ELL409

Machine Learning and Intelligence

Assignment 1

Authors:

Ayushman Kumar Singh (2021CS51004)
Vanchanagiri Alekhya (2020EE10565)

Instructor:

Brejesh Lall

2023/2024 – 1st Semester

1 Question - 1: Implementing Linear Classification models for MNIST and CIFAR-10 datasets

1. **Loading datasets and normalizing:**

   - Dataset choice: Selected the MNIST dataset for this model
   - Data Loading: Used PyTorch's **torchvision.datasets** module to load the datasets.
   - Dataset Details:
     (a) Dataset Size: The MNIST dataset consists of a total of 70,000 images, divided into 60,000 training examples and 10,000 testing examples.
     (b) Number of Classes: There are ten distinct classes in the MNIST dataset, corresponding to the ten digits from 0 to 9.

2. **Data preprocessing:**

   - Normalization: Transformed the image data of the input dataset into the tensors using **torchvision.transforms** and normalised the input dataset to scale features to a common range, mean = 0.5 and variance = 0.5 using the module **torchvision.transforms.normalize**
   - Splitting the dataset: The datasets are split by configuring the train parameter as True for the training set and False for the testing set

3. **Model architecture:**

   - Model Description: Define a linear classification model using PyTorch's nn.Module. The key components of the model include:
     (a) Linear Layer: Linear function is defined using torch.nn.Linear which is responsible for mapping the input data to the outputs.
     (b) Sigmoid Activation Function: Used the function nn.functional.sigmoid() to define an activation function so as to transform the raw output logits into probability scores. Each score corresponds to the likelihood that the input image belongs to a specific class.
   - Input Size: The input size of the model varies depending on the dataset being used.
     (a) MNIST: For the MNIST dataset, each input image is represented as a 28x28 grayscale matrix. This results in a total of 784 input features (28 * 28). The model expects input data to be in this format, with each feature representing the intensity of a pixel in the image.
   - Output Classes: The output layer of our model is configured to have the correct number of units corresponding to the number of classes in the dataset being used.
     For MNIST dataset, the model's output layer has 10 units.
     The sigmoid function applied to these units provides the probability distribution over these 10 classes allowing the model to make predictions about assigning classes.

4. **Loss function and optimizer:**

- Loss Function: We chose the CrossEntropy Loss function for this model. In classification tasks, the goal is to produce a probability distribution over multiple classes and compare it with the true distribution. CrossEntropyLoss achieves this by quantifying the dissimilarity or "cross-entropy" between the predicted probabilities and the actual labels.
- Optimizer Selection: Stochastic Gradient Descent (SGD) was chosen as the optimizer for training our linear classification model. The initial learning rate was taken as lr=0.01 and the momentum value as 0.9

