Build and Implementation of Multilayer Perceptron using MNIST and MPG Public Datasets.

Vimal Thomas Joseph
02/26/2025

Abstract:

With the help of computation advancements, building neural networks have been greatly simplified to a fewer number of code lines. However, to understand the integral parts of how such a network operates, it is necessary to construct a model from the basics. In that context, we have attempted to build a multilayer perceptron model using basic python programming and tested the MLP model using publicly available datasets such as MPG (Miles per Gallon) and MNIST (renowned digit classification problem).

We have found that the MLP works as expected and the model accommodates hyper parameter tunings such as epochs, batch generation techniques, data splitting mechanisms, learning rate adjustments, dropout rate adjustments and turning RMSProp technique on and off using a boolean switch design.

We were able to successfully test the MLP model on both the datasets under different configurations. MPG dataset allowed us to record 0.00227 training loss and 0.065 validation loss while MNIST dataset provided overall test accuracy of 96.78%.

Methodology

We implemented MLP in three phases. 1) Building blocks 2) Training Process and 3) Supported Model Types.

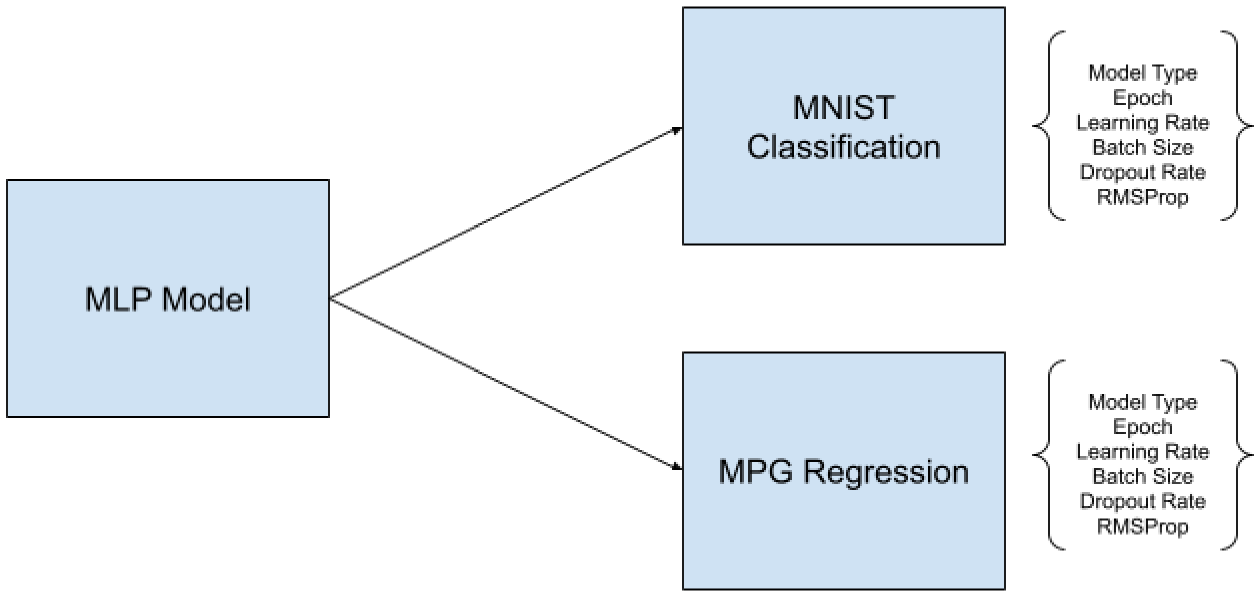
Building Blocks: In this phase, we created classes for activation functions, loss functions, layer handlings and a multilayer perceptron class to orchestrate all the forward passing and backward propagation tasks. Our MLP supports Sigmoid, Tanh, Relu, Softmax, Linear and Mish activation functions. We also support Squared Error and Cross Entropy loss functions.

