Part A

Part 1: Activation and Loss Functions

The activation function is in charge of converting the node's summed weighted input into node activation or output for that input.

We use Rectified Linear Unit (ReLU) for both hidden layers. Non-linearity is introduced by ReLU, which outputs x(input value) for all positive values and 0 for all negative values. It is widely used because it is simple to implement and effective at overcoming the limitations of previously popular activation functions such as Sigmoid and Tanh. It is less susceptible to vanishing gradients, which prevent deep models from being trained, but it can suffer from other issues such as saturated or "dead" units.

$$ReLU(x)=max(0,x)$$

We used Sigmoid activation function for output layer. Because it maps the output to a probability between 0 and 1, Sigmoid is an obvious choice for binary classification problems. It is appropriate for models whose goal is to predict a binary outcome (for example, class 0 or class 1).

$$S(x) = 1/(1 + e−x)$$

We used Binary Cross-Entropy Loss function in our model. We tried using two other loss functions Huber loss and mean square error loss function also, but loss function values were not decreasing over time. Training loss and Validation losses were not decreasing. MLP model is performing better for Binary Cross-Entropy Loss function. As we have binary classification problem and Binary Cross-Entropy is commonly used for binary classification task.

$$L(y,y^)=−(y\cdot\log(y^1−y)\cdot\log(1−y^))$$

Reference:
https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/

Part 2: Learning rate, batch size, initialization

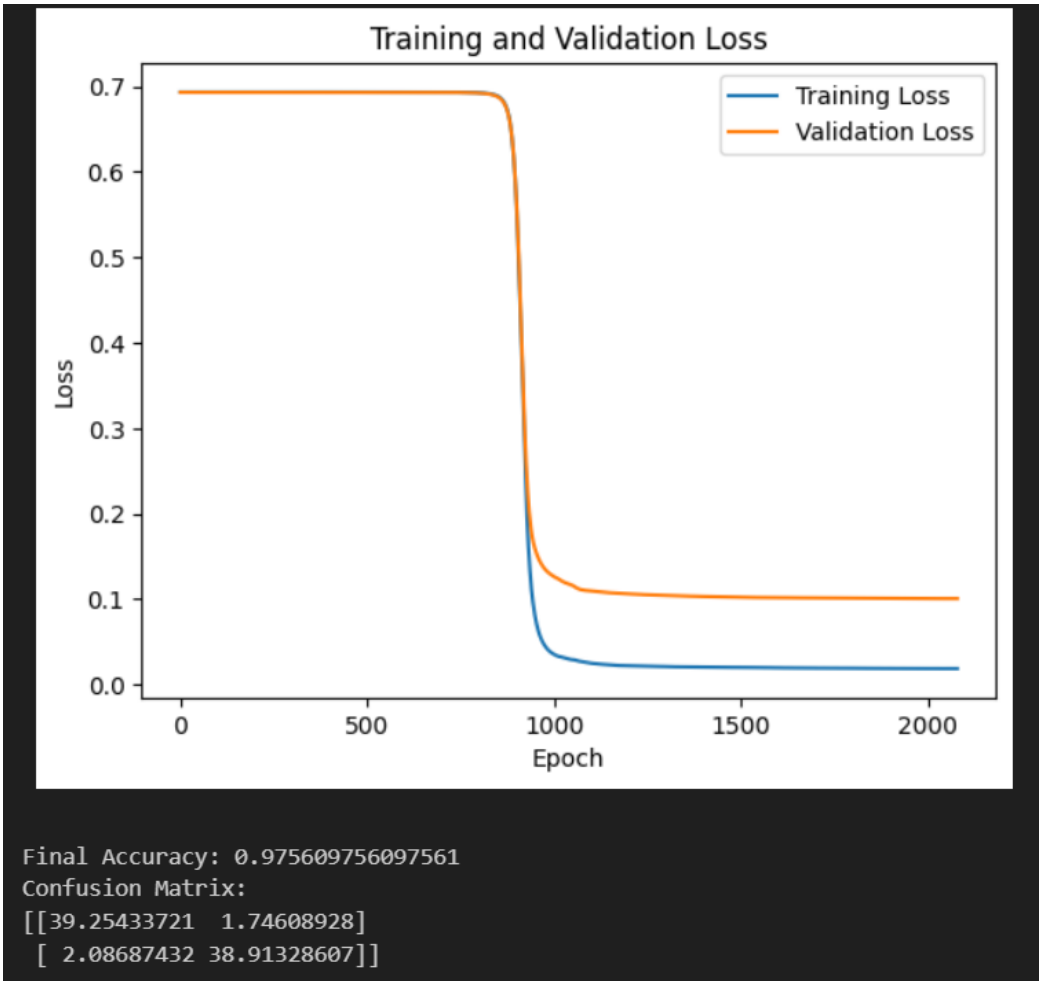
Learning rate : 0.01
Batch size : 16
Epochs: 2500

