CSC 722
Advance Topics in Machine Learning
Project Report

What's Cooking?

By:
Mihir Milind Mirajkar (mmmirajk / 200156826)
Piyush Choudhary (pchoudh2 / 200156175)

Abstract

In this age of globalization, the world has become a smaller place and people are exposed to more cultural diversity than ever before. This has led to people accepting other cultures and trying to while trying to adapt practices from other cultures. One of the biggest effects of the globalization can be seen in the kitchen! People from all around the world are making and eating more food from different cultures. We can see an average American making Pasta, Noodles, Curries, Salads, Sandwiches etc. on a regular basis in their kitchen. But, each type of cuisine requires different ingredients, hence even if a person tries to make these dishes even once, he ends up with a lot of leftover ingredients. But due to some of the similarities in cultures, some ingredients could be used in different dishes as well. Hence, if the person has to cook a dish quickly and has a particular set of ingredients remaining from other dishes, what should he do? Go grocery shopping for remaining ingredients or should he make something with the ingredients he already has? Our project tries to give the answer for the second part. We have trained a neural network which learns the ingredients of a particular cuisine and tries to best of its ability to give a cuisine which could be cooked using the ingredients which you have.

INDEX

Abstract 2

Introduction 4

The Data 5

- Cuisine Ingredients Analysis 5
- Data Preprocessing 6
- One hot encoding 7
- Test Train Split 7

Neural Network 8

- Architecture 8
- Mini Batch Training 9
- Activation Functions 10
- Cost Function 11
- Decaying Learning Rate 12
- Optimizer 13
- Regularization 14

Results 14

Introduction

The idea about this project stems from the belief that there is a way to associate the cuisine category of a dish given the ingredients used to prepare it. Each part of the world has a different cuisine in term of taste, fragrance and texture, this is due to the uniqueness of the ingredients used during the preparation of the dishes. Each cuisine also has a different style and methodology of cooking which gives arise to different styles of similar ingredients. For ages we are able to tell which cuisine is being prepared by just looking or through the aroma of the dish. We are able to do so because we correctly identify the dish due to our ability to guess the ingredients used in it. This project is a small attempt to recreate this unique human ability and empower a machine with this capability. In this project we have examined the capability of the neural network to recreate this human nature. To do this we have created a powerful deep neural network with backpropagation to adjust the weights and biases. The data was obtained from a 2 year old kaggle competition[1] which was hosted by a company named YUMMLY[2]. The company specializes in all types of cuisines and had a lot of data in which all the ingredients to make a particular dish of a particular cuisine are listed. This gave us a huge playground to play with, the only problem being that the data was very sparse. The data was so sparse that our feature vector was less than 0.5% filled. How we tackled this problem and developed a model which predicted with an accuracy of 79% is explained further in the report.

[1] https://www.kaggle.com/c/whats-cooking
[2] https://www.yummly.com/

The Data

The Data we received has the ingredients of about 40,000 dishes. Each dish has a unique ID, a cuisine type and set of ingredients. Across all the dishes we have 1,43,000 ingredients of which we have 6800 unique ingredients. We have 20 types of cuisines which consists of Irish, Mexican, Chinese, Filipino, Vietnamese, Moroccan, Brazilian, Japanese, British, Greek, Indian, Jamaican, French, Spanish, Russian, Cajun Creole, Thai, Southern United States, Korean and Italian. From the initial analysis of data, it can be determined that it is a very sparse dataset as we have total of 6800 ingredients and each cuisine consists no more than 20 ingredients, this means that each row has less than 0.5% of the total ingredients. This creates a problem as sparse feature vectors hamper the learning growth of machine learning models.

