AI201 Report 2
Analysis: Back-propagation Algorithm of Artificial Neural Network

Mary Nathalie Dela Cruz

1. INTRODUCTION

The back-propagation algorithm is a widely used algorithm for training feed-forward artificial neural networks. The training process involves propagating the error from the output layer back to the input layer and updating the weights and biases of the network to minimize a predefined cost function. This is an iterative learning process that employs an optimization algorithm, which is usually the gradient descent, for minimizing a cost function, which is usually the mean squared error. In each iteration of the gradient descent, the weights and biases of each layer are updated based on the derivative of the cost function concerning these parameters. This process continues until the model prediction becomes satisfactory. [1-3]

2. OBJECTIVES

The main objective of this study was to analyze the back-propagation algorithm of a 2-layer multi-layer perceptron neural network. The network architectures were named Network A and Network B, where Network A used a Tanh-Tanh-Logarithmic activation function in its neural layers and Network B used a ReLu-ReLu-Logarithmic activation function in its neural layers. We first partitioned a dataset and dealt with its imbalance using a strategy known as the Synthetic Minority Oversampling Technique (SMOTE). We optimized the number of nodes in each neural layer and the number of epochs for training, and tuned hyperparameters such as the learning rate and the momentum constant. To understand the network's learning progression, we presented learning curves, which are plots of training and validation errors versus the number of epochs number. We also presented the time required to train and test each network, and performance metrics such as confusion matrix, accuracy, precision, recall, and F1 score to evaluate its classification performance.

3. METHODOLOGY

3.1 Data Collection and Pre-processing

Imbalanced classification can be dealt with using SMOTE (Synthetic Minority Oversampling Technique). We used the given dataset that contained 3486 instances with 354 features and a label from 1 to 8. The dataset was imbalanced, where class 1 had 1239 instances while class 3 only had 23 instances. Table 1 shows the frequency distribution of each class before processing. To address the issue of class imbalance, SMOTE was applied to the instances file and the labels file guided by the process discussed in Machine Learning Mastery [1]. According to the article, SMOTE is an oversampling technique that generates new samples for the class in the minority via interpolation. After applying SMOTE, we concatenated the transformed instances and labels file into one dataset. To split the dataset into training and validation sets, we first created an array of indices from 0 to the number of rows in the dataset, and then selected a random subset of them with a probability of 0.7. This subset was used as a mask to filter out the rows that were not in the training set and the inverse of the mask to filter out the rows that were in the validation set. Table 1 shows the effect of SMOTE on the data set where after processing, the total number of instances per class averages around 400 for the training set and around 100 for the validation set. One-hot encoding was also performed on the label set of both training and validation sets. With one-hot encoding, categorical variables were converted into binary vectors to feed our model with binary inputs.

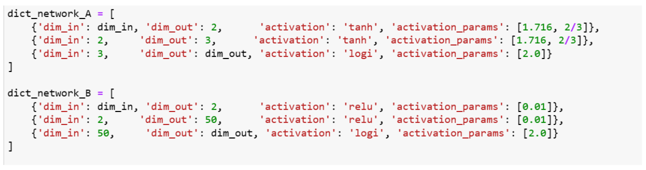
Table 1: Frequency distribution of each class: Full set with no processing vs training set with SMOTE vs test set with SMOTE

Class	Number of Instances		
	No Preprocessing	SMOTE Training	SMOTE Test
1	1625	406	94
2	233	412	88
3	30	406	94
4	483	398	102
5	287	408	92
6	310	389	111
7	52	391	109
8	466	390	110

3.2 Network Training

A method for training a multi-layer perceptron (MLP) was implemented using five custom functions: initialize(), forward_propagation(), backward_propagation(), update_weights(), and train_mlp. We first set a list of dictionaries to represent the network architecture, where each dictionary contains the specifications of a layer, such as the number of input and output nodes, the activation function, and the activation function parameters. Figure 1 shows 2 examples of the list for Network A and Network B with two hidden layers and one output layer.

The following functions were implemented to train our MLP:



```
dict_network_A = [
    {'dim_in': dim_in, 'dim_out': 2,      'activation': 'tanh', 'activation_params': [1.716, 2/3]},
    {'dim_in': 2,     'dim_out': 3,      'activation': 'tanh', 'activation_params': [1.716, 2/3]},
    {'dim_in': 3,     'dim_out': dim_out, 'activation': 'logi', 'activation_params': [2.0]}
]

dict_network_B = [
    {'dim_in': dim_in, 'dim_out': 2,      'activation': 'relu', 'activation_params': [0.01]},
    {'dim_in': 2,     'dim_out': 50,      'activation': 'relu', 'activation_params': [0.01]},
    {'dim_in': 50,     'dim_out': dim_out, 'activation': 'logi', 'activation_params': [2.0]}
]
```

Figure 1: Initial Dictionary of Network Architecture

- initialize(): This function took a list of dictionaries and a seed as inputs. The same list of dictionaries was returned but with additional keys for the weights, biases, and previous weight and bias updates for each layer. In the function, the weights and biases were randomly initialized from a uniform distribution between -1 and 1, and the previous updates were initialized as zero matrices.

