Examining Multi-Layer Perceptron Performance

Effects of Depth, Width, and Hyperparameter Optimization on the German Credit Card Dataset

Name: Aishwarya Shekar Babu

Student ID – 23093811

GitHub URL - https://github.com/aishwarya-shekar-babu/MLP-Performance-Analysis

I. INTRODUCTION

In this tutorial, we're looking at how a type of neural network, Multi-Layer Perceptron (MLP) performs when trying to predict whether someone has good or bad credit. We'll use a dataset with financial information to help the MLP make these predictions with various architectural changes in model.

An MLP works by having layers of neurons that process information. It starts with an input layer where the data enters, followed by one or more hidden layers where the network learns patterns, and finally, an output layer that gives us the prediction (good or bad credit). The MLP can learn complex patterns in the data because it uses non-linear functions, and a technique called backpropagation to adjust itself during training.

Further in the tutorial, we will be experimenting with different configurations of the network to see how they affect the model's ability to make accurate predictions. Specifically, we will be testing how the number of layers (depth) and the number of neurons in each layer (width) change the results. We'll also look at how techniques like regularization and hyperparameter tuning can help prevent overfitting and improve the model's ability to generalize to target data.

These types of models are really useful for financial institutions when they need to analyse credit risk. By accurately predicting whether someone is likely to repay a loan, these models help banks and other lenders make smarter, data-driven decisions. The more accurate the model, the better the institution can manage risk and ensure financial.

II. What is Multi-Layer Perceptron?

A Multilayer Perceptron (MLP) is a type of neural network made up of layers that help process and learn from data. It consists of three main types of layers: input, hidden, and output. Each layer plays a specific role in transforming the input data into a final prediction.

- **Input Layer**: This is where the data enters the network. Each neuron in this layer represents one feature.
- **Hidden Layers**: These layers are in the middle, between the input and output. Each neuron in a hidden layer takes the input, applies weights if the dataset is unbalanced and uses an activation function which decides how to process the information.

- **Output Layer**: This is the last layer that gives the result. It might predict something like "yes" or "no" in a classification task.

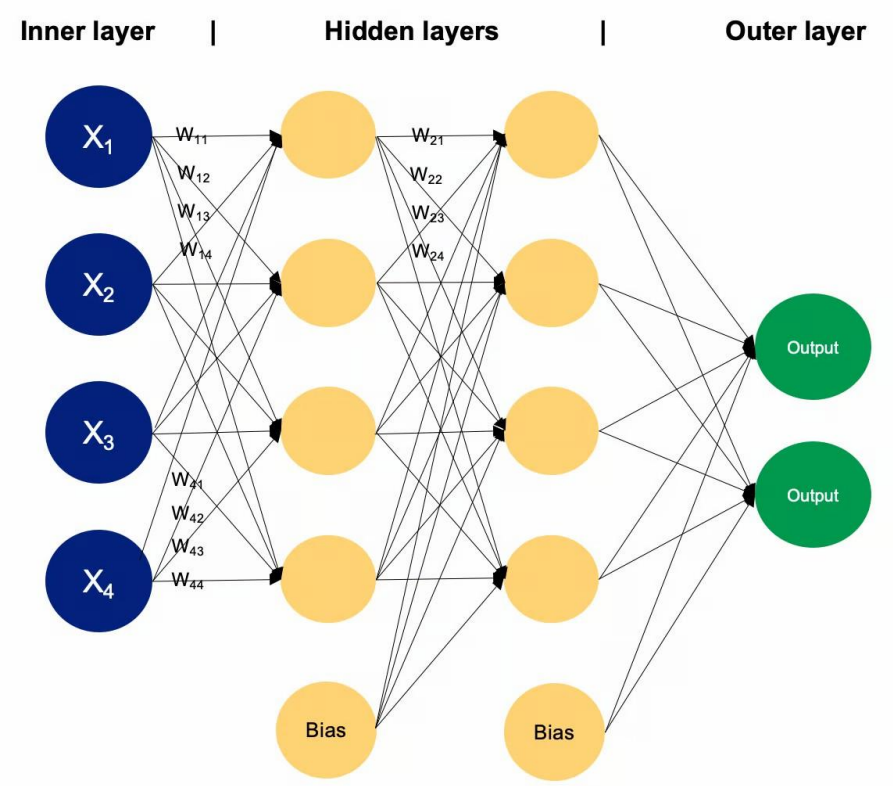


Figure 1: The MLP Architecture diagram

III. PERFORMANCE METRICS.

In the tutorial, we use the below performance metrics to analyse the performance of model