Training Process: MLP model training process includes batch generation step to divide training data into multiple batches before forward propagating through layers. The model as expected forward propagates through layers, calculates losses and then backward propagates the error before updating the weights and biases based on the gradient descent process. It is important to mention that the MLP supports both RMSProp and Dropout parameters. The model also includes two helper functions each calculating evaluation metrics depending on the type of the task (classification, regression).

Model Types: As mentioned above, we implemented the model to support both classification and regression tasks. We introduced a parameter model_type to handle both types. Depending on the model type, the evaluation metrics are calculated separately. Also, the model assumes that class instantiation handles the type of activation functions and loss functions are chosen appropriately.

Design

We created the MLP model as a separate executable notebook (ipynb) and hosted on github. The MLP model is then instantiated in individual python scripts (which are also available in both py and ipynb formats). Individual model scripts handle feature engineering, model training and testing tasks which are detailed below.



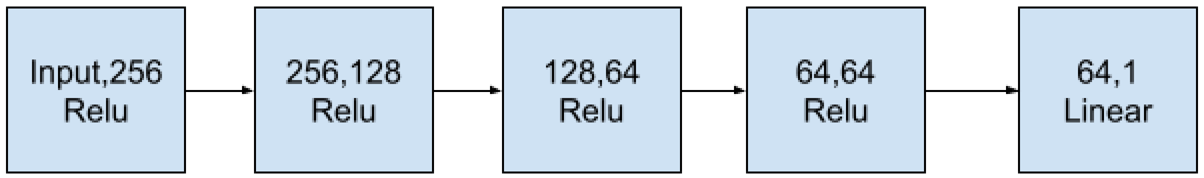
MNIST and MPG Architecture:

MNIST is a classification model and MPG is a regression model and both use MLP as their neural network base layer.

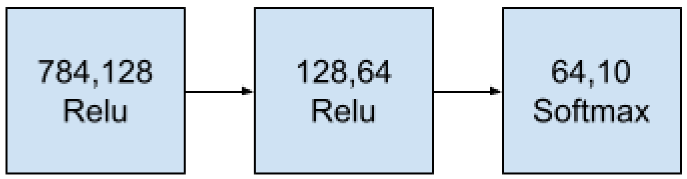
MNIST Model demonstrates the process of classifying handwritten digits from the MNIST dataset using a manually implemented Multilayer Perceptron (MLP). The dataset is downloaded from Kaggle and is normalized to range of 0 and 1 for feature engineering. The dataset is also split into training, validation and testing datasets for model training and validation. An input layer with 784 neurons (representing the 28x28 pixel images) is fed into two hidden layers with 128 and 64 neurons, respectively, using the ReLU activation function and dropout regularization. Finally, an output layer with 10 neurons (representing the 10 digits), using the Softmax activation function is predicted.

MPG model aims to predict miles per gallon (MPG) of cars using a Multilayer Perceptron (MLP) model. The code utilizes the Auto MPG dataset from the UCI Machine Learning Repository.

The data is split into 70%,15%,15% for training, validation and testing respectively. Standardization is applied on the features using mean and the standard deviation of the training data.