Cuisine Ingredients Analysis

The project contains 20 types of unique cuisines from all over the world consisting of Irish, Mexican, Chinese, Filipino, Vietnamese, Moroccan, Brazilian, Japanese, British, Greek, Indian, Jamaican, French, Spanish, Russian, Cajun Creole, Thai, Southern United States, Korean and Italian. For these 20 dishes we have 6714 unique ingredients consisting of salt, tomatoes, onions etc. After doing some analysis on these ingredients we found the top ingredients with the number of dishes in which it appears are:

1. Salt - 18048
2. Onions - 7972
3. Olive oil - 7971
4. Water - 7457
5. Garlic - 7380
6. Sugar - 6434
7. Garlic cloves - 6236
8. Butter - 4847
9. Ground black pepper - 4784
10. All-purpose flour - 4632
11. Pepper - 4438
12. Vegetable oil - 4385
13. Eggs - 3388
14. Soy sauce - 3296
15. Kosher salt - 3113
16. Green onions - 3078
17. Tomatoes - 3058

18. Large eggs - 2948
19. Carrots - 2814
20. Unsalted butter - 2782

Data Preprocessing

Since the data was in a json file with each dish has a unique ID, a cuisine type and set of ingredients was including in a json file with the format as shown below.


```
{
 "id": 48911,
 "cuisine": "filipino",
 "ingredients": [
   "water",
   "jicama",
   "carrots",
   "bamboo shoots",
   "egg roll wrappers",
   "garlic",
   "beansprouts",
   "ground black pepper",
   "vegetable oil",
   "green beans",
   "soy sauce",
   "water chestnuts",
   "diced celery",
   "onions"
 ]
}
```


This was converted to a feature vector with 6714 features and 40,000 tuples and 1 class label. To convert the json file to a feature vector python library "json" was used and a feature vector matrix was created. The data was mapped to different forms and checked for accuracy.

In general neural networks find it difficult to create a model and predict accurately as architectures such as Feed forward network, CNN, RNN, LSTM etc rely on spatial or sequential attributes of the data to learn and tend to learn zeros more than ones i.e. they learn very little. To tackle this problem the solution mentioned on Quora[3] was tried and the data was mapped to [-1,1] where -1 represented absence of a ingredient and 1 represented presence of an ingredient. This yielded an accuracy of 19% which was much lesser than the desired accuracy or the accuracy which was obtained to using [0,1] mapping. Hence [0,1] mapping was used where 0 represented absence of a

[3] https://www.quora.com/Why-are-deep-neural-networks-so-bad-with-sparse-data

ingredient and 1 represented presence of an ingredient. Even though neural networks tend to learn zeros more than one for sparse data, [0,1] encoding worked the best for this situation.

One hot encoding

The class had 20 unique types of cuisines, in the initial phase each cuisine was mapped to a unique numerical value (numerical encoding) and prediction of the value was done. We could have used an integer encoding directly and rescaled where needed. This may work for problems where there is a natural ordinal relationship between the categories, and in turn the integer values, such as labels for temperature ‘cold’, warm’, and ‘hot’. But since our problem had no ordinal relationship and allowing the representation to lean on any such relationship might be damaging to learning to solve the problem. In this case, we gave the network more expressive power to learn a probability-like number for each possible label value. This helped in making the problem easier for the network to model. When a one hot encoding is used for the output variable, it offered a more nuanced set of predictions than a single label. It was more suitable and a better choice to classify using neural network and softmax as the last activation function. Hence our output was a list of 19 zeros and 1 one e.g. for a cuisine which was third on the index we had an output of [0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0].

Test Train Split

Since the test data available on kaggle doesn’t have the actual labels it was not possible for us to test the model on the data given. Hence, we created our own test data which was randomly sampled from the training set. The ratio for train to test data was 75:25. This allowed us to test our trained model on the data which was never seen by the model, but in the process it also reduced the number of samples on which the training could be done. Due to this it was possible that the model had never seen a type of cuisine which was present in the test set. Hence, it could lead to reduction in reliability of the neural network. But since we had a huge training set the chances of this happening was very low and the data was randomly split with different seeds to make sure this does not happen.

