Project

Training Neural Networks for Handwritten Digit Recognition

Neural Networks

Alia Medhat Mohamed | Artificial Intelligence|20221440735

Malak Mahmoud Aref | Artificial Intelligence|20221445867

Faculty of Computers and Data Sciences,

Alexandria University.

May 2023

Table of Contents

Introduction

PyTorch is a popular open-source machine learning framework that provides tools and libraries to build and train deep neural networks. In this project, we will be using PyTorch to build a neural network for recognizing handwritten digits from the MNIST dataset. The MNIST dataset is a widely used benchmark dataset for image classification tasks. It consists of 60,000 28x28 grayscale images of handwritten digits from 0 to 9, with 48,000 images in the training set and 12,000 images in the validation set. Our goal is to build a neural network that can classify these images into their respective digit categories with high accuracy.

Importing Packages

Will import all the needed packages we will use in the project

- numpy → is a library for numerical computing in Python.
- matplotlib → is a popular library to plot graphs in Python.
- pandas → is a Python package providing fast, flexible, and expressive data structures
- Sklearn (short for scikit-learn) → is a library for machine learning in Python. It provides tools for data preprocessing, feature selection, model selection, and model evaluation, among other things.
- Torchvision → is a package in the PyTorch framework that provides datasets and transforms for computer vision tasks, such as image classification.
- Math → is a built-in Python module that provides mathematical functions and constants.
- Torch → is a popular open-source machine learning framework that is used for building neural networks. It provides tools for tensor computation.
- torch.nn → is a subpackage of PyTorch that provides tools for building neural networks, including various types of layers (e.g., convolutional layers, recurrent layers, etc.) and loss functions.
- torch.optim → is a subpackage of PyTorch that provides tools for optimizing the parameters of a neural network during training, using techniques like stochastic gradient descent (SGD) and its variants.
- torch.nn.functional → is a subpackage of PyTorch that provides functional versions of neural network layers and activation functions, which can be used in building neural networks.

Loading MNIST Dataset:

- We can load the dataset Using Torchvision.datasets which can contains multiple built-in datasets.
- To import our dataset , it needs to be called from the torch datasets using `torchvision.datasets.MNIST`.

Dataset

- The MNIST dataset contains 48000 x 784 training examples of handwritten digits 1.
    - Each training example is a 28-pixel x 28-pixel grayscale image of the digit.
        - Each pixel is represented by a floating-point number indicating the grayscale intensity at that location.
        - Each training examples becomes a single row in our data matrix X_train.

$$X_{train} = \begin{pmatrix} ---(x^{(1)})--- \\ ---(x^{(2)})--- \\ \vdots \\ ---(x^{(m)})--- \end{pmatrix}$$

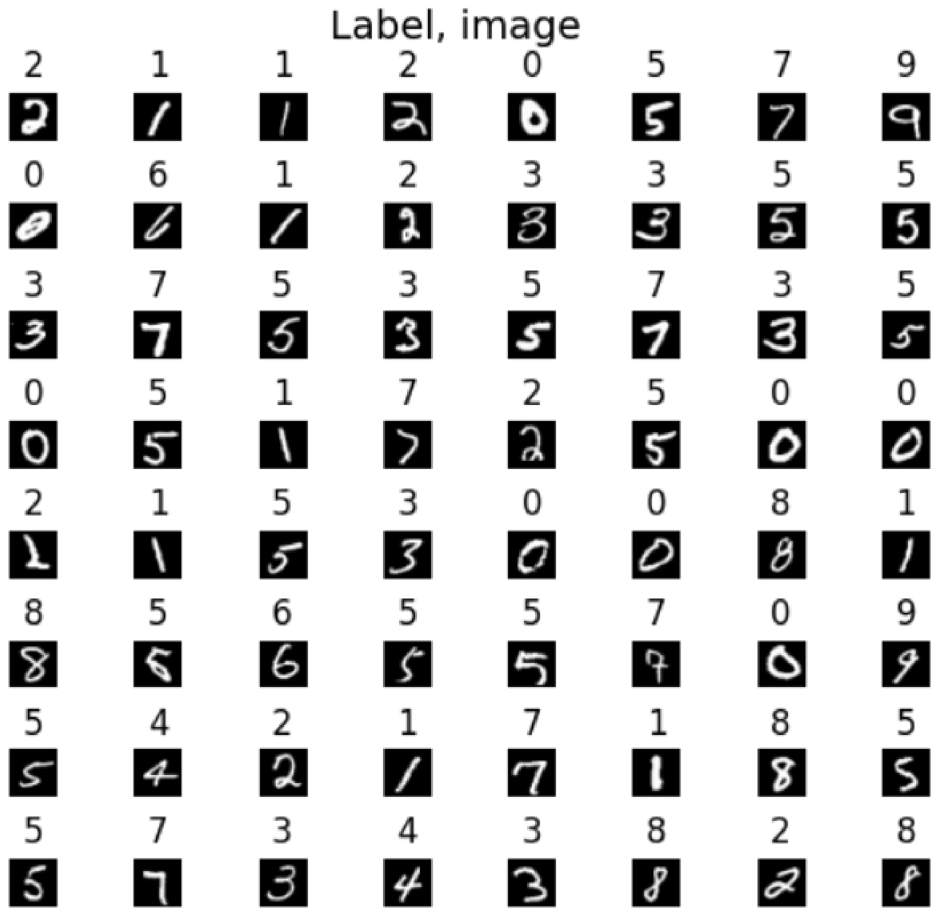
- The second part of the training set is a 12000 x 784 dimensional vector y_train that contains labels for the training set
    - y = 0 if the image is of the digit 0, y = 4 if the image is of the digit 4 and so on.

Start by loading the data from the CSV file using the `read_csv` function from the Pandas library. We then split the data into input features, `X`, and target labels, `y`. We normalize the input features by subtracting their mean and dividing by their standard deviation. Finally, we use the `train_test_split` function from the Scikit-learn library to randomly split the data into training and validation sets, with the validation set being 20% of the total data. The `random_state` parameter ensures that the split is reproducible, while the `test_size` parameter controls the proportion of the data that goes into the validation set. We print the shapes of the resulting arrays to verify that the data has been split correctly.

```
Training set shapes:
X_train:(48000, 784)
y_train(48000,)
Validation set shapes:
X_val:(12000, 784)
y_val(12000,)
```

Data Visualization

We will begin by visualizing a subset of the training set by randomly generating a grid of 8 rows and 8 columns of subplots using 'plt.subplots'. The size of the figure is set to 5 inches by 5 inches. The 'tight_layout' function is used to adjust the spacing between the subplots and ensure that there is no overlap. The code then enters a loop that iterates over all the subplots in the grid using 'axes.flat'. For each subplot, a random index is generated using 'np.random.randint(m)', where 'm' is the number of samples in the training set. The corresponding image from the training set is then reshaped into a 28 by 28 pixel matrix and displayed in grayscale using 'ax.imshow'. The label of the image is displayed as the subplot title using 'ax.set_title'. Finally, the 'ax.set_axis_off()' function is used to remove the x and y axis labels from the subplots, and the 'fig.suptitle' function is used to set the title of the entire figure.



Building Neural Network

MLP Function

First, define a multi-layer perceptron (MLP) neural network that inherits from the 'nn.Module' class in PyTorch. An MLP is a type of feedforward neural network that consists of one or more layers of fully connected neurons.

Constructor method for the MLP class

Then define the constructor method for the MLP class. It takes two arguments 'num_hidden1' and 'num_hidden2' to specify the number of neurons in the first and second hidden layers, respectively. We then created three 'nn.Linear' objects for the three layers of the MLP: 'self.layer1', 'self.layer2', and 'self.layer3'. The first layer has 28*28 input neurons and 'num_hidden1' output neurons, the second layer has 'num_hidden1' input neurons and 'num_hidden2' output neurons, and the third layer has 'num_hidden2' input neurons and 10 output neurons. We also store the values of 'num_hidden1' and 'num_hidden1' as attributes of the MLP object.