MPG

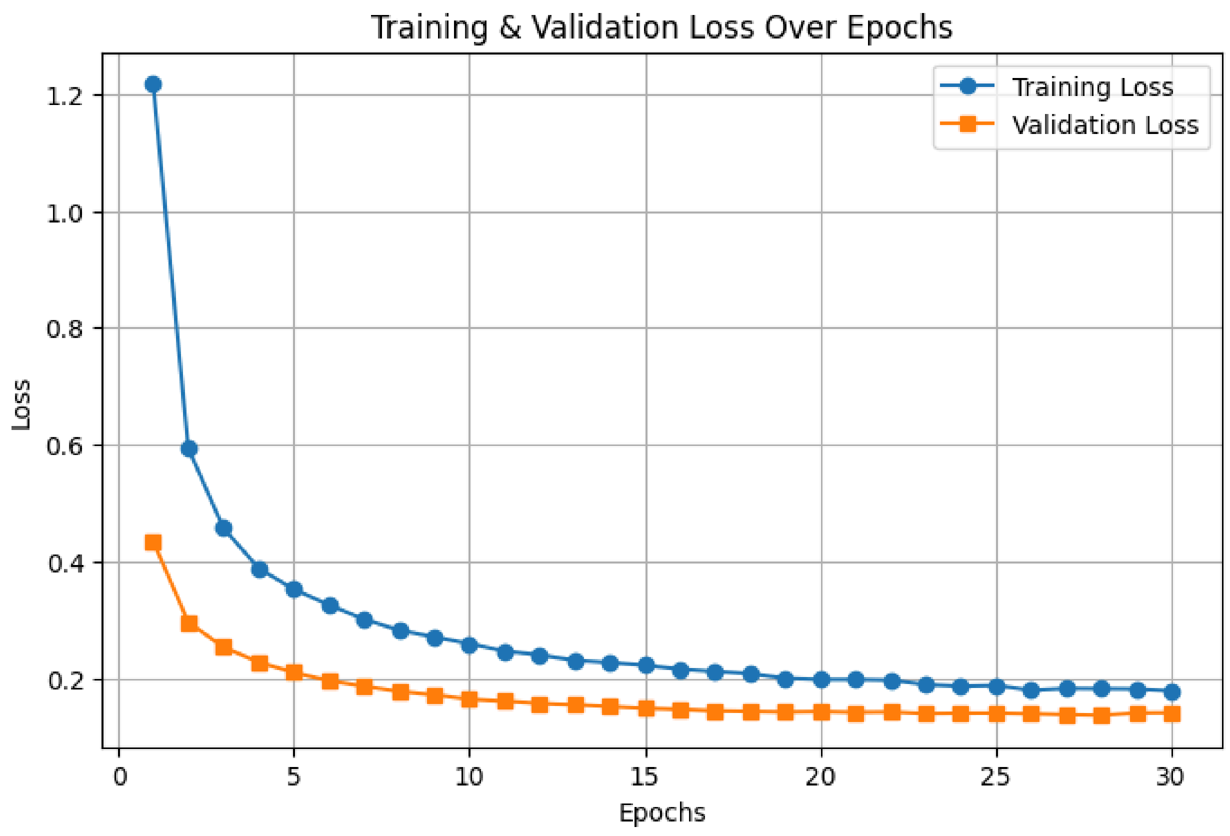


MNIST

Results

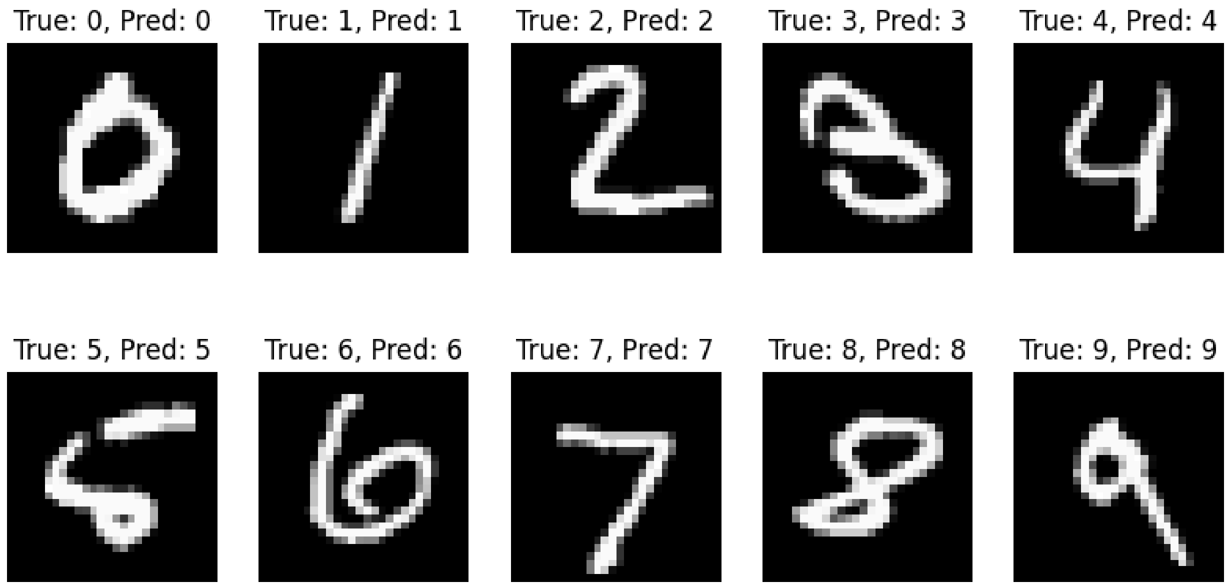
MNIST:

The model is trained for different hypertuning combinations and achieved upper 90 percentile accuracy in the beginning. However,the model started overfitting when RMSProp and dropout properties were introduced. Then, we increased the dropout rate to 0.5 and decreased the learning rate to 0.0001 to achieve the current performance. The model overfit was evident from the fact that training loss was decreasing while validation loss was gradually increasing. Below output shows the final training and validation losses.

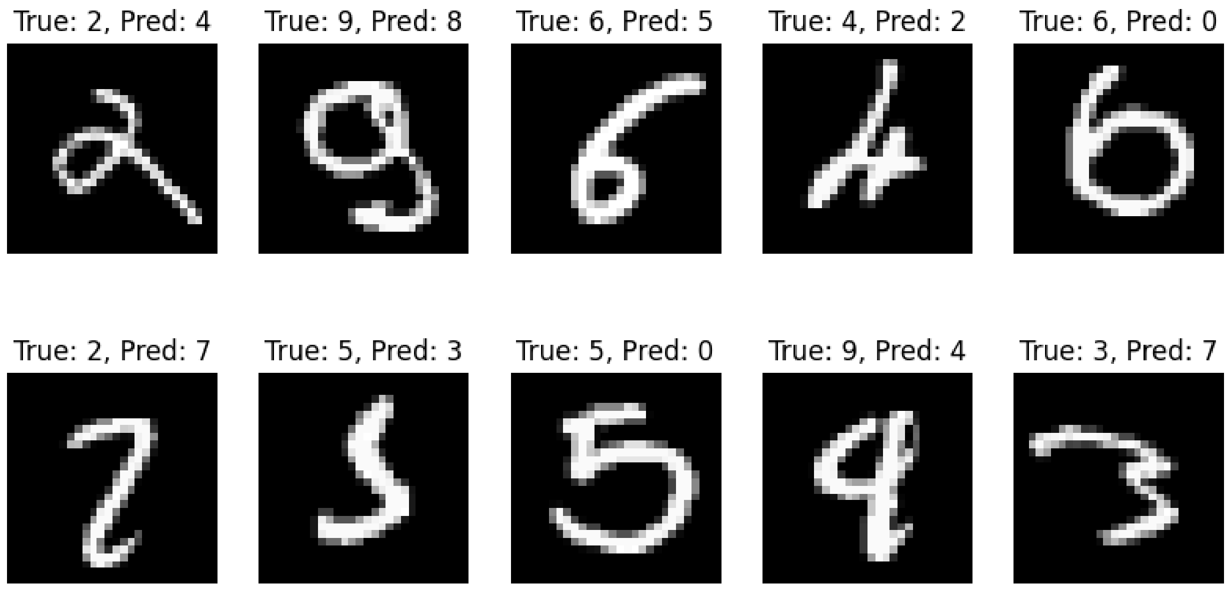


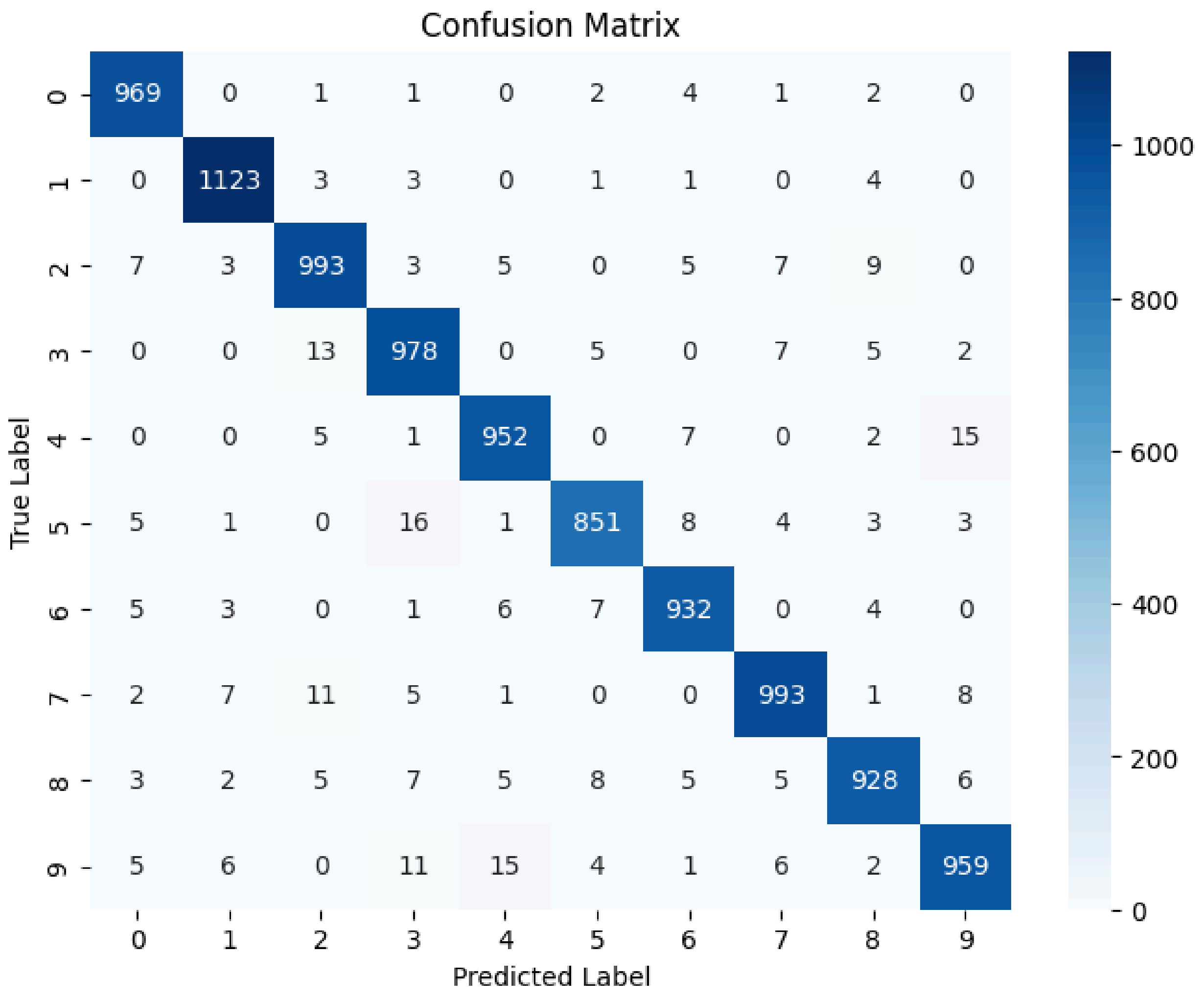
The model archives a good overall accuracy of 96.78% and predicts the images as expected. Misclassified images often exhibit characteristics that make the prediction difficult (sample attached). The result set contains individual accuracy of each digit which is also reflected in the confusion matrix provided.