Neural Network

Architecture

The neural network used to solve this problem is a deep feed forward network with backpropagation. The structure of our network is a classic feed forward network difference being that the size of neurons in the hidden layers decreases with the depth. Diagrammatic representation of our neural network structure is as follows:

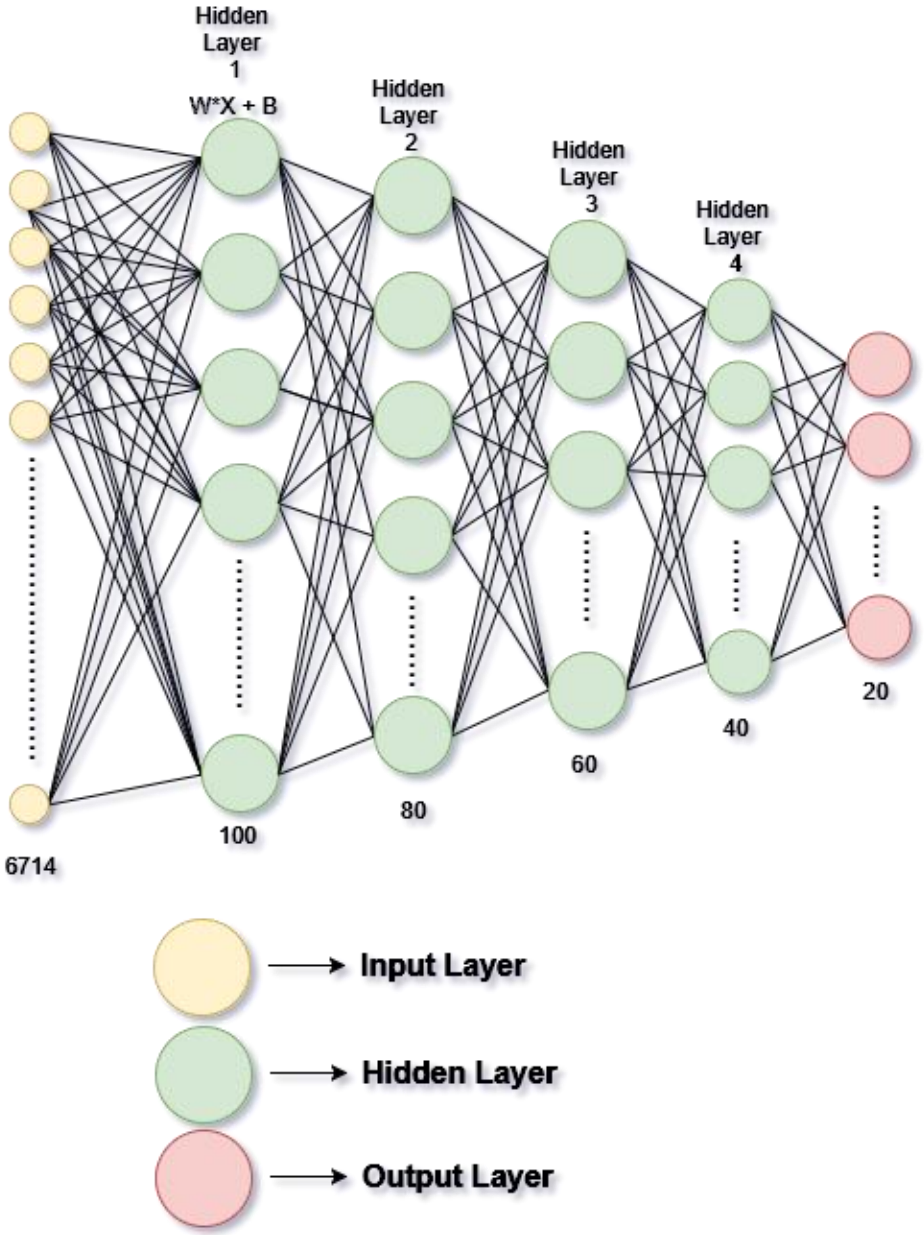

Fig: Neural Network Architecture used in the project

The drastic decrease in the number of neurons from the input layer to the first hidden layer is due to computational reasons and to avoid overfitting. The depth of the neural network was decided by trial and error as a deeper network gave no better results but a shallower network gave a worse accuracy.