- forward_propagation(): This function took the input data and the list of dictionaries as inputs. The same list of dictionaries was returned but with additional keys for the input, output, and v values for each layer. In the function, the input value was the output of the previous layer or the input data for the first layer, while the output value was the result of applying the activation function to the v value, which is the linear combination of the input value, the weight matrix, and the bias vector. The activation function can be logistic, hyperbolic tangent, or rectified linear unit, depending on what we set in the dictionary of a layer.

- backward_propagation(): This function took the output data, the list of dictionaries, and the learning rate as inputs. The same list of dictionaries was returned but with additional keys for the delta, delta_weight, and delta_bias values for each layer. The delta value was the error signal that propagated from the output layer to the input layer. The delta_weight value represented the weight update for each layer while the delta_bias value represented the bias update for each layer.

- update_weights(): This function took the initial list of dictionaries, the updated list of dictionaries, the delta_weights, the delta_biases, and the momentum as inputs. A new list of dictionaries was returned with updated weights and biases for each layer. The weight update was the sum of the delta_weight value and the momentum times the previous weight update. The bias update was the sum of the delta_bias value and the momentum times the previous bias update. Here, the momentum controlled the amount of the previous update that was added to the current update.

- train_mlp(): This function took several inputs to train a multilayer perceptron (MLP) network. It initialized the network with the initialize() function, created lists for storing errors and dictionaries for storing layer parameters, and then entered a loop for the specified number of epochs. In each epoch, it shuffled the training data and split it into mini-batches. For each mini-batch, it initialized lists to store the delta weights and biases. Then, for each instance in the mini-batch, it performed forward propagation with the forward_propagation() function, calculated the error, and performed backward propagation with the backward_propagation() function. It then updated the weights and biases of the network with the update_weights() function using the average delta weights and biases. If the current epoch was a multiple of "per_epoch", it calculated the sum of squared errors, not only for the training data but also for the validation data, and stored these values. Finally, it returned the updated network, the predicted and true values for the training and validation data, and the sum of squared errors for each epoch.

3.3 Performance evaluation

To assess model performance, we implemented a confusion matrix and several evaluation metrics. The distribution of predicted classes against the actual classes can be visualized with the confusion matrix. Accuracy can give a measure of the model's correctness, while precision and recall can give a measure of the model's performance on positive instances. Lastly, f1 score offered a balancing measure between precision and recall.

4. EXPERIMENTAL RESULTS

4.1 Initial Performance for Network A

Learning curves of MLP for Network A were acquired before tuning the number of nodes in each neural layer, the number of epochs for training, the learning rate, and the momentum.

Figure 2 shows the learning curve of the initial MLP with the training set of Network A. A logarithmic decrease in MSE was observed as the number of epochs increased. Increasing the number of epochs allows the MLP to learn more complex patterns in the training data, reduce the MSE and the chance of underfitting, and increase the chance of overfitting.

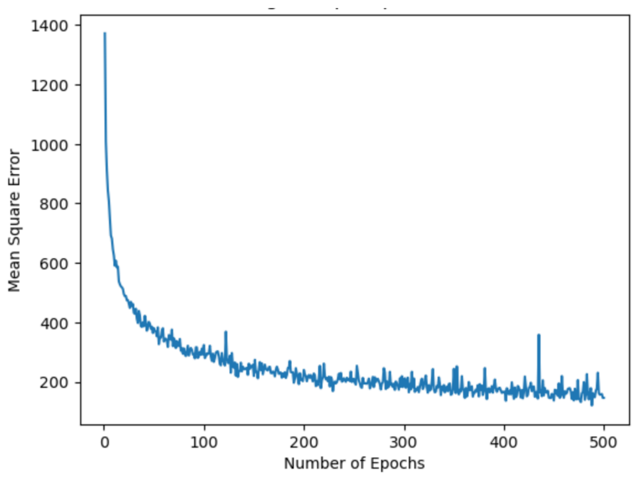


Figure 2: Training MSE per epoch of Network A without optimization

Figure 3 shows the learning curves of the initial MLP with the validation set of Network A. Although the learning curve for the validation set has more MSE spikes than the learning curve for the training set, it still follows a downward trend, which is an indication of the generalizability of the initial MLP.

