**WINTER OF CODE 6.0**

**ML Bootcamp**

**PROJECT REPORT**

**By Dhruv Agrawal**

**23JE0060**

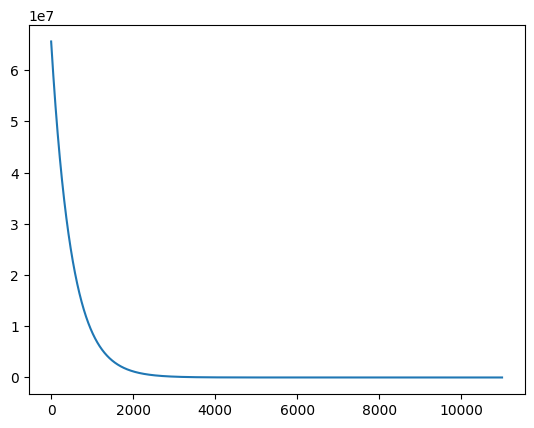
**TABLE OF CONTENTS**

1. Linear Regression
2. Polynomial Regression
3. Logistical regression
4. K-nearest neighbours
5. Neural network
6. K-means clustering

**Linear Regression**

The aim was to fit a straight line through the provided data. When I initially wrote this algorithm I didn’t apply any vectorisation which resulted in exorbitant training time, taking nearly 1 hour to run 1000 iterations. As a result my first reaction was to apply complete vectorisation over the algorithm. Next important step was applying normalisation over the dataset (I applied Z-score Normalisation) which was one of the most important steps. Then I experimented with different values of learning rate alpha to choose the optimal learning rate.

The final alpha obtained was **1e-3,** the r2 score at this alpha is **1.0.**

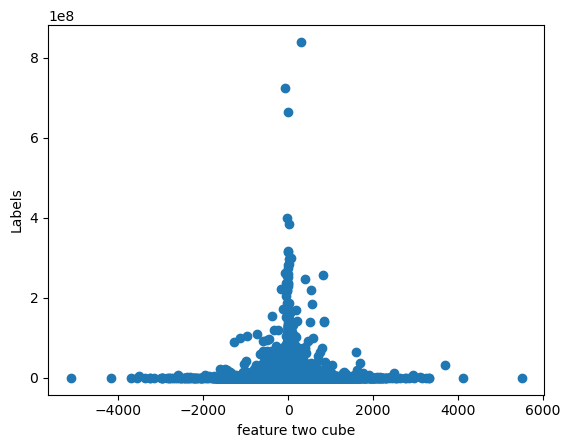
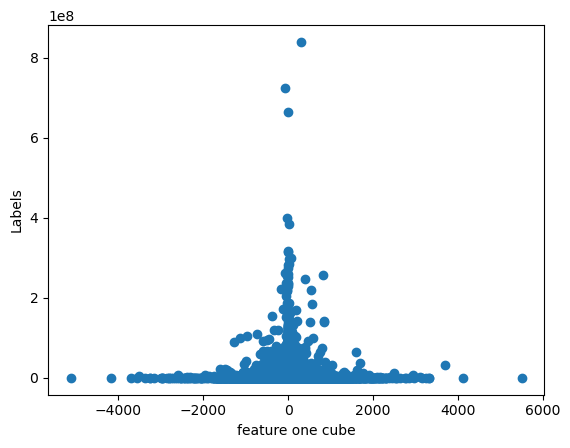
**Polynomial Regression**

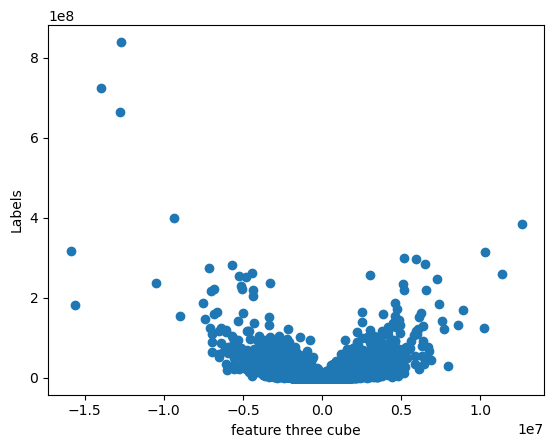
* ***Dataset Analysis:***

The provided dataset had 50000 training examples with each one containing 3 features, say x1, x2, x3. I divided the dataset into two parts namely the training dataset and the cross validation set with the training set containing 40000 examples and cv set containing the remaining 10000.