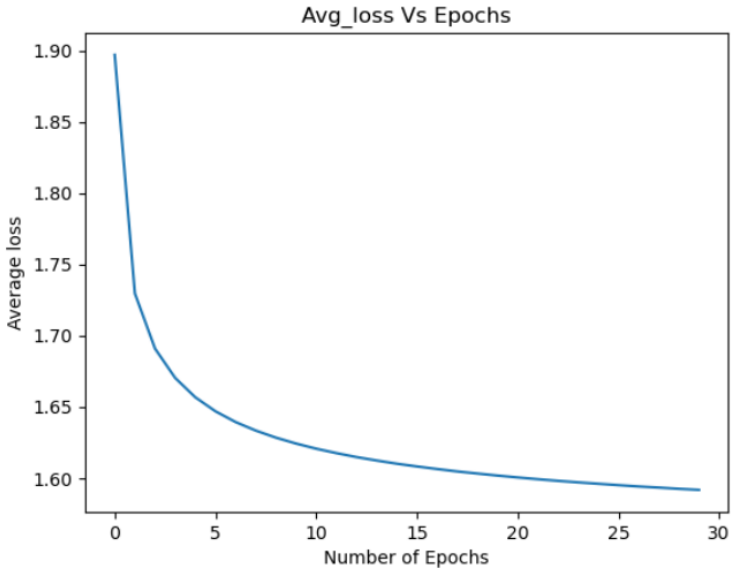
5. **Training loop:**

- primary purpose is to iteratively update the model's weights to minimize the chosen loss function. This iterative process allows the model to learn and adapt to the training data over multiple epochs
- Training Steps:
  (a) Forward Pass: In the forward pass, an input mini-batch of training data is passed through the model. The model computes predictions for each example in the mini-batch. These predictions are typically raw scores or logits, indicating the model's confidence in each class.
  (b) Loss Calculation: After the forward pass, the CrossEntropyLoss is applied to compare the model's predictions with the actual ground truth labels for the mini-batch. The loss quantifies how well or poorly the model is performing on this mini-batch. After every epoch we calculate the cost or average loss for that epoch.
  (c) Backpropagation: Backpropagation is the process of computing gradients of the loss with respect to the model's weights and biases.
  (d) Parameter Updates: Using the computed gradients, the SGD optimizer updates the model's weights and biases. The magnitude and direction of these updates depend on the learning rate and optimizer's specific update rule. This step iteratively refines the model's parameters to minimize the loss.
  (e) Zeroing Out Gradients: Before computing gradients during each iteration of the training loop (different mini batches), it is crucial to zero out the gradients of the model's parameters. This is typically done using the optimizer.zerograd() function.

6. **Evaluation:**

- The primary objective of the evaluation phase is to assess the model's performance on previously unseen test dataset.
- Used model.eval() function to evaluate the test dataset and find the accuracy.
- Accuracy quantifies the model's ability to make correct predictions about the class labels of the test data.
- Accuracy = (Number of Correct Predictions) / (Total Number of Predictions)

7. Results and analysis:



(a) Final Accuracy on MNIST test set for a batch size of 64 , 30 epochs, a learning rate of 0.01, using SGD optimizer with StepLR scheduler 89.84 percent

(b) Training time for all 30 epochs: 304.1020436286926 secs

(c) From the graph, we observe that the average loss for the model prediction decreases as the number of epochs increases. We can say that loss curve converges as we train the data and evaluate for more epochs.

2 Question 2: Hyperparameter Tuning

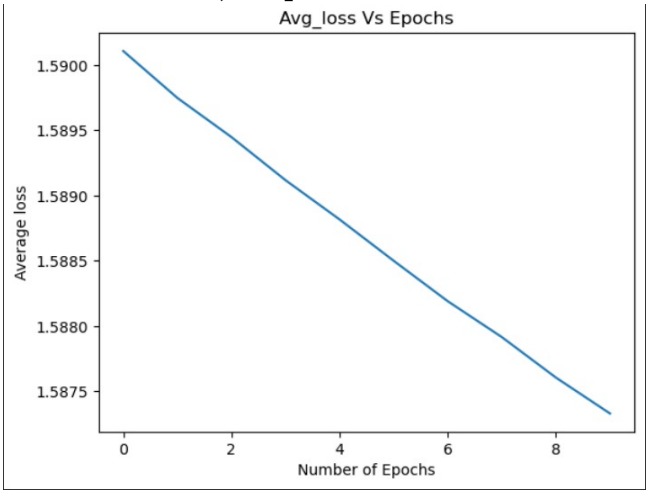
1. **Implementing Learning Rate Schedules:**

   - Scheduler Selection: We used stepLR learning rate scheduler.
   - Initial Learning Rate= 0.01
   - Gamma is the multiplicative factor by which gamma reduced for each step
   - step size is the number of epochs, the learning rate decays

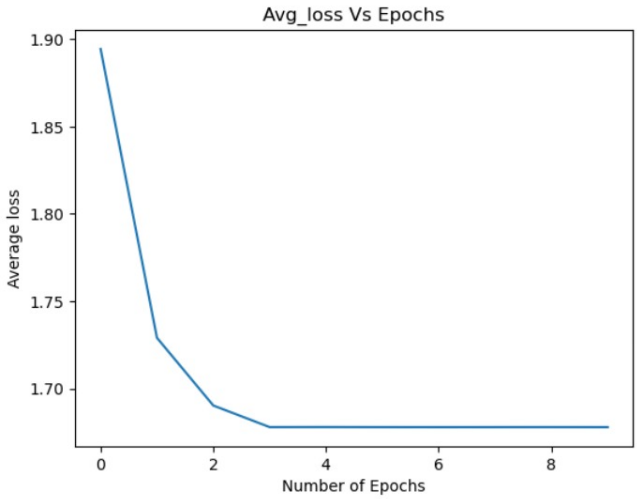
Gamma	Epochs	Step Size	Accuracy
0.01	10	10	90.04
0.001	10	3	87.08
0.001	10	1	87.57
0.1	10	1	87.94
0.1	10	3	88.52

