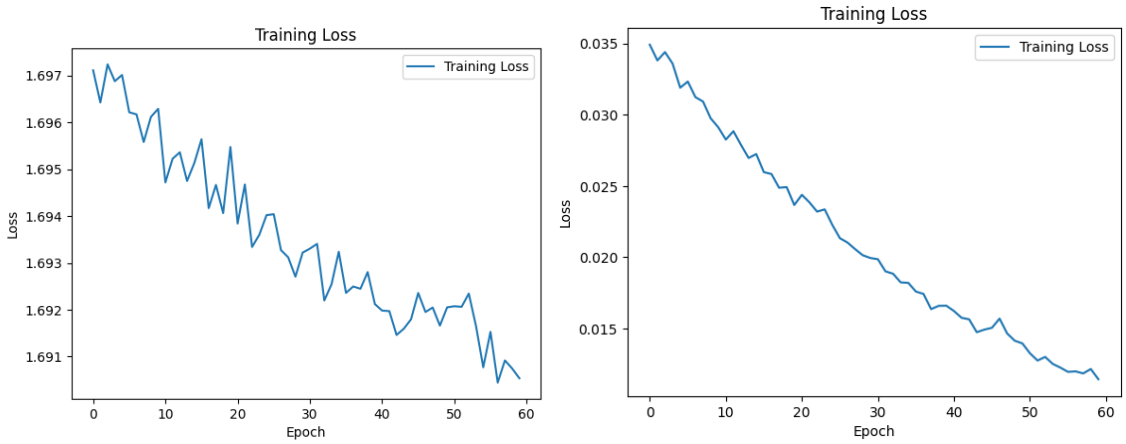
The topology of the networks being investigated consists of 3 layers the input layer, a single hidden layer and an output layer. The input layer is flattened into a vector of size 784 (28 x 28 pixels). The hidden layer consists of a single fully connected(dense) linear layer (nn.Linear) with 60 nodes. The output layer is another fully connected linear layer (nn.Linear) that produces the final output logits for each of the 10 classes (digits 0 to 9). After the hidden layer, a Rectified Linear Unit (ReLU) activation function is applied (nn.ReLU). This neural network architecture is called a feedforward neural network.

Adding the ReLU (Rectified Linear Unit) activation function after the linear layer introduces non-linearity into the model. This non-linearity allows the neural network to learn more complex relationships in the data, which can help improve the model's ability to capture intricate patterns and make more accurate predictions.