Accuracy – This is the most straightforward measure. It tells you the percentage of correct predictions out of all predictions made

Precision – Precision focuses on how many of the positive predictions made by the model are actually correct.

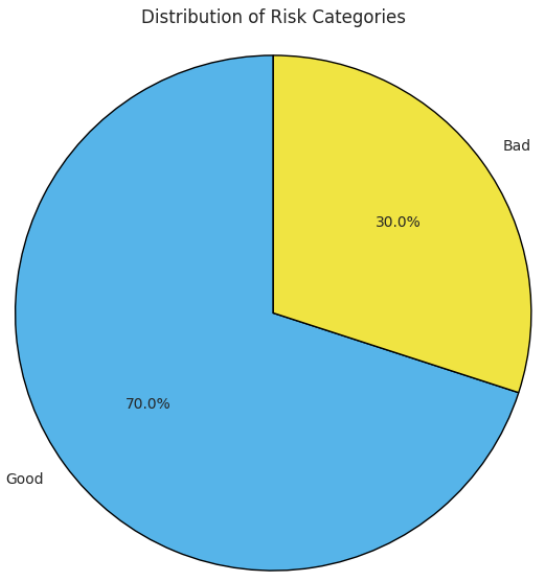
Recall – Recall is about how many of the actual positive cases were correctly identified by the model

F1-Score – The F1-Score combines precision and recall into one number to give a better sense of the model's performance when there's a trade-off between precision and recall

ROC-AUC (Receiver Operating Characteristic - Area Under Curve) – The ROC curve is a graph that shows how well the model distinguishes between positive and negative classes by plotting True Positive Rate (Recall) against False Positive Rate.

IV. Data preprocessing:

The dataset German credit was obtained by Kaggle, it includes both Categorical and numerical columns with more than 1000 rows, the column named “Risk” is identified to be target variable. Upon, Checking further by plotting pie chart to find out the distribution of each category,
The “good” is 70% where as “bad” is 30%, which appears to be unbalanced data, which has to be fined tuned before proceeding with building the model



Data Cleaning & Preprocessing

- Dropped unnecessary columns.
- Filled missing values with "**unknown**".
- Converted target variable (Risk) to binary: "good" → 0, "bad" → 1.

Feature Engineering

- Separated features (X) and target (y).
- Identified categorical and numerical columns.

Encoding & Feature Scaling

- Since MLPs can't process categorical data directly, Applied ***One-Hot Encoding (OHE)*** to categorical features, converting them into binary vectors (0s and 1s) to avoid incorrect numerical relationships.
- Standardized numerical features using ***StandardScaler()*** to ensure balanced feature distribution.

Train-Test-Validation Split

- dataset: 60% training, 20% validation, 20% testing.
- Stratified split for class distribution.
- Introducing a random seed ensures that the random processes involved in training the MLP, such as weight initialization or data shuffling, produce the same results each time the model is run. This helps with reproducibility, allowing experiments to be consistent and comparable across different runs.