   - Results:
     Trials with changing Gamma and step size parameters

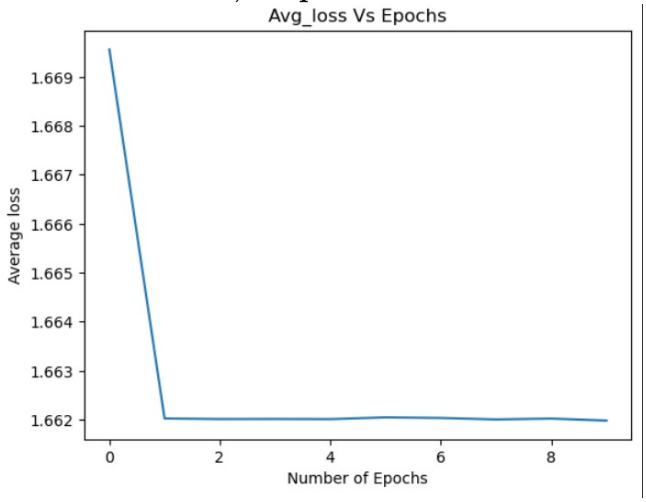
Gamma=0.01, step size=10



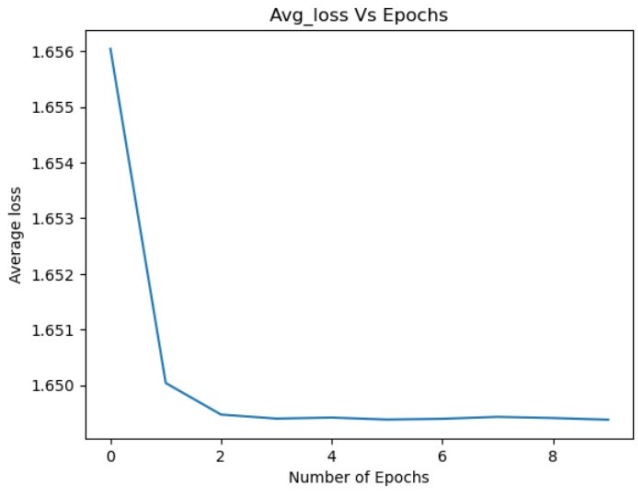
Gamma=0.001, step size=3



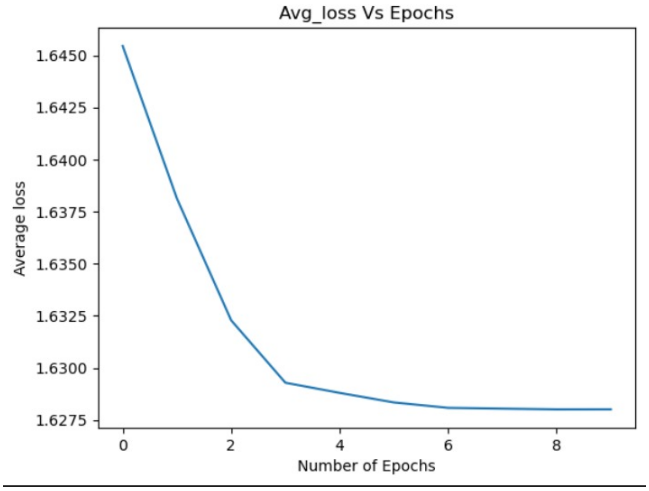
Gamma=0.001, step size=1



Gamma=0.1, step size=1



Gamma=0.1, step size=3



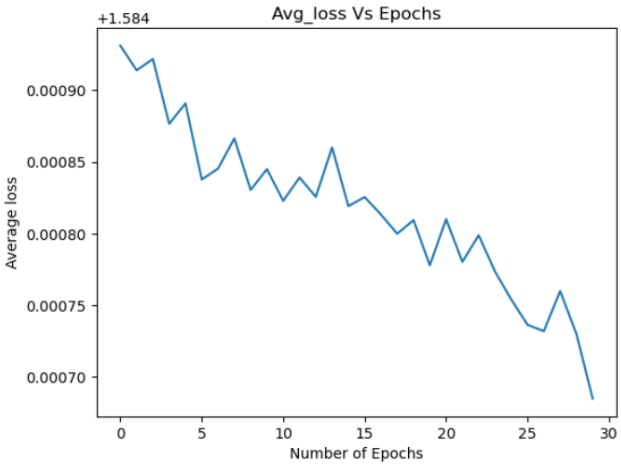
- keeping the gamma same and decreasing the step size , the accuracy decreases.