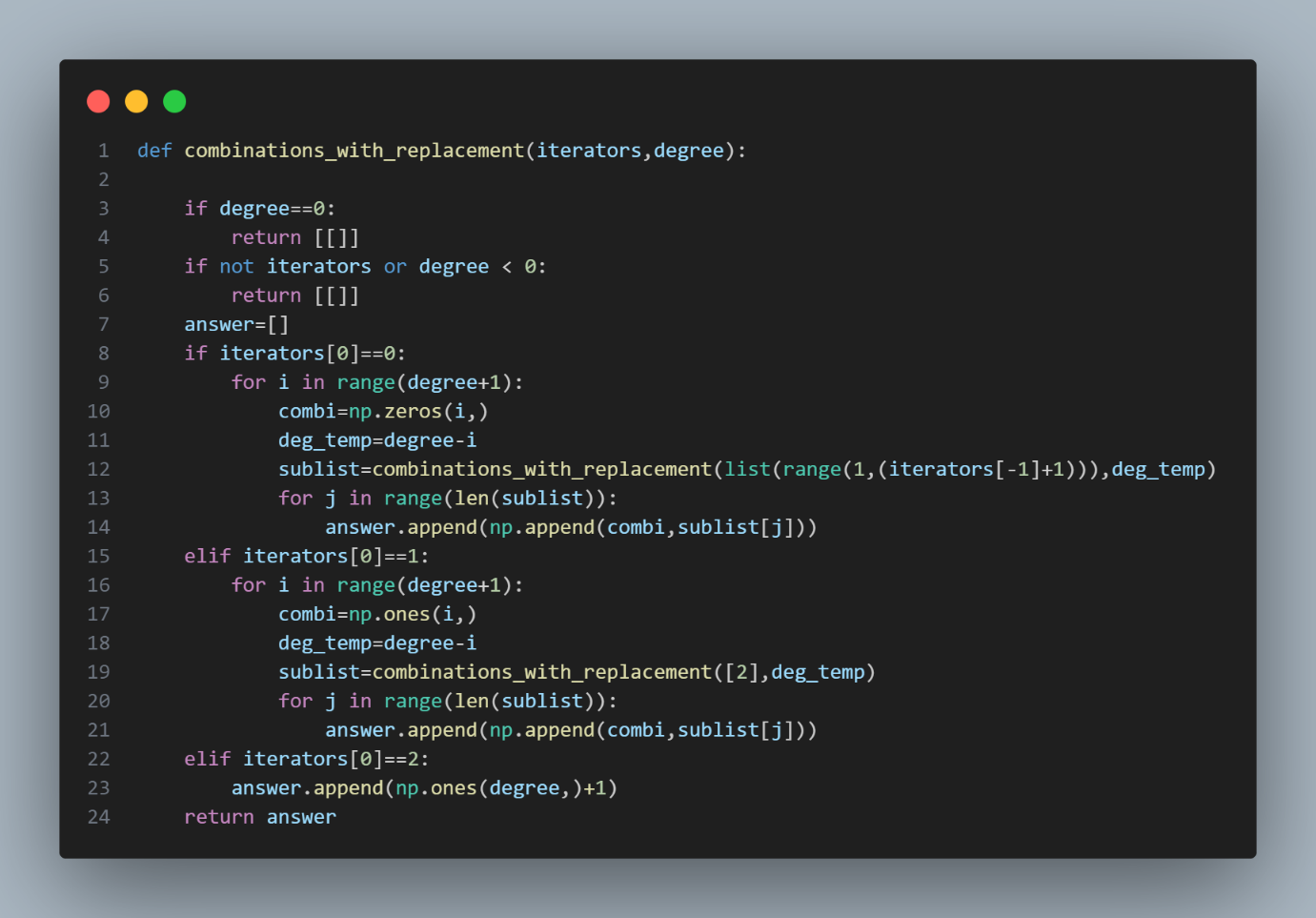
* ***Creating the polynomial:***

The first step in creating this algorithm was to decide the degree of polynomial, so I decided to plot different features against the labels in order to decide on the degree of polynomial.





Since the cube of third feature looks like it will fit in a parabola I decided to create a polynomial in even degrees of x\*\*3 ,i.e., x\*\*6 or x\*\*12. The next step was creating the polynomial, at first I tried to manually create the terms necessary for polynomial but I soon realised that it was not a sensible task also it did not yield great results either. So, I tried to create a function which returns all possible combinations of three features upto any degree entered.

