) + self.bias2)

    # Get the number of neurons in the hidden layer
    num_neurons = activation.shape[1]

    # Create a DataFrame to store the activation values, class labels, and data point IDs
    df_activation = pd.DataFrame(activation, columns=[f'Neuron {i + 1}' for i in range(num_neurons)])

    return df_activation, num_neurons
```

Using the function **visualize_hidden_layer_output_ordered** we organize and display the activation values of neurons in a chosen hidden layer using a heatmap. It sorts the activation values based on class labels (ordered per class) and data point IDs, creating a visual representation that shows how different neurons react across various classes in the validation set.

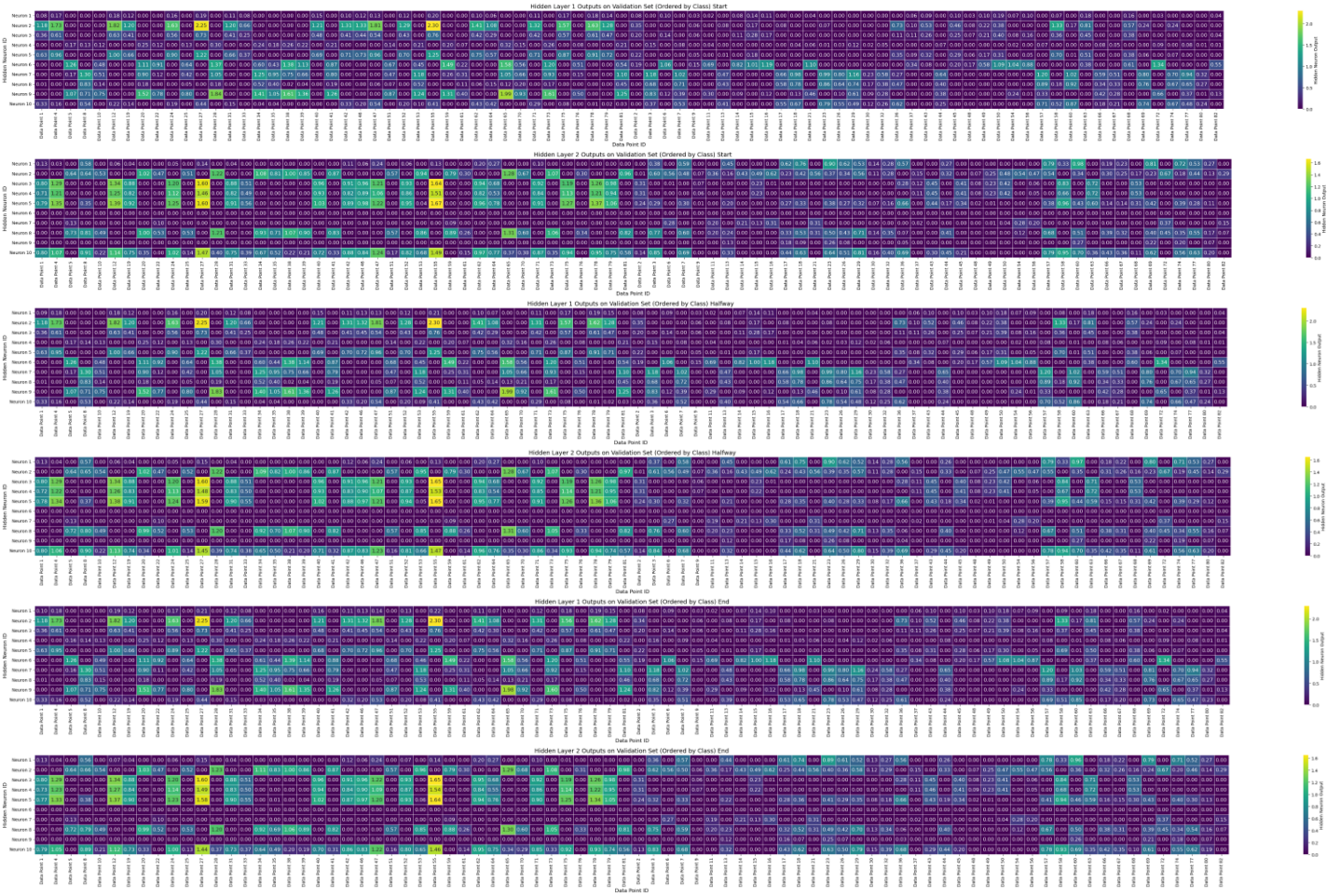
```
def visualize_hidden_layer_output_ordered(self, X_val, y_val, layer_id, title):
    # Compute hidden neuron outputs on the validation set
    df_activation, num_neurons = self.compute_hidden_outputs_ordered(X_val, layer_id)