2. **Introducing L2 Regularization:**

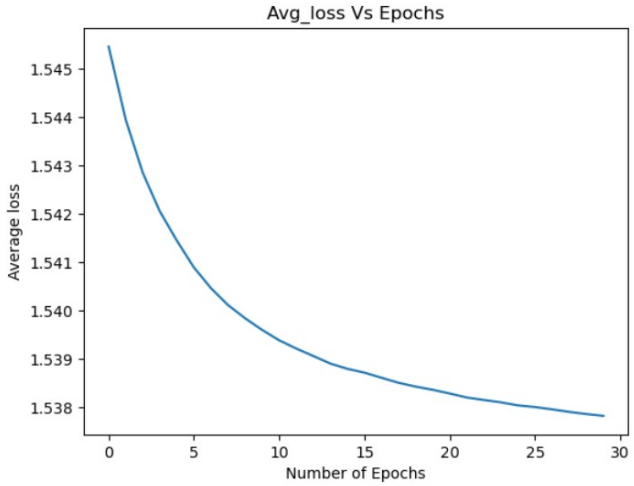
Regularisation Parameter	**Accuracy**	**Epochs**
0.001	90.37	30
0.0001	91.44	10
0.1	85.05	10
0.0000001	89.25	10

- Results:

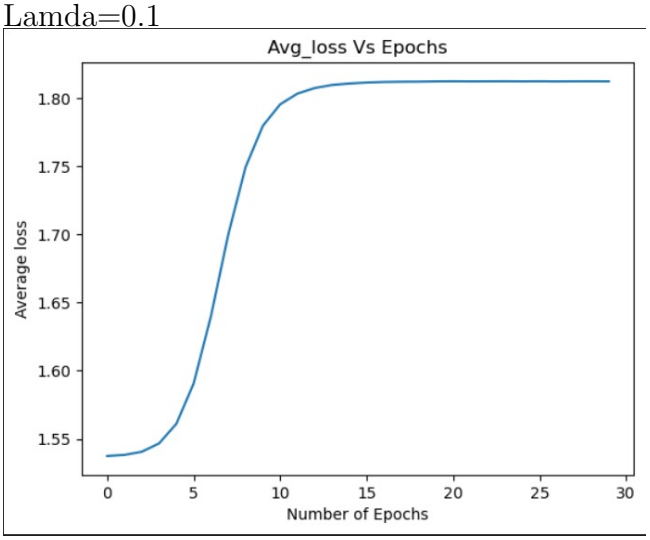
Lamda=0.01



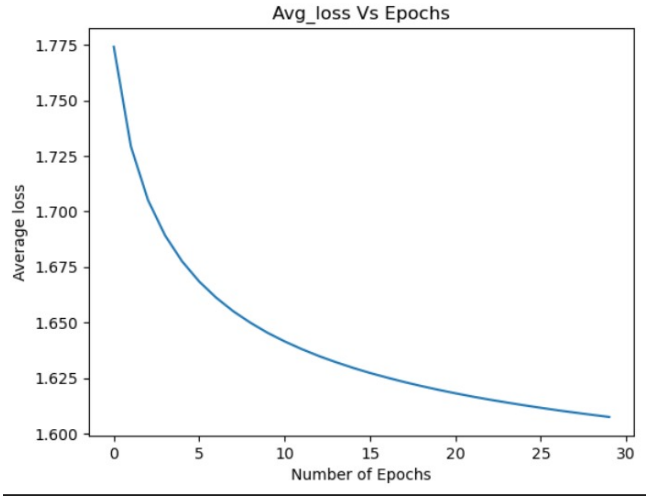
Lamda=0.0001



Lamda=0.1



Lamda=0.00000001



Observations:

(a) From the above graph, the avg loss decreases as the number of epochs increases.

(b) For same number of epochs, for very high regularization parameter, the accuracy is low and for very low regularization parameter, the accuracy is low.