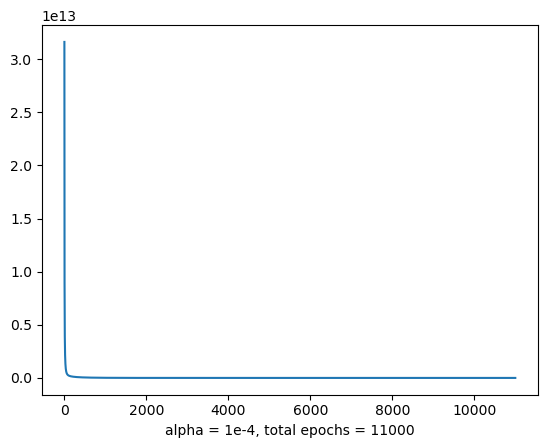


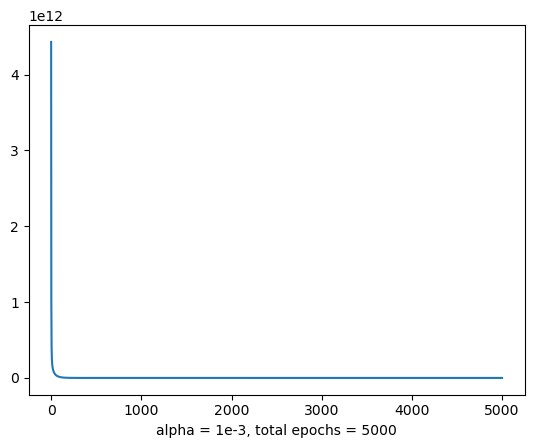
* ***Applying gradient descent***

The training dataset was divided into two sets namely the training dataset containing 40000 examples and the cross validation set of 10000 examples. To speed up computation I divided the training dataset into batches of 32 each. Also, I used z-score normalization.

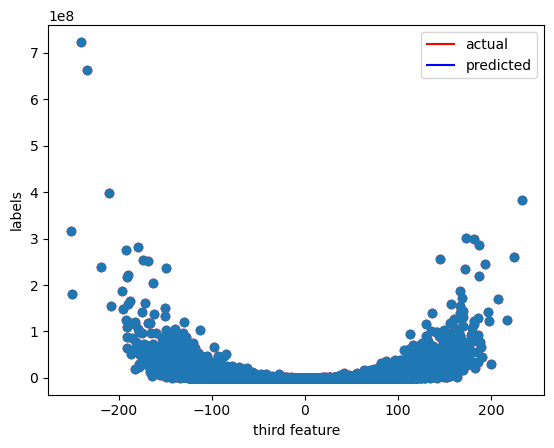
The next step was choosing hyperparameters at first I initialised it with **α=1e-4 and λ=0,** because this is a large dataset it does not really require normalisation but I still kept in in the model for generalisation purposes. Anyways, the r2 score with this setting was nearly one at **0.999999999.** Then I increased the alpha to **1e-3** and lambda was kept same at 0, The r2 score obtained with this set of features was **1.0**

As cost tended to zero(*precisely cost came out to be* ***4.04275e-16*** *after 5000 epochs*) and much quicker than previous alpha so it is computationally cheap as well.





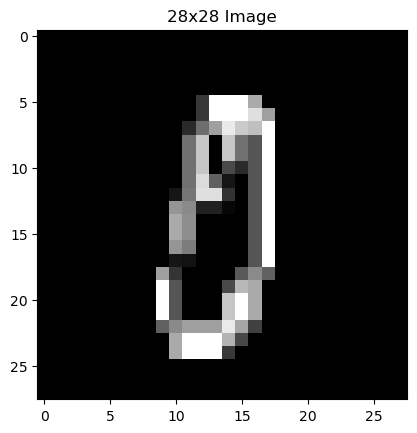
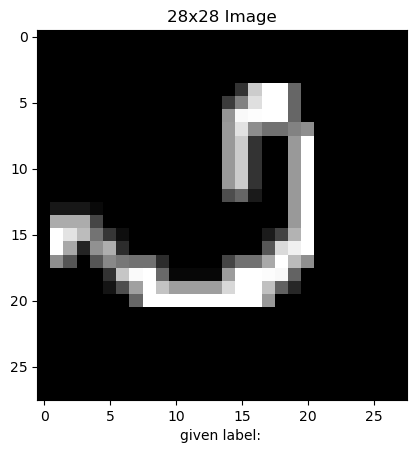
***Actual v/s predicted labels***