Handling Class Imbalance

- Computed class weights to address imbalance (70% Good, 30% Bad).
- Assigned higher weight to minority class (Bad).

```
# Compute class weights
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(y_train), y=y_train)

# Convert class weights to integers or round the values
class_weights_dict = {i: float(weight) for i, weight in enumerate(class_weights)}

print("Class Weights:", class_weights_dict)
```

```
Class Weights: {0: 0.7142857142857143, 1: 1.6666666666666667}
```

Figure 3: The code snippet for class weights assigning

V. MLP MODEL ANALYSIS WITHOUT HYPERTUNING:

```
# Function to build a simple MLP model (No dropout, regularization, or early stopping)
def build_simple_mlp(depth=2, width=64):
    model = keras.Sequential()
    model.add(layers.Input(shape=(X_train.shape[1],)))  # Input Layer

    # Add hidden layers (Simple Dense Layers with ReLU)
    for _ in range(depth):
        model.add(layers.Dense(width, activation='relu'))

    # Output layer for binary classification
    model.add(layers.Dense(1, activation='sigmoid'))

    # Compile model
    model.compile(optimizer=optimizers.Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Figure 4: Code snippet of function to build MLP without any hyperparameter tuning

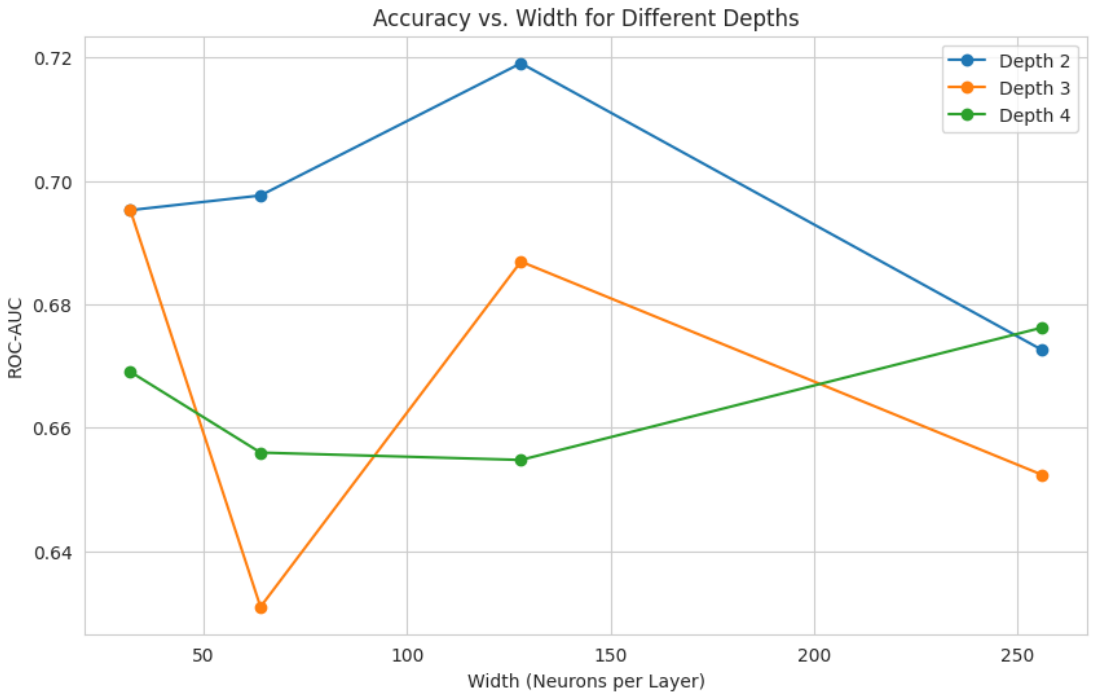
This script is designed to **analyse how an MLP works without hyperparameter tuning**. It explores how the number of layers (**depth**) and neurons per layer (**width**) affect performance but does **not fine-tune** settings like **learning rate, dropout, or regularization.**

1. **Building the Model:** The MLP consists of an **input layer**, multiple **hidden layers with ReLU activation**, and an **output layer with sigmoid activation** for classification. The model's architecture changes based on the chosen depth and width.
2. **Training & Evaluating:** The model is trained using **fixed settings** (50 epochs, batch size of 32) and tested on unseen data. Key performance metrics like **accuracy, precision, recall, F1-score, and ROC-AUC** are calculated to understand how the model performs.
3. **Experimenting:** Different MLP structures are tested (**depths = [2, 3, 4]**, **widths = [32, 64, 128, 256]**) to see how varying layers and neurons impact performance. Since no hyperparameter tuning is applied, the results purely reflect how the architecture affects learning.

PERFORMANCE EVALUATION:

	Depth	Width	Accuracy	Precision	Recall	F1-Score	ROC-AUC
0	2	32	0.740	0.564516	0.583333	0.573770	0.695238
1	2	64	0.750	0.586207	0.566667	0.576271	0.697619
2	2	128	0.740	0.555556	0.666667	0.606061	0.719048
3	2	256	0.675	0.470588	0.666667	0.551724	0.672619
4	3	32	0.740	0.564516	0.583333	0.573770	0.695238
5	3	64	0.710	0.520000	0.433333	0.472727	0.630952
6	3	128	0.735	0.557377	0.566667	0.561983	0.686905
7	3	256	0.720	0.537037	0.483333	0.508772	0.652381
8	4	32	0.710	0.515152	0.566667	0.539683	0.669048
9	4	64	0.745	0.604651	0.433333	0.504854	0.655952
10	4	128	0.730	0.560000	0.466667	0.509091	0.654762
11	4	256	0.740	0.574074	0.516667	0.543860	0.676190

Figure 5: Performance metrics of impact of depth and width



Impact of width and depth

The plotted ROC-AUC values against neuron width for different depths provide further insights:

1. **Depth 2 shows a peak performance at width 128, then declines**, suggesting that increasing the width beyond a certain point leads to diminishing returns.
2. **Depth 3 exhibits a dip at width 32**, recovers at width 128, but drops again at width 256. This indicates that deeper models may require tuning to find an optimal width.

Conclusions

- **Increasing depth does not necessarily improve performance unless width is tuned correctly.**
- **Overfitting might be a concern for deeper models with lower performance at smaller widths.**

VI. MLP MODEL ANALYSIS HYPERTUNING:

```python
# Function to build an MLP model with dropout & L2 regularization

def build_mlp(depth=2, width=64, l2_lambda=0.01, dropout_rate=0.3):
    model =  keras.Sequential()
    model.add(layers.Input(shape=(X_train.shape[1],)))  # Input Layer (Automatically derived from X_train)

    # Add hidden layers with L2 regularization & dropout
    for _ in range(depth):
        model.add(layers.Dense(width, activation='relu', kernel_regularizer=regularizers.l2(l2_lambda)))
        model.add(layers.Dropout(dropout_rate))  # Dropout for regularization

    # Output layer for binary classification
    model.add(layers.Dense(1, activation='sigmoid'))

    # Compile the model
    model.compile(optimizer=optimizers.Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Figure 6: Code snippet of function to build MLP with hyperparameter tuning

This script is designed to analyze how an MLP performs when hyperparameters like learning rate, dropout, and L2 regularization are fine-tuned. The objective is to explore the impact of different tuning strategies on model performance and generalization.

1. **Building the Model:** The MLP consists of an input layer, multiple hidden layers with ReLU activation, and an output layer with a sigmoid activation function for binary classification. The architecture allows for tuning depth, width, dropout, and L2 regularization.
2. **Training & Evaluating:** The model is trained using adaptive settings, where hyperparameters such as learning rate, dropout rate, and L2 regularization strength are varied. The training runs for 50 epochs with a batch size of 32, and key performance metrics like accuracy, precision, recall, F1-score, and ROC-AUC are calculated. The best-performing configuration (Depth=4, Width=32) is selected.

3. **Experimenting:** Different MLP structures are tested with hyperparameter tuning applied. Depths ([2, 3, 4]) and widths ([32, 64, 128, 256]) are examined to assess performance improvements when optimized settings are used.