We are initializing weights and biases also. Weights and biases are parameters learned by neural networks during the training process. Before applying the activation function, they determine the strength of connections between neurons (weights) and the bias added to the weighted sum (biases). There are three sets of weights and biases in our task, corresponding to the connections between the input layer and the first hidden layer, between the two hidden layers, and between the second hidden layer and the output layer. Small random values drawn from a normal distribution with a mean of 0 and a standard deviation of 0.01 are used to initialize the weights (np.random.randn(...) * 0.01). The biases are all set to zero. It is common practice to set biases to zero because it allows neurons to start with a non-zero output, introducing some level of activation from the start. We are using small random values to initialize weights it helps to prevent problems like vanishing or exploding gradients during the early stages of training. Weights are multiplied by 0.01 so that we can control the scale of the activations and keep them from being too large.

If we set learning rate too high it may cause the algorithm to overshoot the minimum, resulting in divergence. A low learning rate, on the other hand, can result in slow convergence. We tried it with different learning rates with 0.001 it was converging too slowly for 0.01 training was complete at

4100 epochs and for 0.1 it was too fast and training completed model stoped at approximately 500 epochs. So we chose 0.01. As when the model was stoping the graph was coming out to be same. For 0.001 and 0.0001 model was performing very slow at even not stopping at 50,000 epochs.
Smaller batch sizes introduce more noise to the optimization process. If we set batch size larger it provide more stable gradient estimates, they slow convergence. On batch size 32 model was trained at 4100 epochs and for 16 it was trained at 2000 epochs and training accuracy was same for both. So we chose 16 as smaller batch size is more efficient for convergence.
Number of epochs I chose here are 2500 and model converges at 2077 epochs.

Part 3: Training



Early Stopping:

This early stopping mechanism prevents overfitting by stopping training when the model's performance on the validation set ceases to improve or begins to degrade. If the current validation loss (validity_loss) is better (lower) than the best_loss, the best_loss is updated to the current loss, and the stop_counter is reset to 0. If the validation loss does not improve, increment stop_counter. If the stop_counter reaches or exceeds the specified wait, it indicates that there has been no improvement for a predetermined number of epochs. The training is halted in this case, and the loop is broken.
Our model stopped training at epoch 2077.
Final accuracy obtained is 97.56%.
Confusion Matrix is [[39.25433 1.7460]
[2.0868 38.9132]]

This model using Binary Cross Entropy loss is giving us more accuracy than the other two keeping all the other parameters unchanged.

For Mean Square error loss

Final Accuracy: 0.5
Confusion Matrix:
[[20.49970018 20.50029031]
[20.49963522 20.50030282]]

For Huber loss

Final Accuracy: 0.45
Confusion Matrix:
[[19.49970018 21.50029031]
[20.49963522 20.50030282]]