***Classification***

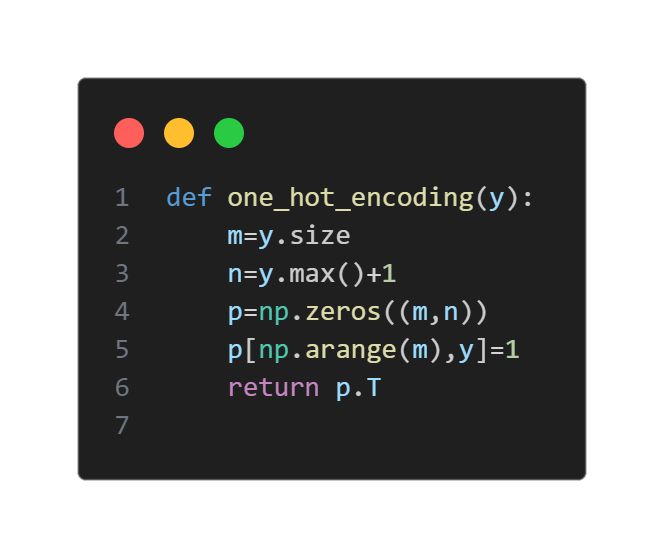
* Dataset Analysis

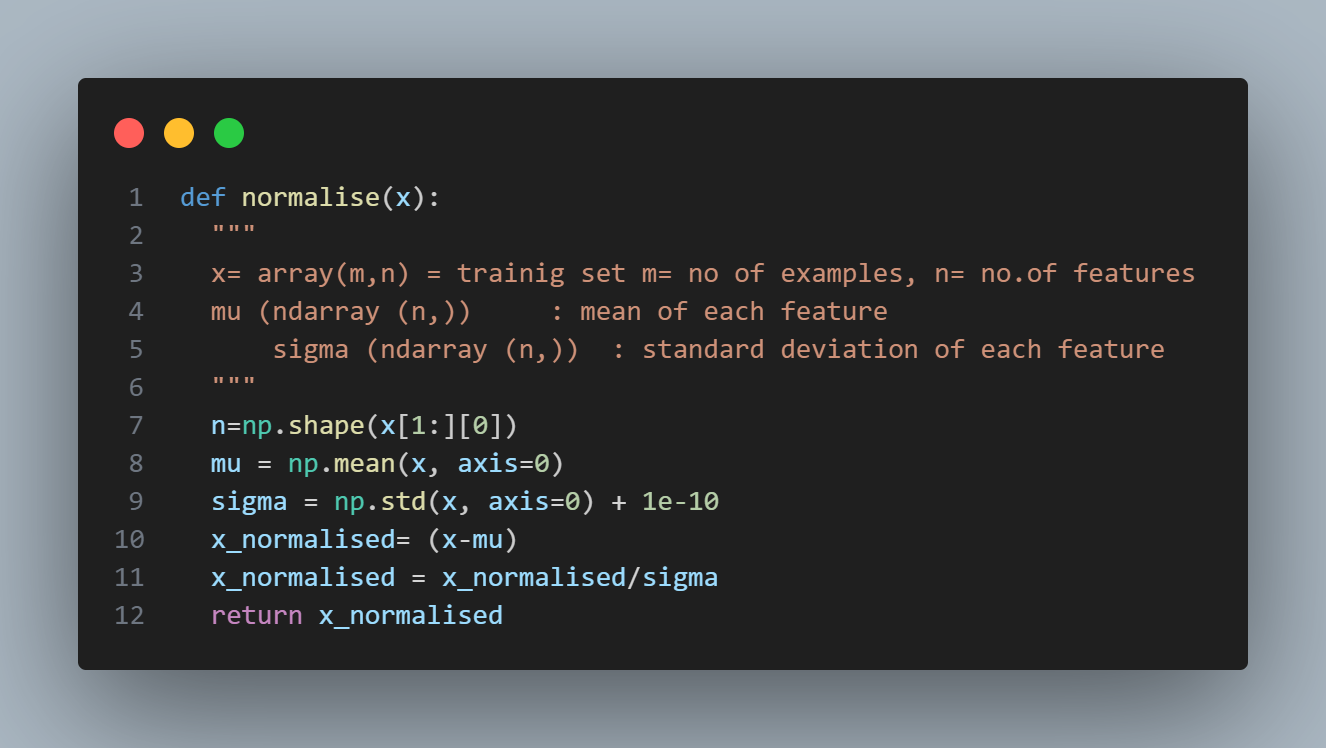
The provided dataset has 30000 examples with 784 features each, which represent the values for all the pixels of a 28\*28 grid with each such grid showing the picture of a digit written in a foreign language.

******The first column shows which number it is supposed to be.

**Logistical Regression**

Since we had to classify the data into more than two classes I decided to go with the one vs all approach. A crucial part for this approach is one hot encoding of the labels. Since the training dataset was large I decided it will be best to vectorise the entire code the code I ended up using for one hot encoding was:



The training data had a large range (0-255) so I normalised the training data with the z-score normalization method.