	Depth	Width	Accuracy	Precision	Recall	F1-Score	ROC-AUC
0	2	32	0.730	0.534091	0.783333	0.635135	0.745238
1	2	64	0.725	0.528090	0.783333	0.630872	0.741667
2	2	128	0.700	0.500000	0.716667	0.589041	0.704762
3	2	256	0.715	0.517241	0.750000	0.612245	0.725000
4	3	32	0.735	0.542169	0.750000	0.629371	0.739286
5	3	64	0.710	0.512195	0.700000	0.591549	0.707143
6	3	128	0.715	0.517241	0.750000	0.612245	0.725000
7	3	256	0.720	0.525641	0.683333	0.594203	0.709524
8	4	32	0.745	0.555556	0.750000	0.638298	0.746429
9	4	64	0.740	0.552632	0.700000	0.617647	0.728571
10	4	128	0.710	0.512195	0.700000	0.591549	0.707143
11	4	256	0.735	0.544304	0.716667	0.618705	0.729762

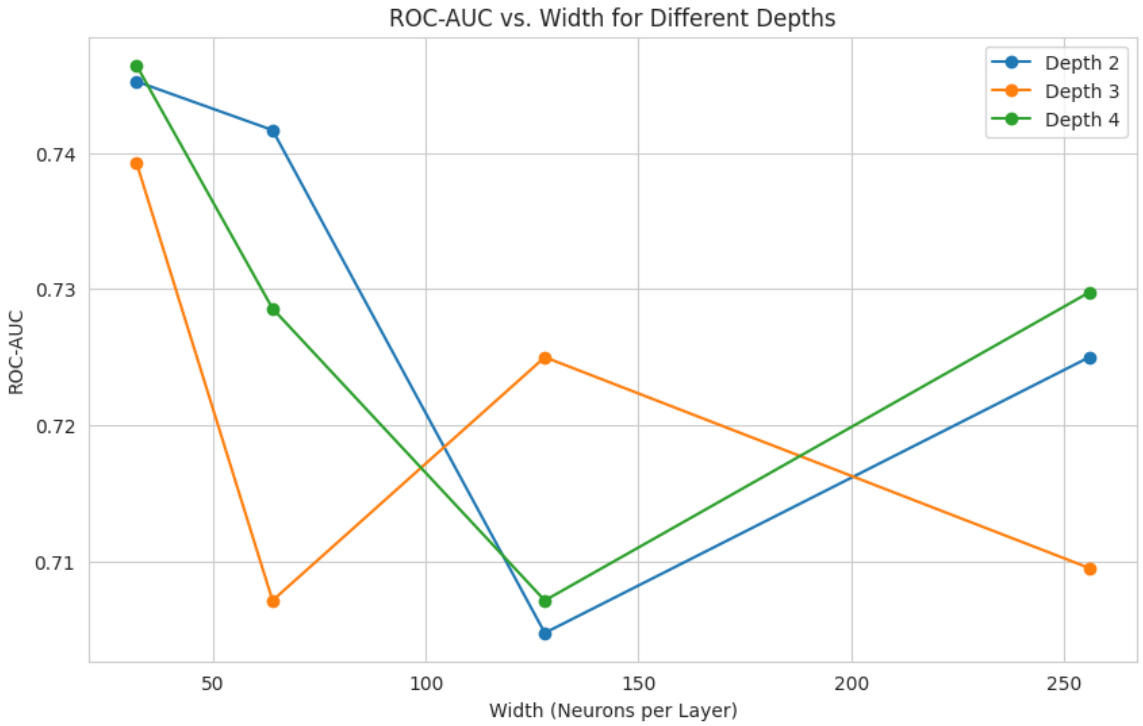


Figure 7: Performance metrics of impact of depth, width, and tuning

PERFORMANCE EVALUATION

Impact of Hyperparameter Tuning on Width and Depth

The plotted ROC-AUC values against neuron width for different depths, considering hyperparameter tuning, provide deeper insights:

1. Depth 2 shows improved stability compared to the untuned model, with consistent performance across different widths, especially around 128 neurons.
2. Depth 3 exhibits reduced fluctuations compared to the untuned version, showing steady improvement at width 128 and 256, demonstrating the benefits of optimized settings.
3. Depth 4 maintains better performance at higher widths, suggesting that deeper models require both width expansion and appropriate tuning to generalize well.