Forward Method

Then define the 'forward' method, first we flatten the input image with the 'view' method which reshapes the tensor to have a single dimension of size 28*28. We then pass the flattened tensor through the first layer 'self.layer1' and apply the ReLU activation function using 'torch.relu'. The output of the first layer is then passed through the second layer 'self.layer2'and the ReLU activation function is applied again. Finally, the output of the second layer is passed through the third layer 'self.layer3' to produce the final output tensor.

Print number of parameters Method

Create a function called 'print_num_parameters' that accepts a PyTorch model and a model name as arguments. The function then calculates the sum of the elements in each parameter tensor of the model and prints the total number of parameters in the model. The name of the model and the total number of parameters in the model are then included in a string that is printed by the function. The total number of parameters is computed using a generator expression that iterates over the model's parameters in each tensor using the 'numel' method. The total number of elements in each parameter tensor are then added up by the 'sum' function to determine the total number of parameters in the model. The 'print_num_parameters' function is called in the second line of code with the text "MLP(200,200)" as the "name" argument and an instance of the 'MLP' class as the 'model' argument. This calculates and outputs the total number of parameters in the MLP and generates a new MLP object with two hidden layers and 200 neurons each.

Train Function

Create the train function. The ‘train’ function takes several arguments to train a PyTorch model on a given dataset.

- model: the PyTorch model to be trained.
- data: the dataset to be used for training.
- batch_size: the number of samples to be processed in each batch during training.
- weight_decay: a regularization parameter that penalizes large weights in the model to prevent overfitting.
- optimizer: the optimization algorithm to be used for training. The function supports two options: "sgd" for stochastic gradient descent and "adam" for the Adam optimizer.
- learning_rate: the learning rate for the optimizer, which determines the step size at each iteration during optimization.
- momentum: a parameter used by the SGD optimizer to accelerate convergence in the relevant direction and dampen oscillations.
- data_shuffle: a boolean flag indicating whether to shuffle the order of the samples in each epoch during training.
- num_epochs: the number of epochs to train the model for.

Then the ‘DataLoader’ object is initialized to load the training data in batches. A cross-entropy loss function called ‘nn.CrossEntropyLoss()’ is then defined to calculate the discrepancy between predicted and actual labels. It initializes an optimizer object with the relevant hyperparameters based on the optimizer that was supplied. After initializing the optimizer based on the specified optimizer type, the ‘train’ function initializes four empty lists to track the learning curve during training: ‘iters’, ‘losses’, ‘train_acc’, and ‘val_acc’.

- iters: a list that stores the number of iterations (batches) processed during training, incremented by 1 for each batch
- losses: a list that stores the loss values computed on each batch during training
- train_acc: a list that stores the accuracy of the model on the training set at the end of each epoch
- val_acc: a list that stores the accuracy of the model on the validation set at the end of each epoch

The function then initializes ‘n’ to 0, which will be used to track the number of iterations processed during training for plotting purposes. The function then enters a for-loop that iterates over the specified number of epochs. For each epoch, the function iterates over the batches in the ‘train_loader’ object and trains the model on each batch. After each batch, the function computes the loss, backpropagates to update the model parameters, and tracks the loss in the ‘losses’ list.

After each epoch, the function evaluates the accuracy of the model on the training and validation sets and stores the values in the 'train_acc'and 'val_acc' lists, respectively. At the end of each epoch, the function increments 'n' by the number of batches processed during that epoch, and appends 'n' and the average loss for that epoch to the 'iters' and 'losses' lists for plotting purposes. Finally, the function returns the trained model and the four tracking lists.

After the training loop, the code plots the learning curve of the training process. The first plot displays the change in loss over the iterations (batches) in the training process. The second plot displays how training and validation accuracy changed across the iterations (batches) in the training process.

Then we specified a 'window_size' parameter, which determines the number of previous values to use when calculating the moving average. In this case, we set the 'window_size' to 10. We then calculate the moving average of the training loss and training accuracy using the 'rolling' function from the pandas library. The 'rolling' function takes the window size as input and returns a rolling window object. The 'mean' function is then applied to this object to calculate the moving average.

Use 'matplotlib' is to create the first plot by creating a new figure with the 'plt.figure()' method, and the plot's title is specified using 'plt.title()'. The plot is then generated using the 'plt.plot()' method, with 'iters' representing the x-axis (iterations) and 'losses' representing the y-axis (loss). We also created a line plot of the moving average of the training loss over a set of iterations, the resulting plot will have the iteration numbers on the x-axis and the moving average values of the training loss on the y-axis. This can be useful for visualizing how the training loss is changing over time and detecting trends or patterns in the data. In order to label the x and y axes, respectively, the methods 'plt.xlabel()' and 'plt.ylabel()' are needed. 'plt.show()' is then used to display the plot.

Also 'matplotlib' is also used to create the second plot. The plot is created by the 'plt.plot()' method, with `iters` representing the x-axis (iterations) and 'train_acc' and 'val_acc' representing the two lines in the plot. The line 'val_acc' displays the change in validation accuracy. The line 'train_acc' displays the change in training accuracy over the iterations. We also created a line plot of the moving average of the training accuracy over a set of iterations, the resulting plot will have the iteration numbers on the x-axis and the moving average values of the training accuracy on the y-axis. This can be useful for visualizing how the training accuracy is changing over time and detecting trends or patterns in the data. The x and y axes are labelled using the methods 'plt.xlabel()' and 'plt.ylabel()' correspondingly. The 'plt.legend()' method is used to create a legend for the plot. To display the plot, 'plt.show()' is finally executed.

The function then prints the model's final training accuracy and validation accuracy after the plots have been displayed. The final training accuracy is obtained by indexing the last element of the 'train_acc' list, while the final validation accuracy is obtained similarly by indexing the last element of the 'val_acc' list.

Get Accuracy Function

Create a function named ‘get_accuracy’ that accepts a PyTorch model as an input and, depending on the value of the ‘train’ argument, outputs the model's accuracy on either the validation set or the training set. If ‘train’ is True, the function loads the training set (`mnist_train`) into a PyTorch DataLoader object with a batch size of 4096. Otherwise, it loads the validation set (`mnist_val`) into a PyTorch DataLoader object with a batch size of 1024.

The model is put into evaluation mode using ‘model.eval()’, which disables any dropout or batch normalization layers that may be present in the model.

Both the ‘correct’ and ‘total’ variables are initialized to zero. The ‘correct’ will keep track of the number of correctly classified examples, while ‘total’ will keep track of the total number of examples evaluated.

Then the function then loops over each batch of examples in the data loader. For each batch, it passes the batch of images ‘imgs’ through the model to obtain a batch of predicted class probabilities ‘output’. The class that has the highest probability for each example is the predicted class, which is found using the expression ‘output.max(1, keepdim=True)[1]’. The ‘eq’ method is used to compare the predicted classes ‘pred’ with the true labels ‘labels.view_as(pred)’, and the number of correct predictions is added to ‘correct’. Finally, the number of examples in the batch is added to ‘total’. After processing all batches in the data loader, the function returns the overall accuracy of the model by dividing ‘correct’ by ‘total’.