After running into invalid outputs many times from both the normalisation function (due to division by zero) and the cost function (due to zero in logarithmic function) I added a small value epsilon (1e-10) to prevent such an error in future.

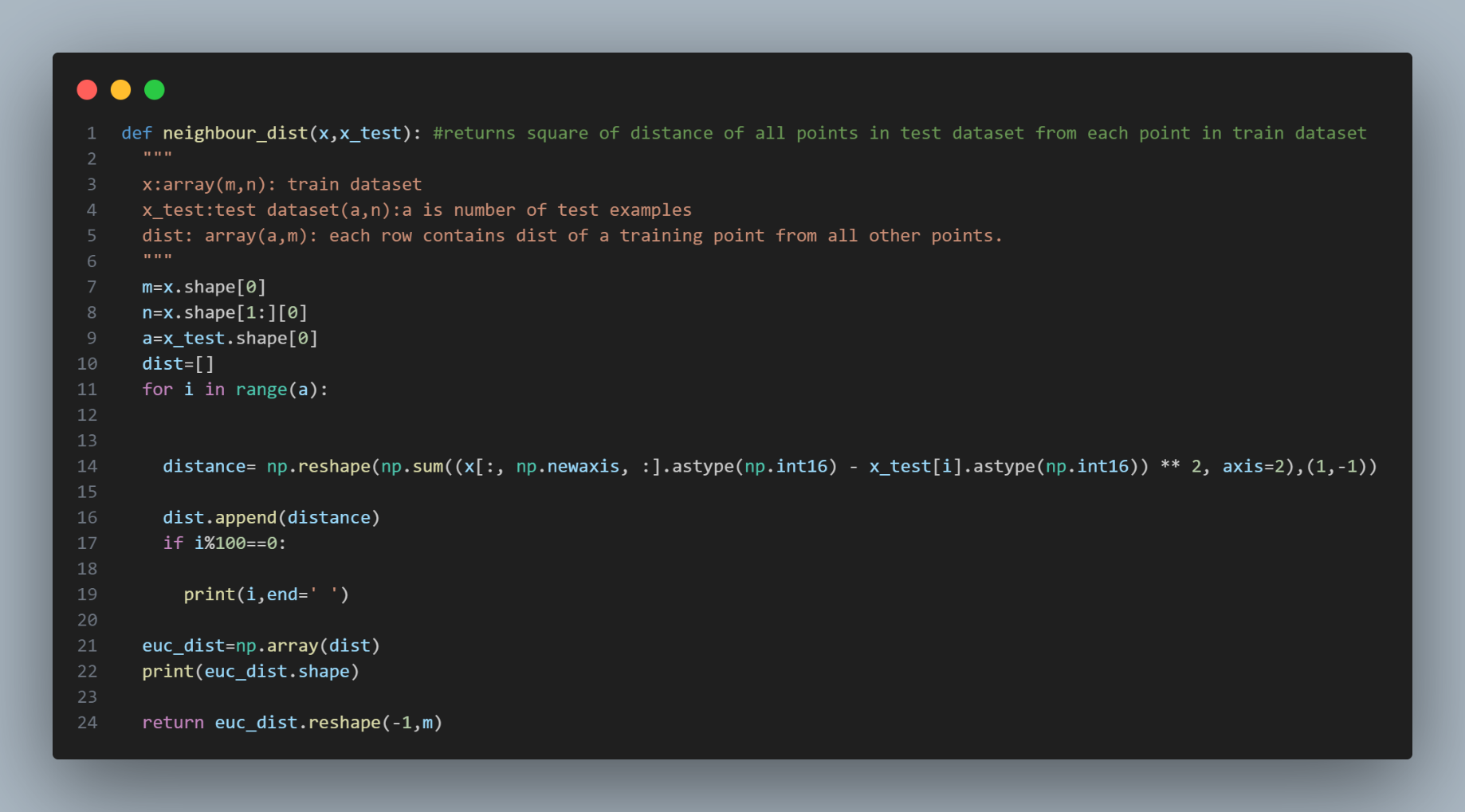
Surprisingly, choosing the value for learning rate (alpha) for this function was not an issue as even values of alpha as high as 10 didn’t cause the cost to increase.

I split the training set into training set and the cross-validation set with the former containing 24000 examples and latter the remaining 6000 examples from the original training set. I shuffled the training set before splitting it to make sure the model is trained on every type of data, as a result I attained a total accuracy of **98.5%** on training set and **96.5%** on cross validation set which indicated there was no overfitting.