    # Add class labels and data point IDs to the DataFrame
    df_activation['Class'] = y_val
    df_activation['Data Point ID'] = range(1, len(y_val) + 1)

    # Sort the DataFrame by class labels and then by data point IDs
    df_activation = df_activation.sort_values(by=['Class', 'Data Point ID'])

    # Extract the sorted activation values
    activation_sorted = df_activation.drop(columns=['Class', 'Data Point ID']).values

    fig_height = (0.5 * num_neurons)
    fig_width = min (5 * len(y_val),50)

    # Create a heatmap
    plt.figure(figsize=(fig_width, fig_height))
    sns.heatmap(activation_sorted.T, cmap="viridis", annot=True, fmt=".2f", linewidths=.5,
        xticklabels=[f'Data Point {i}' for i in df_activation['Data Point ID']],
        yticklabels=[f'Neuron {i + 1}' for i in range(num_neurons)],
        cbar_kws={'label': 'Hidden Neuron Output'}, annot_kws={"size": 12})
```

Using the function **model** we test the model's performance on the validation data during training. It calculates accuracy, loss, and confusion matrix for the validation set.
Additionally, it includes conditional statements to visualize the hidden layer outputs at different training stages based on specific hyperparameter values **(sg == 0.4 and lr is 0.000001 or 0.001)**. The code visualizes the hidden layer outputs at the **start, midway**, and **end** of training for both the first and second hidden layers under certain conditions.
It also collects and stores the validation accuracy for different hyperparameter combinations (**sg** and **lr**). Finally, it prints the validation accuracy for specific hyperparameter combinations of interest and maintains a history of accuracy across epochs for various hyperparameter combinations.

```
        #Test the data
        val_accuracy, val_loss, confusion_matrix = self.test(X_val, y_val)
        val_loss_history.append(val_loss)

        if sg == 0.4 and (lr == 0.000001 or lr == 0.001):
            if epoch == 0:
                # Visualize hidden layer output on validation set at start
                self.visualize_hidden_layer_output_ordered(X_val, y_val, 1, 'Start')
                self.visualize_hidden_layer_output_ordered(X_val, y_val, 2, 'Start')
            if epoch == epochs // 2:
                # Visualize hidden layer output on validation set at halfway
                self.visualize_hidden_layer_output_ordered(X_val, y_val, 1, 'Halfway')
                self.visualize_hidden_layer_output_ordered(X_val, y_val, 2, 'Halfway')

            if epoch == epochs - 1:
                # Visualize hidden layer output on validation set at the end
                self.visualize_hidden_layer_output_ordered(X_val, y_val, 1, 'End')
                self.visualize_hidden_layer_output_ordered(X_val, y_val, 2, 'End')