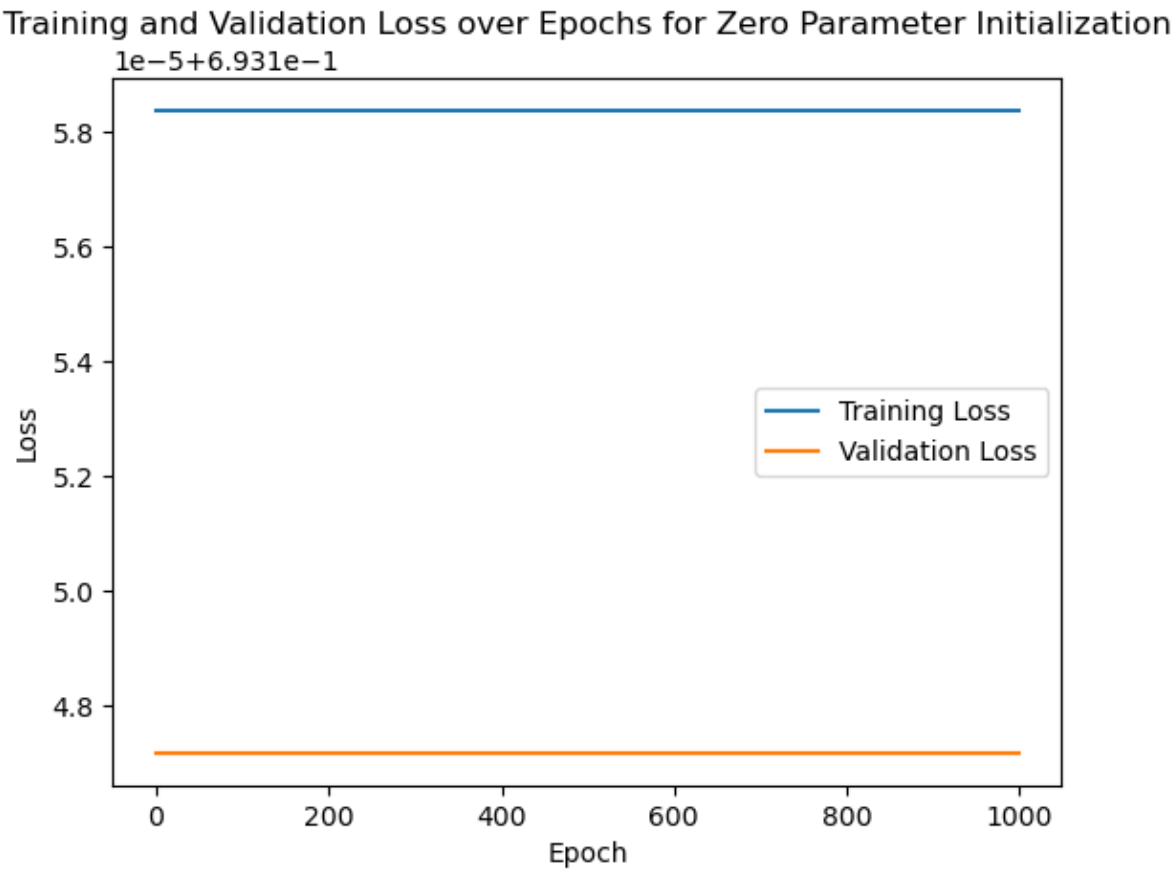
Reference:
https://machinelearninggeek.com/multi-layer-perceptron-neural-network-using-python/
https://github.com/Soham-Kelkar/Deep-Learning-without-Libraries
https://thecodacus.com/posts/2022-01-07-build-neural-network-from-scratch-in-python-no-libraries/

Part B

Part 1: Parameter initialization

The best performing model from part A in terms of classification accuracy is the model using Binary cross entropy loss. We initialize all its parameters (connection weights and biases) to zero and retrain it using all the other settings from Part A. The performance of this new trained model is as follows:

Final Accuracy: 0.5
Confusion Matrix:
[[20.49977063 20.50022937]
[20.49977063 20.50022937]]



We further note that we have the same training loss and validation loss for all epochs: Training Loss: 0.693158372382391, Validation Loss: 0.6931471806225411

We notice that when we initialize the parameters, the model doesn't learn. This is because each layer produces the same output, 0, during forward and backward passes. Thus, the model cannot learn different features from the input data. The model further becomes too simple in terms of complexity and cannot reflect the underlying patters in the data. All of this prevents the learning process.
Compared to the best model from part A, we note that this model has lower final accuracy, higher training loss, and higher validation loss. All of this is reflected in the plot above: We can see that the model is not performing well on the training data. Furthermore, we see that the model never converges.

Part 2: Learning rate vs parameter initialization

Preprocessing Techniques

1. **Normalization**: Our approach involves normalizing the features by subtracting the mean and dividing by the standard deviation, ensuring a standardized scale across all features.
2. **One-Hot Encoding**: We use this technique to convert categorical variables into a machine-readable form, facilitating the processing of non-numeric data.