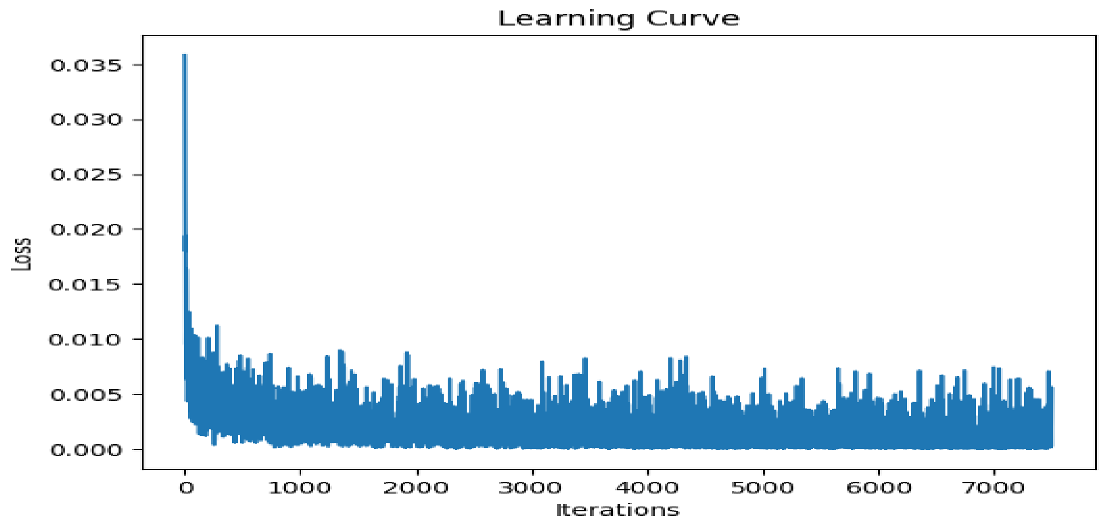
Training / Validation Loss Plot with Change in Number of Epochs

First Trial

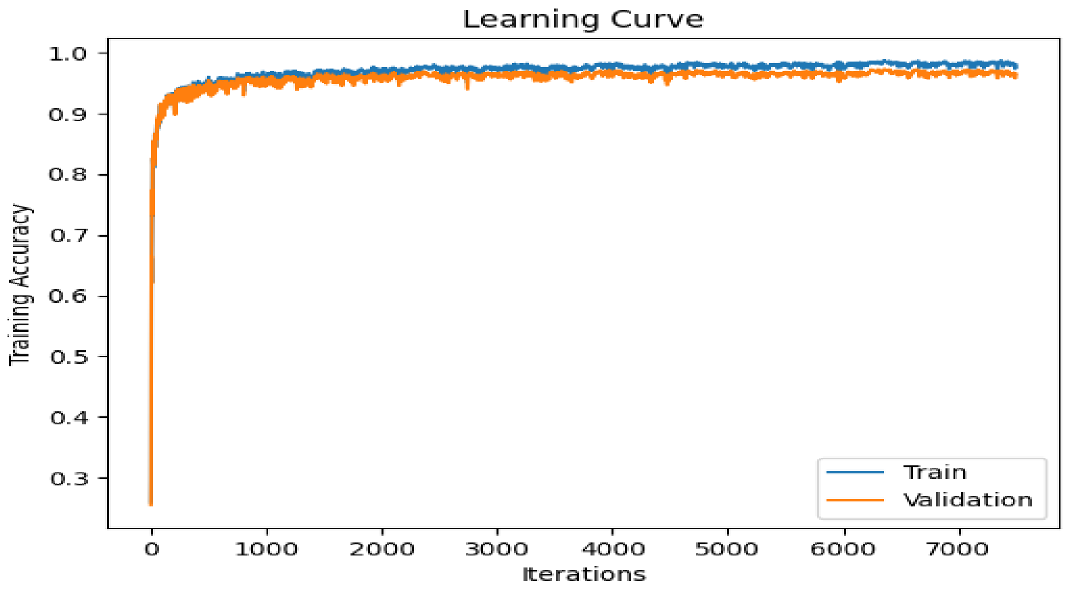
We trained the model with 100 hidden neurons of layer 1 and 50 hidden neurons of layer 2. We used in the second layer less neurons to reduce the features.

Using Optimizer: Adam, Learning rate: 0.01, Loss Function: Cross Entropy, and number of epochs=10 by default.

- *Training / Validation Loss*



- *Training / Validation Accuracy*



Final Training Accuracy: 0.9792916666666667

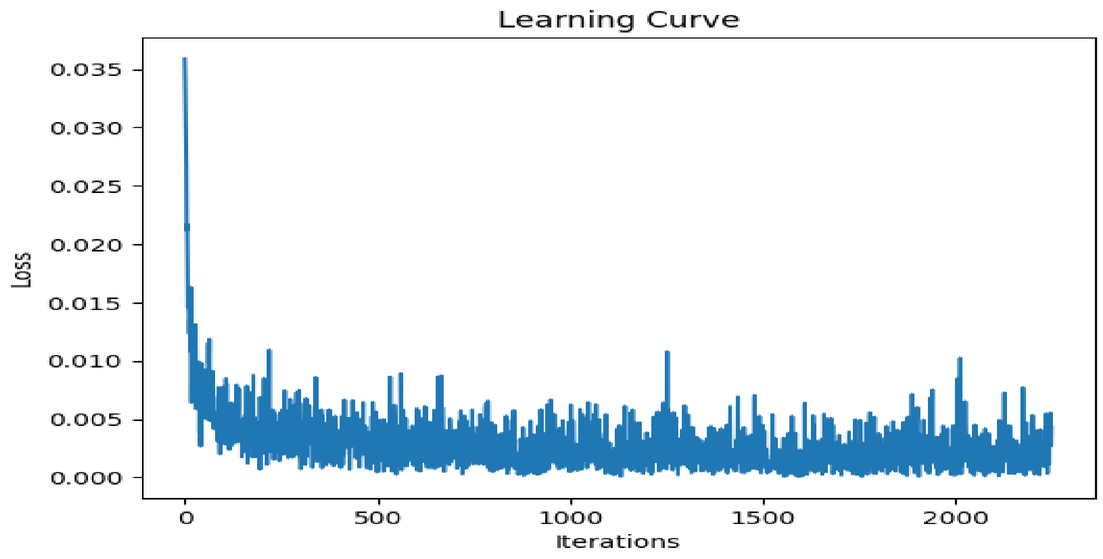
Final Validation Accuracy: 0.9638333333333333

Second Trial

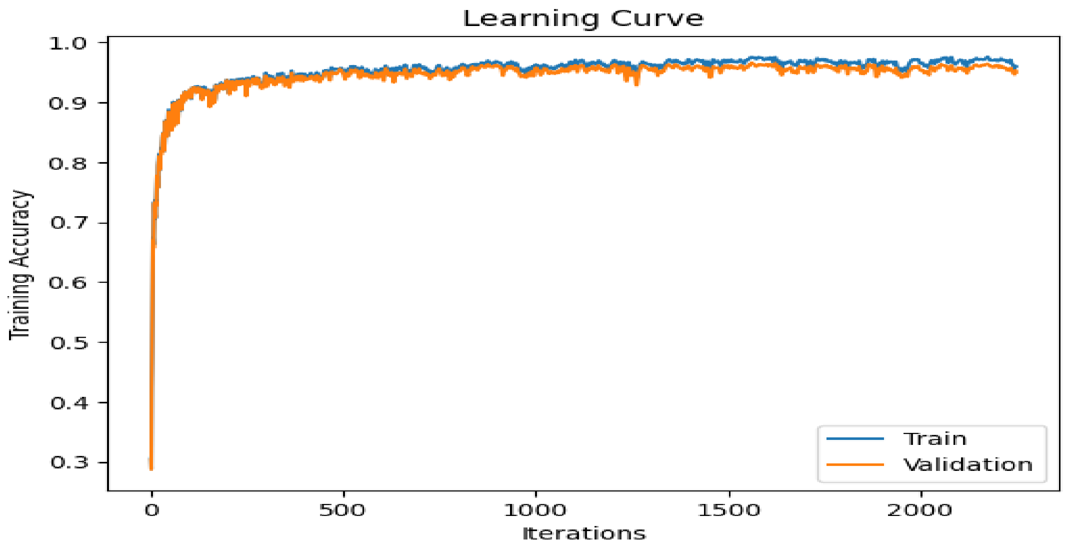
We trained the model with 100 hidden neurons of layer 1 and 50 hidden neurons of layer 2. We used in the second layer less neurons to reduce the features.