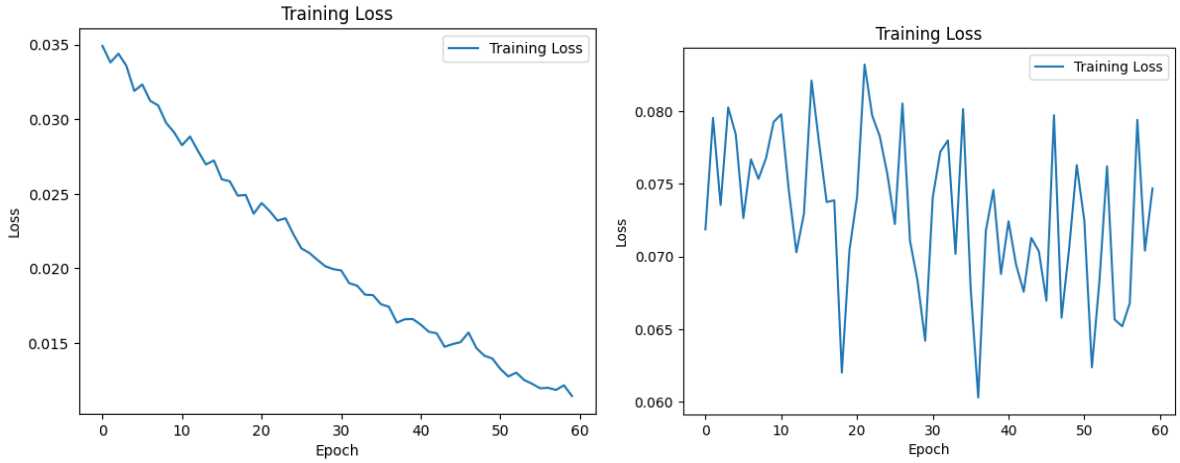


The graph on the left has only 2 layers. It consists of linear layer (nn.Linear) followed by a softmax activation function applied to its output. On the contrary, the graph on the right has 3 layers. It represents the model, clearly, it can be observed that ReLu (non-linear) activation function is better than softmax (linear). Unlike softmax the losses produced are below 0.035 – the losses approach 0 much more faster. Therefore, the model uses ReLu for the rest of the experiment.

Preprocessing steps are performed on the training set. The dataset is normalized before being passed into the neural network. The pixel values are floating-point numbers and divided by 255.0 to scale them to the range [0,1]. The dataset is further reshaped. The images are originally stored as 28x28 matrices. Each image is reshaped into a 1D vector of size 784 using the ‘reshape’ function. Neural networks operate on flattened vectors as inputs. This transformation ensures that each image is represented as a feature vector, making it compatible with the input layer of the neural network.

The model uses the cross-entropy loss function (nn.CrossEntropyLoss). It is well-suited for multi-class classification in our case the MNIST dataset involves classifying handwritten digits into one of ten possible classes (digits 0 through 9). It can handle situations where each input sample belongs to exactly one class, making it appropriate for the MNIST classification task where each image corresponds to a single digit. It aims to make its predicted probabilities for the correct classes as close as possible to 1 and for the incorrect as close as possible to zero leading to accurate classification.

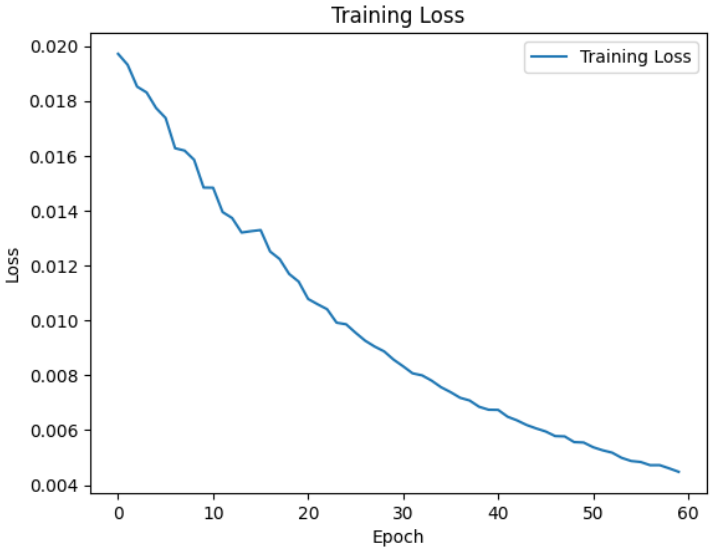
The model uses Stochastic Gradient Descent with Momentum (torch.optim.SGD). SGD is a variant of the standard stochastic gradient optimizer. It incorporates momentum, which accelerates SGD by adding a fraction of the update vector of the past time step to the current update vector. This momentum term helps out the oscillations in the parameter updates and accelerates convergence, especially in noisy gradients or saddle points. The learning rate (specified by the 'lr' parameter) determines the step size for parameter updates.



