Deep Learning Assignment 1 Report
Shikhar Dave
B22CH032

Colab file link : B22CH032.ipynb

Objective : This assignment was based on making a neural network from scratch in python .

Dataset Details : The dataset used is MNIST which contains handwritten digits from 0 to 9 . The size of the image is 28*28 which is flattened to 784 dimensional vectors which were **normalised** further on . Tensorflow is used by me for importing the dataset . **One-hot encoding** is applied to make the output vector dimension the same as the number of classes as I am using the categorical cross entropy loss function.

Experimentation details : I experimented with various weight initialization techniques such as **Xavier , He and random initialisation** . Batch size is taken as 22 as I am from B22. I also experimented with the activation function so I took **ReLU , sigmoid and tanh** , along with their derivatives defined from scratch. **Softmax function** is used for the multiclass classification task . Cross entropy loss is used made from scratch and combined with **L2 regularization to prevent overfitting** . This makes the model more generalised instead of just mugging up the training set data .

Forward Propagation : The `forward_pass` function computes the output of a neural network by propagating input data (X) through its layers. It first performs a linear transformation (`Z1`) using weights (`W1`) and biases (`b1`) for the first layer, then applies an activation function like ReLU, sigmoid, or tanh to produce `A1`. Similarly, for the second layer, it computes `Z2` using `A1`, weights (`W2`), and biases (`b2`), and applies the **softmax function to get the final probabilities** (`A2`). The intermediate computations (`Z1`, `A1`, `Z2`, `A2`) are **stored in a cache** for use during backpropagation.

Backpropagation : The `backward_pass` function calculates the gradients of the loss function with respect to the weights and biases for both layers. It starts with the output layer by computing the difference between the predicted (`A2`) and actual (`y_true`) values (`dZ2`). Using this, it calculates gradients for the weights (`dW2`) and biases (`db2`) of the second layer. These gradients are propagated back to the first layer, where derivatives of the chosen activation function are applied to compute `dZ1`, followed by `dW1` and `db1`. `L2` regularization is incorporated to prevent overfitting.

Parameter updation : The `update_parameters` function updates the weights and biases of the neural network using gradient descent. It reduces the values of weights and biases (`W1`, `W2`, `b1`, `b2`) by a fraction of their respective gradients (`dW1`, `dW2`, `db1`, `db2`), scaled by the learning rate. This process ensures that the network iteratively minimizes the loss function and improves its predictions over multiple training epochs.

The main training loop : The `train_model_with_gradient_descent` function trains a neural network using different optimization modes (stochastic, batch, or mini-batch gradient descent). It initializes weights and biases based on the selected method and iterates over the specified number of epochs. Depending on the optimization mode:

- **Stochastic Gradient Descent (SGD)**: Updates weights for each training sample.
- **Batch Gradient Descent**: Uses the entire dataset to compute gradients and update weights.
- **Mini-Batch Gradient Descent**: Splits the data into batches and updates weights for each batch.

It calculates the validation loss and accuracy at the end of each epoch, storing these metrics in the `history` dictionary. Early stopping halts training if the validation loss doesn't improve for a set number of epochs (`patience`). The nested loops systematically iterate over all combinations of activation functions, weight initialization methods, and optimization modes. For each configuration:

1. **Data Splitting**: Splits the data into training and validation sets using `train_test_split`.
2. **Training**: Calls the `train_model_with_gradient_descent` function with the specific configuration, enabling early stopping.
3. **Plotting**: Visualizes the validation loss and accuracy curves across epochs using matplotlib. This helps analyze how each configuration impacts model performance over time.

In the code file , You will find a total 27 plots for validation loss and validation accuracy .

Best model pipeline : After doing this rigorous experimentation and concepts learned from class , the best performing model is the neural network with ReLU activation function , He weight initialization strategy and the mini-batch gradient descent algorithm . I have run this model again explicitly and have integrated the learning rate scheduling for better performance .

Lr_scheduler : The function `learning_rate_scheduler` dynamically adjusts the learning rate during training based on validation loss trends. It aims to optimize model convergence and prevent overfitting by gradually reducing the learning rate if there's no improvement in validation loss over a specified number of epochs (`patience`).