Using Optimizer: Adam, Learning rate: 0.01, Loss Function: Cross Entropy, and number of epochs=3.

- *Training / Validation Loss*



- *Training / Validation Accuracy*



Final Training Accuracy: 0.9592083333333333
Final Validation Accuracy: 0.94975

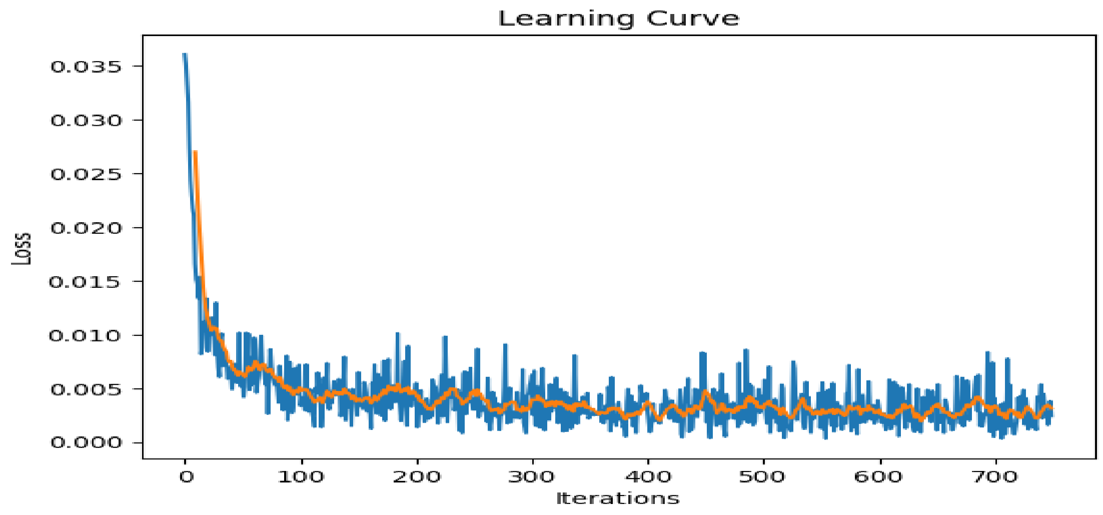
We concluded that the difference between the first and second trial is as we increase the number of epochs the accuracy increases that's why in this trial the accuracy decreased compared with the first trial.

Third Trial

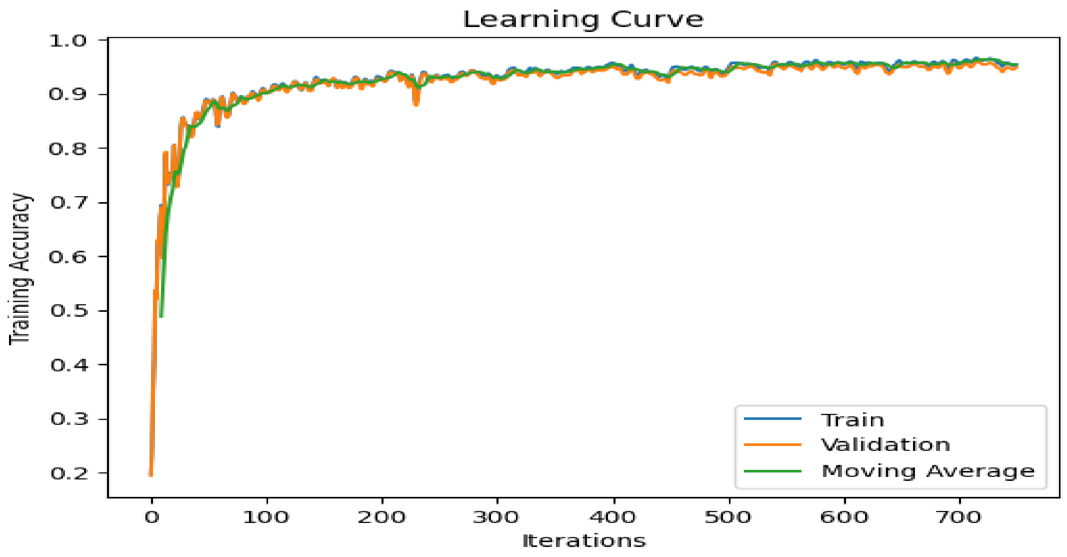
We trained the model with 100 hidden neurons of layer 1 and 50 hidden neurons of layer 2. We used in the second layer less neurons to reduce the features.

Using Optimizer: Adam, Learning rate: 0.01, Loss Function: Cross Entropy, and number of epochs=1.

- *Training / Validation Loss*



- *Training / Validation Accuracy*



Final Training Accuracy: 0.9525833333333333
Final Validation Accuracy: 0.9478333333333333

Also, the accuracy decreased compared to the other trials because number of epochs here is 1. In this trial we added to plot the moving average graph. The moving average graph helps to reduce the noise and better visualization.

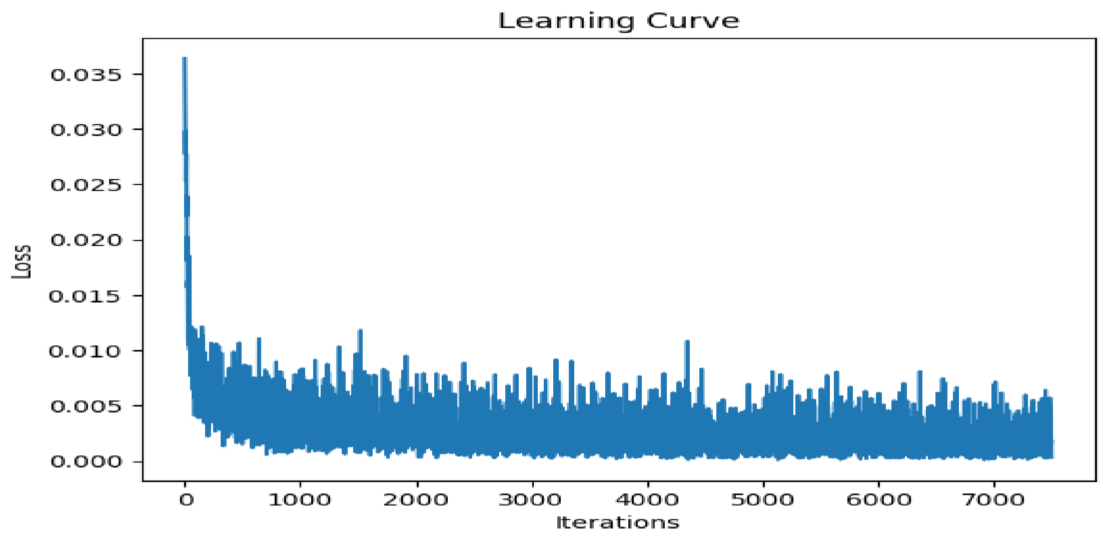
Training / Validation Loss Plot with Change in Number of Hidden Neurons and Number of Epochs