**K-nearest neighbours**

This algorithm focuses on classifying data by checking the most probable neighbour in its vicinity. The number of neighbours to be considered is decided by the value of k. I used Euclidean distance for my project. I decided to not take square roots of [ ∑(x-y)^2] to save computation as it makes no difference on our algorithm’s functioning. I vectorised the code so that on each iteration the squared distance of a test example from all training examples. Passing the whole test array to the distance equation was not possible because of computational limitations (Memory error).

After trying to divide the test set into mini batches the most efficient method was found to be sending just one example at a time.



I also vectorised the remaining algorithm resulting in training time reducing from a few hours to nearly **8 minutes**. I split the original training set into training set and cross validation set of size 24000 and 6000 respectively.

Choosing optimal value of k is crucial for this algorithm’s performance. If the value of K is too small it might cause overfitting subsequently if k is too large it may cause underfitting. The graph of K v/s accuracy obtained is.

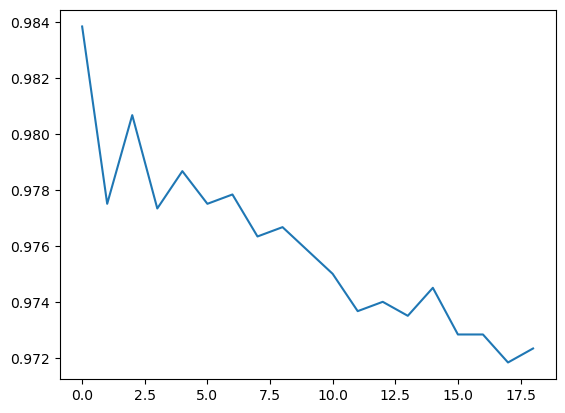


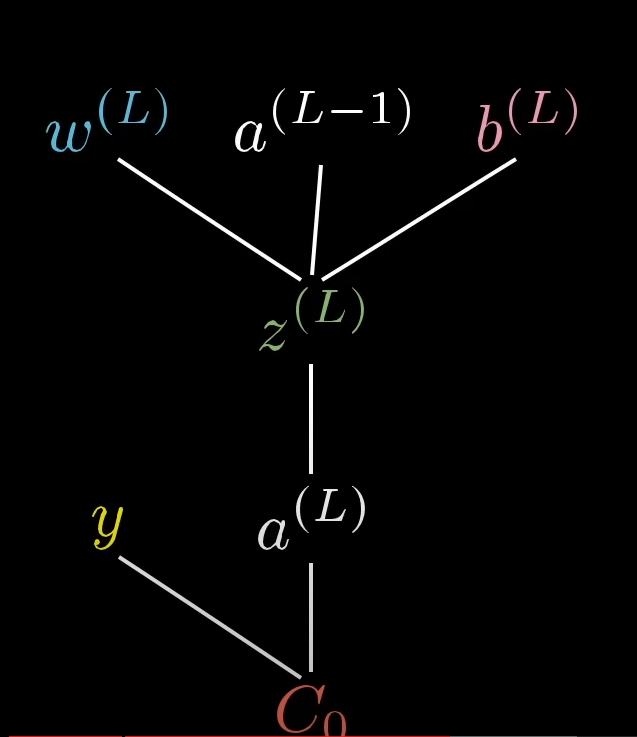
Figure 1: K v/s Accuracy

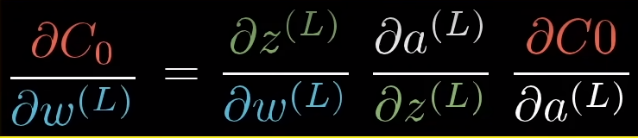
**Neural Network**

The first step in creating a neural network was deciding the architecture of the network, since we are dealing with images and multiclass classification I decided it is best to use softmax function as output layer and Relu function in hidden layers because we are passing pixel brightness values so a layer returning a negative value makes no sense as neural networks learn to recognise patterns.

The next important step was deriving the equations for **back-propagation**, mainly how to calculate the gradients of different layers with respect to the loss which in my case is **cross entropy loss**. The gradient can be calculated by using chain rule of differentiation.