4.2 Nodes and Hyperparameter Tuning

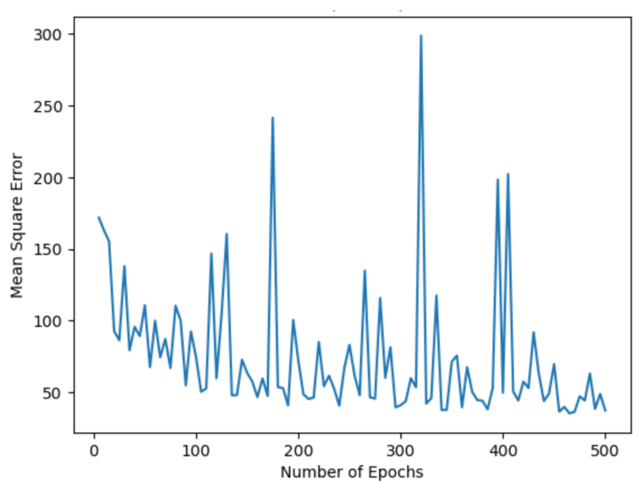


Figure 3: Validation MSE per 5 epochs of Network A without optimization

Optimization of the number of nodes per neural layer, the momentum, the learning rate, and the number of epochs of the MLP was done by evaluating how each parameter affects the mean MSE of the last 5 instances and the time interval.

Figure 4 shows the plot for optimizing the number of nodes in the 1st hidden layer of the initial MLP with the training set of Network A. Mean MSE decreases and the training time increases as the number of nodes in the 1st hidden layer increases. By increasing the number of nodes in the 1st hidden layer, MLP fits the training data better but it requires more computational resources. The logarithmic decrease in mean MSE implies that there is a diminishing return in adding more nodes. Hence, mean MSE and training time must be balanced when looking for the optimal number of nodes in the 1st hidden layer.

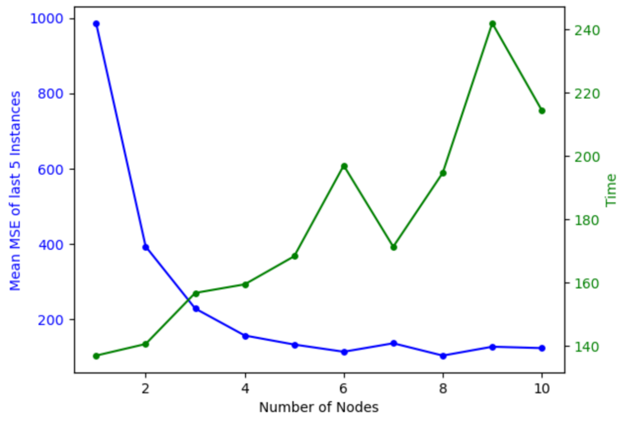


Figure 4: Optimizing the number of nodes in the 1st hidden layer by evaluating the effect of its variation on the mean MSE of the last 5 instances and the time interval

Figure 5 shows the plot for optimizing the number of nodes in the 2nd hidden layer of the initial MLP with the training set of Network A. Same as in Figure 4, mean MSE has a logarithmic decrease in Figure 5 as the number of nodes in the 1st hidden layer increases. Training time on the other hand randomly varies. The random variation in training time can be due to the composition of mini-batches which can affect how the weights are updated. External factors such as system load can also affect the training time.

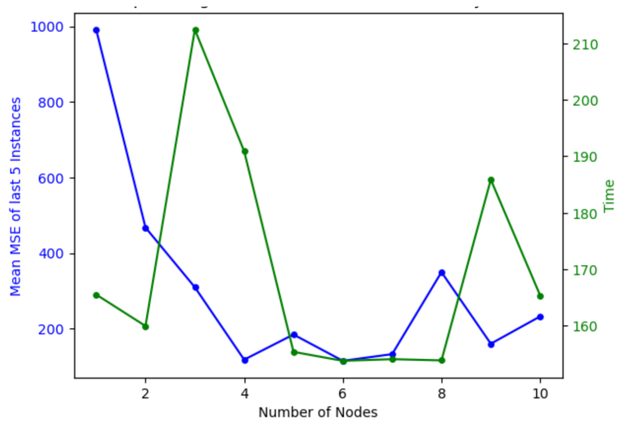


Figure 5: Optimizing the number of nodes in the 2nd hidden layer by evaluating the effect of its variation on the mean MSE of the last 5 instances and the time interval

Figure 6 shows the plot for optimizing the number of nodes in the 1st and 2nd hidden layers of the initial MLP with the training set of Network A. As the number of nodes in the 1st and 2nd hidden layers increases, training time increases while mean MSE initially has a logarithmic decrease which then subtly increases at the end. Effective performance can thus be achieved by balancing mean MSE and training time. Considering Figures 4 to 6, the optimal number of nodes in the 1st and 2nd hidden layers of the MLP is 5.