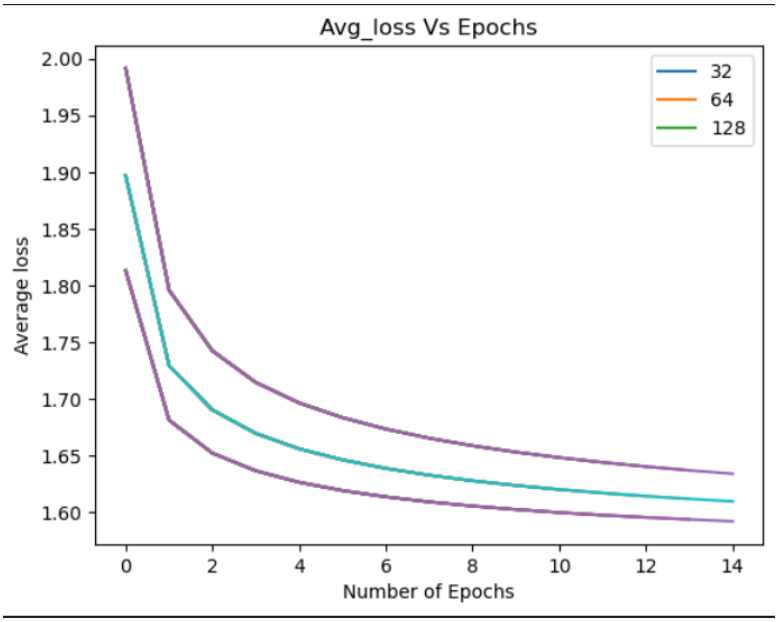
(c) For optimal regularization parameter 0.0001 for 10 epochs, the accuracy is high. From the table and graphs, it can be seen that for very low regularization parameter, the model overfits and for high regularization parameter, the model underfits.

3. **Experimenting with Batch Sizes:**

- Batch Size Variations: We have tarined the mode for batch sizes of 32, 64 and 128.
- Convergence Analysis: Loss Vs Number of epochs graphs for differnt batch sizes

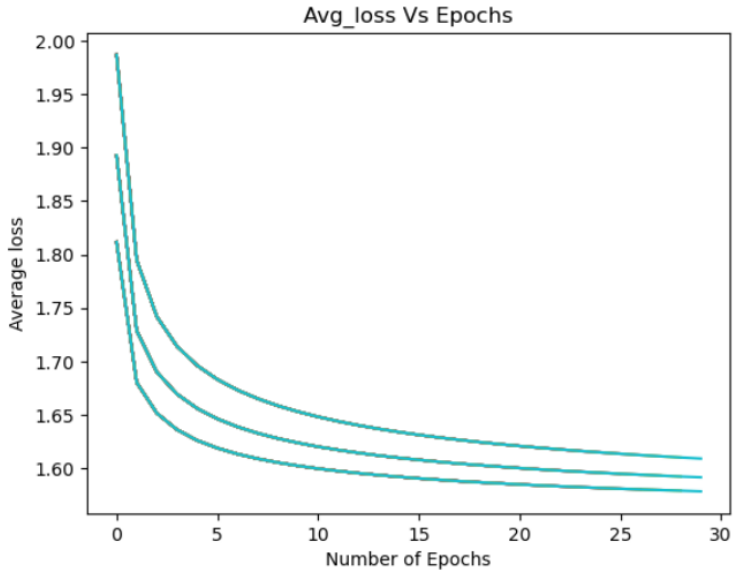
Batch Size	**Accuracy**	**No. of Epochs**
32	89.80	15
64	89.15	15
128	88.30	15
32	90.33	30
64	89.75	30
128	89.19	30

- Results:
  No. of Epochs 15
  Upper one corresponds to 128 batch size, middle one corresponds to 64 batch size and lower one for 32 batch size.



No. of Epochs 30

Upper one corresponds to 128 batch size, middle one corresponds to 64 batch size and lower one for 32 batch size.



- Observations: For same number of epochs, as the batch size increases, accuracy decreases.
  By keeping the batch size same, as the number of epochs increases the accuracy increases.