    accuracy_list.append(val_accuracy)
    if sg == 0.4 and (lr == 0.000001 or lr == 0.001):
        print(f"Learning rate: {lr}, Sigma: {sg} ------------------> Validation Accuracy: {val_accuracy:2f}")
accuracy_hist.append(accuracy_list)
```

OUTPUTS

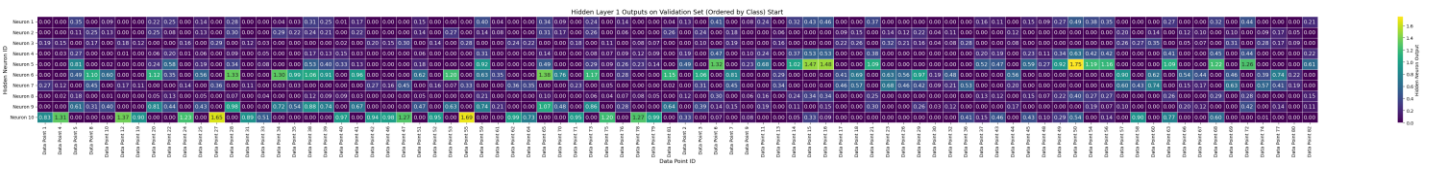
Below are the hidden layer outputs given for each hidden layer 1 and 2 at the **start, midway**, and **end** of training performed on validation set for the worst performing model with learning rate = 0.00001 and sigma = 0.4. The first half of the datapoints represents the class of 0's and the second half of the datapoints represents class of 1's.

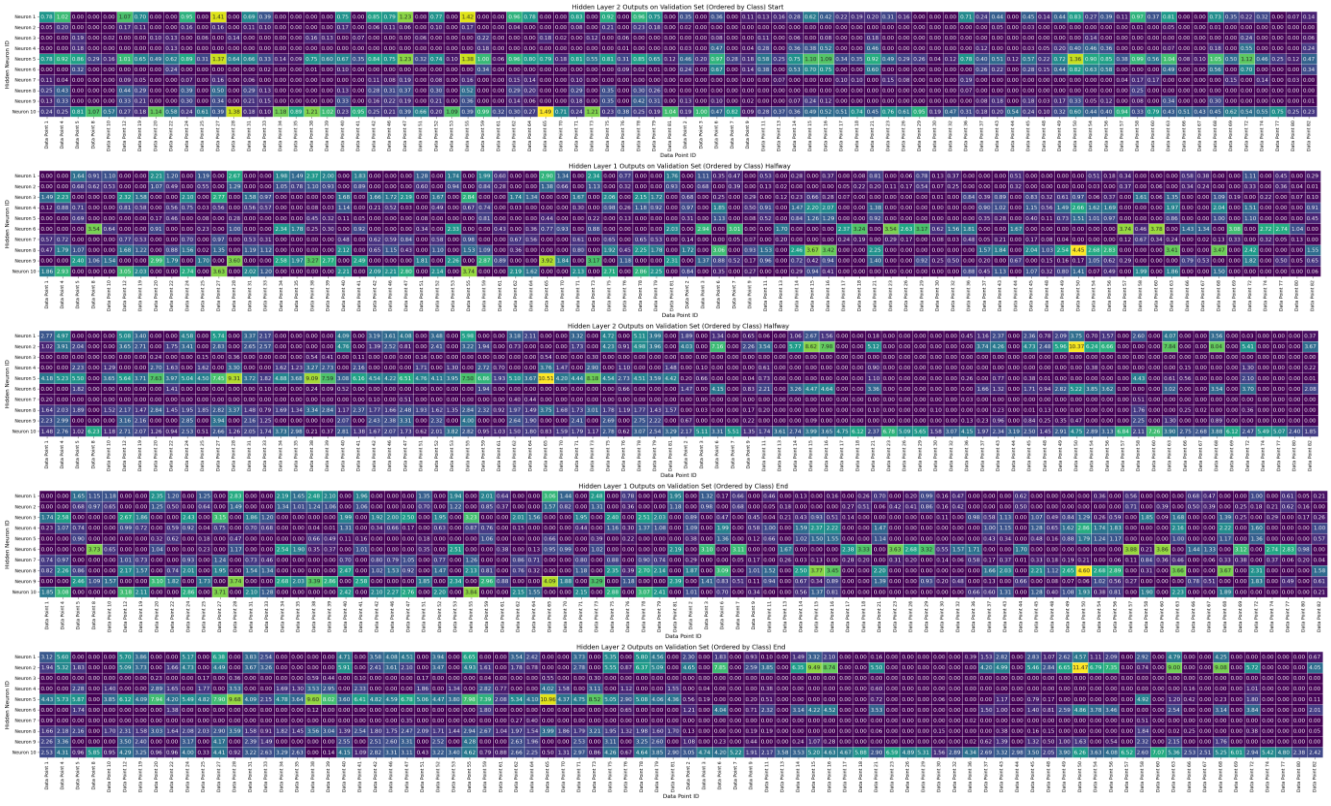


The Learning rate, Sigma and Validation Accuracy values for the worst performing model

