ML Bootcamp

Project Report

Prachi Vakshi

IIT (ISM) Dhanbad

The problem statement of the project asked us to implement linear regression, polynomial regression, logistic regression, KNN for classification and N – layer Neural Network with the help of numpy, pandas and matplotlib libraries only.

Implementation of Linear Regression from scratch:

The training data set had 50000 examples with 20 features each. Using the data, we had to fit such a linear function to the data which would minimize the cost or the error. To work on this idea, we created an array for weights of suitable size and a variable for bias and initialized them with zeros. Then we predicted the output and found the error for all the examples. The mean squared error function was used as the cost function because it is a two-degree polynomial so will have only one minimum where we'd get the lowest cost. With an arbitrary value for the learning rate, gradient descent was carried out. Several values for gradient descent were tried unit we found the value nearest to optimal. By value nearest to optimal, we mean, a value which would make the cost converge faster than other values, without overshooting at any point. When the cost saturates, we use the value of the parameters (weights and bias) to predict the labels for unknown data.

Initially, I used loops to calculate the predicted value and cost by iterating over each example. This took a lot of time. So, I vectorised my code which made the training much faster and made sure to use the same approach in further programs as much as possible.

During training, several values for learning rate were tried:

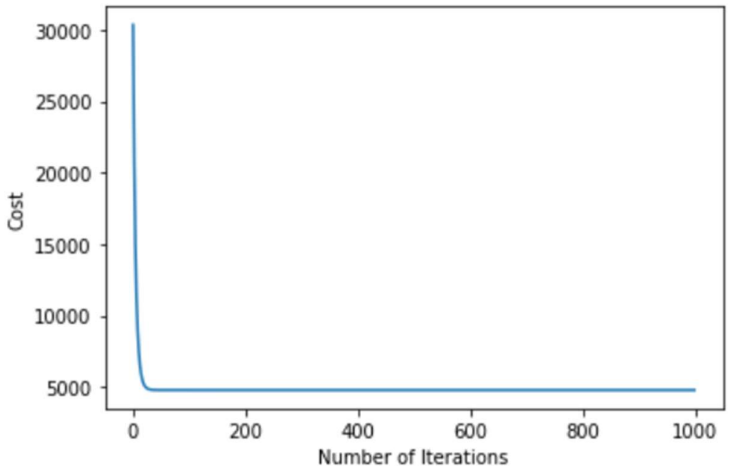
- 0.001: With this value of learning rate, the cost decreased constantly but slowly so I tried a higher learning rate.
- 1: With this learning rate, the cost decreased after certain iterations and then increased after some. This meant that the learning rate was too high and the random movement of the cost was nothing but "overshooting". So, I tried a lower learning rate which is still higher than 0.001.
- 0.01: With this value of the learning rate, the cost decreased constantly and faster than in the case of 0.001.
- 0.1: To find the best case, I further tried 0.1 as the learning rate. The cost decreased constantly and saturated much faster.

- 0.3: To check if a higher learning rate would do the job, I tried 0.3 but encountered overshooting.

So, I concluded that 0.1 should be the learning rate for which the cost decreased in the following way with the number of iterations:

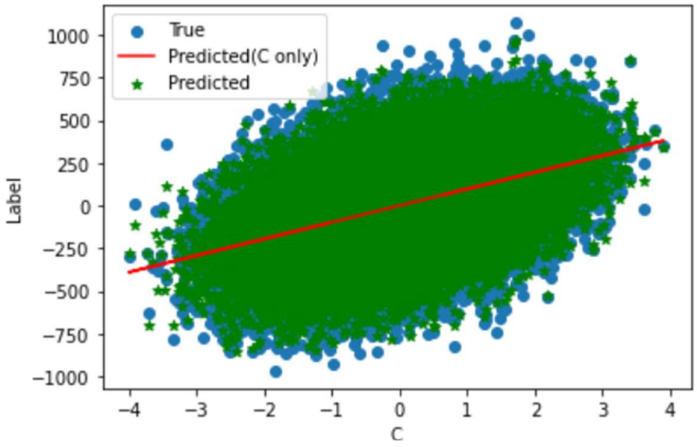
Number of Iterations	Cost
100	4769.76872476984
200	4769.768702764219
300	4769.768702764219
400	4769.768702764218
500	4769.768702764218
600	4769.768702764218
700	4769.768702764218
800	4769.768702764218
900	4769.768702764218
1000	4769.768702764218

The cost was plotted with the number of iterations for the same learning rate and the graph came out to be like this:



We see how the cost has saturated.