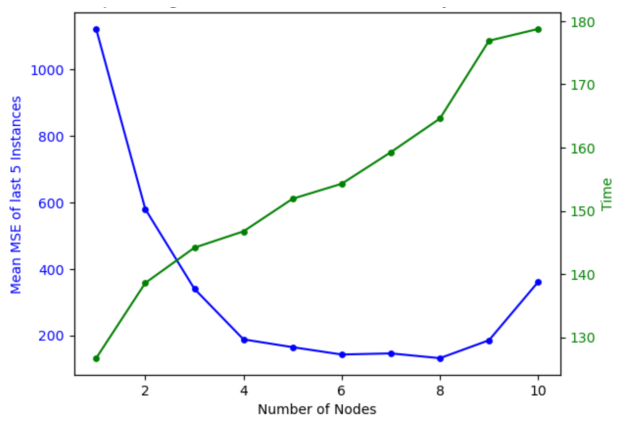


Figure 6: Optimizing the number of nodes in the 1st and the 2nd hidden layer by evaluating the effect of its variation on the mean MSE of the last 5 instances and the time interval

Figure 7 shows the plot for tuning the momentum of the initial MLP with the training set of Network A. Increasing momentum also increases the mean MSE and the likelihood of obtaining a smaller training time. This happens because momentum adds a fraction of the previous weight to the current weight to speed up the convergence, decreasing training time and increasing the likelihood of overshooting the optimal point. Taking the trade-off into consideration, the optimal momentum for the MLP was deemed to be 0.7. With this value, training time can be reduced while not compromising the mean MSE.

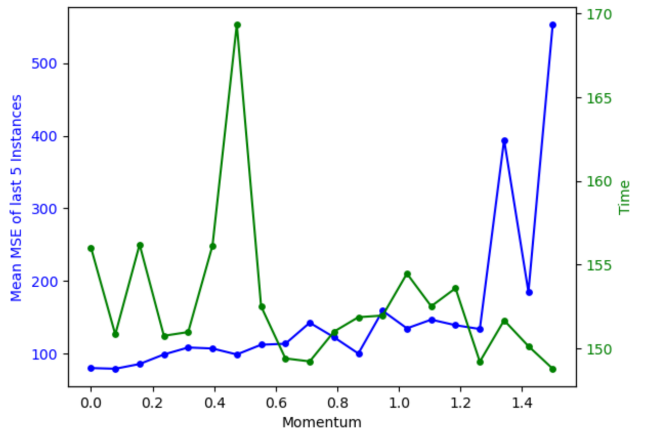


Figure 7: Tuning momentum by evaluating the effect of its variation on the mean MSE of the last 5 instances and the time interval

Figure 8 shows the plot for tuning the learning rate of the initial MLP with the training set of Network A. As expected, the learning rates near the minimum value and the maximum value have a large mean MSE while the learning rates in between have a small mean MSE. A larger MSE can be observed if the learning rate is too small and the model takes a long time to converge or if the learning rate is too large and the model overshoots the optimal point. Training time, on the other hand, has a steady increase with sudden spikes as the learning rate increases. Considering these findings, the optimal learning rate for the MLP is 0.14.

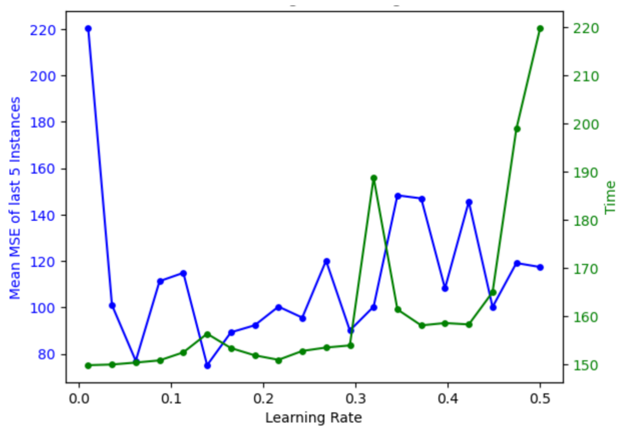


Figure 8: Tuning learning rate by evaluating the effect of its variation on the mean MSE of the last 5 instances and the time interval

Figure 9 shows the plot for optimizing the number of epochs done on the initial MLP with the training set of Network A. The plot shows a decrease in mean MSE and an increase in training time as the number of epochs increases to 500. This is expected because increasing the number of epochs needs more computations and longer training time. This also allows the MLP to learn more complex patterns, improving MSE. Considering the trade-off, the optimal number of epochs for the MLP is 400.

