REPORT OF PROJECT ON

BASIC APPLICATIONS OF MACHINE LEARNING

June, 2018

Submitted by

Athul Prakash

Based on the work done at

Naval Physical and Oceanographic Laboratory,

Defence Research and Development Organization,

~~Kochi~~

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to **Shri S.Kedarnath Shenoy**, Outstanding Scientist, Director, NPOL-DRDO and **Mr. Suresh M**, Scientist-G,~~ Chairman HRD Council~~, for permitting me to carry out my project work at his prestigious organization.

I also express my heartful gratitude to **Mr. K.V. Rajasekharan Nair**, Scientist-G, ~~GH~~(P&A), **Dr. Sapna Pavitran**, Scientist-E, DH(HRD), and **Mr. K.V. Surendran**, Technical Officer B, NPOL, for providing the opportunity, facilities and administrative support to do my project.

I am indebted to **Mrs. R. Pradeepa,** Scientist-F, DH(SPA), NPOL for permitting me to do this project under her division. I would like to express my deep sense of respect and gratitude towards my guide, supervisor and mentor **Dr. Sooraj K Ambat,** Scientist-E, for his patience, technical guidance and keen interest in my project, all of which have been invaluable to me.

DECLARATION

I, undersigned, hereby declare that the project report ‘Basic Applications of Machine Learning’, is a bonafide work done by me under supervision of NPOL and Dr. Sooraj K Ambat, Scientist-E.

The Submission represents my ideas in my own words and where ideas or words of others have been included, I have adequately and accurately cited and referenced the original sources. I also declare that I have adhered to ethics of academic honesty and integrity and have not misrepresented or fabricated any data or idea or fact or source in my submission. I understand that any violation of the above will be a cause for disciplinary action and can also invoke penal action from the sources which have thus not been properly cited or from whom prior permission has not been obtained.

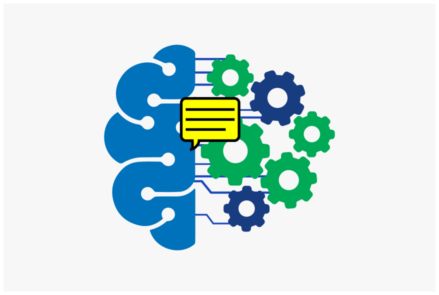
Place: Kochi Athul Prakash

Date:

TABLE OF CONTENTS

INTRODUCTION

1 Machine Learning



Machine Learning is the use of statistical techniques to make computers 'learn' with data, without being explicitly programmed. It is a subset of Artificial Intelligence in computer science. Arthur Samuel coined the term machine learning, and the field evol[illegible]om studies of pattern recognition in the late 1950s.

It involves the construction of algorithms that can learn from and make predictions on data – instead of following static programmed instructions. This is done by building a model from sample inputs through various learning algorithms, such as logistic regression or neural networks. It can solve problems whose solutions are difficult, if not infeasible, to explicitly program algorithmically – for example: Optical Character Recognition, Email spam filtering and computer vision.

Machine Learning is mainly concerned with 4 learning tasks –

Supervised Learning The computer is given example inputs (called data) and their known outputs (also called labels) and the objective is to learn a rule that maps each inpu[illegible]he given output.

Unsupervised Learning No target labels are given to the program, instead only the input data is presented. The goal of the program is to find some structure in the given data. It is like saying to the computer 'Here, take some data and make sense out of it'.

Semi-supervised Learning Incompletely labelled data is given to the program. Often, most target labels are missing and less than 10% of the data are labelled.

Reinforcement Learning The program is asked to make decisions in a dynamic environment and the training data consists of rewards or punishments given as a consequence of the action the program took. For example, teaching a computer to play a video game.

2 Classification

Classification is an application of machine learning where all the labels belong to a finite set of values that is known to the program. As such, classification falls squarely in the category of supervised learning. For eg: classifying emails as spam or not spam.

In classifications, the labels can be represented by integers starting from zero, by binary digits or by a vector of one-hot coded values.

Often in classification tasks, the purpose is to classify objects (data) into either of two categories – this is called Binary Classification.

Besides classification, two other common applications of machine learning are:

Regression where the output variable (ie the label) belongs to a range of continuous values instead of a discrete set.

Clustering where a set of inputs should be divided into groups and the set of groups is not known beforehand.