The weights were initialized randomly with the range of [0,1] with a normal distribution using tensorflow's truncated_normal function. One another method was to initialize the weights to zero, this is a approach which is sometimes used in preliminary testing as it is easier to use. But this might cause the model to get stuck in a saddle point as backpropagation identifies the set of weights that minimizes the weighted squared difference between the target and observed values, and initializing the weights to zeros would make the model become stuck in the same point as it will not be able to orient itself to the correct direction of gradient descent. Hence, this method was rejected and random initialization of weights was done. Biases in our neural network were initialized to zero.

Mini Batch Training

One of another performance and robustness boosting aspect of this project is using [4]Mini batch training vs Online training or batch training. Since our model uses back propagation, the key to back-propagation training are quantities called gradients. During training, the current neural network weight and bias values, along with the training data input values, determine the computed output values. The training data has known, correct target output values. The computed output values might be greater than or less than the target values.

It's possible to compute a gradient for each weight and bias. A gradient is just a number like -0.83. The sign of the gradient (positive or negative) tells you whether to increase or decrease the value of the associated weight or bias so that the computed output values will get closer to the target output values. The magnitude of the gradient gives you a hint about how much the weight or bias value should change.

In our case we have taken batches of 100 to analyze and set the weights and biases. In stochastic training (also called online training), after the gradients are computed for a single training item, weights and biases are updated immediately. Put another way, in stochastic training, the accumulated gradient values of batch training are estimated using single-training data items. This is more computationally expensive and takes

[4] https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/

significantly longer to train models on large datasets. The frequent updates can result in a noisy gradient signal, which may cause the model parameters and in turn the model error to jump around (have a higher variance over training epochs). The noisy learning process down the error gradient can also make it hard for the algorithm to settle on an error minimum for the model

The reasons for choosing mini batch over batch processing is that even though batch processing results in more stable models, the parameters can get stuck in a local minima due to premature optimization of parameters and also requires a lot of memory as our dataset is large and it tries to store all the training samples in a single matrix and perform operations on it. While mini batch processing is more stable than stochastic gradient it offers more robust solution to local minima compared to batch processing.

Activation Functions

The activation function which were tested for the project were:

1. Sigmoid activation function
2. Tanh activation function
3. ReLU activation function
4. Softmax activation function

In the initial phases of the project when the class was numerically encoded, sigmoid was used as the activation function for all the layers excluding the last layer. This did not yield a very good result as the sigmoid function has a very drawback of vanishing gradient. At the end of the network gradient is small or has vanished (cannot make significant change because of the extremely small value). The network refuses to learn further or is drastically slow (depending on use case and until gradient /computation gets hit by floating point value limits). Since our problem has a very sparse matrix, this problem could have been accentuated, the vanishing gradient could have caused the neural network to learn zeros more than ones and hence in turn learn nothing.

Later ReLU was used as activation function for the hidden layers and Softmax is used for the output layer. Since ReLU is a nonlinear function it does not have the drawbacks of a linear activation function as well as since the range of ReLU is [0, inf] it overcomes the problem of vanishing gradient.

One of the problems in using ReLU is called dying ReLu problem. [5]Because of the horizontal line in ReLu(for negative X), the gradient can go towards 0. For activations in that region of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This problem can cause several neurons to just die and not respond making a substantial part of the network passive. To tackle this problem we introduced dropout in the network.