```
Learning rate: 1e-06, Sigma: 0.4 ------------------> Validation Accuracy: 0.304878
```

Below are the hidden layer outputs given for each hidden layer 1 and 2 at the **start, midway**, and **end** of training performed on validation set for the best performing model with learning rate = 0.001 and sigma = 0.4. The first half of the datapoints represents the class of 0's and the second half of the datapoints represents the class of 1's.





The Learning rate, Sigma and Validation Accuracy values for the best performing model

```
Learning rate: 0.001, Sigma: 0.4 ------------------> Validation Accuracy: 0.975610
```

4. Hyperparameters Optimization

From the part B3 and B2, we achieved the highest accuracy (98 %) with the parameters, Learning Rate = 0.001, Activation Function = Sigmoid (output layer), Relu (Hidden layers). Hence, we decided to retain the same activation functions and tune the below hyperparameters.

- learning_rates = [0.001, 0.01, 0.1]
- hidden1_sizes = [5, 10, 20]
- hidden2_sizes = [5, 10, 20]
- batch_sizes = [32, 64, 128]

This code below performs a hyperparameter search for an MLP model by iterating through various combinations of hyperparameters such as learning rates, batch sizes, and sizes for two hidden layers. It creates an instance of the MLP class for each combination, trains the model, and evaluates its performance on the validation set. The best and worst performing hyperparameter combinations are tracked, along with their validation accuracies. Additionally, the code generates a Parallel Coordinates Plot using Plotly's Graph Objects to visualize the relationship between hyperparameters and accuracy. This plot helps in

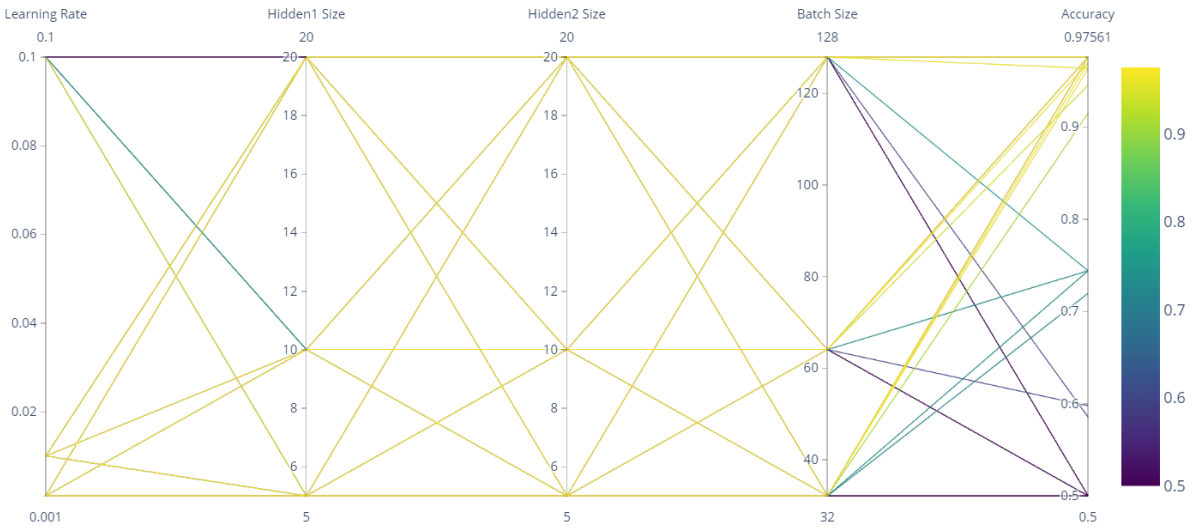
understanding how changes in hyperparameters affect the model's performance. Overall, it systematically explores different hyperparameter settings to find the combination that results in the best validation accuracy and provides a visual representation of this exploration.

In [6]:
```
# Assuming X_train, y_train, X_val, y_val are already defined

# Define hyperparameter values to iterate over
learning_rates = [0.001, 0.01, 0.1]
hidden1_sizes = [5, 10, 20]
hidden2_sizes = [5, 10, 20]
batch_sizes = [32, 64, 128]

best_val_accuracy = 0
worst_val_accuracy = 100
best_hyperparameters = {}
worst_hyperparameters = {}
similar_val_accuracy = []

results = []