Neural Network Architecture and Functions

- **Activation Functions**:
    - *ReLU Function*: Used in hidden layers, the ReLU (Rectified Linear Unit) function outputs zero for negative inputs and retains positive values. This is mathematically represented as f(x) = max(0, x).
    - *Sigmoid Function*: Essential in binary classification tasks, it compresses any real-valued number into a 0 to 1 range, ideal for probability predictions. It's defined as f(x) = 1 / (1 + e^(-x)).
- **Loss Function**: We employ the binary cross-entropy loss function, which effectively penalizes deviations from the actual values in binary classification.

Network Configuration

- The output layer of our neural network is set to have 2 nodes.
- Similarly, the input layer is configured with 2 nodes.
- The hidden layer is designed with 10 neurons, offering a balance between complexity and computational efficiency.

Training Process

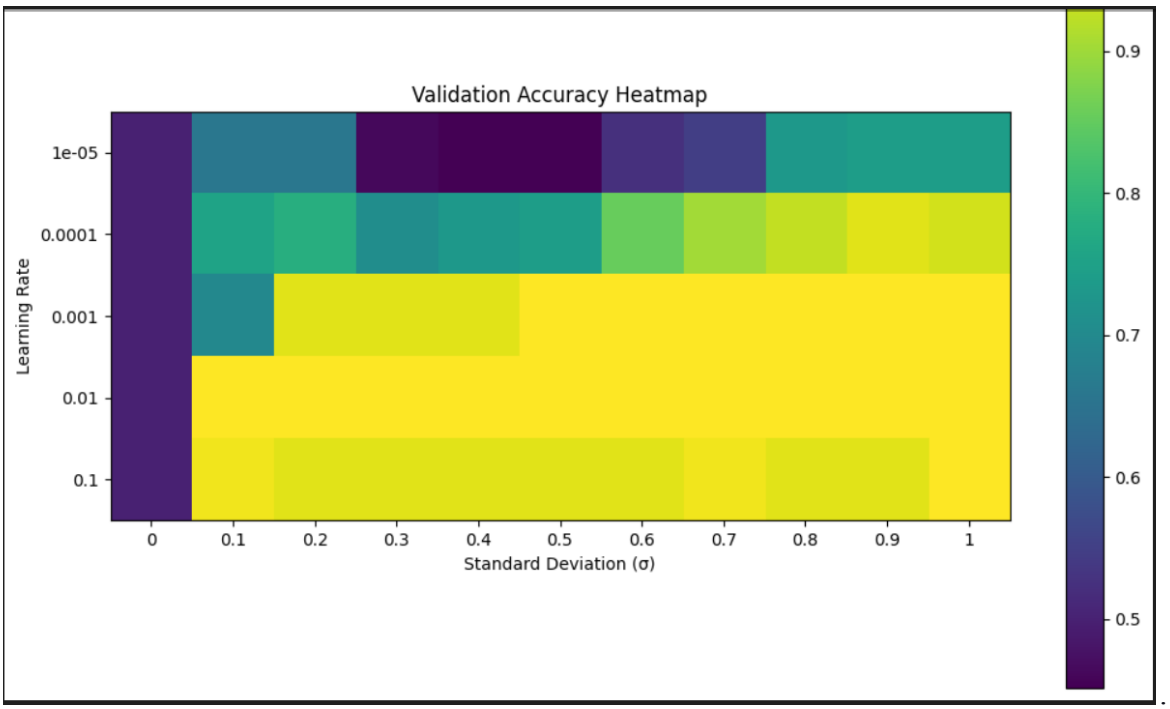
- **Epochs**: Each epoch represents a complete pass through the training dataset, determining the extent of learning.
- **Batch Size**: Set at 32, it refers to the number of training examples used in each iteration, crucial for weight adjustments.

Hyperparameter Tuning

- We explore various learning rates and standard deviations for weight initialization. These hyperparameters significantly influence the learning process and model performance.
- Our approach involves nested loops to test each combination of learning rate and standard deviation. This comprehensive testing includes initializing weights and biases, processing training data in mini-batches, and updating weights using gradient descent.