3 Data Sets Used

Two different datasets were used in this project. They are outlined below:

3.1 Iris F[illegible]er Data Set

The Iris flower data set is a multivariate data set introduced by statistician and biologist Ronald Fisher in his 1936 paper The use of multiple measurements in taxonomic problems. It was collected by Edgar Anderson in order to quantify the morphologic variation of Iris flowers of three closely related species, viz:

- *Iris setosa*
- *Iris virginica*
- *Iris versicolor*

The data set consists of 50 samples from each of the species. Each sample/record contains measurements of 4 attributes of the flower-

- Petal length
- Petal Width
- Sepal length
- Sepal Width

The data is stored and read from a CSV file where each row contains 1 record followed by the name of that species, ie, data followed by label.

It has been shown by previous analysis that out of the 3 species, *Iris setosa* is fully linearly separable from the other 2 species whereas *Iris virginica* and *Iris versicolor* are not completely linearly separable, although it is only a few examples that make the separation non-linear. Hence, with this dataset, we can investigate both the performance on linearly separable data and non-separable data for any given model.

3.2 MNIST Data Set

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used in machine learning.

The MNIST database contains 60,000 training images and 10,000 testing images.

The dataset was created by Yann LeCun by combining a similar dataset written by high school students with another dataset of digits written by staff in the United States Postal Service.

The MNIST dataset is especially used in Computer Vision tasks involving classification and object detection.

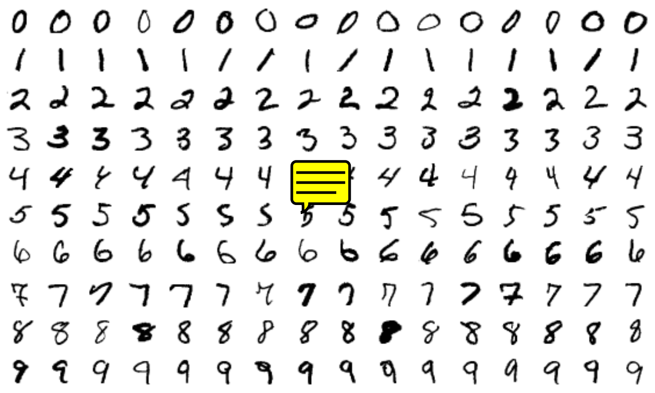


Figure 1: Some samples from the MNIST dataset.

PART I
Logistic Regression

DATASET USED: Iris Flower Dataset

1 Basics of Logistic Regression

Logistic regression is a widely used classification technique in Machine Learning.

Logistic Regression is natively suited to binary classification problems, but certain variaitons have been developed that extend its applicability beyond 2-class problems.

It has a relatively modest number of parameters – just one parameter for every input feature – making for a very simple model; it is not suited to complex classification tasks. However, by using a simple model such as this, the risk of overfitting the data is eliminated. In addition, model simplicity also saves processing power which results in faster development of the model.

In the logistic model, each set of inputs is vectorized into a single input vector and further, all the parameters of the model are also vectorized into another parameter vector. Then, the scalar product of these 2 vectors is calculated. This product is then passed through the logistic function to get the final output of the model – always a real number between 0 and 1. A full mathematical description can be found in Section 2.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

The Logistic(Sigmoid) Function

2 Mathematical Description and Notation

$$X = [X_1, X_2, X_3, \dots] \qquad \textit{are the inputs to the model}$$

$$\theta = [\theta_0, \theta_1, \theta_2, \dots] \qquad \textit{are the parameters of the model}$$

$$X = [X_0 = 1 : X_1, X_2, X_3, \dots] \qquad \textit{bias term is appended}$$

$$Z = X \cdot \theta \qquad \textit{take the dot product}$$

$$h = \frac{1}{1 + e^{-Z}} \qquad \textit{output is the sigmoid function of } Z$$

3 Training The Model

3.1 What is Training?

In classification, the function of any model is to take a set of inputs and predict some output values that decide which class the input example belongs to. In binary classification though, it predicts just a single output. Typically, this output ranges from 0 to 1 and a decision boundary of 0.5 is placed on this range to infer which class is predicted.

To produce an output, the model has some parameters that the input is computed with. Thus, the value of the parameters defines wholly what the output is going to be for every input shown to the model.