# Iterate over hyperparameter combinations
for lr in learning_rates:
    for bs in batch_sizes:
        for h1_size in hidden1_sizes:
            for h2_size in hidden2_sizes:
                # Create an instance of the MLP class with current hyperparameters
                mlp_model = MLP(input_size=2, hidden1_size=h1_size, hidden2_size=h2_size, output_size=2)
                print(f"Training with Hidden1 Size:{h1_size}, Hidden2 Size:{h2_size}, Batch Size:{bs}, Learning Rate:{lr}")
                # Train the model
                mlp_model.model(X_train, y_train, X_val, y_val, epochs=100, learning_rate=lr, batch_size=bs)

                # Test the model on the validation set
                val_accuracy, _, _ = mlp_model.test(X_val, y_val)
                print(f"Accuracy: {val_accuracy}")

                # Check if current hyperparameters result in better validation accuracy
                if val_accuracy > best_val_accuracy:
                    best_val_accuracy = val_accuracy
                    best_hyperparameters = {'learning_rate': lr, 'hidden1_size': h1_size, 'hidden2_size': h2_size, 'batch_size':
                elif val_accuracy == best_val_accuracy:
                    similar_val_accuracy.append((h1_size,h2_size,bs,lr))

                # Check if current hyperparameters result in worse validation accuracy
                if float(val_accuracy) < worst_val_accuracy:
                    worst_val_accuracy = val_accuracy
                    worst_hyperparameters = {'learning_rate': lr, 'hidden1_size': h1_size, 'hidden2_size': h2_size, 'batch_size'

                result_entry = {
                    'Learning Rate': lr,
                    'Hidden1 Size': h1_size,
                    'Hidden2 Size': h2_size,
                    'Batch Size': bs,
                    'Accuracy': val_accuracy
                }
                results.append(result_entry)