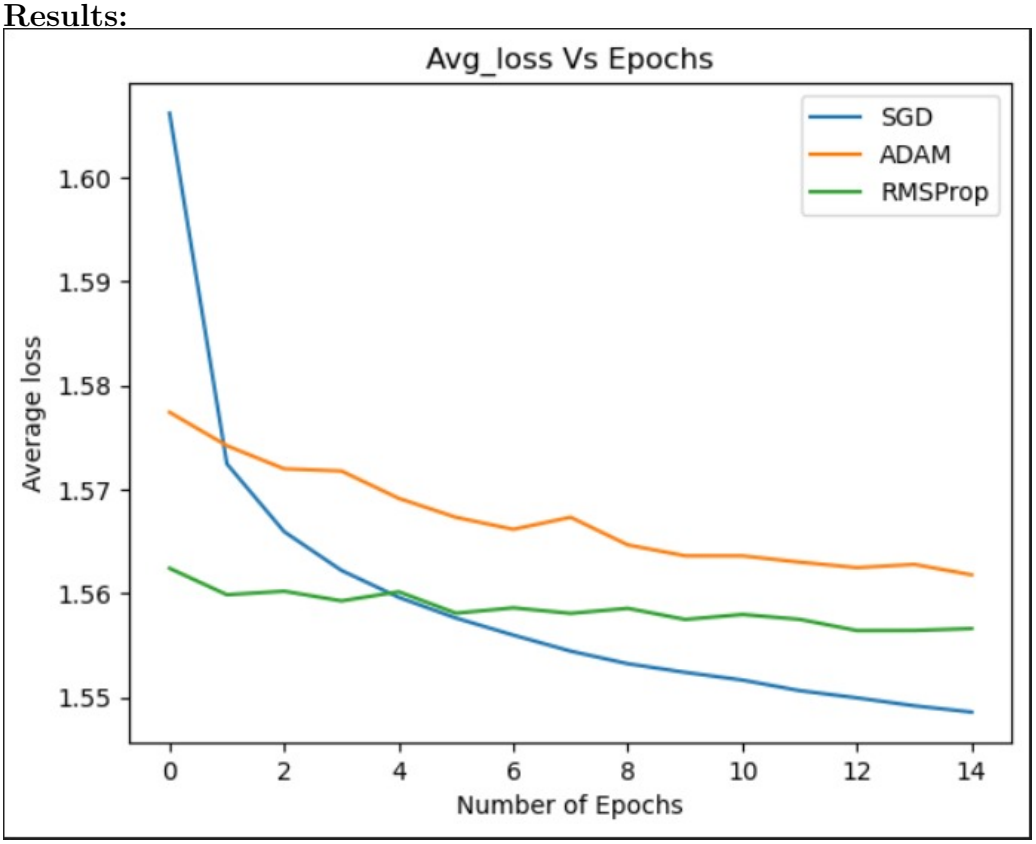
4. **Comparing Different Optimizers:**

   - **Optimizers chosen:**
     1. Stochastic Gradient Descent,
     2. Adam,
     3. RMSprop
   - **Performance comparision:**
     SGD Performed better for 15 epochs.
     We can see the performance comparission from the below graph.

   - **Accuracy and Training Time Analysis:**

Optimiser	**Accuracy**	**Training Time for 15 Epochs**
SGD	91.12	164.7250738143921
ADAM	88.24	170.9005675315857
RMSProp	88.38	150.74425625801086

   - **Results:**



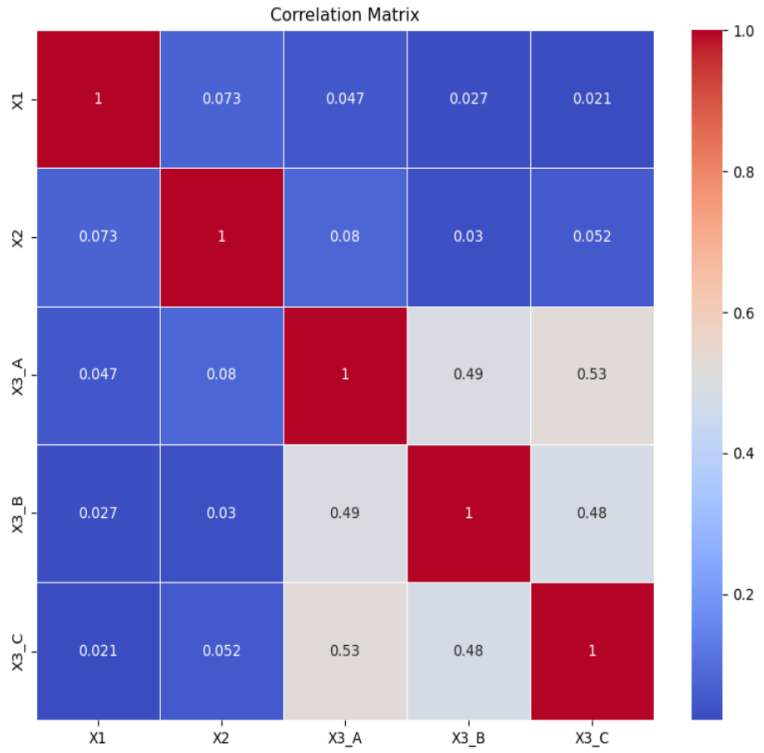
5. **Analysis:**
   From the above table and graph, we observe that for 15 epochs, the accuracy for SGD is more followed by RMSprop, and then ADAM optimiser
   Trainig time for ADAM optimiser is more , followed by SGD and least for RMSprop
   Average loss in the initial epochs is high for SGD but as epochs increased the avg loss is less compared to other two.