First Trial

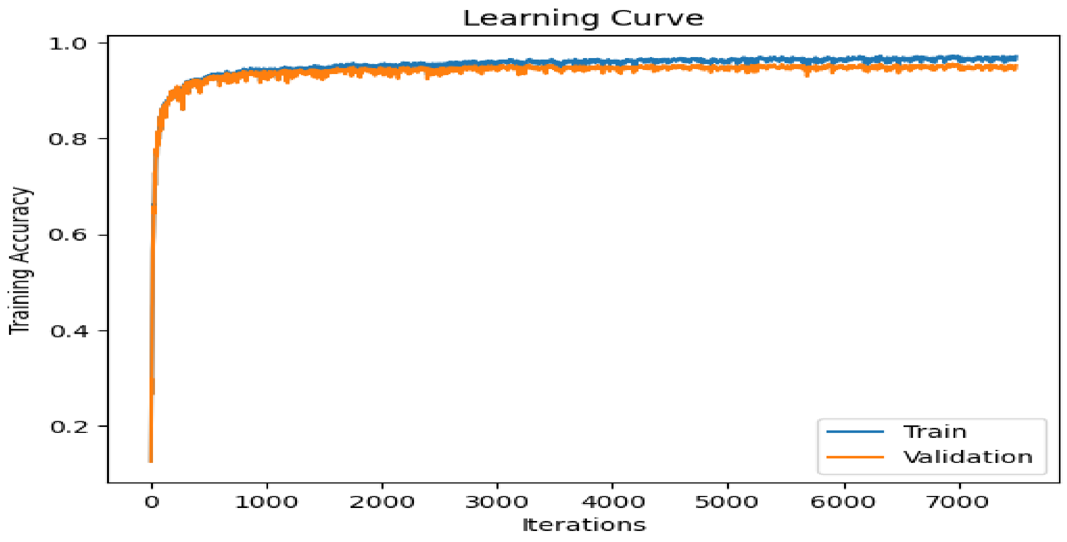
We trained the model with 30 hidden neurons of layer 1 and 5 hidden neurons of layer 2. We used in the second layer less neurons to reduce the features.

Using Optimizer: Adam, Learning rate: 0.01, Loss Function: Cross Entropy, and number of epochs=10 by default.

- *Training / Validation Loss*



- *Training / Validation Accuracy*



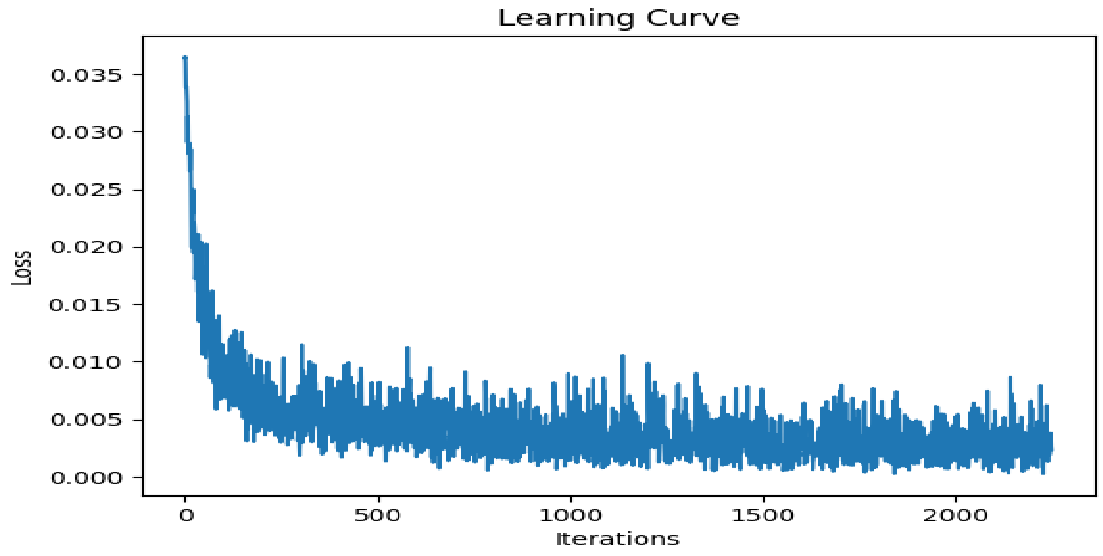
Final Training Accuracy: 0.9713333333333334
Final Validation Accuracy: 0.95225

Second Trial

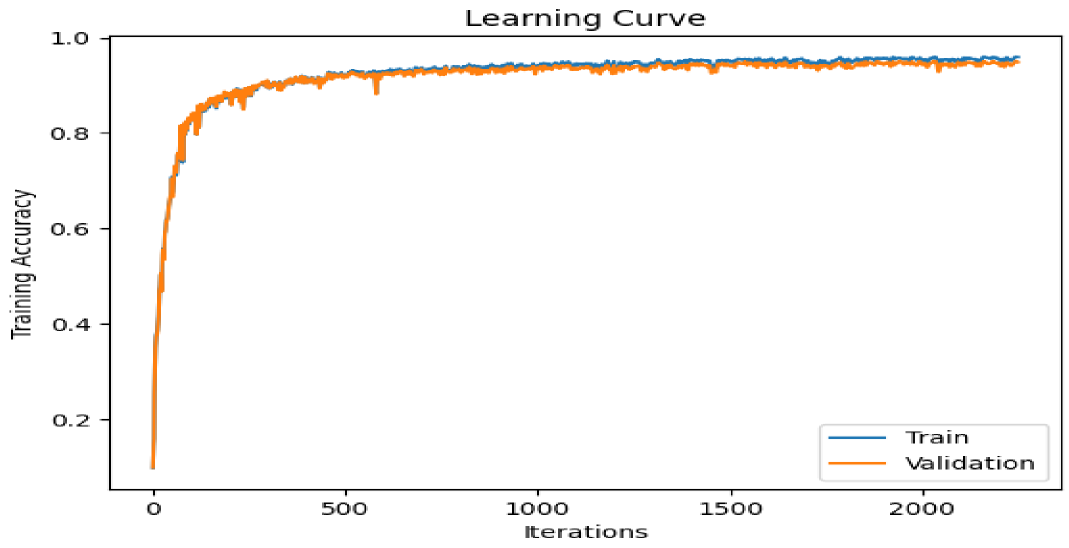
We trained the model with 30 hidden neurons of layer 1 and 5 hidden neurons of layer 2. We used in the second layer less neurons to reduce the features.

Using Optimizer: Adam, Learning rate: 0.01, Loss Function: Cross Entropy, and number of epochs=3.

- *Training / Validation Loss*



- *Training / Validation Accuracy*



Final Training Accuracy: 0.9595625
Final Validation Accuracy: 0.9484166666666667

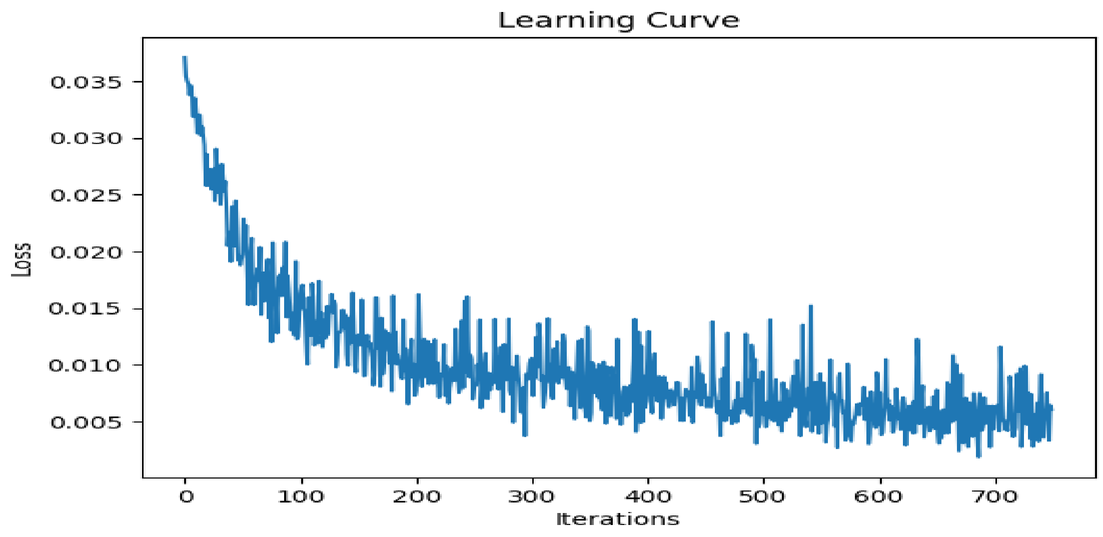
We concluded that the difference between the first and second trial is as we increase the number of epochs the accuracy increases that's why in this trial the accuracy decreased compared with the first trial.