Results Analysis

- **Accuracy Matrix**: We compile an accuracy matrix from the results list to visually represent the model's performance across different hyperparameters.
- **Heatmap Visualization**:
    - The heatmap, generated using Python's matplotlib library, vividly illustrates the effects of learning rates and weight initialization on validation accuracy.
    - The optimal performance is observed at a learning rate of 0.001 and a standard deviation of around 0.8.
    - The heatmap reveals that extreme values of the learning rate and very small weights generally yield poorer performance.

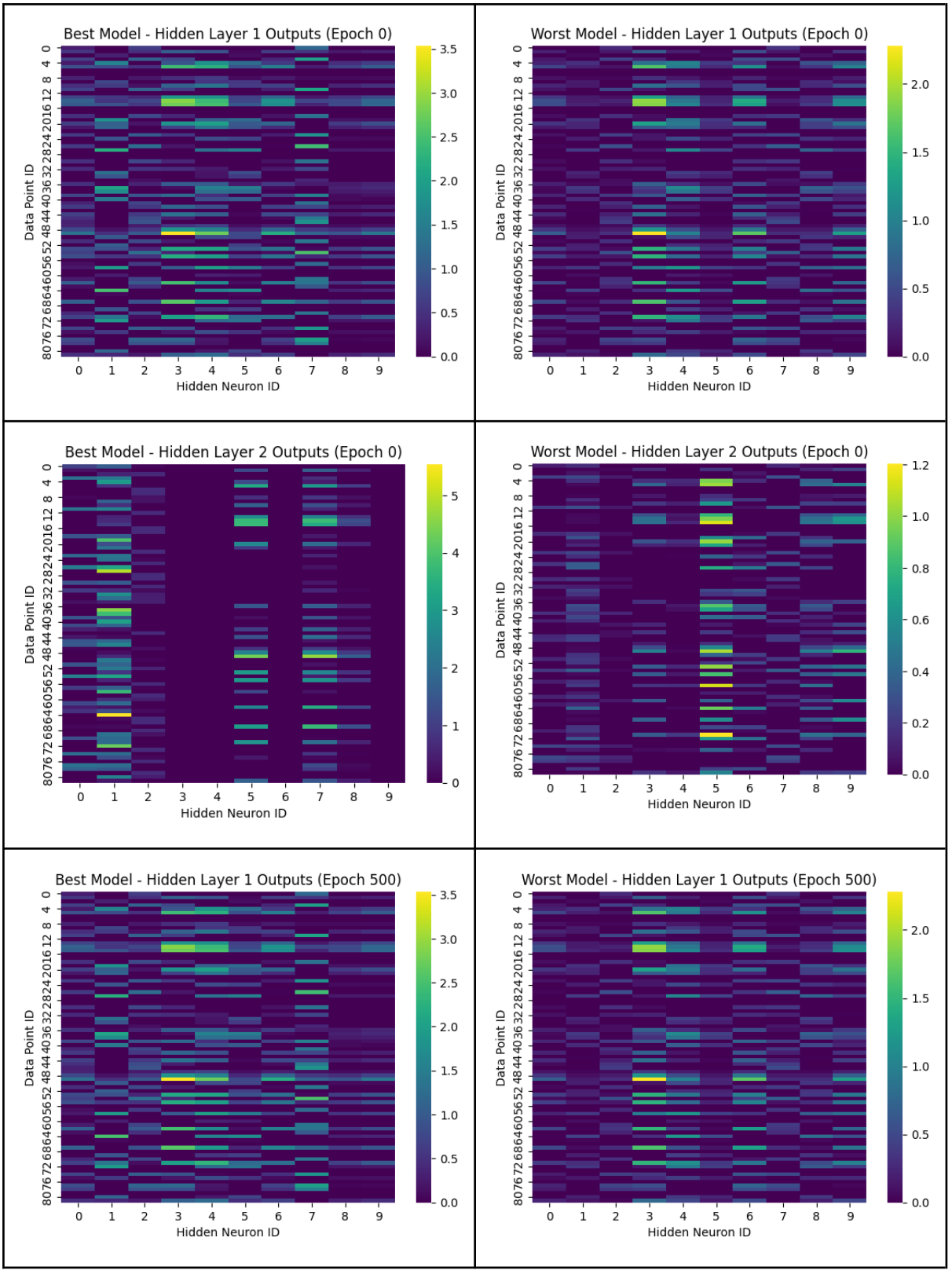


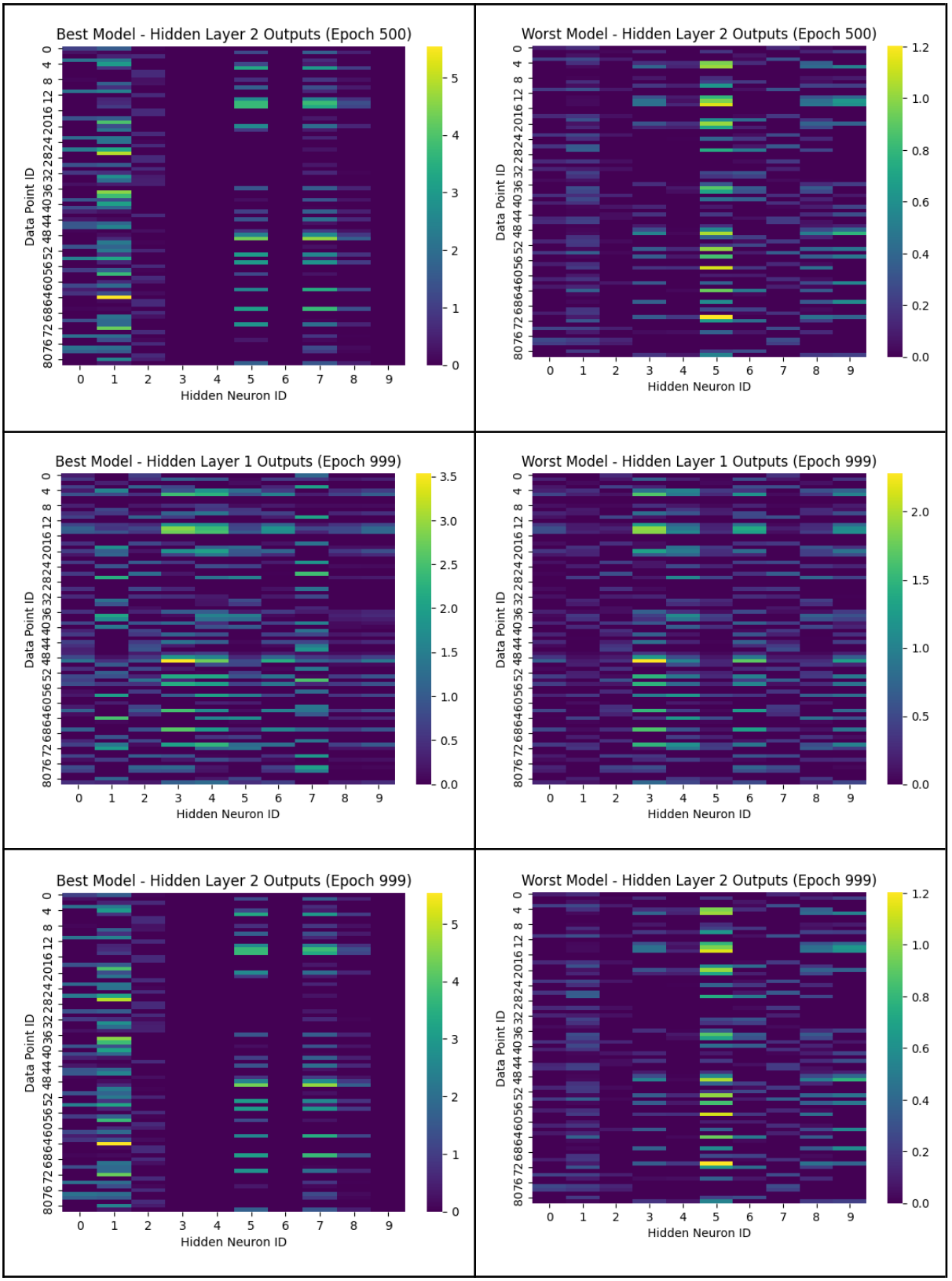
Part 3: Activations

We use heatmaps, where the x-axis represents the hidden neuron ID, the y-axis represents the data points ID, and the colors represent the hidden neuron outputs, to visualize the hidden layer outputs for the best model and the worst model. We visualize the said layers at epoch 0 (i.e., after connection initialization before to start the training process), at epoch 500 (i.e., at half of the training process), and at epoch 999 (i.e., the end of the training process).