The following plot shows how the predicted values fit the given values for the training dataset:



The red line shows how the values vary with s single feature ‘C’.

Implementation of Polynomial Regression:

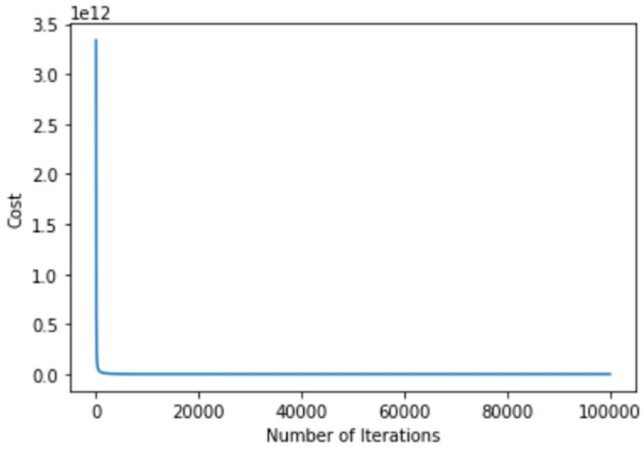
The training data set given to us had 50000 examples and 3 features each. Using feature engineering, we created more features of higher degrees by multiplying the given features with each other in different proportions. Thus, we fit an n-degree polynomial to the data in a way which minimizes the cost. We started with a 1-degree polynomial and increased the degree until the cost got minimised. The mean squared error function was used as the cost function because it is a two-degree polynomial so will have only one minimum where we'd get the lowest cost. For each degree, an array of weights and the bias was initialised to 0 and gradient descent was carried out. We iterated till the cost was minimised. On increasing the degree from 1 to 5, the saturated cost decreased. For polynomials of higher degrees, the cost reduction was very slow. So, the final function was taken to be a 5-degree polynomial. For a 5-degree polynomial, I tried different learning rates and got the following observations:

- 0.001: The cost decreased constantly but very slowly which indicated that we should try a higher learning rate.
- 1: Overshooting was observed which indicated that the learning rate is too high.
- 0.01: The cost decreased constantly but faster than it did in the case of 0.001.
- 0.03: Overshooting was detected implying that the learning rate is too high.

So, I concluded that 0.1 should be the learning rate, for which, the cost decreased in the following way with the number of iterations:

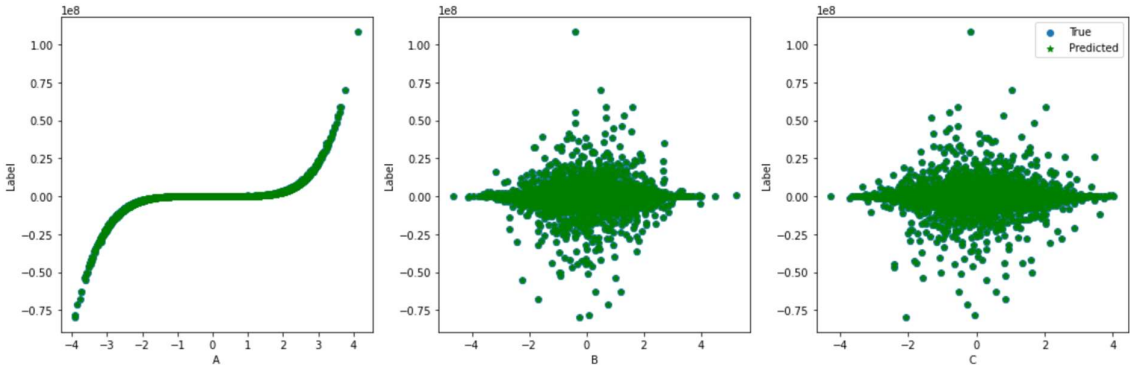
Number of Iterations	Cost
10000	69878806.48381823
20000	278440.7315606625
30000	1112.4816758378572
40000	4.447734229413477
50000	0.017787931870940493
60000	7.115356997298977e-05
70000	2.846577508306762e-07
80000	1.1390158136056154e-09
90000	4.601985135227778e-12
100000	2.0493384335477914e-14