3 Question 3: Predicting Sales Revenue

1. **Data Preprocessing**: Read the sales data in csv format. Encoded categorical value which is X3 in our case using 1 of K encoding (or 1 hot encoding) by using the function **pandas.get_dummies**. I am replacing X3 by 3 coulums X3_A, X3B and X3C. If $X3 = \{i\}$ then $X3\_\{i\} = 1$ for i in {A,B,C}
Why we are converting categorical values to 1 Hot encoding?
Categorical values are converted to 1-hot encoding in Python because machine learning models typically only accept numerical data. In 1-hot encoding, each unique category is represented by a new column, and each column contains a binary value indicating whether the original categorical variable belongs to that category or not. This allows machine learning models to more effectively learn the relationships between the categorical variables and the target variable.

2. **Exploratory Data Analysis (EDA):** Defining traning set as {X,y} where X includes all those columns after dropping y, and y includes only target variables.

3. **Feature Selection:** I am dropping feature coulumns i and j if $absolute\_corr\{i, j\} > 0.8$
The absolute correlation matrix is found by calling the funtion **X.corr().abs()**
The corellation matrix is shown below with a number near 1 means higher correlation.



Why we are removing highly co-related features?
Highly correlated features refer to variables that have a strong linear relationship with each other. When two or more variables are highly correlated, they carry almost the same

information, making it redundant to include all of them in a model.

4. **Linear Regression Model:** We Split the data into traing set $\{X\_train, y\_train\}$ and test set $\{X\_test, y\_test\}$.
We ran gradient descent on $\{X\_train, y\_train\}$ to minimise the error function:-

$$E(\theta) = \frac{\Sigma_{i=1}^{100}(X\_test^{\intercal}\theta(i) - y\_train(i))^2}{2}$$

At each step of gradient decent we updated weights θ as

$$\theta_{k+1} = \theta_k - \lambda \boldsymbol{\nabla} E$$

Where we took λ= 0.01 and k <= 99. (for 100 iterations) and k <= 999 for 1000 iterations.

Finally we used the above model to get predictions for $\{X\_test\}$ as:

$$y\_pred = X\_test^{\intercal}\theta$$

5. **Model Evaluation:** And then we calculated the Mean absolute error by the following code in python:

```
(y_test - y_pred)**2.mean()
```

Similarly other errors where calculated as:

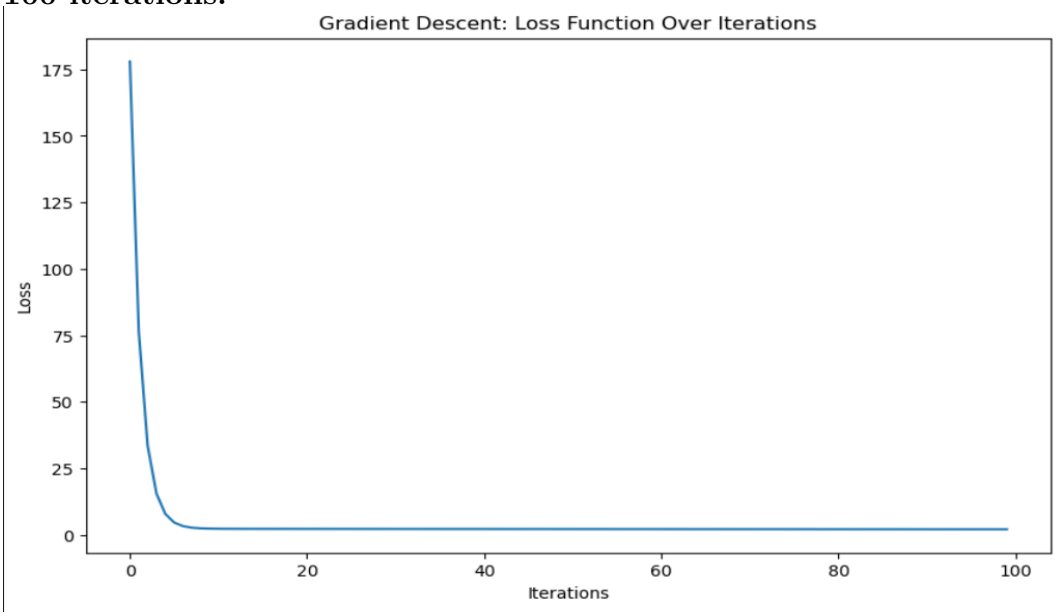
```
mean squared error = ((y_test - y_pred1) ** 2).mean()

root mean squared error = np.sqrt(mse1)
```

Here we do a comparision of errors obtained by running different epochs sizes.