Third Trial

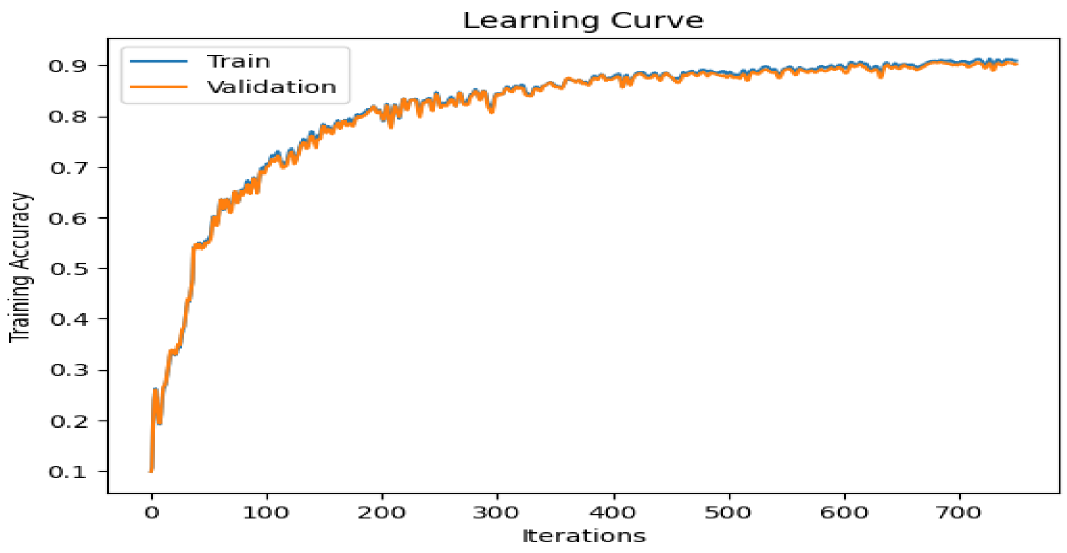
We trained the model with 30 hidden neurons of layer 1 and 5 hidden neurons of layer 2. We used in the second layer less neurons to reduce the features.

Using Optimizer: Adam, Learning rate: 0.01, Loss Function: Cross Entropy, and number of epochs=1.

- *Training / Validation Loss*



- *Training / Validation Accuracy*



Final Training Accuracy: 0.9092083333333333
Final Validation Accuracy: 0.9023333333333333

Here you can see the difference when we decreased the number of epochs to 1 and decreased the hidden neurons the training and validation accuracy decreased a lot.

Building Neural Network with Dropout and Normalization Layers

NMLP Function with Dropout and Normalization Layer to NMLP Function

First, define a multi-layer perceptron (NMLP) neural network that inherits from the 'nn.Module' class in PyTorch. An NMLP is a type of feedforward neural network that consists of one or more layers of fully connected neurons.

Constructor method for the NMLP class

Then define the constructor method for the NMLP class. It is the same steps as in the MLP constructor but we added a dropout layer after the after the second fully connected layer (fc2). By adding a single dropout layer, we can prevent overfitting and improve the generalization ability of the model. Dropout works by randomly dropping out some neurons during training, which forces the remaining neurons to learn more robust features and prevents the model from becoming too specialized to the training data. Then we added normalization layers after the first and second fully connected layers (fc1 and fc2). By applying layer normalization to the output of each layer, we can ensure that the distribution of the activations remains stable throughout the network and helps to reduce the internal covariance shift. In summary, adding multiple normalization layers and a dropout layer can help to further stabilize the training process, reduce overfitting, and improve the performance of the neural network model on the MNIST dataset.

Forward Method

Then define the 'forward' method, it is the same steps as in the Forward function in MLP and we applied the dropout and the normalization layers after each linear layer and before the activation function. we used the 'F.relu' function instead of the 'torch.relu' function, which allows us to apply the layer normalization and dropout layers in between the linear and activation layers.

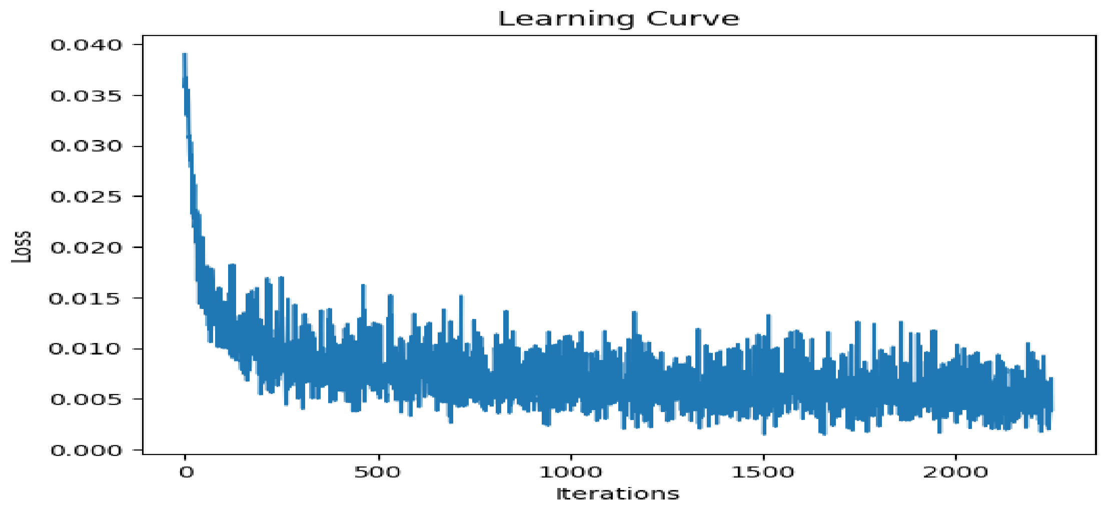
Then Retrain the model using the same training function we used before.

Training / Validation Loss Plot with Dropout and Normalization Layers

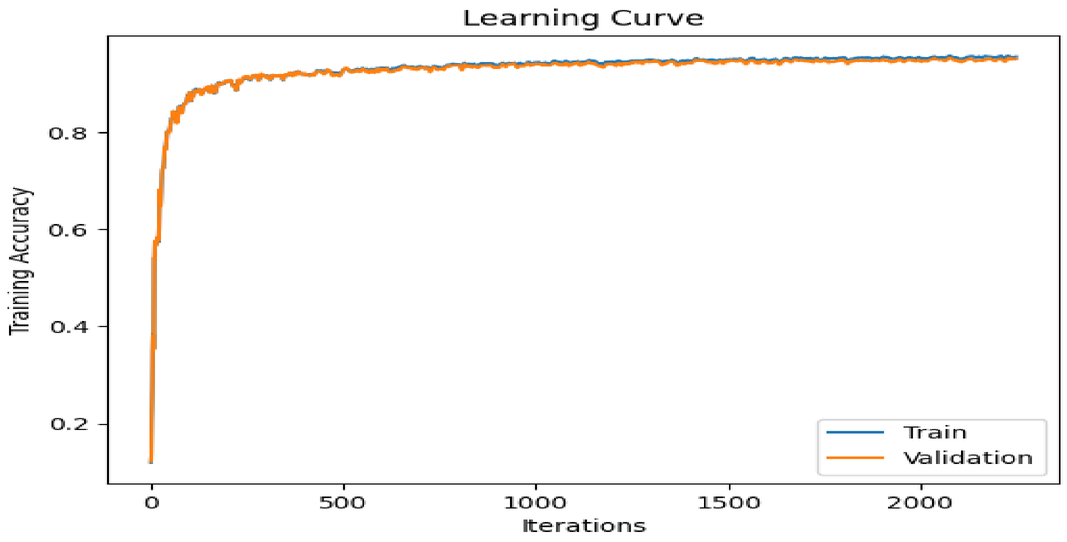
We trained the model with 30 hidden neurons of layer 1 and 5 hidden neurons of layer 2. We used in the second layer less neurons to reduce the features.