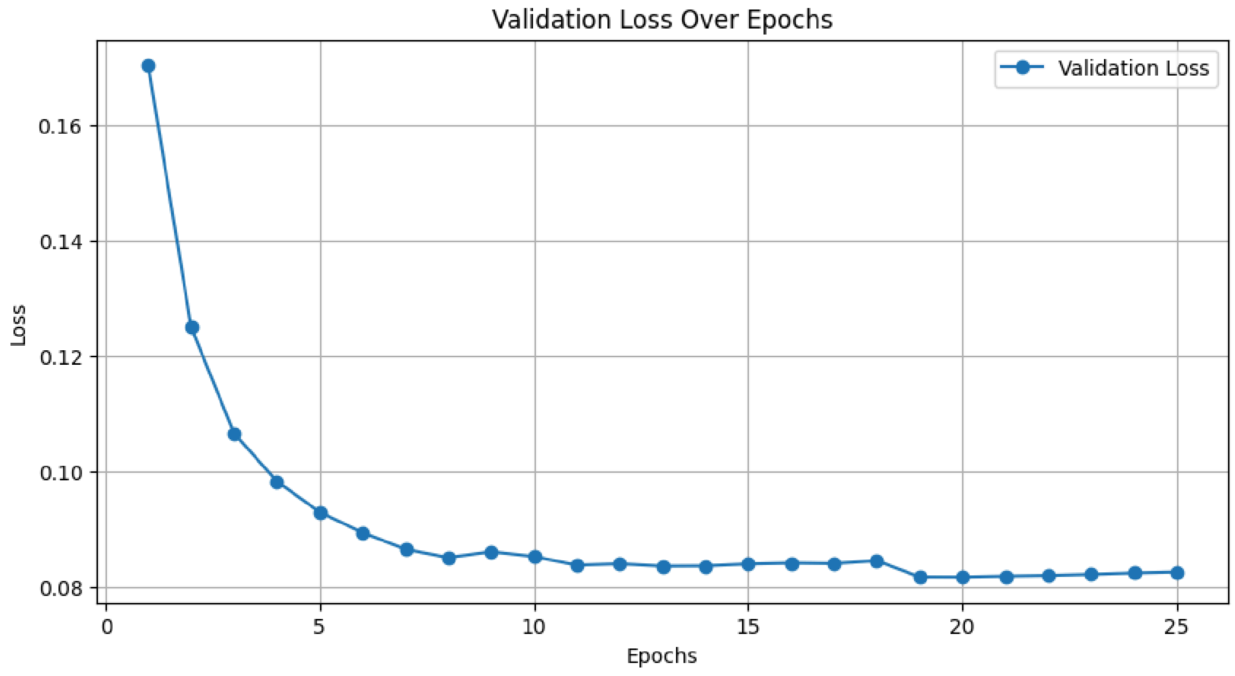
Best model results : **`Epoch 22/25: Validation Loss = 0.0819, Validation Accuracy = 0.9791`**

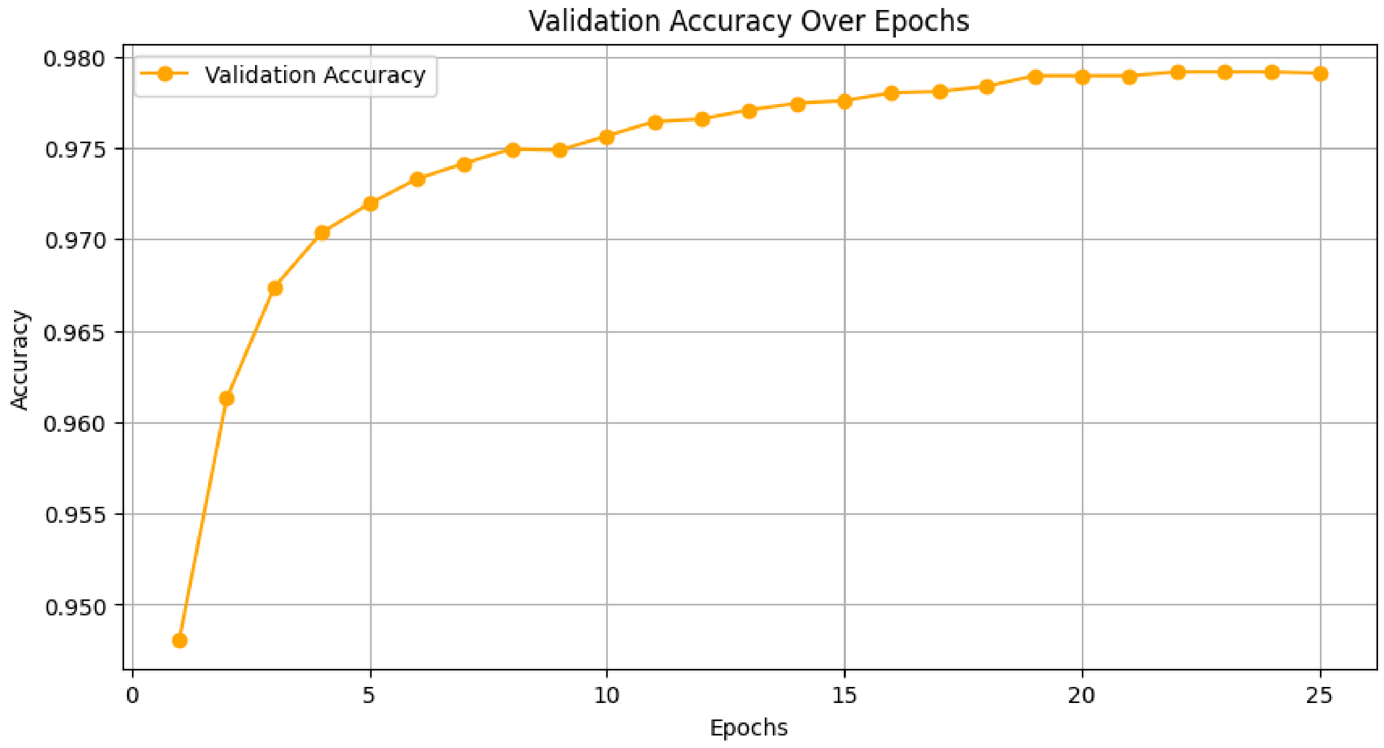
ReLU is an efficient activation function that avoids vanishing gradients, promotes sparsity by outputting zeros for negative inputs, and handles non-linearity effectively, allowing the network to learn complex patterns. It outperformed Sigmoid and Tanh by providing faster convergence and better stability.