Correctly Classified Images (One per Digit)



Misclassified Images



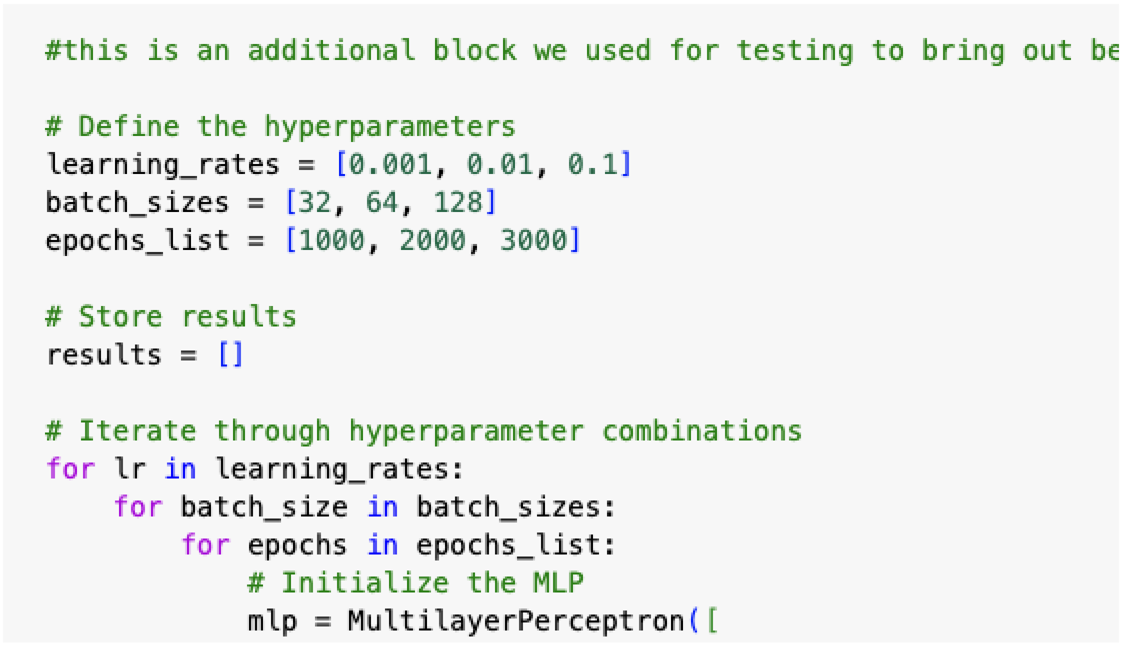


MPG:

Similarly, for the MPG model, the feature engineering was a difficult task. Initially, We created cross features between horsepower, cylinders, acceleration and displacement to see if the combined features provide higher correlation for MPG prediction. We also introduced minmax scaling to the numerical features and applied one hot encoding on the boolean features.

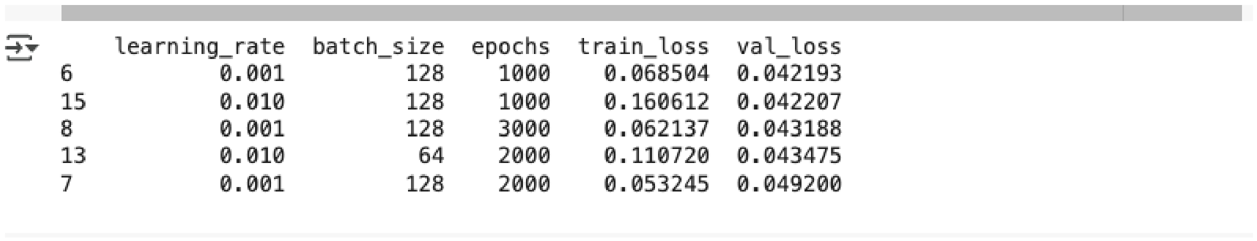
However, these feature engineering efforts did not yield better performance than the regression model example given. Hence, removed the additional features and kept the feature engineering basics such as standardization using the mean and the standard deviation method.