# Print the best hyperparameters and validation accuracy
print("================================================================================")
print("Best Hyperparameters:", best_hyperparameters)
print("Best Validation Accuracy:", best_val_accuracy)
print("Worst Hyperparameters:", worst_hyperparameters)
print("Worst Validation Accuracy:", worst_val_accuracy)
print("Other models(hyperparameters) with best validation accuracy(hidden1_size1, hidden1_size2, batch_size, learning_rate):", s
print("================================================================================")
```

From the above code we get the following output

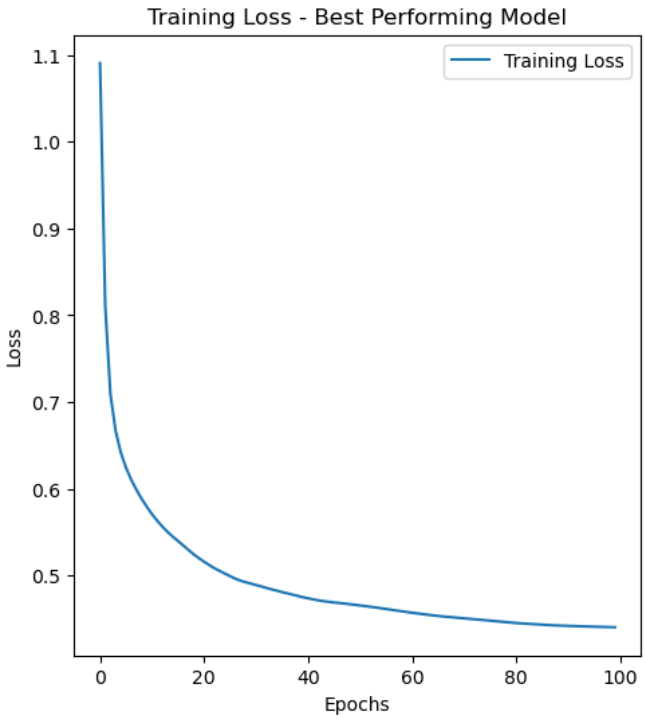
```
================================================================================
Best Hyperparameters: {'learning_rate': 0.001, 'hidden1_size': 5, 'hidden2_size': 5, 'batch_size': 32}
Best Validation Accuracy: 0.975609756097561
Worst Hyperparameters: {'learning_rate': 0.1, 'hidden1_size': 10, 'hidden2_size': 10, 'batch_size': 32}
Worst Validation Accuracy: 0.5
Other models(hyperparameters) with best validation accuracy(hidden1_size1, hidden1_size2, batch_size, learning_rate): [(5, 20,
32, 0.001), (10, 5, 32, 0.001), (10, 10, 32, 0.001), (10, 20, 32, 0.001), (20, 5, 32, 0.001), (20, 10, 32, 0.001), (20, 20, 32,
0.001), (5, 20, 64, 0.001), (10, 10, 64, 0.001), (10, 20, 64, 0.001), (20, 5, 64, 0.001), (20, 10, 64, 0.001), (20, 20, 64, 0.0
01), (5, 10, 128, 0.001), (5, 20, 128, 0.001), (10, 5, 128, 0.001), (10, 10, 128, 0.001), (10, 20, 128, 0.001), (20, 5, 128, 0.
001), (20, 10, 128, 0.001), (20, 20, 128, 0.001), (5, 10, 32, 0.01), (5, 20, 32, 0.01), (10, 5, 32, 0.01), (10, 10, 32, 0.01),
(10, 20, 32, 0.01), (20, 5, 32, 0.01), (20, 10, 32, 0.01), (20, 20, 32, 0.01), (5, 10, 64, 0.01), (5, 20, 64, 0.01), (10, 5, 6
4, 0.01), (10, 10, 64, 0.01), (10, 20, 64, 0.01), (20, 5, 64, 0.01), (20, 10, 64, 0.01), (20, 20, 64, 0.01), (5, 5, 128, 0.01),
(5, 10, 128, 0.01), (10, 5, 128, 0.01), (10, 10, 128, 0.01), (10, 20, 128, 0.01), (20, 5, 128, 0.01), (20, 10, 128, 0.01), (20,