He Weight initialization is tailored for ReLU, scaling weights optimally to maintain output variance across layers. This prevents issues like exploding or vanishing gradients and ensures smoother and faster convergence, outperforming Xavier and random initialization methods.

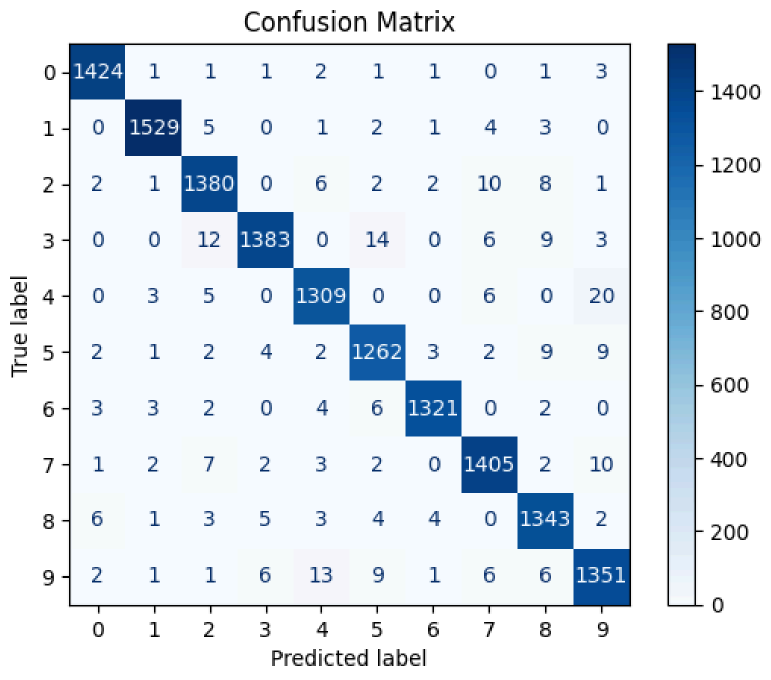
Mini-Batch Gradient Descent strikes a balance between computational efficiency and gradient accuracy. It reduces oscillations seen in stochastic gradient descent and introduces randomness that helps the model generalize better. This method enabled stable and efficient learning compared to batch or stochastic approaches, making it ideal for this task.

The following are the validation loss and validation accuracy curve for the best possible model :





Also the confusion matrix obtained in this case :



	0	1	2	3	4	5	6	7	8	9
0	1424	1	1	1	2	1	1	0	1	3
1	0	1529	5	0	1	2	1	4	3	0
2	2	1	1380	0	6	2	2	10	8	1
3	0	0	12	1383	0	14	0	6	9	3
4	0	3	5	0	1309	0	0	6	0	20
5	2	1	2	4	2	1262	3	2	9	9
6	3	3	2	0	4	6	1321	0	2	0
7	1	2	7	2	3	2	0	1405	2	10
8	6	1	3	5	3	4	4	0	1343	2
9	2	1	1	6	13	9	1	6	6	1351