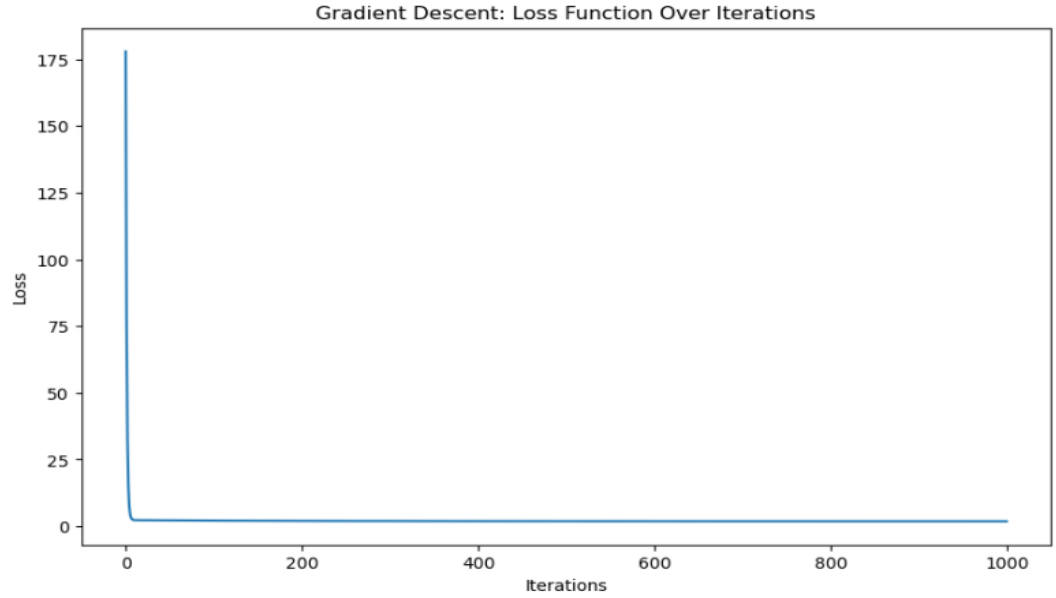
No. of Epochs	**Mean absolute Error**	**Mean Square error**	**Root Mean Square error**
100	1.7817667922248157	4.820463918186302	2.195555491939637
1000	1.71511920634073	4.547392187075927	2.1324615323789375
10000	1.718132688023921	4.530103524263401	2.1284039852113135

Interpretations:
Thus increasing the number of iterations much more may lead to overfitting to the training data $\{X\_train, y\_train\}$ and thus poor performance on test data $\{X\_test, y\_test\}$
For example if we increase the iterations to around 10000, we see an increase in mean absolute error loss as given in the table above.

Graphs of Loss function Vs no. of iterations:
Below is the graph plotted for loss function with respect to number of iterations:
100 iterations:



1000 iterations:



6. **Model Interpretation:**
   Coefficeints of Linear regression model obtained (θ) after running 100 gradient descent iterations are:-

$\theta_{X1} = 3.101279$
$\theta_{X2} = 0.695296$
$\theta_{X3A} = 0.141923$
$\theta_{X3B} = 0.122751$
$\theta_{X3C} = 0.257776$

Coefficeints of Linear regression model obtained (θ) after running 1000 gradient descent iterations are:-
$\theta_{X1} = 3.010739$
$\theta_{X2} = 1.939450$
$\theta_{X3A} = -0.066712$
$\theta_{X3B} = -0.082384$
$\theta_{X3C} = 0.370290$

We clearly see a difference between the weights for the 2 models.
Interpretations:
On increasing the number of iterations from 100 to 1000 weights tend to decrease and thus fit more to the training dataset: $\{X\_train, y\_train\}$
Thus increasing the number of iterations much more may lead to overfitting to the training data $\{X\_train, y\_train\}$ and thus poor performance on test data $\{X\_test, y\_test\}$
For example if we increase the iterations to around 10000, we see an increase in mean ebsolute error loss as given in the table above.

7. **Predictions:** We did predictions for three data points from the model corresponding to (θ_{1000}) given by:
($\theta_{X1} = 3.010739$, $\theta_{X2} = 1.939450$, $\theta_{X3A} = -0.066712$, $\theta_{X3B} = -0.082384$, $\theta_{X3C} = 0.370290$) and got the following result:

X1	**X2**	**X3**	**y_pred**
10.5	1	A	33.485497
12.3	0	B	36.949705
8.7	1	C	28.503169

End.