Using Optimizer: Adam, Learning rate: 0.01, Loss Function: Cross Entropy, dropout probability = 0.5, and number of epochs = 3.

- *Training / Validation Loss*



- *Training / Validation Accuracy*



Final Training Accuracy: 0.9556041666666667
Final Validation Accuracy: 0.9506666666666667

Comparison

Comparing between the graph before adding the dropout and normalization layers and after adding dropout and normalization layers both graphs with 30 hidden neurons of layer 1 and 5 hidden neurons of layer 2 using Optimizer: Adam, Learning rate: 0.01, Loss Function: Cross Entropy, dropout probability = 0.5, and number of epochs = 3.

You can see that by using dropout layers the test accuracy increased from 76.92% to 80.77%. This is a good improvement and shows that this model performs well in both training and testing. Therefore, using dropout regularization we have handled overfitting in deep learning models. With dropout (dropout rate less than some small value), the accuracy will gradually increase and loss will gradually decrease first (That is what is happening in our case). When you increase dropout beyond a certain threshold, it results in the model not being able to fit properly.

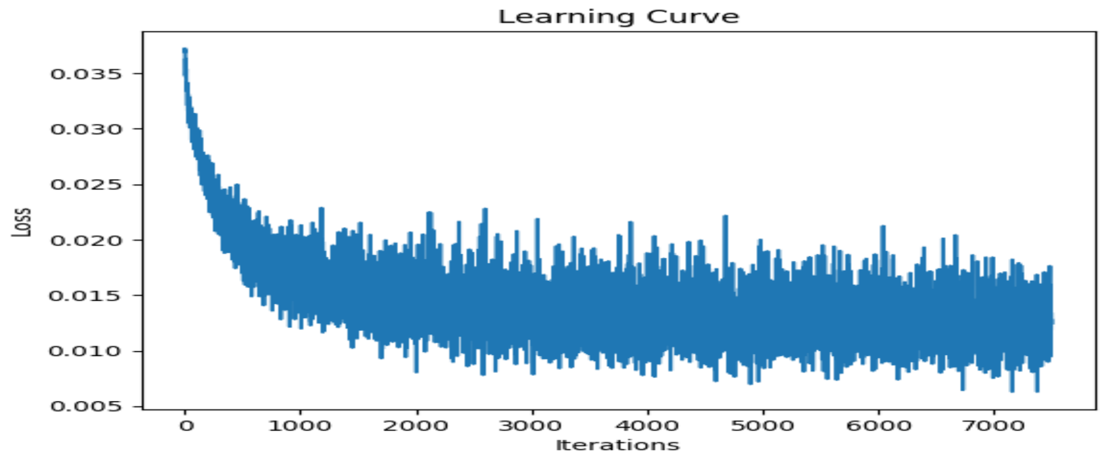
Retraining 3 Models

First Model

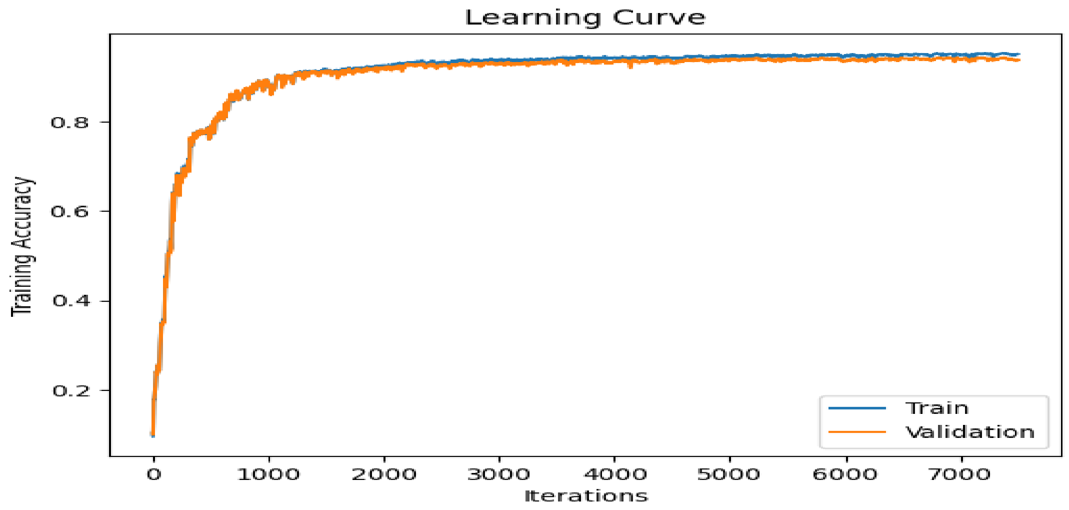
We trained the model with 30 hidden neurons of layer 1 and 5 hidden neurons of layer 2. We used in the second layer less neurons to reduce the features.

Using Optimizer: Adam, Learning rate: 0.001, Loss Function: Cross Entropy, dropout probability = 0.2, and number of epochs = 10 by default.

- *Training / Validation Loss*



- *Training / Validation Accuracy*



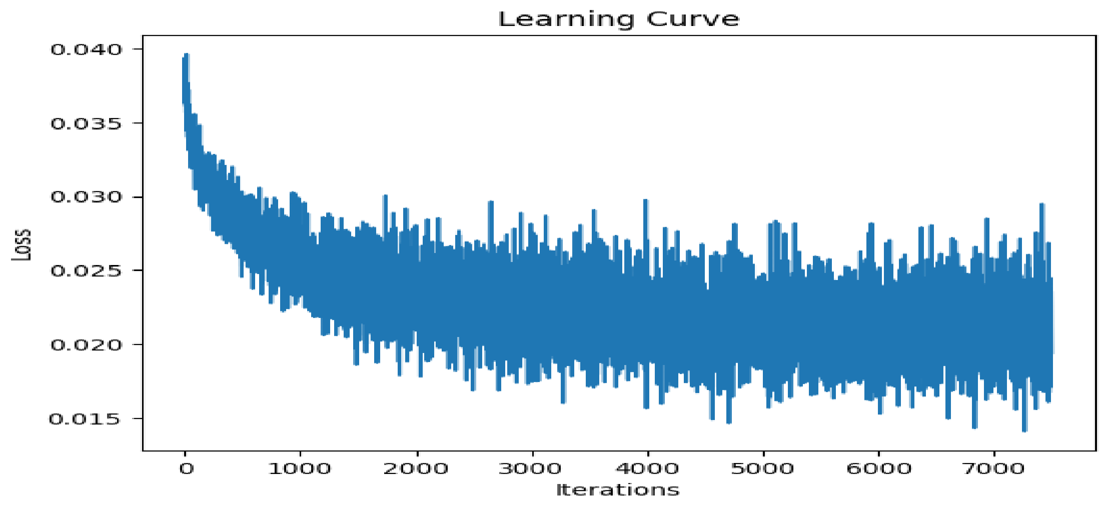
Final Training Accuracy: 0.9505833333333333
Final Validation Accuracy: 0.93675