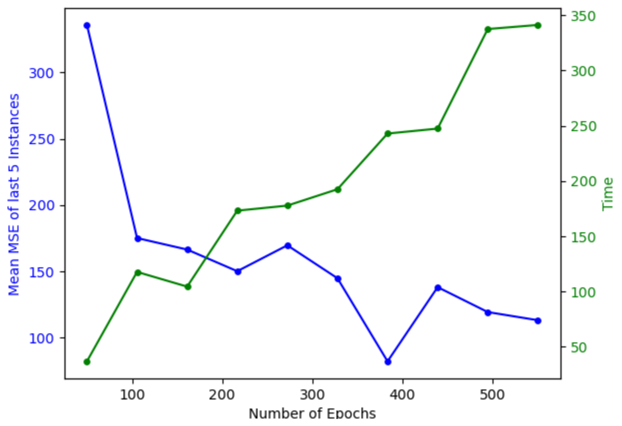


Figure 9: Optimizing the number of epochs by evaluating the effect of its variation on the mean MSE of the last 5 instances and the time interval

4.3 Optimized Performance for Network A

To optimize the performance of the MLP with Network A, we set the number of nodes in the 1st and the 2nd hidden layers to be 5, the momentum to be 0.7, the learning rate to be 0.14, and the number of epochs to be 400. These were the optimal values we found.

Figure 10 shows the learning curve of the optimized MLP used on the training set of Network A. Similar to Figure 2, a logarithmic decrease in MSE was observed as the number of epochs increased. Increasing the number of epochs used in training allows it to learn more complex patterns in the training data and to reduce MSE.

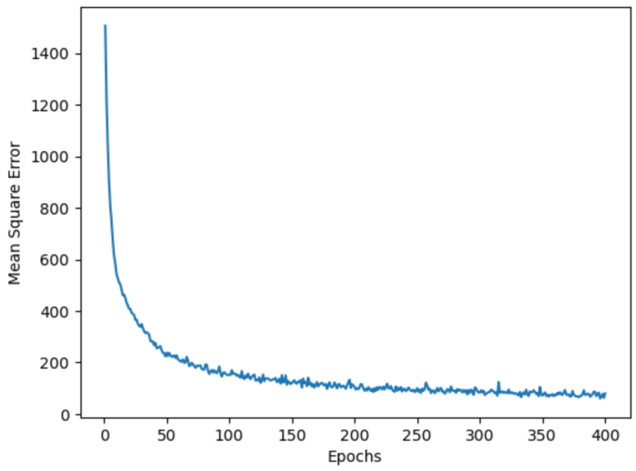


Figure 10: Training MSE per epoch of Network A with optimization

Table 2 shows the confusion matrix of the optimized MLP with the training set of Network A. Here, several instances that were supposed to be class 1 were predicted to be not class 1. This anomaly is due to the application of SMOTE where the number of instances that were initially class 1 was significantly reduced. The model also wrongly predicted the several instances to be class 8.

Table 2: Confusion matrix of optimized training of Network A

		Actual							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Predicted	**1**	400	2	1	0	0	2	1	0
	2	6	402	1	0	0	1	1	1
	3	2	1	403	0	0	0	0	0
	4	3	0	0	389	0	0	0	6
	5	0	0	0	0	408	0	0	0
	6	8	2	0	1	0	368	0	10
	7	7	0	0	0	0	1	383	0
	8	29	4	0	10	0	9	1	337

Figure 11 shows the learning curve of the optimized MLP with the validation set of Network A. Similar to Figure 3, the learning curve for the validation set has more MSE spikes than the learning curve for the training set in Figure 10. The learning also follows a downward trend, indicating the generalizability of the optimized MLP.

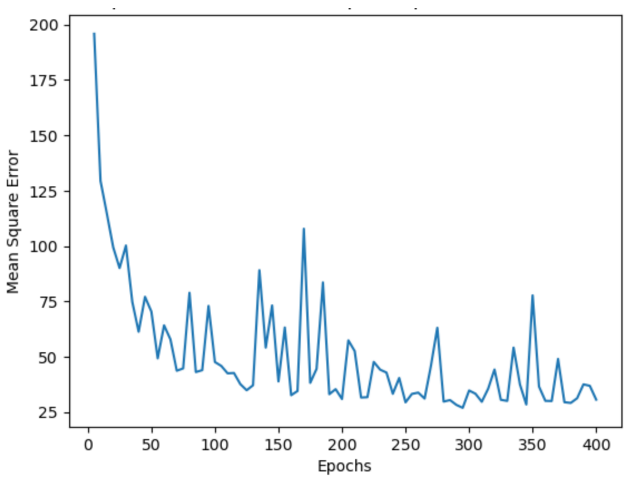


Figure 11: Validation MSE per 5 epochs of Network A with optimization

Table 3 shows the resulting confusion matrix of the optimized MLP with the validation set of Network A. Similar trends in performance were seen in Table 2 and Table 3 where several instances that were class 1 were predicted to be not class 1. Several instances were also wrongly predicted to be class 8.