The cost was plotted with the number of iterations for the same learning rate and the graph came out to be like this:



We see how the cost has saturated. The value of these parameters (weights and bias), obtained after the cost has saturated, are used to predict the labels for unknown data.

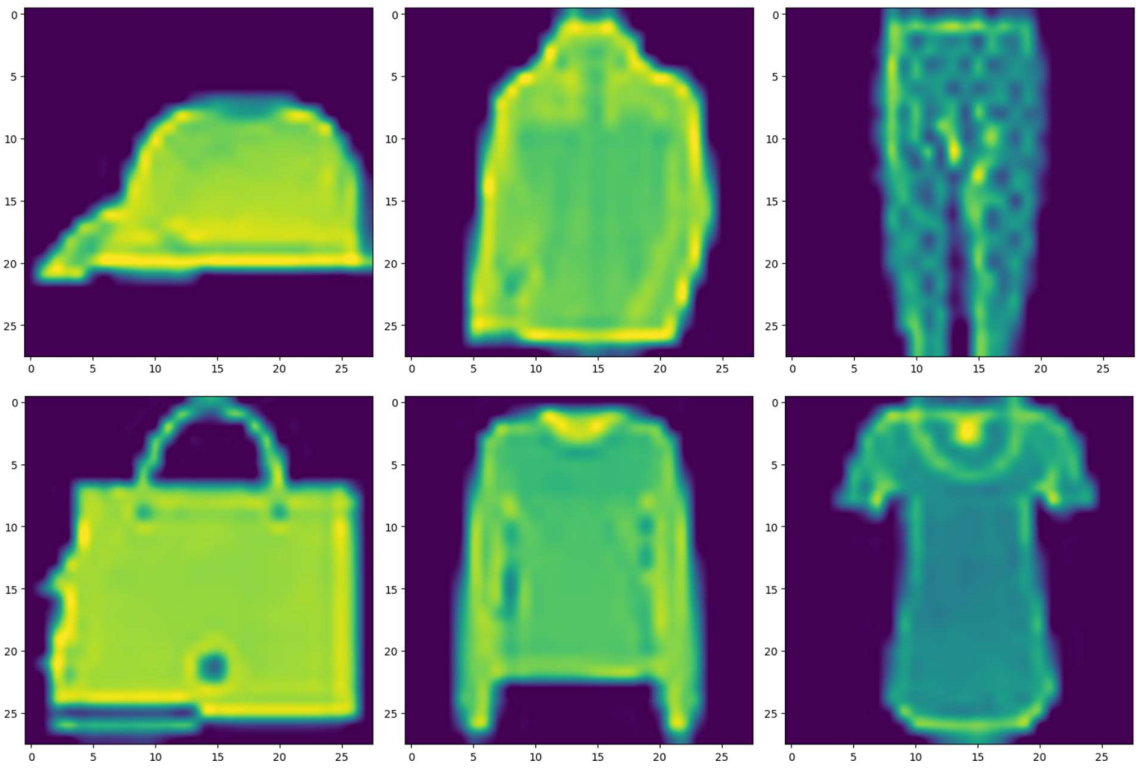
The following plot shows how the predicted values fit the given values for the training dataset:



These plots show how the values vary with the three features given to us originally.

Implementation of Logistic Regression:

The training data set had 30000 examples. Each example was a set of pixels which means, we need to classify the pictures. Plotting the first 6 examples of the training dataset turned out to be like this:



So, the pictures given to us were of clothes, footwear and accessories.