To train the MPG model using MLP, we introduced a hypertuning mechanism where we grouped learning rates, batch sizes and epoch lists so that the training was done using this collection. The result is stored for future reference. This process is very similar to experiments in Databricks Model Serving process. The snapshot below shows the additional setup in the MPG model.

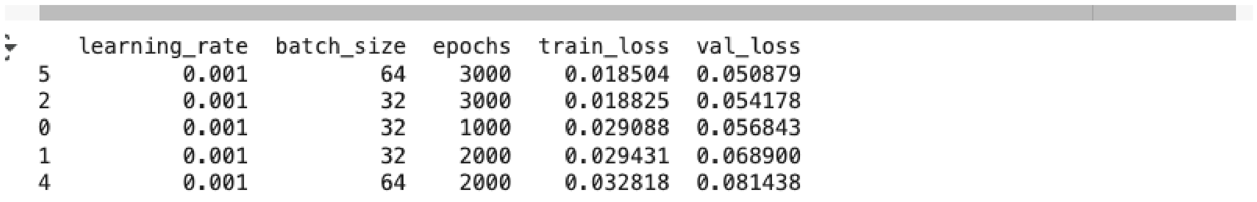


```
#this is an additional block we used for testing to bring out be

# Define the hyperparameters
learning_rates = [0.001, 0.01, 0.1]
batch_sizes = [32, 64, 128]
epochs_list = [1000, 2000, 3000]

# Store results
results = []

# Iterate through hyperparameter combinations
for lr in learning_rates:
    for batch_size in batch_sizes:
        for epochs in epochs_list:
            # Initialize the MLP
            mlp = MultilayerPerceptron([
```

Based on the experiments, we chose the desired hyperparameters and applied them to the MPG model. The snapshot below shows the top 5 result sets based on training loses and validation loses.

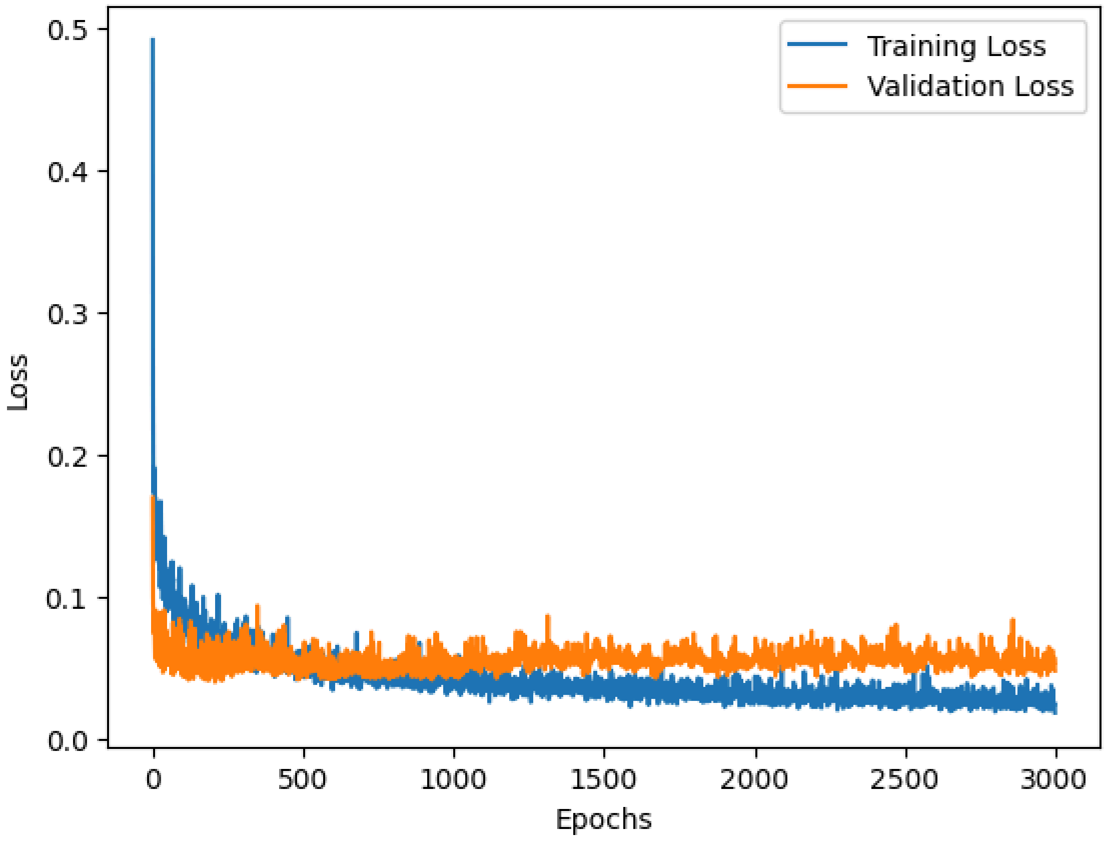


```
     learning_rate  batch_size  epochs  train_loss  val_loss
 6           0.001         128    1000    0.068504  0.042193
 15          0.010         128    1000    0.160612  0.042207
 8           0.001         128    3000    0.062137  0.043188
 13          0.010          64    2000    0.110720  0.043475
 7           0.001         128    2000    0.053245  0.049200
```