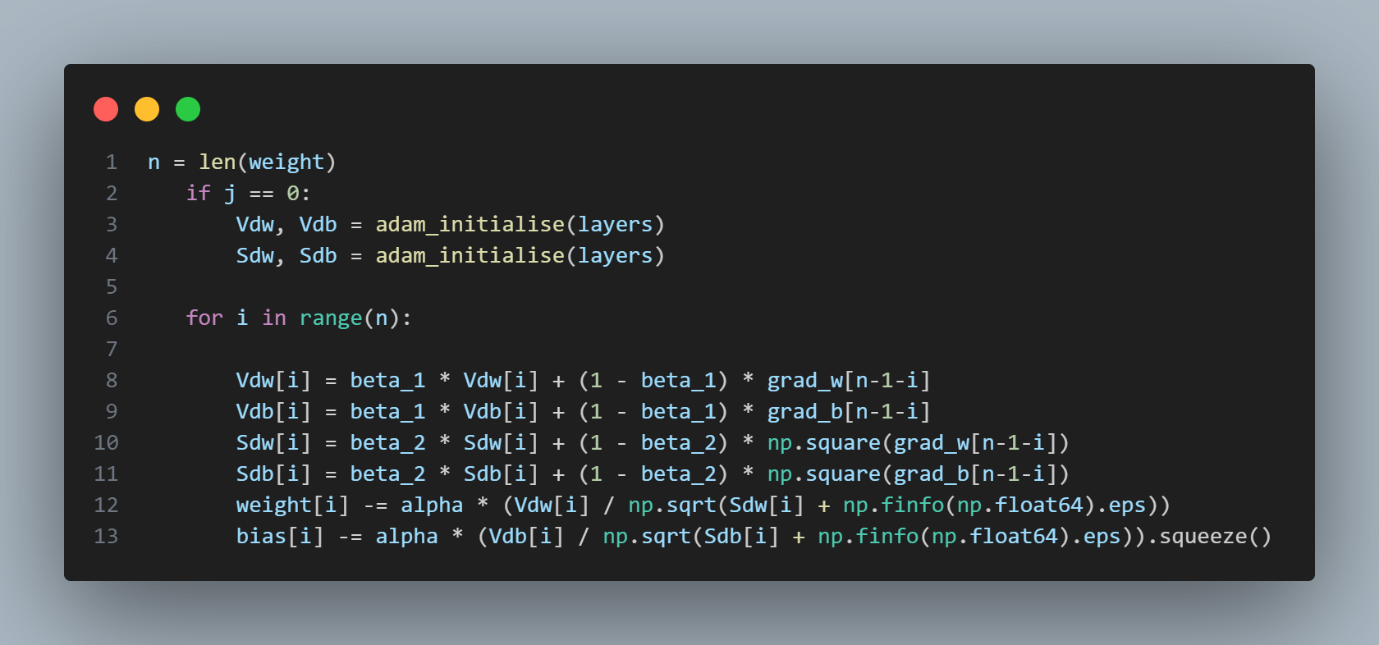
Here is an example of how I approached calculating the equations for the derivatives.





* Optimisation algorithm

The next step was applying the **adam optimisation** function to speed up convergence of the function.



* ***Initialisation***

The next step is initialising the weights initially I simply used **numpy.random.randn** for initialisation. But later I resorted to using the

**He initialisation.**



\

* Hyperparameter Tuning

The biggest challenge for me in creating the neural network was hyperparameter tuning.

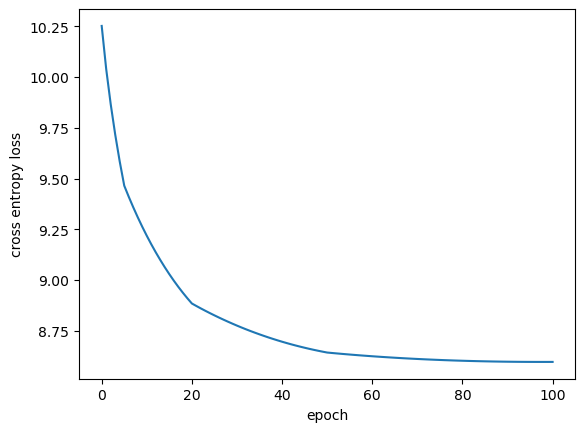
* + - 1. learning rate(alpha)
      2. parameter for momentum step(beta1)
      3. parameter for RMS step(beta2)
      4. batch size
      5. Number of epochs

What made it particularly difficult was that varying any of these parameters even slightly could result in the function overflowing. I tried a set with:

* + - 1. learning rate (alpha) =0.00000025
      2. parameter for momentum step (beta1) =0.7
      3. parameter for RMS step (beta2) =0.999
      4. batch size =128
      5. Number of epochs=200

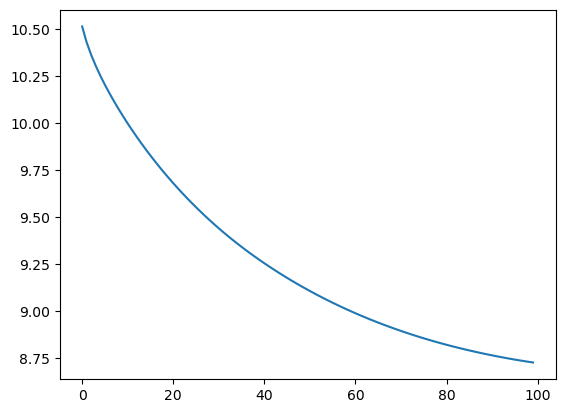
The cost decreased till 150 epochs before suddenly increasing.

