Multilayer Perceptron on Regression and Classification

Abstract: This project implements a Multilayer Perceptron (MLP) from scratch and tests it on two different datasets: classifying handwritten digits from the MNIST dataset and vehicle mileage prediction using the Auto MPG dataset. The model achieves an accuracy of 90.30% on the MNIST dataset and a total test loss of 0.9593 for vehicle MPG prediction. For digit classification, the MLP has a dense layer of six layers using ReLU activation and Softmax in the output layer. The regression Auto MPG model, combines ReLU, Mish and Linear activations to predict vehicle mileage.

Methodology: My MLP engine entirely using NumPy by focusing on key aspects of deep learning such as forward propagation, backpropagation, activation functions, loss functions, optimization, and weight updates. I have selected mini-batch generator technique to divide the training dataset into smaller batches to make more efficient and improve generalization. Instead of using all training data at once, mini batches allow the model to update weights iteratively, reducing computational overhead and improves convergence. For my activation functions, I have chosen non-linearity to help the model to learn complex patterns. I have implemented activation functions such as Sigmoid, Tanh, ReLU, Softmax, Linear, Softplus and Mish. To measure the difference between model predictions and actual values, I have implemented loss functions such as Mean squared error and cross-entropy. Mean squared error works great for regression problem and cross entropy works great for classification tasks. To implement the layer class, each layer consists of weights (w), biases (b), activation function, dropout rate which helps to prevent overfitting by randomly deactivating neurons during training. The layer class has forward and backward method. The forward method computes the linear transformation and applies to activation function. The backward method of layer computes gradients of the activation function and updates weights and biases based on computed gradients. Weights of each layer initializes using Glorot (Xavier) Initialization to ensure balanced weight distribution and biases in each layer is initialized to zero. As activation functions brings non-linearity, it applies non-linear function to layer outputs. I have implemented dense layer for my MLP classification task which also has forward and backward propagation. For the Multilayer perceptron class, it has forward propagation, backward propagation, predict and train method. The forward propagation (forward(x)) passes input through each layer. Backward propagation takes input as loss gradients and input data and computes weight and bias gradients using the chain rule and passes errors backward through the layers. The train method handles mini batch gradient descent updates and iterates each weight updates using small batches. The RMSprop uses running averages of squared gradients to adapt learning rates. In each epoch, the model trains on training data and computes training and validation losses for performance. While training in each epoch, it shuffles training data

to avoid learning order bias. Each layer includes weights, biases, and an activation function, with Xavier initialization which ensures stable learning.

The MLP models were designed based on the requirements of each dataset: for MNIST focuses on classification and for Auto MPG focuses the model on regression. To train my model for the auto MPG dataset, I converted the input data to NumPy arrays as my MLP for mpg is also structured for NumPy arrays to compute numerical values efficiently. The MLP architecture is designed with multiple hidden layers, each using different activation functions and dropout rates to prevent overfitting while training. The first layer passes the input features into 256 hidden layers of neurons using ReLU activation and second layer with Mish activation function and for the next two hidden layers with ReLU activation function. The final layer uses Linear as activation function which is necessary for regression tasks. The model is trained, and loss function I have used for regression task is mean squared error with RMSprop optimization. The training process runs for total of 50 epochs by using mini batch of size 16, which helps stabilize updates and prevent local minima issues. After 50 epochs of training sessions, the model is evaluated on the test dataset and computed the total test loss to identify the model performance. Lower loss value indicates better performance.

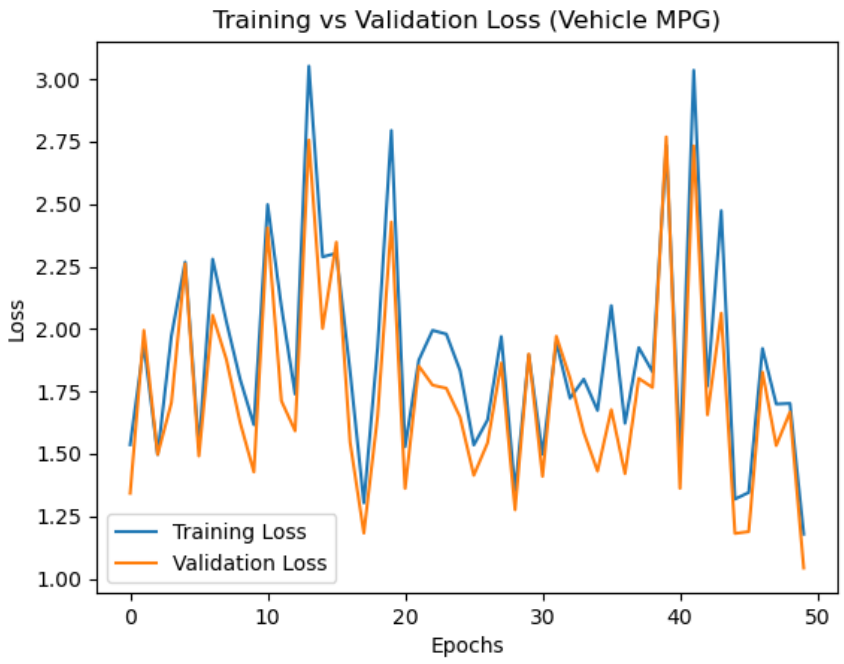
MLP model for the MNIST dataset which consist of 28x28 grayscale handwritten digit images. To start with the model architecture, I have normalizes the pixel values by dividing with 255.0 which ensures all values will be between [0,1] range. The images are flattened from 28x28 matrix to 784-dimensional vector for further processing. For classification tasks, the loss function I have used is cross entropy which expects labels in a one hot encoded format which means each label has to be represented as a vector of length equal to the number of classes (MNIST has 10 classes, number 0-9). This method of using one hot encoding converts each class in binary vectors allows the model to compute gradients efficiently and works best for cross entropy. The MLP architecture consists of six fully connected dense layers each using ReLU activation function. The final layer uses Softmax activation function which is the compatible with cross entropy and classification tasks. The model is trained for 50 epochs with a batch size of 64 and learning rate of 0.001. The model computes the total test accuracy by comparing the predicted labels with the true labels.