In this case, we do not use the mean squared error as the cost function because it has several local minima (because we are using the non-linear sigmoid function) which would make gradient descent difficult. To get a convex error function, we use the log loss function. Everything else remains the same as earlier – the parameters and the gradient descent but the labels are "one-hot encoded" to facilitate cost calculation.

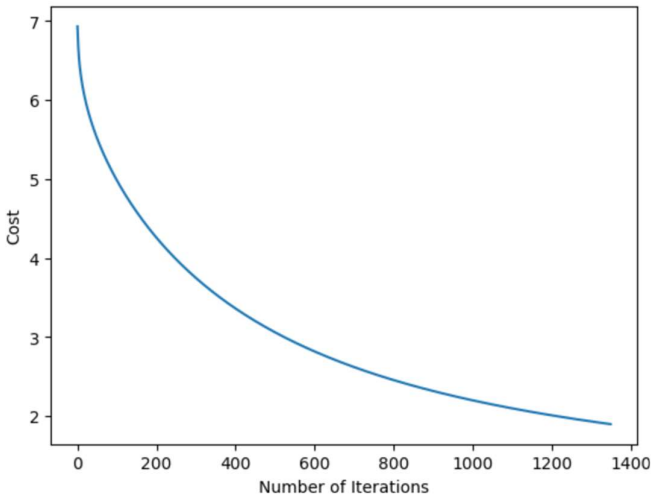
We used the one vs all approach and calculated the cost for different learning rates:

- 0.001: The cost decreased constantly but very slowly which indicated that we should try a higher learning rate.
- 0.003: The cost decreased constantly but faster than it did in the case of 0.001.
- 0.006: The cost decreased constantly but faster than it did in the case of 0.003.
- 0.009: Overshooting was detected implying that the learning rate is too high.

So, I concluded that 0.006 should be the learning rate, for which, the cost decreased in the following way with the number of iterations:

Number of Iterations	Cost
135	4.700064156753328
270	3.8887091830343854
405	3.347810830697584
540	2.9591627797460673
675	2.667598245477146
945	2.262775817420553
1080	2.117636708851244
1215	1.9979527590662902
1350	1.8977593897806304

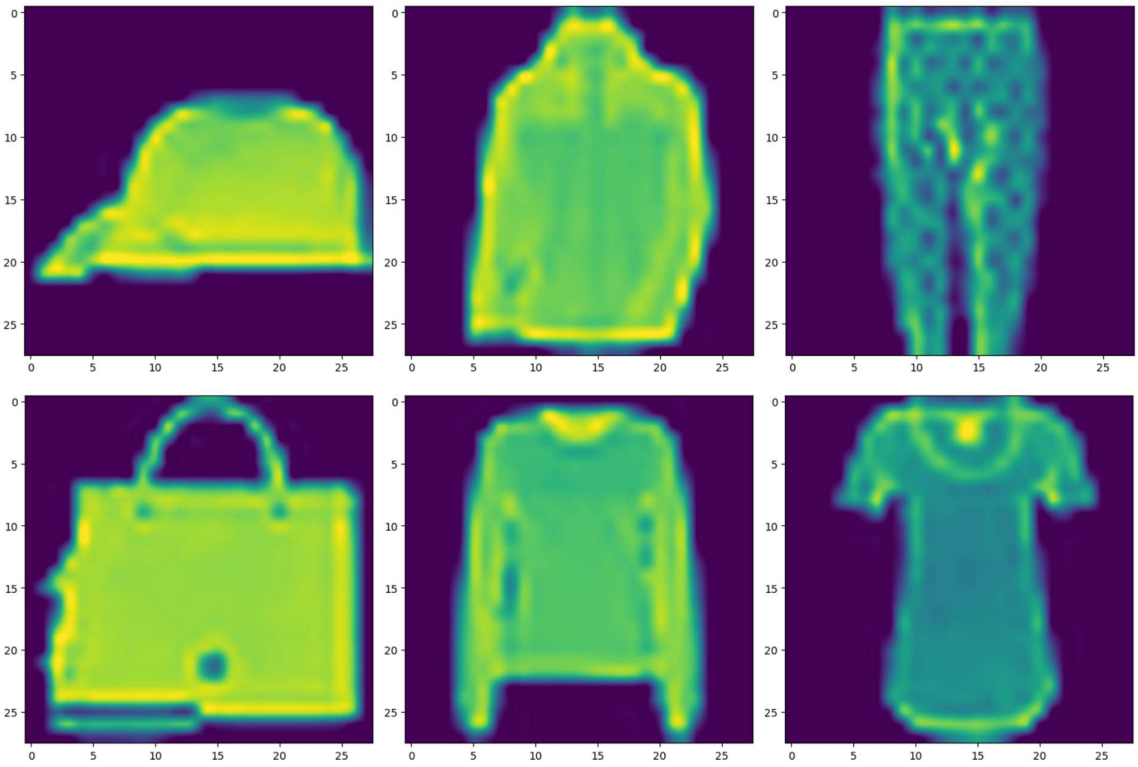
The cost was plotted with the number of iterations for the same learning rate and the graph came out to be like this:



We see how the cost has saturated. The value of these parameters (weights and bias), obtained after the cost has saturated, are used to predict the labels for unknown data.

Implementation of KNN for classification:

The training data set had 30000 examples. Each example was a set of pixels which means, we need to classify the pictures. Plotting the first 6 examples of the training dataset turned out to be like this:



In this case, we divide the training dataset into two parts. One part will be used as known values while we will try to predict the labels of the other part and compare the answer with the labels to know how well the code works. This algorithm follows "birds of a feather flock together", i.e, it considers something to be of the same kind as that of the majority of its k nearest members. So we try different values of k and see the accuracy.

For the following values of k, we got the following observations:

- 7: The accuracy was 84.14%.
- 9: The accuracy dropped to 81% approximately.
- 5: The accuracy was around 83%.
- 6: The accuracy was higher than 83% but lower than 84.14%.
- 8: The accuracy was higher than 81% but lower than 84.14%.

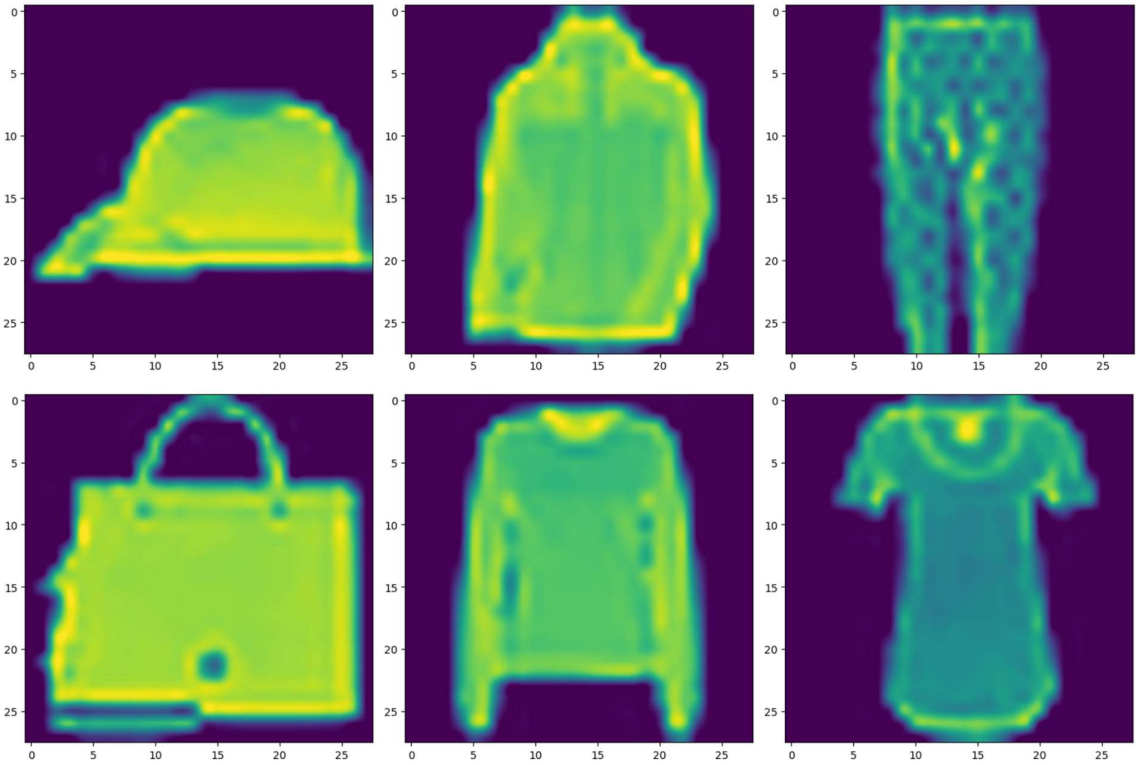
This implied that 7 is the optimal value which would give us the highest accuracy. As we move away from 7, the accuracy falls because we consider too many or too less neighbours and end up getting a faulty idea about the category of the unknown point.