Although some good results were also obtained:



alpha=0000005; beta1=0.6; beta2=0.999; batch size=32

accuracy obtained by this setting on cross validation set was 85%.



Alpha= 0.0000003

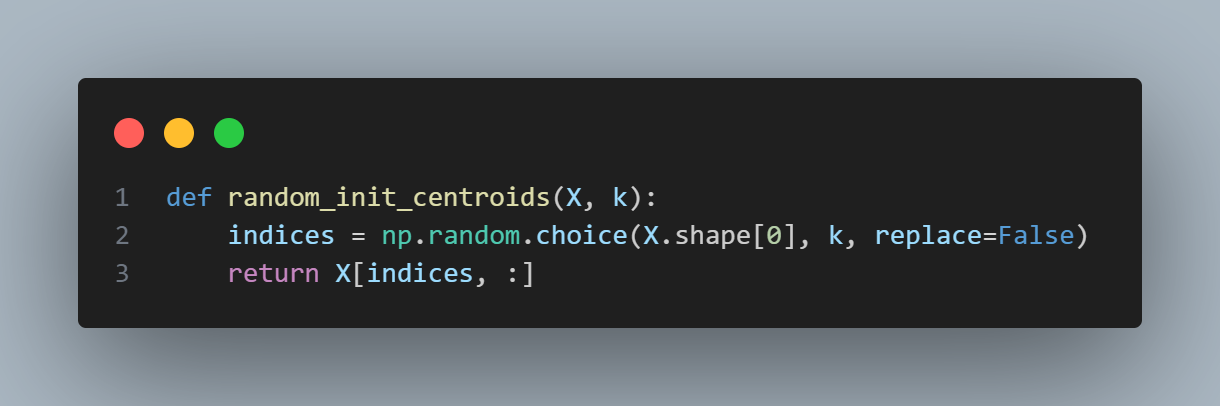
beta1=0.7; beta2=0.999; batch size=128

accuracy=80%

**K-means Clustering**

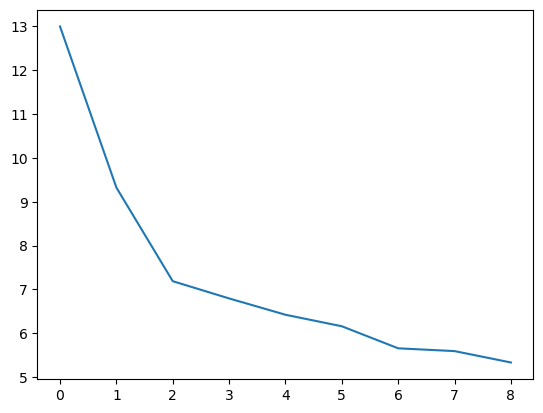
This algorithm works by initialising random centroids initially then updating them over iterations. The problem I was initially facing was with initialisation of centroids as all my centroids were being initialised at random locations far away from each other so all the data points ended up choosing the same centroid thus preventing the centroids from updating.

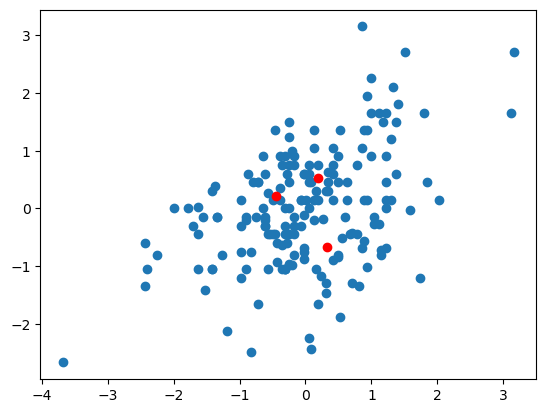
So I changed my initialisation strategy instead of random points I initialised the centroids at random points which are present within the dataset.



This allowed the centroids to be updated.

Now, to find the optimal number of centroids I used the elbow method.

The graph obtained was:

Thus, the optimal value of number of centroids obtained was 3.