20, 128, 0.01)]
================================================================================
```

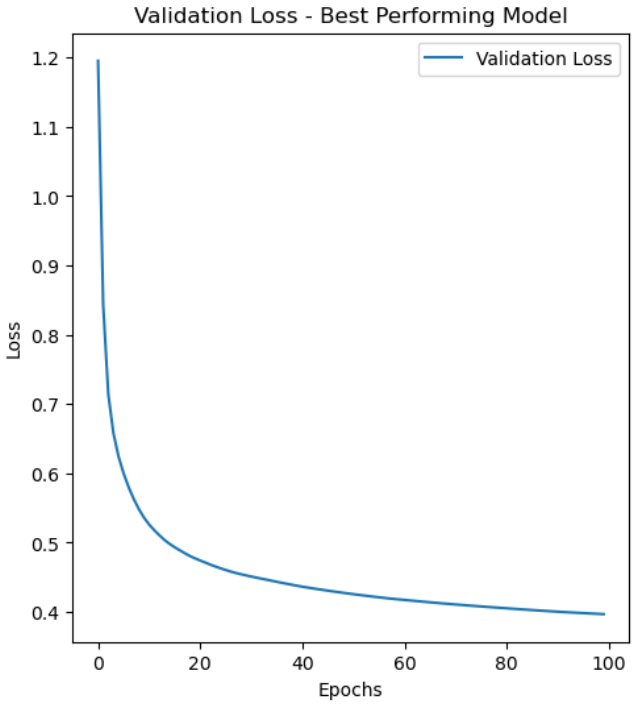
Plotting the Parallel Coordinates Plot for the above mentioned hyperparameters



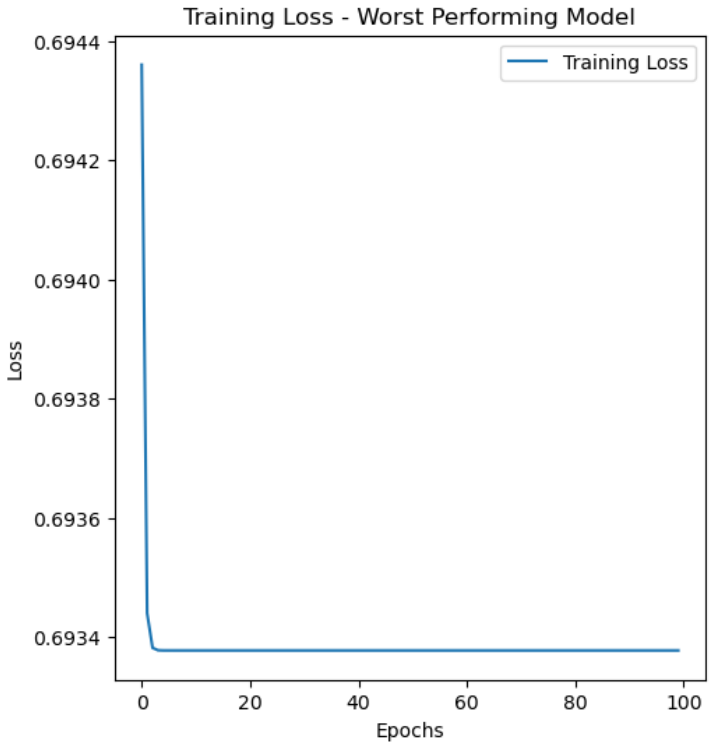
This visualization shows the co-relation between the accuracy and the hypermeter values. The colors of the lines which connect various hyperparameter settings represent the accuracy on the validation set.
From the above graph we can see all the yellow lines show the best accuracy, as the color turns darker it shows the lower accuracy. We can clearly understand that by tuning different hyperparameters we could attain different accuracy results.

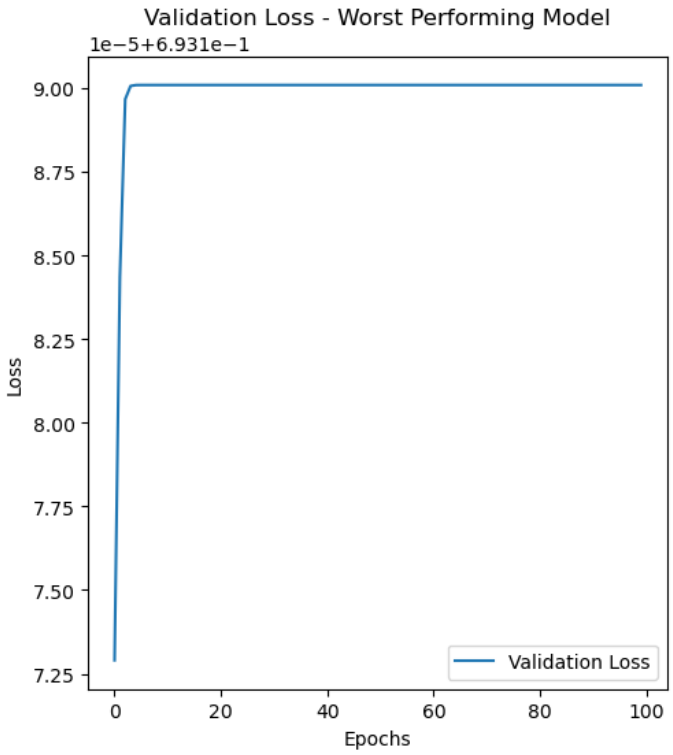
For the worst and best performing models, we have plotted the values of the loss function computed separately over the training and validation sets respectively as given below.





The above graphs show the Training loss and Validation Loss for the best performing models





The above graphs show the Training loss and Validation Loss for the worst performing models

Best performing model generalization performance

This model with the best hyperparameters likely demonstrates good generalization. It achieved the highest validation accuracy among all combinations, suggesting it learned patterns from the training data that apply well to unseen validation data. However, there's no possibility of overfitting where a model performs significantly better on the training set than on the validation set.

Worst performing model generalization performance

The worst performing model probably demonstrates poor generalization. Its low validation accuracy suggests it didn't learn patterns from the training data that can be applied effectively to new data. This model might suffer from underfitting, where it failed to capture the underlying patterns in the training data. The model might be too simple to comprehend the complexities present in the data, resulting in poor performance both on training and validation sets.