4.4 Optimized Performance for Network B

The parameters that were investigated to be optimal were used to train MLP on Network B. Same as before, the optimal number of nodes in the 1st and the 2nd hidden layers is 5, the optimal momentum is 0.7, the optimal learning rate is 0.14, and the optimal number of epochs is 400.

Figure 12 shows the learning curve of the optimized MLP used on the training set of Network A. Similar to Figure 2

Table 3: Confusion matrix of optimized validation of Network A

		Actual							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
8*Predicted	**1**	86	1	1	0	0	3	1	1
	2	0	88	0	0	0	0	0	0
	3	2	0	92	0	0	0	0	0
	4	1	0	0	101	0	0	0	0
	5	0	0	0	0	92	0	0	0
	6	7	2	0	0	0	101	0	1
	7	0	0	0	0	0	1	108	0
	8	12	1	0	4	1	5	0	87

and Figure 11, a logarithmic decrease in MSE was observed as the number of epochs increased.

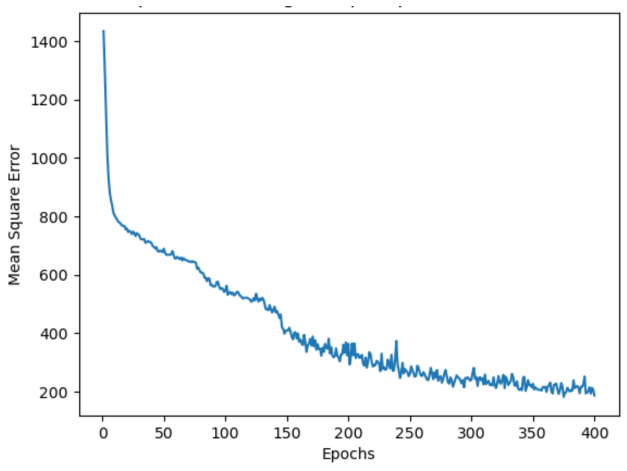


Figure 12: Training MSE per epoch of Network B with optimization

Table 4 shows the resulting confusion matrix of the optimized MLP with the training set of Network B. Similar trends in performance were seen in Table 2, Table 3, and Table 4 where there were several instances that the actual class was 1 but the predicted class was different. Several instances were also wrongly predicted to be class 6 or class 8, where most should be predicted to be class 1.

Table 4: Confusion matrix of optimized training for Network B

		Actual							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
8*Predicted	**1**	391	0	4	2	0	1	4	4
	2	14	388	5	0	0	4	0	1
	3	3	1	401	0	0	0	0	1
	4	11	0	0	379	0	0	0	8
	5	8	0	0	0	398	0	2	0
	6	49	4	0	7	9	300	10	10
	7	24	0	0	0	6	1	358	3
	8	46	2	4	14	1	19	4	300

Figure 13 shows the learning curve of the optimized MLP with the validation set of Network B. Similar to Figure 3 and Figure 11, the learning curve for the validation set has more

Table 5: Confusion matrix of optimized validation for Network B

		Actual							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
8*Predicted	**1**	92	0	2	0	0	0	0	0
	2	2	84	0	0	0	1	0	1
	3	0	0	94	0	0	0	0	0
	4	5	0	0	92	0	0	0	5
	5	1	0	0	0	91	0	2	0
	6	30	1	0	0	3	68	3	6
	7	1	0	0	0	0	0	108	0
	8	12	1	1	6	2	2	3	83

MSE spikes than the learning curve for the training set in Figure 12. The learning curve also has a decreasing trend, implying that the optimized MLP can still be generalized and used on another dataset.