Cost Function

Cross Entropy: Using cross entropy have been a popular technique to compute and optimize the loss of the neural network, based on which the weights and biases are updated. Because of this, “correctly” computing the cost function is a paramount step which initiates the coveted backpropagation procedure.
In case of classification problems using the cross entropy loss function usually yields better results. After trying with a few other loss functions (L1, L2 norm) even we got the same results and hence without experimenting any further we opted to use cross entropy for our project.
In order to implement the cross entropy in our use case, since we had one-hot encoded our labels, we had to calculate the difference between 20 values. So the output from the last layer of the neural network is compared with the corresponding one-hot encoded (of size 20) correct output. Using these 2 vectors the cross-entropy is then computed.

$$L = -\sum_i y_i log\tilde{y}_i + (1 - y_i)\log\,(1 - \tilde{y}_i) =$$

$$= -\sum_i y_i log\sigma(z_i) + (1 - y_i)\log\left(1 - \sigma(z_i)\right)$$

$$where\ z_i = \bar{w}^T \cdot \bar{x}_i \quad and \quad \sigma(z_i) = \frac{1}{1 + e^{-z_i}}$$

[5] https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

Decaying Learning Rate

Generally when deciding the learning rate value, it is tricky to work with just value which will be used for all the epochs. This is because, having a low learning rate might be nice towards the end, but would be excruciatingly slow at the beginning when weights and biases could be updated using larger steps but because if the the low learning rate would be taking smaller steps. This resulted in in an enormously large amount of time for the training process to complete across all the epochs. Similarly on the other side when we use a large learning rate is beneficial for the starting process when taking large steps are a good thing and helps in reaching the near goal faster. But the problem arises after reaching near the minimum value, and then continuing with large steps. This will lead to going over the minimum value from one side to the other without actually reaching the minimum point.

To solve this we needed to change our thinking from the concept of a fixed learning rate to an dynamic learning rate. So the learning rate starts off with a high value and as the training progresses through the epochs the learning rate slowly “decays” in value.This way we get the “best of both worlds”, i.e a large initial learning rate to move quickly when moving towards the minima and then gradually decreasing the step size in order to make more fine movements when approaching the minima.

The formula shown below is how we implemented the learning decay rate. Using max and min learning rates ensures that the learning starts from max earning rate and does not go below the min learning rate.

```
# learning rate decay
max_learning_rate = 0.01
min_learning_rate = 0.0001
decay_speed = 2000.0
learning_rate = min_learning_rate + (max_learning_rate - min_learning_rate) \
                                    * math.exp(-epoch / decay_speed)
```

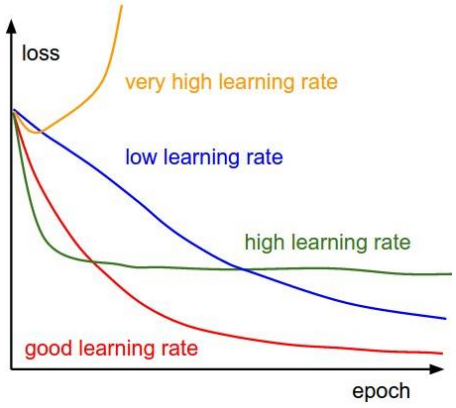

Fig: loss vs epoch as a function of different learning rates

Optimizer

In our work we experimented with 2 optimization techniques, Adam Optimizer and Gradient Descent Optimizer. Almost always the Adam Optimizer gave us considerably better results and hence we decided to go with it.

In broad terms when optimizing the cost functions, the crux of the backpropagation process, the chain rule differentiation is utilized. Here we try to gauge the effect of the each of the weights in all the layers on the computed cost. This is done by finding the partial derivative of the cost(J) with respect to the each weight. The chain rule come because of the operations performed on the weights to reach the cost function. First we compute (Z) i.e multiply the weights with the inputs to the layer and add biases. Then we perform the activation function (σ) of this computed value, the result is $\sigma(Z)$. This value is then used in the cost function (L) to compute the final cost (J). This cost is then used in the partial derivation, to determine effect of the the weights on the cost function and that how they should be update, both in terms of direction and magnitude.
The following figure gives an intuition as to how the chain rule of back propagation works.