Conclusions

- Hyperparameter tuning significantly stabilizes model performance, reducing drastic drops observed in untuned models.
- Increasing depth yields better performance only when regularization and dropout are optimized.
- Overfitting is mitigated when L2 regularization and dropout are correctly applied, especially for deeper models.
- The optimal width depends on depth, but mid-sized widths (100-150 neurons) still show inconsistent performance, emphasizing the need for careful tuning.

Actionable Takeaways for MLP Model Analysis & Hyperparameter Tuning

- Start with smaller widths (e.g., 32-64 neurons) and gradually increase while monitoring ROC-AUC.
- Avoid mid-sized widths (~100 neurons) without tuning, as they tend to reduce performance consistency.
- Increase width for deeper models (200+ neurons works well for Depth 4) but with appropriate regularization.
- Apply L2 regularization and dropout strategically to prevent overfitting.

VII. Hyperparameter Tuning: Heatmap Insights

The first heatmap visualizes Accuracy across different Depth and Width values.

The second heatmap displays ROC-AUC, assessing the model's ability to distinguish between classes.

- Accuracy Trend: Higher depth (4 layers) generally improves accuracy, peaking at Depth = 4, Width = 32 (0.7450).
- ROC-AUC Trend: The highest ROC-AUC score (0.746) is observed at Depth = 4, Width = 32, confirming an optimal balance between sensitivity and specificity.

- Trade-offs: Increasing depth beyond a certain point (e.g., Depth 4, Width 256) does not always improve performance and may lead to diminishing returns.