The task of training (called fitting the model) is to tune these parameters such that the model can accurately distinguish between inputs of different classes. Hence, labelled data - data for which the correct outputs are known - is fed into the model and the model's predictions on each of these data are calculated. Then a cost function needs to be used . . .

3.2 Cost Function

The Cost Function measures the error in classifying using a given model. The more the predictions correlate with the actual labels, the lesser should be the value of the cost function.

Thus, the task of fitting the model accurately is accomplished by monitoring the cost function and reducing it's value by adjusting the model's parameters. Therefore, the cost function should also suggest which way to tune each of the parameters of the model.

For logistic regression, the following cost function is used:

$$J(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)), & \text{if y = 1} \\ -log(1 - h_\theta(1 - x)), & \text{if y = 0,} \end{cases}$$

Since the behavior of the function is defined piecewise, its shape is as shown in figures 1 & 2. It is apparent that the function's tendency is towards zero as the prediction tends toward 1, in figure 1 (where the true label is 1), or 0 in figure 2 (where the true label is 0).

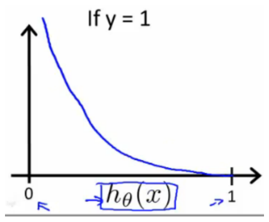


Figure 1: Logistic Cost Function when y=1

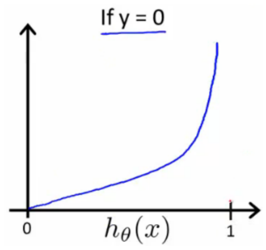


Figure 2: Logistic Cost Function when y=0

The parameters of the model are tuned in a manner that minimizes the cost function. This is achieved with an optimization algorithm...

3.3 Gradient Descent – The Cost Minimization Algorithm

Gradient Descent is a first-order iterative minimization algorithm for finding the minimum of a function. It is based on the following observation: That if a multivariate function F(X) is defined and differentiable in the neighbourhood of a point 'a', then it decreases fastest in the direction opposite the gradient of F at 'a', i.e. - ∇ F(a).

So in every step, a chosen point 'a' is moved a distance proportional to the local gradient, in a direction opposite to the gradient. This logic is summarized as an update step:

$$a_{n+1} = a_n - \alpha \nabla F(a_n) \tag{2}$$

α is called the Learning Rate. It scales the length covered in every step and hence, it defines how fast the optimization proceeds.

The reasoning behind this update step is as follows:

Since the cost decreases in every step, and since the updates converge only when the gradient = 0, the point 'a' would have to converge on the minimum.

However, there are 2 main caveats to this reasoning:

- The minimum that is converged on may be just a local minimum.
- If α is too large, an update might skip over a minimum and cause a cost increase

3.4 Gradient Descent Applied to Logistic Regression

The cost function, J(Th) from Section 3.2 is rewritten as:

$$J(\theta) = -y\log(h_\theta(X)) - (1-y)\log(1-h_\theta(X)) \tag{3}$$

This form makes it readily differentiable.

Note: J is a function of θ here since θ are the variables in optimizing J

Hence,

$$\frac{\partial J}{\partial \theta_j} = \frac{\partial[-y\log(h_\theta(X)) - (1-y)\log(1-h_\theta(X))]}{\partial \theta_j}$$

Replacing $h_\theta(X)$ using ~~Equation 1~~,

$$\frac{\partial J}{\partial \theta_j} = \frac{\partial[-y\log(\frac{1}{1+e^{-\theta^T x}}) - (1-y)\log(1-\frac{1}{1+e^{-\theta^T x}})]}{\partial \theta_j}$$

On simplifying and differentiating,

$$\frac{\partial J}{\partial \theta_j} = (h_\theta(X) - y)x_j$$

Averaged over m examples,

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(X) - y)x_j$$

Thus, from Equation 2,

$$\theta_j := \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(X) - y)x_j \tag{4}$$

Equation 4 is the final update step used in Logistic Regression.

4 The One v Rest Approach to Classification

The dataset chosen, the Iris Dataset, contains examples belonging to 3 classes. However, the logistic regression model is best suited for binary classification problems. Therefore, the data labels must be changed to make a binary classification problem out of the Iris Dataset. This is achieved using one-vs-rest classification, where each example is classified as belonging to one particular class (label 'ONE') vs not belonging to that class (label 'REST'). This class needs to be chosen beforehand and typically, the choice of this class influences the ease of classification. The OnevRest approach can be used to easily convert an n-class problem to a 2-class one, so that Logistic Regression becomes possible.