Initially, the MLP was not generating good results for the MNIST dataset. The model accuracy was 10% which is very less as random guess will also give the same accuracy level. After reading multiple sites, I have added dense layer with forward and backward propagation. After adding dense layer, the MLP model generated quite well accuracy of 80%-90% depending on the hyperparameters. Also, adding Mish in activation function for regression tasks was giving Nan values depending on the learning rate. If the learning rate is too high or too low, the training and validation loss becomes nan.

Results:

Vehicle MPG Regression Dataset:

- Total Testing Loss: 0.9593
- Loss Curve:

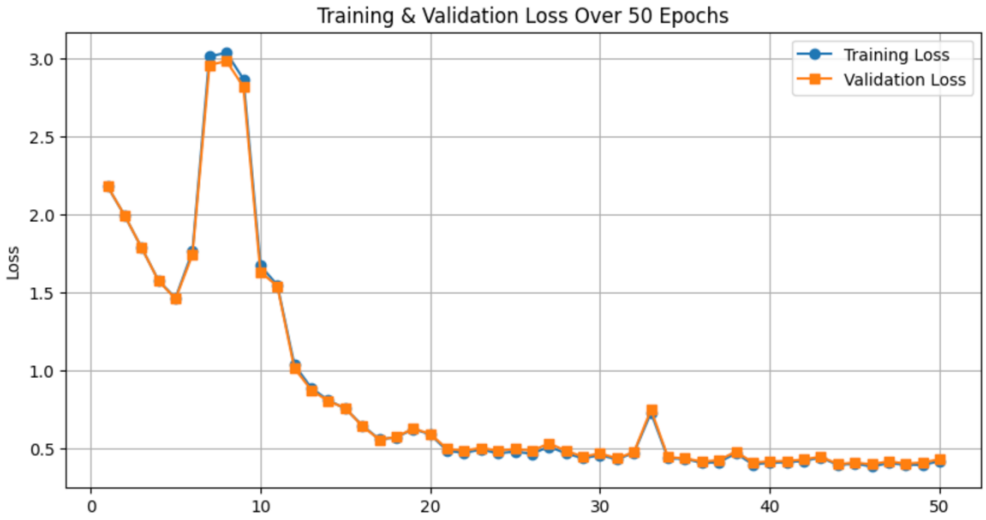


- True vs. Predicted MPG (10 Samples):

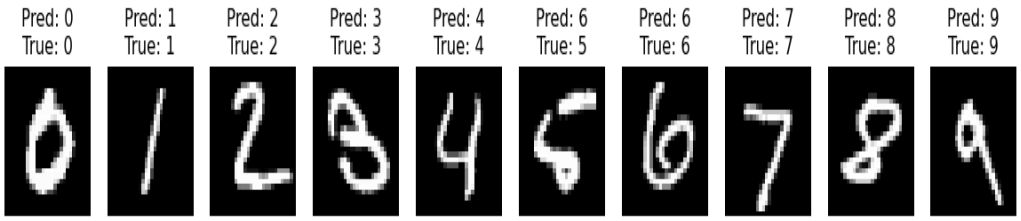
TRUE MPG	PREDICTED MPG
-0.45	-0.37
-0.95	-0.44
0.42	0.09
-1.33	-0.80
-0.33	-0.15
-0.53	-0.32
-1.08	-0.82
-1.08	-0.74
-0.17	-0.09
-0.70	0.04

MNIST Classification Dataset:

- Accuracy on Full Testing Dataset: 90.30%
- Loss Curve:



- Example Predictions:



Code Repository: [GitHub Repo Link]

Discussion & Conclusion: for vehicle MPG regression, the total testing loss was 0.9593 which indicates the model predictions aligned well with the actual MPG values. Also, the training and validation loss curve shows significant fluctuations for 50 epochs. The spikes in the loss curve explains how the model performance is different significantly due to small dataset or hyperparameter. The table shows difference between the true mpg and predicted mpg. It shows the model is predicting quite well but there's still scope of improvement. For the MNIST dataset, the final accuracy was 90.30%, which is quite reasonable for a basic MLP architecture. The loss curve shows the models adaptability with the MNIST dataset. To conclude, the vehicle MPG dataset showed good results, but we can improve the test loss with more fine tuning of the MLP architecture.