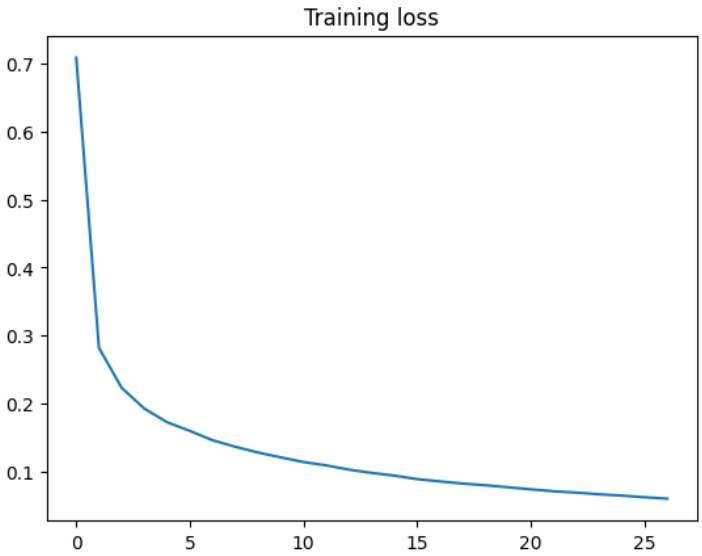
In our model shown on the left, the learning rate is kept constant at 0.001, unlike the graph on the right which uses AdamW another type of optimizer. In this graph, AdamW adapts the learning rates for each parameter individually based on the first and second moments of the gradients. It maintains separate adaptive learning rates for each parameter. In the graph, it clearly can be seen that the loss fluctuates at a higher degree. Therefore, in the model, we use SGD throughout the experiment.

Unlike SGD, AdamW also separates the weight decay (L2 regularization) from the optimization step, ensuring that weight decay affects all parameters uniformly after the parameter updates. In our case, SGD incorporates weight decay in the training loop. Same as the learning rate the regularisation strength is kept constant at 0.001.

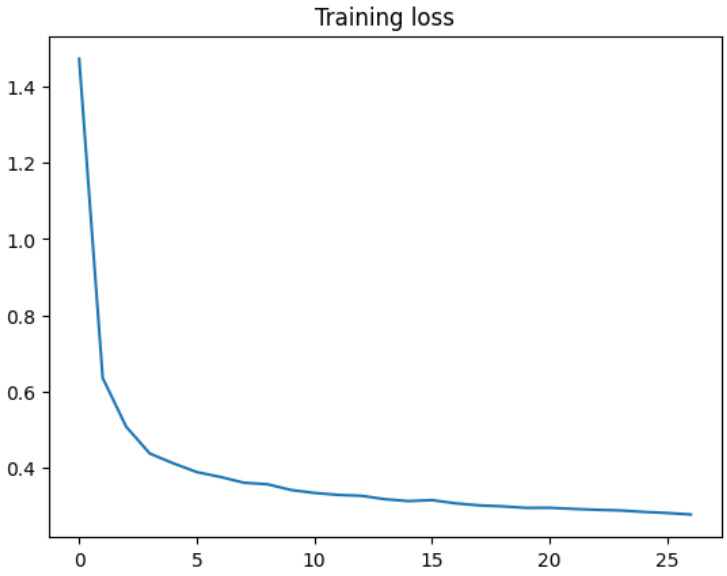
The graph below shows improvement after applying L2 regularisation in the training loop. In the previous training loss graph recall all losses were below 0.035, in the graph below all losses are below 0.020, and by observing the curve unlike the previous graph it is much smoother. The graph is approaching 0 much faster than before it even reaches the low of 0.004.



The model further applies early stopping criteria in the model to prevent overfitting and to determine the optimal number of training epochs. The curve below shows the results of applying early stopping. The curve is less steep than before showing that the loss decreases much faster.



To experiment another regularization technique was used instead of L2 regularization still applying the early stopping criteria above to check the performance. The dropout regularization method was applied where different the dropout values were used. The graph below shows the result of using a low dropout value of 0.2.