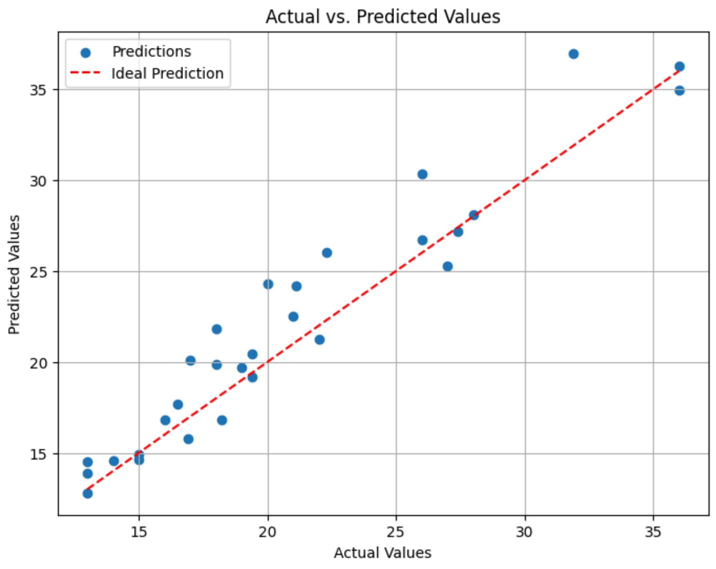
```
    learning_rate  batch_size  epochs  train_loss  val_loss
 5          0.001          64    3000    0.018504  0.050879
 2          0.001          32    3000    0.018825  0.054178
 0          0.001          32    1000    0.029088  0.056843
 1          0.001          32    2000    0.029431  0.068900
 4          0.001          64    2000    0.032818  0.081438
```

After applying the desired hypertuning parameters, the MPG model started performing well. However, we observed that even a slightest change of learning rate or the drop rate introduced varied loss curves. Finally, the model is set for a 0.001 learning rate with a epochs of 3000 and batch size of 64 and achieved 0.0179 training loss and a 0.0465 of test loss.



Following table shows the predicted and actual values for the MPG model and we observed that most of the values are pretty close to the actual values. However, there is a spread and variance for some of the predicted values.

	Predicted	Actual
0	13.906656	13.0
1	22.510850	21.0
2	19.194222	19.4
3	14.636901	15.0
4	14.912963	15.0
5	26.055179	22.3
6	21.849788	18.0
7	27.164241	27.4
8	36.985292	31.9
9	19.699472	19.0



Conclusion:

This activity enabled us to understand the core components of the neural networks and how the hidden layers contribute to solving real world problems and how different activation functions work and how backward propagation self corrects the weights and biases for a better prediction and accuracy.