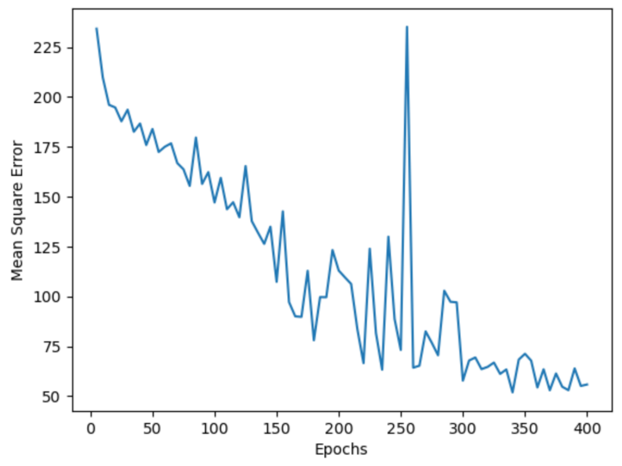


Figure 13: Validation MSE per 5 epochs of Network B with optimization

Table 5 shows the resulting confusion matrix of the optimized MLP with the validation set of Network B. Similar trends in performance were seen in Tables 2 to 5 where several instances should have class 1 but instead got a different class. Same as Table 4, several instances were wrongly predicted to be class 6 or class 8, where most of them should instead be class 1.

5. ANALYSIS AND DISCUSSION OF RESULTS

Table 6 compares the accuracy, precision, recall, f1 score and mcc of the initial and optimized MLP with Network A. For both initial and optimized MLP, the performance metrics are high where the performance metrics are all above 94% for the training and validation of the initial MLP and the validation of the optimized MLP, while above 96% for the training of the optimized MLP. It is also evident that the optimized MLP performed better than the initial MLP for both training and validation in all metrics. This implies that the parameters were optimized successfully.

Figures 14 and 15 compare the learning curve of the initial MLP and the optimized MLP for training and validation of

Table 6: Evaluation and Comparison of Performance of Initial vs Optimized MLP for Network A

Performance Metrics	Initial Network A		Optimized Network A	
	Training Dataset	Validation Dataset	Training Dataset	Validation Dataset
Accuracy	94.38%	94.12%	96.56%	94.38%
Precision	94.40%	94.19%	96.69%	94.61%
Recall	94.31%	94.35%	96.51%	94.69%
F1 Score	94.30%	94.16%	96.53%	94.48%

Network A, respectively. As expected, the optimized MLP performs better in terms of MSE as the number of epochs increased compared to the initial MLP, where the initial MLP has higher peaks of error and reached a smaller minimum MSE. This implies that the parameters were tuned successfully.

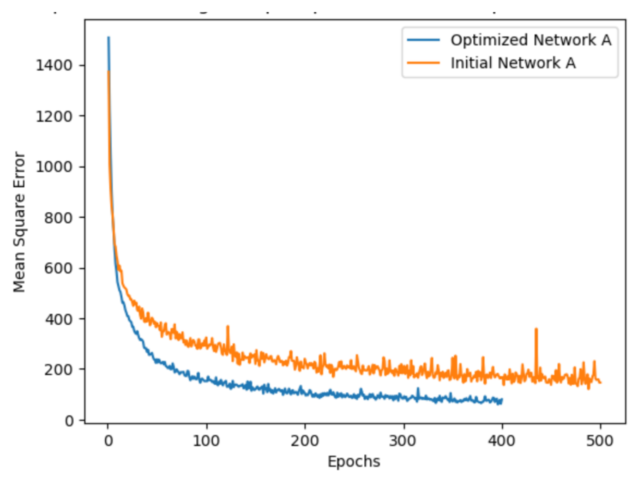


Figure 14: Training MSE per epoch of Initial vs Optimized Network A

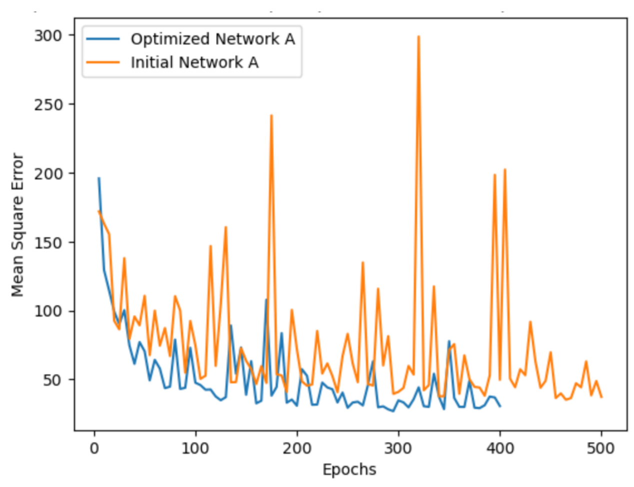


Figure 15: Validation MSE per 5 epochs of Initial vs Optimized Network A

Table 7 compares the accuracy, precision, recall, f1 score and mcc of the optimized MLP for Network A and Network B. Both cases performed well, where the values of all performance metrics are above 96% for the training of the optimized MLP for Network A, above 94% for the validation of the optimized MLP for both networks, above 90% for the training of the optimized MLP for Network B, and above 89% for the validation of the optimized MLP for Network B. It is also evident that the optimized MLP of Network A performed better than the optimized MLP of Network B for both training and validation in all metrics. This is because the tanh function performed better than the ReLu function given the distribution of the input data and the hyperparameters and the network architecture that we selected.