Further, OnevRest approach also suggests a method to run logistic regression on a fully multi-class problem. This method is commonly used for the logistic technique; however, it will not be explored in this project. If there are C classes, C classifiers can be trained, each of which determines whether the input example belongs to one of the C classes or not, a la OnevRest. Then, by comparing the classification scores of the C classifiers, the best class that the example would fall into may be selected.

Since the Iris Dataset has three classes of inputs, there are three ways to implement One v Rest classification – each one classifying a specific class vs the rest.

5 Experiment 1

5.1 Objective

To train logistic regression classifiers on the Iris dataset using the One v Rest approach

5.2 Procedure

* 3 Classifiers were trained to respectively distinguish between *Iris setos*a and the rest, *Iris virginica* and the rest, *Iris versicolor* and the rest.
* Data was loaded from Iris.csv and no pre-processing was done apart from that required for One v Rest. The data was divided, in each case, into an 80:20 ratio of training:testing data.
* The model's parameters were initialized to all 0's and trained for 10k iterations using Logistic Regression
* Hyperparameters: $\alpha = 0.1$;   regularization was not used.
* The Cost was monitored after every iteration and the graph of Cost-vs-Iterations was plotted every time.

5.3 *Iris setosa* v Rest Classifier

5.3.1 Observed Data

From Training

Initial Cost = 66.23   Final Cost = 0.066

From Testing

Correct Examples: 30/30 (100%)
Incorrect Examples 0/30 (0%)

		True Class		
		One	Rest	Total
Predicted Class	One	10	0	10
	Rest	0	20	20
	Total	10	20	30

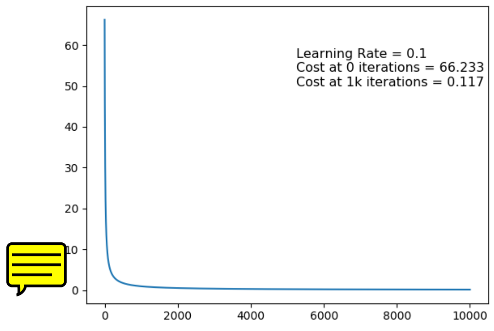


Figure 3: Graph of first 1k iterations of training

5.3.2 Conclusions

* The cost reduces sharply for the first few iterations but slowly for later iterations.
* The Cost plateaus out after a few hundred iterations, and it does not tend to zero.
* The Cost consistently falls after every iteration, although the fall is less later on.

5.4 *Iris virginica* v Rest Classifier

5.4.1 Observed Data

From Training

Initial Cost = 78.147;   Final Cost = 8.77

From Testing

Correct Examples: 29/30 (97%)
Incorrect Examples 1/30 (3%)

		True Class		
		One	Rest	Total
Predicted Class	One	10	0	10
	Rest	1	19	20
	Total	11	19	30

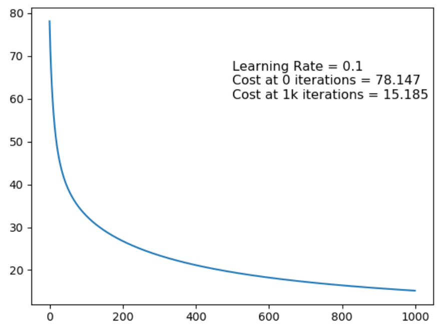


Figure 4: Graph of first 1k iterations of training

5.4.2 Conclusions

* The cost falls quickly for the first iterations, but not as quickly as for *Iris setosa*.
* The Cost plateaus out after a few hundred iterations, and it tends to a value higher than for *Iris setosa*.
* The Cost consistently falls after every iteration, although the fall is less later on.

5.5 *Iris versicolor* v Rest Classifier

5.5.1 Observed Data

From Training

Initial Cost = 78.278;   Final Cost= 60.6

From Testing

Correct Examples: 22/30 (73%)
Incorrect Examples 8/30 (27%)

		True Class		
		One	Rest	Total
Predicted Class	One	3	7	10
	Rest	1	19	20
	Total	4	26	30

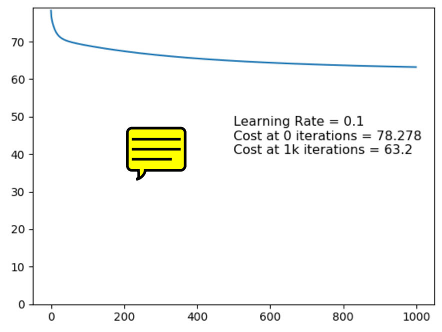


Figure 5: Graph of first 1k iterations of training

5.5.2 Conclusions

* The cost reduces appreciably only for a few starting iterations.
* The cost plateaus out far earlier than for 1. Iris setosa or 2. Iris virginica
* The total reduction in cost is minimal overall and does not tend to improve with more iterations.