Second Model

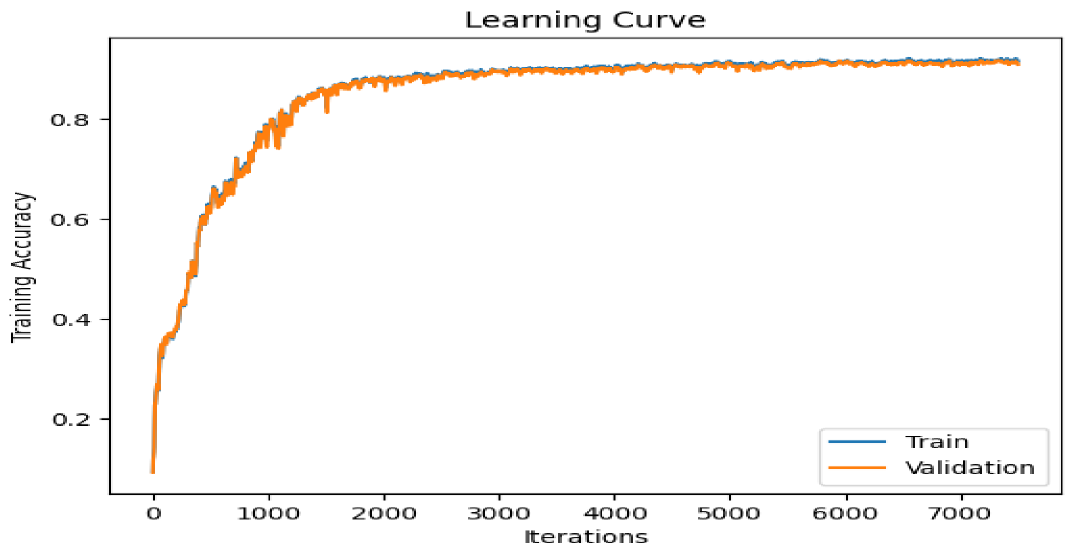
We trained the model with 30 hidden neurons of layer 1 and 5 hidden neurons of layer 2. We used in the second layer less neurons to reduce the features.

Using Optimizer: Adam, Learning rate: 0.0005, Loss Function: Cross Entropy, dropout probability = 0.4, and number of epochs = 10 by default.

- *Training / Validation Loss*



- *Training / Validation Accuracy*



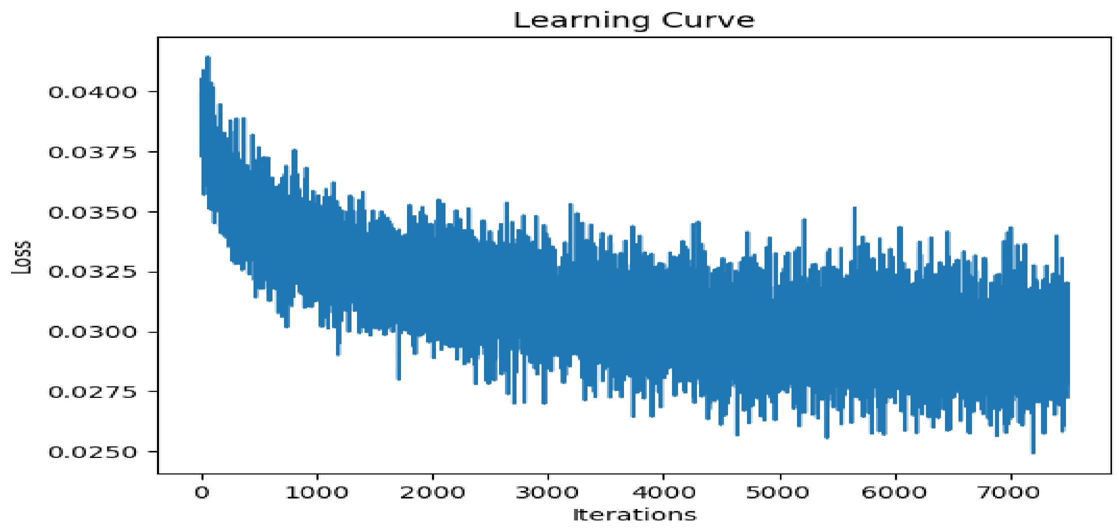
Final Training Accuracy: 0.9157916666666667
Final Validation Accuracy: 0.9095833333333333

Third Model

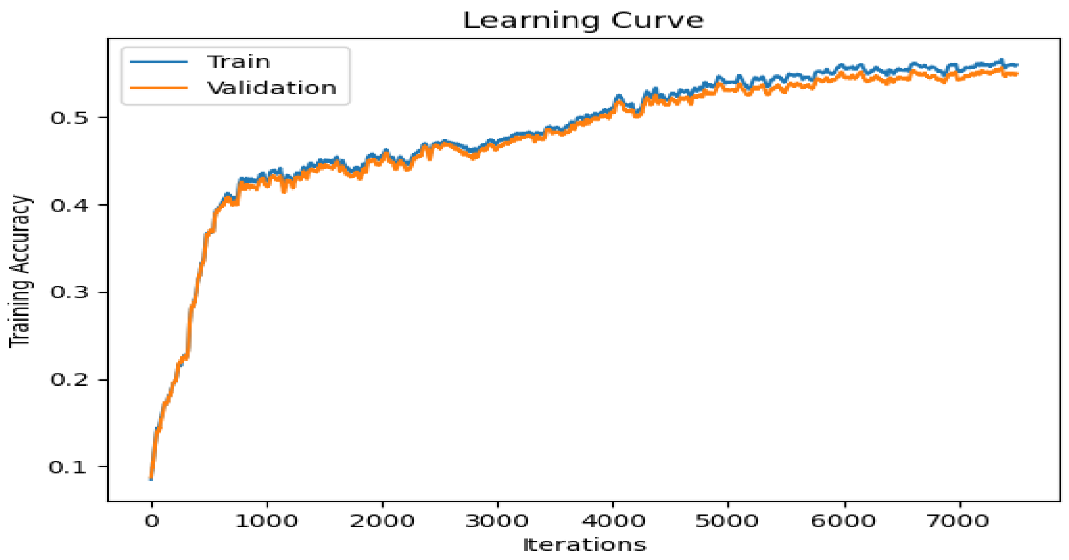
We trained the model with 30 hidden neurons of layer 1 and 5 hidden neurons of layer 2. We used in the second layer less neurons to reduce the features.

Using Optimizer: Adam, Learning rate: 0.0001, Loss Function: Cross Entropy, dropout probability = 0.6, and number of epochs = 10 by default.

- *Training / Validation Loss*



- *Training / Validation Accuracy*



Final Training Accuracy: 0.5596666666666666
Final Validation Accuracy: 0.5495833333333333

Conclusion

We concluded from model 1, model 2, and model 3 that it is common to observe a decrease in accuracy when increasing the dropout probability. Dropout is a regularization technique that helps prevent overfitting by randomly dropping out some units in the neural network during training. This forces the remaining units to learn more robust features that are not dependent on the presence of specific units.

However, as the dropout probability increases, the network is more likely to drop out important units, which can negatively impact the network's performance. Additionally, higher dropout probabilities can also lead to underfitting, where the network is not able to capture the underlying patterns in the data.

It is important to find an appropriate balance between regularization and model complexity. An accuracy near 0.5 in model 3 could indicate that the model is not learning the patterns in the data very well.

In general, a dropout probability of 0.6 is relatively high, and can cause the model to underfit the training data, especially if the model has a large number of parameters. Additionally, a learning rate of 0.0001 is quite small, and may cause the model to converge very slowly, or get stuck in a local minimum of the loss function. For the MNIST dataset, a dropout probability of 0.6 and a learning rate of 0.0001 may not be optimal.

The MNIST dataset is relatively small and simple, so a smaller model with a lower dropout probability and a higher learning rate may be more appropriate. As in model 1 we reduced the dropout probability to 0.2 or 0.3, and increased the learning rate to 0.001 or 0.01. Additionally, we could try adding more layers to the model, or increase the number of hidden units in each layer, to make the model more powerful.