We observe that the hidden layer 2 of the best model has more clear structure and distinct patterns compared to the worst model's hidden layer 2. Furthermore, we see sparsity in the hidden layer 2 of the best model, which indicates that only a subset of its neurons is highly active for its data points. This means that the best model is efficiently processing the input data.

We note that the hidden layer 1 of the best model has more greens and yellows compared to the hidden layer 1 of the worst model. This indicates that the best model's neurons are more strongly activated in response to the input, and thus, the model learns faster at the different stages of training (i.e., the epochs). Furthermore, the hidden layer 1 of the best model shows more diversity in the patterns and colors (which indicate how strongly are the neurons activated). This indicates that the best model captures the various features of the input data more efficiently compared to the worst model.


Best Model - Hidden Layer 1 Outputs (Epoch 0)
Worst Model - Hidden Layer 1 Outputs (Epoch 0)
Best Model - Hidden Layer 2 Outputs (Epoch 0)
Worst Model - Hidden Layer 2 Outputs (Epoch 0)
Best Model - Hidden Layer 1 Outputs (Epoch 500)
Worst Model - Hidden Layer 1 Outputs (Epoch 500)
Data Point ID
Hidden Neuron ID


Best Model - Hidden Layer 2 Outputs (Epoch 500)
Worst Model - Hidden Layer 2 Outputs (Epoch 500)
Best Model - Hidden Layer 1 Outputs (Epoch 999)
Worst Model - Hidden Layer 1 Outputs (Epoch 999)
Best Model - Hidden Layer 2 Outputs (Epoch 999)
Worst Model - Hidden Layer 2 Outputs (Epoch 999)
Data Point ID
Hidden Neuron ID

Part 4: Hyperparameters Optimization

Activation Functions and Their Roles

1. **Sigmoid Function**: Predominantly used in binary classification, the sigmoid function maps neuron outputs to a range between 0 and 1, ideal for probability predictions. During backpropagation, its derivative is crucial for gradient calculation and weight updates.
2. **ReLU (Rectified Linear Unit)**: This function introduces non-linearity to the model by returning positive inputs as-is and zeroing out negatives. It's favored in deep neural networks for its efficiency. Its derivative, straightforward with values of 1 or 0, facilitates backpropagation.
3. **Tanh (Hyperbolic Tangent)**: Offering outputs between -1 and 1, Tanh is similar to the sigmoid but often achieves faster convergence. Its derivative plays a pivotal role during backpropagation.

Loss Metrics: Binary Cross-Entropy

Binary Cross-Entropy Loss is a staple in binary classification models. It quantifies the difference between predicted probabilities and actual labels, guiding the optimization algorithm to adjust weights for minimizing loss.

Initialization and Training Functions

- **Parameter Initialization**: This function initializes weights and biases across the neural network. It leverages a random seed for reproducibility and adopts small random values and zero vectors for weights and biases.
- **Training Function**: This comprehensive function facilitates neural network training. It accepts inputs like data, labels, weights, biases, and learning rate. The forward pass involves computing activations layer by layer, culminating with the sigmoid function in the output layer. Backpropagation is conducted if it's not a validation pass, updating weights and biases through gradient descent.
- **Model Training**: The train_mlp_model function embodies a holistic approach for training multilayer perceptron models. It encompasses weight and bias initialization, loss tracking, and forward-backward propagation across epochs. Validation accuracy is computed by comparing predictions against actual labels.