Table 7: Evaluation and Comparison of Performance of Optimized MLP for Network A versus Network B

Performance Metrics	Network A		Network B	
	Training Dataset	Validation Dataset	Training Dataset	Validation Dataset
Accuracy	96.56%	94.38%	91.09%	89.00%
Precision	96.69%	94.61%	92.02%	90.68%
Recall	96.51%	94.69%	90.95%	89.78%
F1 Score	96.53%	94.48%	91.08%	89.25%

Figures 16 and 17 compare the learning curve of the training and validation of the optimized MLP of Network A versus Network B, respectively. The optimized MLP performs better for network A in terms of MSE as the number of epochs increased compared to Network B, where the learning curve of the optimized MLP used on Network B had higher peaks of error and reached a smaller minimum MSE.

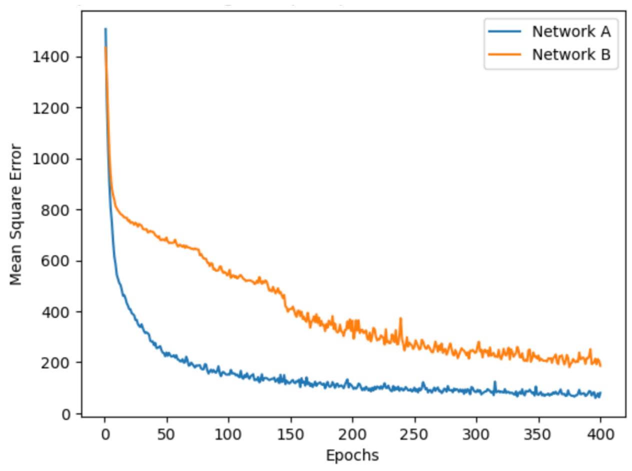


Figure 16: MSE of optimized training per epoch of Network A vs Network B

The MLP used on Network A took 258 seconds without optimization and 440 seconds with optimization because momentum was decreased in favor of decreasing classification errors. The optimized MLP took 441 seconds when used on Network B. The difference in time interval of Network A and Network B is minimal.

6. CONCLUSION

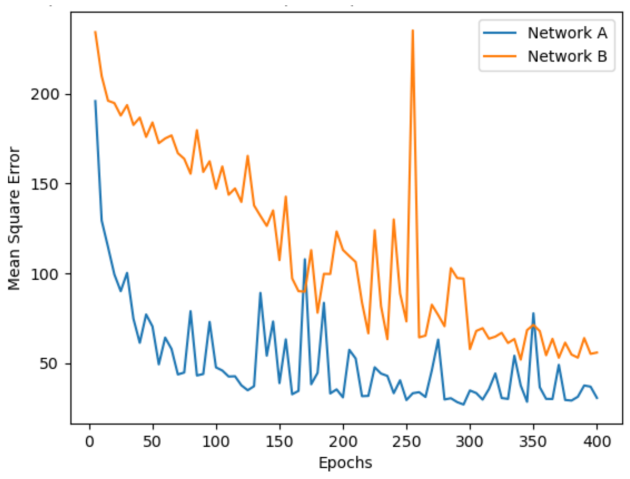


Figure 17: MSE of optimized validation per 5 epochs of Network A vs Network B

Analysis of the backpropagation algorithm of ANN for varying scenarios was performed on a dataset that was applied with SMOTE. Learning curves, confusion matrices, and metrics such as accuracy, precision, recall, and f1 score were used for performance evaluation. By tuning the number of nodes, the momentum, the learning rate, and the number of epochs, we were able to improve the performance of the ANN. Given the hyperparameters used and the network architecture, the tanh function worked better than the relu function as an activation function.

7. REFERENCES

[1] "SMOTE for Imbalanced Classification with Python." Machine Learning Mastery, 17 Mar. 2021, machinelearningmastery .com/smote-oversampling-for-imbalanced-classification/.

[2] Zweiri, Yahya H., James F. Whidborne, and Lakmal D. Seneviratne. "A three-term backpropagation algorithm." Neurocomputing 50 (2003): 305-318.

[3] Leonard, James, and Mark A. Kramer. "Improvement of the backpropagation algorithm for training neural networks." Computers Chemical Engineering 14.3 (1990): 337-341.