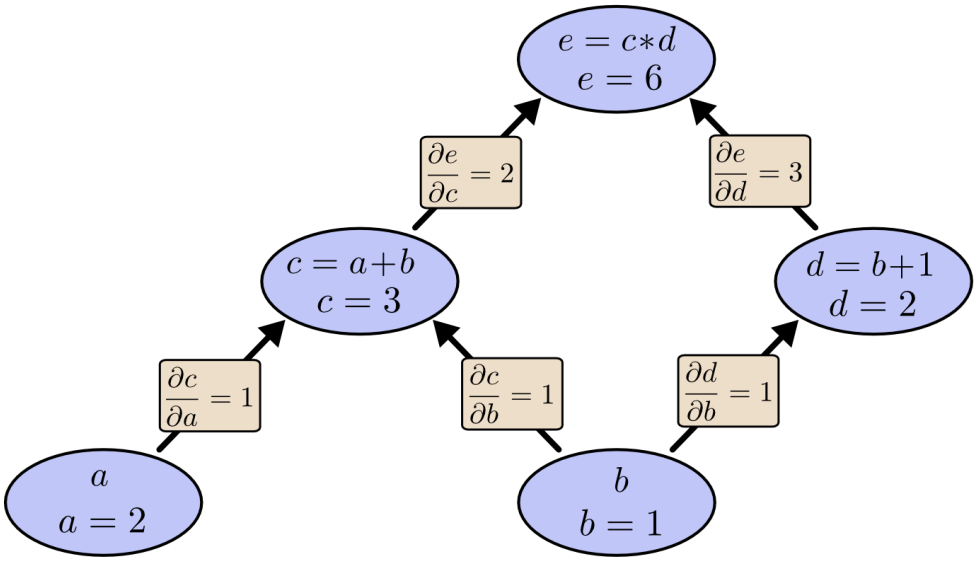

Fig: Chain rule used in backpropagation

The main reason for the difference in performance can be associated with 2 concepts that are present in the Adam Optimizer but not in the simple Gradient Descent. These 2 concepts are:

1. Root Mean Square Propagation: Also acronymed as RMSprop the Root Mean Square Propagation allows the optimizer to consider the the past values as well

as the current value, giving a decaying weight to the past values while maintaining a constant weight for the current value.

2. Momentum: This term factors in the several dimensions that the gradient descent could choose to move in the maximizes the movement in the dimension which would be optimal in it's quest to reach the minima while optimizing the cost function.

Regularization

After setting up the architecture of the neural network we started to experience severe overfitting problems. The training accuracy would quickly overfit and lead to declining performance on the test data set. Because of this we thought of introducing some kind of regularization in an attempt to tackle this problem.

We introduced three kinds of regularizations:

1. L2-norm regularization: In this method we added a penalty term when the weights were update. So updating the weights too much would mean very high penalty. This may seem counter-intuitive, but this is the point of regularization. We do not want the neural network to train too much on the training data as we know it results in overfitting. So penalising the weights would results in a more general approach rather than custom weights just for the training set which results in decreased performance over unseen data. The λparameter used to decide the severity of the penalty term is set to 1.0 and is then normalised over the batch size.
2. Dropout regularization: In this method we introduced the concept of randomly ignoring certain neurons in each epoch. So for each each node in each layer we maintain a probability of keeping that neuron. We have set set this probability parameter as 0.75. This meant that each neuron in each layer had a 25% chance of being dropped for that iteration of weight updation.
3. Early stopping: Another concept we implemented was to stop the training prematurely. This resulted in the reduction of the severity of the overfitting as the model was not trained for a long period of time which meant that too high of a variance was not introduced which would make the model biased towards the training data set.