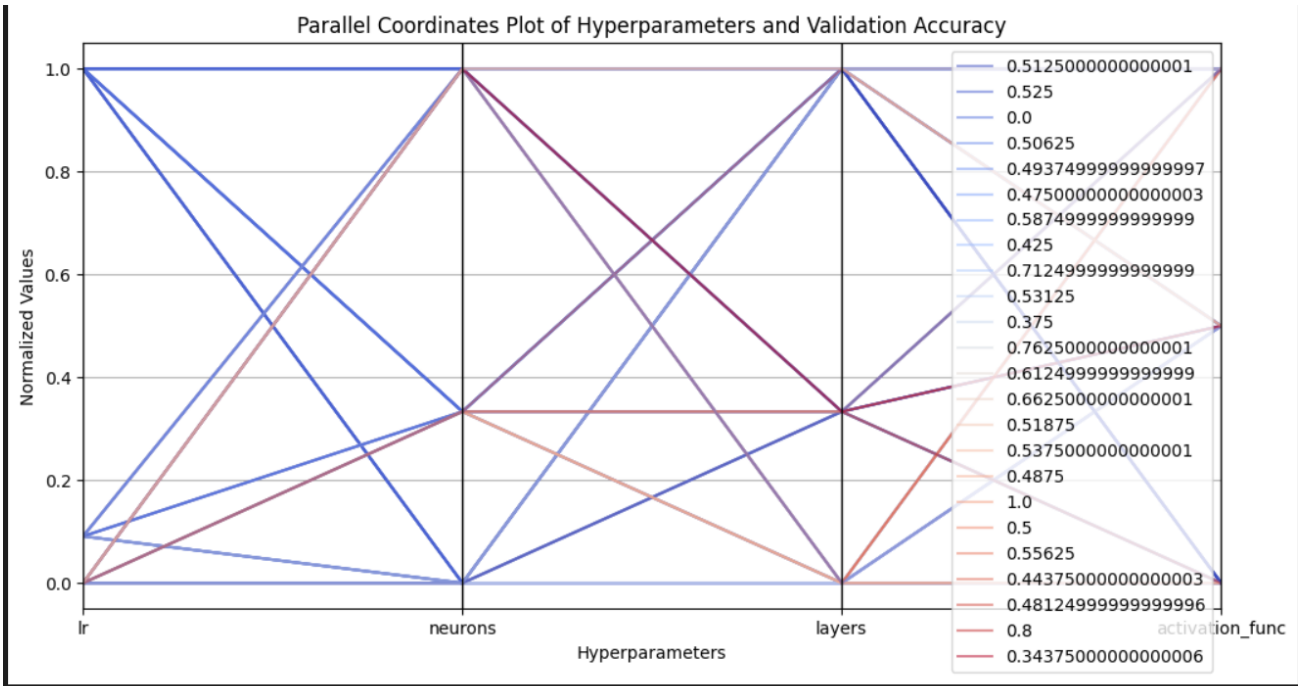
Hyperparameter Tuning and Visualization

- **Hyperparameter Exploration**: The code systematically explores combinations of hyperparameters, sorting results by validation accuracy to identify the best-performing model.
- **Visualization with Parallel Coordinates**: The performance of various hyperparameter combinations is visualized using a parallel coordinates plot. This visualization assists in understanding the impact of each hyperparameter on model accuracy.

Conclusion

The provided approach offers a detailed roadmap for neural network optimization. From selecting the right activation function to fine-tuning hyperparameters, each step is vital for enhancing model performance. The parallel coordinates plot emerges as a powerful tool, revealing how different hyperparameters interact and impact the overall efficacy of the

model. However, while informative, it may require supplementary quantitative analysis for deeper insights.



Part C: Pear Review

Arooba Arshad

All three of us did the whole assign together. All the fridays were dedicated for Machine Learning assignments. In first assign we divided 3 questions in all of us, everyone did their parts and reports respectively and then we compiled. For assign 2 first all of us worked on A part, researched it together and when we come up on something one of us tried to complete part A and its report and handed over to other two members and then they divided it. All of us worked equally.

Sirisha Dhavala

For our project, there were three of us in the group, and I'm happy to say that everyone did their part. We split the work evenly and helped each other out. Everyone got involved, and we all put in the same amount of effort. Because of this, I think we all should get the same grade. We worked well together and everyone did a fair share of the work.

Alisa Todorova

We worked together and equally on both assignments. We split the first assignment into 3 and each worked on one part. Then we discussed what everyone has done. We worked together on the second assignment, as its tasks were dependent on each other. We split the writing of the report into three. We worked every Friday together, and what was left, we finished at home. Everyone's contribution was equal, fair, and very good!