This value of 7 is now fixed and used for predicting the labels for unknown data. We use the same portion of the training data which we had used while training.

Implementation of Neural Network:

We begin with a 1-layer neural network. This would have an input layer and an output layer. The output layer has softmax as the activation function.

This network will be used for classification. The training data set had 30000 examples. Each example was a set of pixels which means, we need to classify the pictures. Plotting the first 6 examples of the training dataset turned out to be like this:



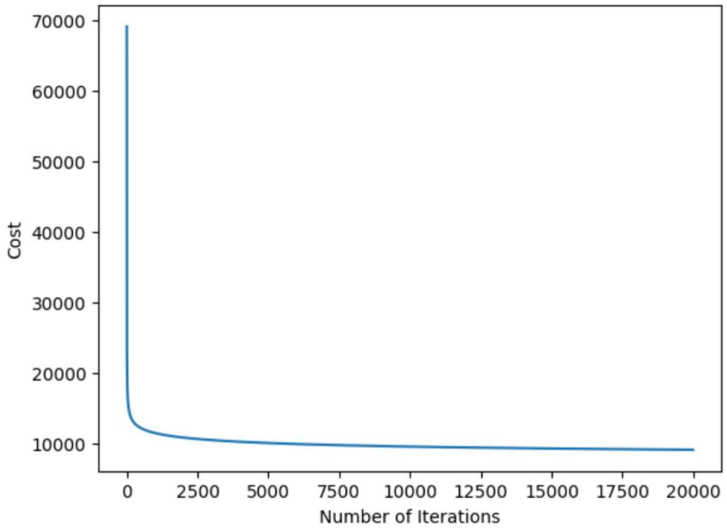
Since we have softmax as the activation function in the output layer, to get a convex curve, we use the cross-entropy function as the cost function. We make an array of weights and biases for this layer with the one-hot encoded label array. We apply gradient descent using backpropagation to get the best values for the parameters which would minimize the cost. Backpropagation is a way of calculating the gradient by using the chain rule. It is faster than the traditional way of gradient descent. The following values of the learning rate were used and the observations were noted:

- 0.001: A runtime error was encountered because the code tried to find the exponential of a value which would give a huge value almost tending to infinity. This meant that the learning rate was too high.
- 0.000001: The cost decreased constantly and saturated at a point.
- 0.000003: The cost decreased constantly and faster than the previous time.
- 0.000006: The cost decreased for a good number of iterations but there was overshooting when the cost was near saturation. This meant that this value is still high for the learning rate.

So, I concluded that 0.000003 should be the learning rate, for which, the cost decreased in the following way with the number of iterations:

Number of Iterations	Cost
2000	10791.718082669398
4000	10199.652481328985
6000	9880.980253392889
8000	9668.515268939665
10000	9511.504004704608
12000	9388.198045646652
14000	9287.41870745959
16000	9202.70317768868
18000	9129.989980575725

The cost was plotted with the number of iterations for the same learning rate and the graph came out to be like this:



At this point, we achieve 89.36% accuracy for training data.

We see how the cost has saturated. The value of these parameters (weights and bias), obtained after the cost has saturated, are used to predict the labels for unknown data for this architecture of the neural network.