Results

The above mentioned neural network was tested on a system with the following specifications:

1. Processor: 4.0 GHz
2. Memory: 16GB
3. GPU: Nvidia GTX 1070
4. OS: Ubuntu KDE flavour
5. IDE: pycharm

Since we had a GPU at our disposal, we used the Tensorflows GPU version which uses CUDA. Each epoch took ~0.9 seconds to complete i.e. our neural network was able to analyze and backpropagate ~30,000 samples with 6714 features in less than a second. We trained the model for 500 epochs and simultaneously tested its accuracy on the test set.

Our model consistently performed well with different splits and was stable giving an accuracy of ~79% over several runs. The state-of-the-art model to solve this problem has an accuracy of 83%.

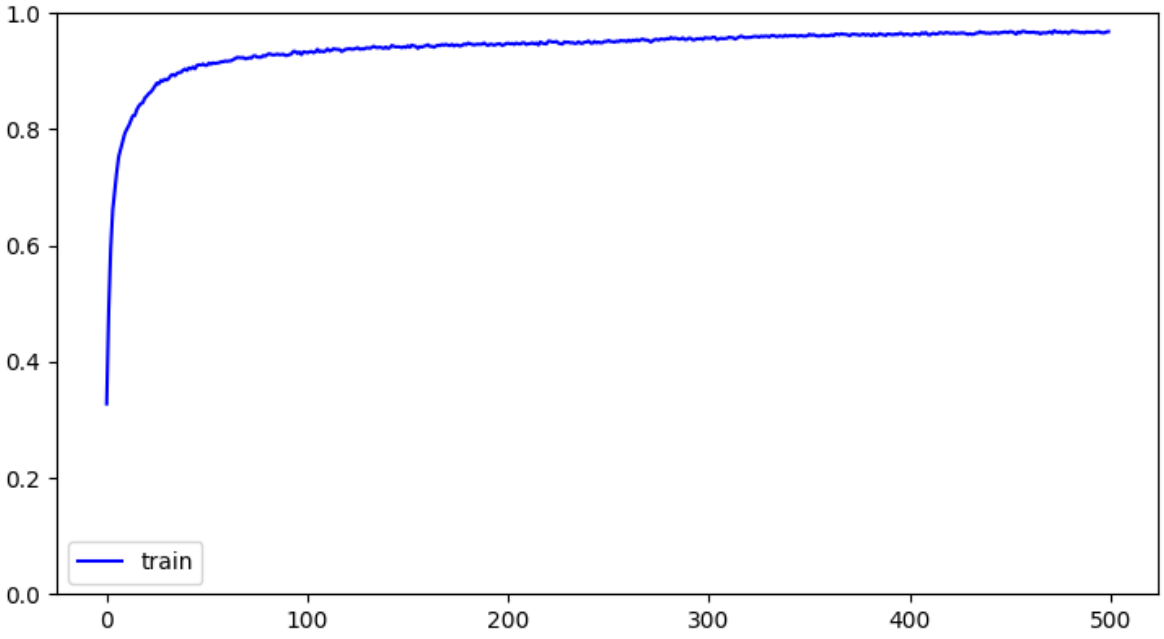

Fig: Training accuracy

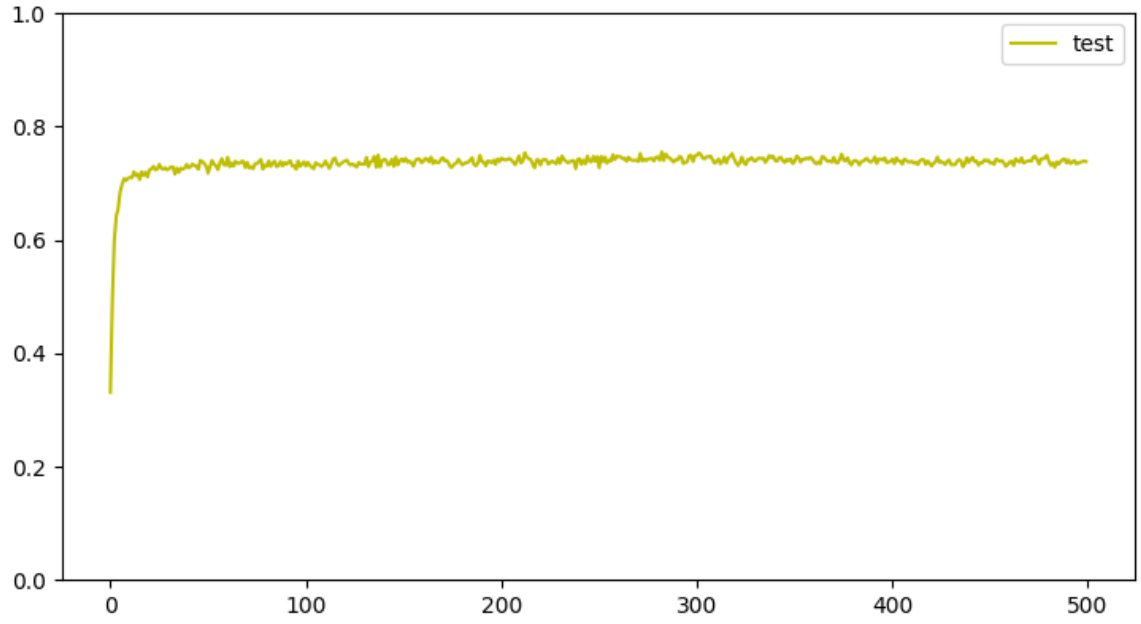

Fig: Test accuracy

The graph below shows the cost of the model on training data over 500 epochs.

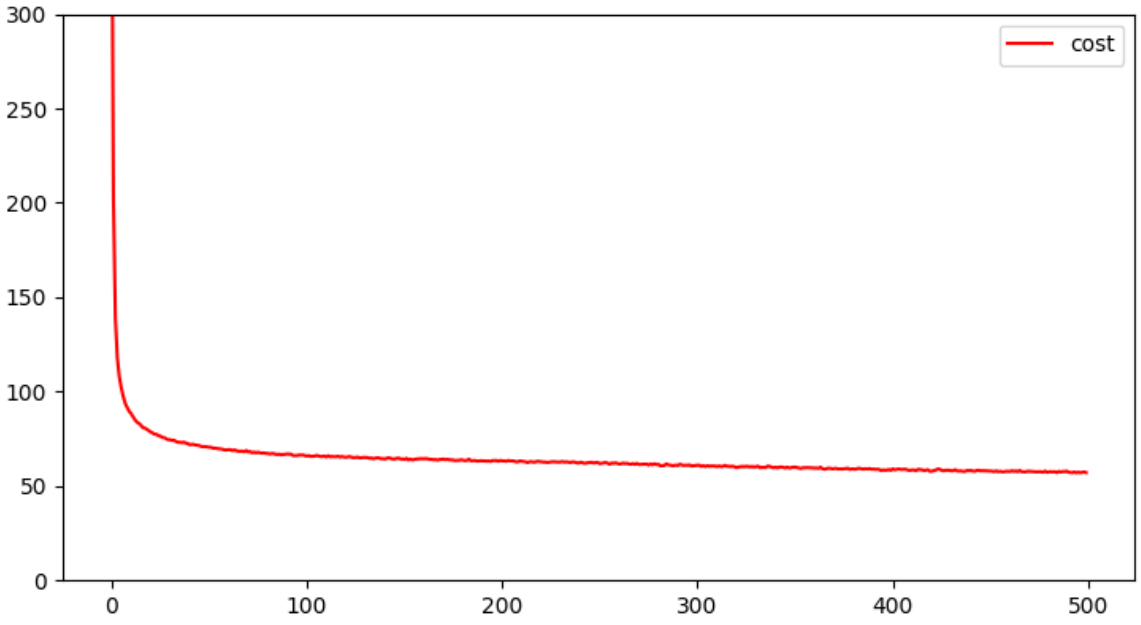

Fig: Training Cost