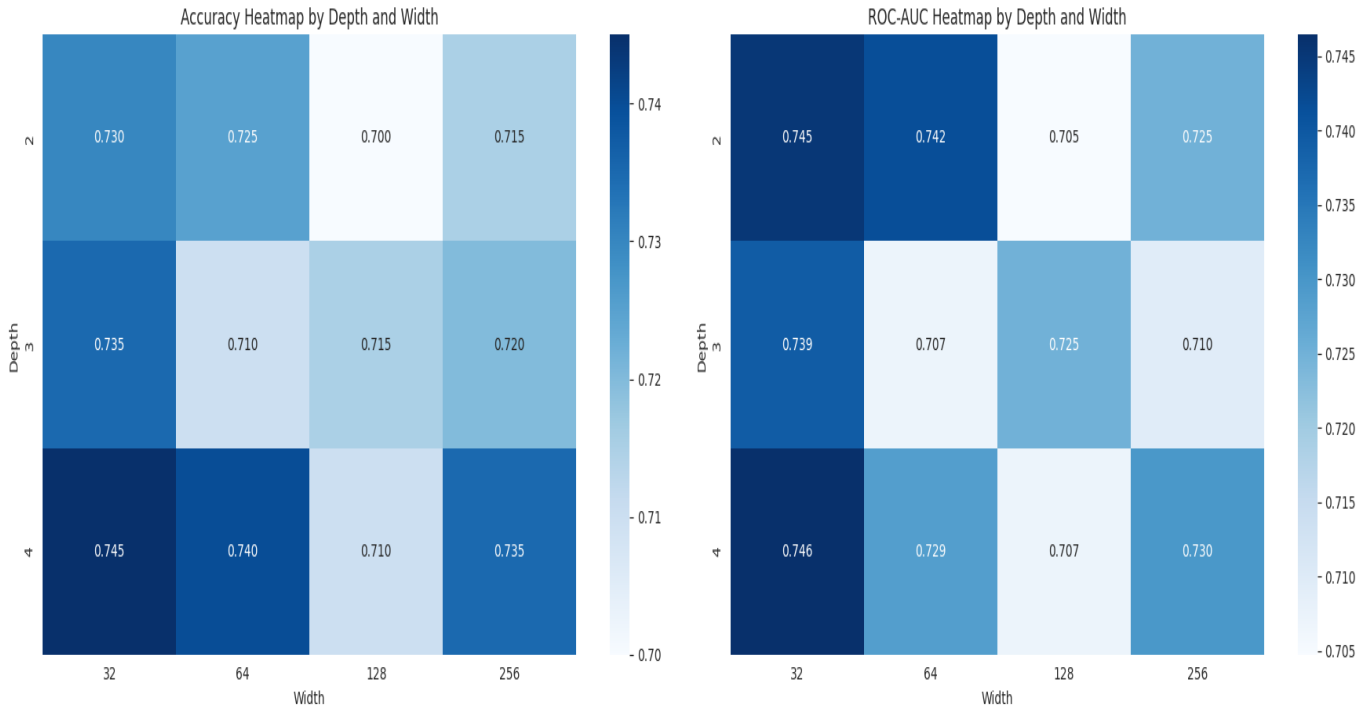


Figure 8: Heatmap analysis

VIII. Confusion Matrix Insights

- True Negatives (104): Correctly classified as Class 0 (Non-Event).
- False Positives (36): Incorrectly classified as Class 1 (Event) when they were actually Class 0.
- False Negatives (17): Misclassified as Class 0 when they were actually Class 1.
- True Positives (43): Correctly identified as Class 1 (Event).

Observations

- The false positive rate (36 misclassifications) is relatively high, meaning normal cases are sometimes misclassified as events.
- The false negative rate (17 misclassifications) indicates that some actual events are missed, though the recall (0.7167) remains reasonably high.
- The balance between precision (0.5443) and recall (0.7167) suggests that the model prioritizes detecting actual events over avoiding false positives.

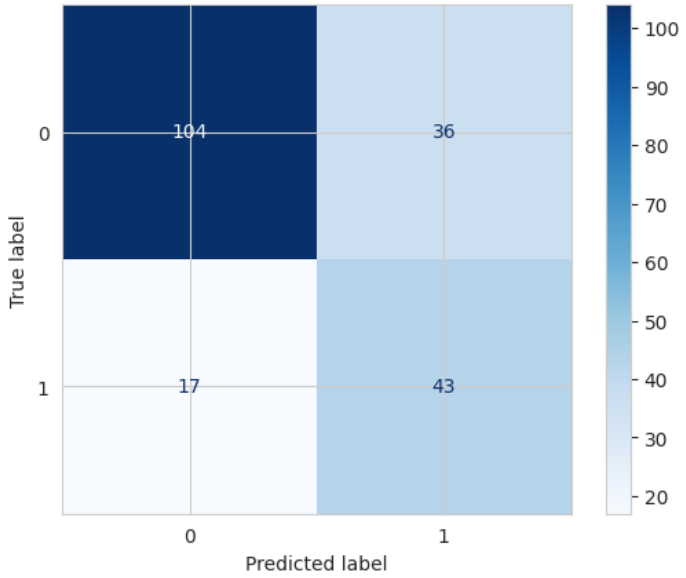


Figure 9: The confusion matrix for best fit model

IX. Final Conclusion

The best-performing model (Depth=4, Width=32) achieves **0.745** accuracy, **0.555** precision, **0.638** recall, and **0.746429** ROC-AUC.

The model prioritizes recall (detecting actual positive cases) at the cost of precision (more false positives), making it suitable for applications where missing positive cases is costly.

X. Real Time Applications using MLP

- **Instant Loan Decisions** – MLP enables real-time credit scoring, allowing banks to approve or reject loans within seconds.
- **Better Fraud Detection** – Identifies high-risk applicants quickly, reducing financial losses due to fraud or defaults.
- **Scalability** – Can handle large volumes of applications simultaneously, improving efficiency for financial institutions.

References:

Kaggle Dataset: https://www.kaggle.com/datasets/kabure/german-credit-data-with-risk/data

Multilayer Perceptrons in Machine Learning: A Comprehensive Guide

https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning

Investigation and improvement of multi-layer performance

Investigation and improvement of multi-layer perceptron neural networks for credit scoring - ScienceDirect

Hyperparameter Optimization with Factorized Multilayer Perceptron. schilling2015-ecml.pdf