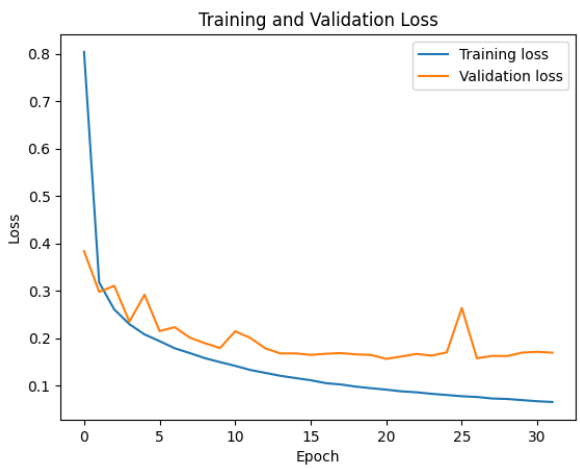


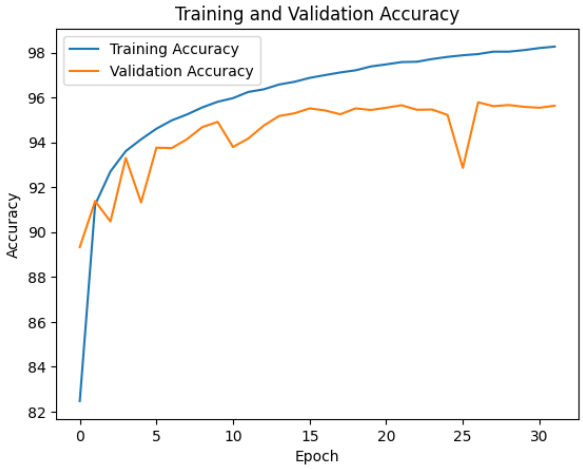
The model's performance did not improve even when trying different dropout values. The dropout value 0.2 was used because low dropout values tend to give better results but it can be observed that the model did not perform better than using L2 regularization. In the graph above unlike in L2 regularization, the loss value does not even reach the lows below 0.1.

During the training phase, the model is trained on the training dataset. The training data is fed into the model in mini-batches using a data loader ('train_loader'). For each mini-batch, the input data is passed through the model, and predictions are obtained. The loss is calculated using the cross-entropy loss function, which compares the model's predictions with the true labels. L2 regularization is applied to the loss by adding a regularization term penalizing the weights. Gradients of the loss for the model parameters are computed using backpropagation. The optimizer SGD updates the model parameters based on the computed gradients to minimize the loss. This process is repeated for multiple epochs (specified by 'num_epochs').

In the validation phase after each of training, the model's performance is evaluated on a separate validation dataset. The validation data is fed into the model in mini-batches using a validation data loader ('val_loader'). For each mini-batch, the input data is passed through the model to obtain predictions. The loss is calculated using the same cross-entropy loss function as in the training phase, without applying L2 regularization. The average validation loss and accuracy are computed over all mini-batches to evaluate the model's performance on unseen data.

Throughout the training process, the model's performance on the validation set is monitored. The model parameters yielding the best validation accuracy are saved and used to plot the graphs below.





The graph on the right clearly shows that the model's training accuracy is approximately 98% and the validation accuracy is approximately 95%. Training loss is approximately 0.1 while validation loss is approximately 0.2. These are not huge differences.

In summary, these are the final parameters used:

Parameter	**Value**
Batch size	64
Learning rate	0.001
The number of nodes at the input layer	784
The number of nodes at the hidden layer	60
The number of nodes at the output layer	10
Regularization strength	0.001
Activation function	ReLU, Linear
Optimizer	AdamW
epoch	60
Early stopping	5

The model uses 60 nodes for the hidden layer before using this value 30 nodes were used and the performance decreased – loss values were high. The batch size is 64 – different batches were used like 100 and the model did not show any significant changes because the batches are created randomly.

Overall the performance of the feedforward neural network gives reasonable accuracy to predict the input digit. Even though for some input image digits it gives incorrect classification, a 95% accuracy is still efficient for image classification.