6 Experiment 2

6.1 Ideation

* In Gradient Descent, the size of the update step is proportional to the magnitude of the gradient of the Cost Function, i.e. $\Delta\theta = -\alpha\nabla\theta$.
* Therefore, as the magnitude of Cost and that of it's gradient fell in the previous experiment, the size of the update step became smaller and smaller until finally, towards the end of the training, the update step size was negligible.
* How can this effect be counteracted? How to ensure that the update step size does not become negligible?

6.2 Approach

For the following experiment, the learning rate was made adjustable. The value of α was increased as the value of Cost fell. The objective was to see how far and how fast the cost would fall as compared to Experiment 1, where α was a constant.

A scaling factor(S.F.) was used in the updation of α: every time the cost fell by a factor of S.F., α would be increased by a factor of S.F.

Thus, with $\alpha_{start} = 0.1$, the following update of α was run after every iteration:

$$\alpha = \alpha_{start} \times (S.F)^{\lfloor \log(\frac{J_init}{J}) \rfloor} \tag{5}$$

Three values of S.F were used: $S.F = \{\mathit{2},\ \mathit{5},\ \mathit{10}\}$

6.3 Observed Data

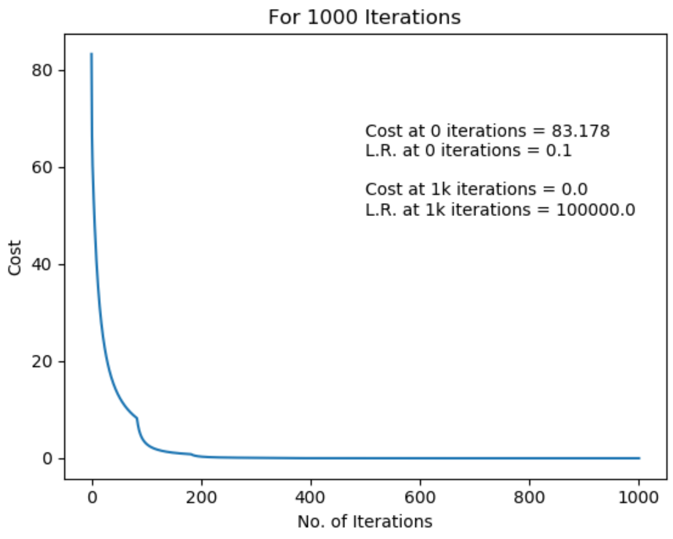


Figure 6: Falling Cost (with S.F = 10)

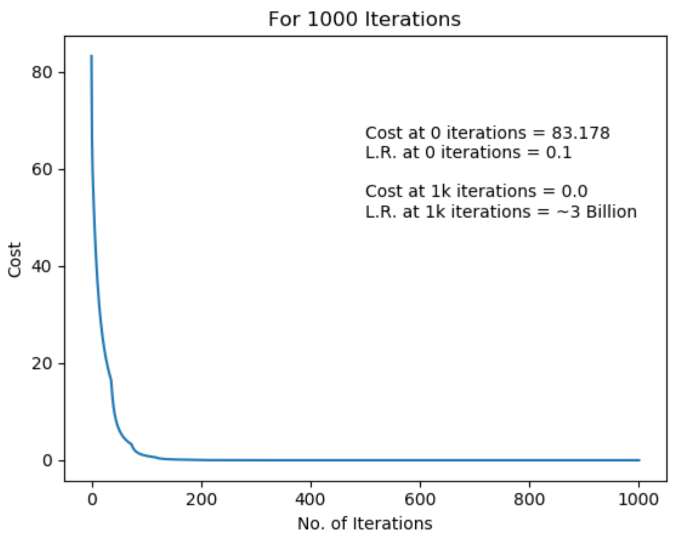


Figure 7: Falling Cost (with S.F = 5)

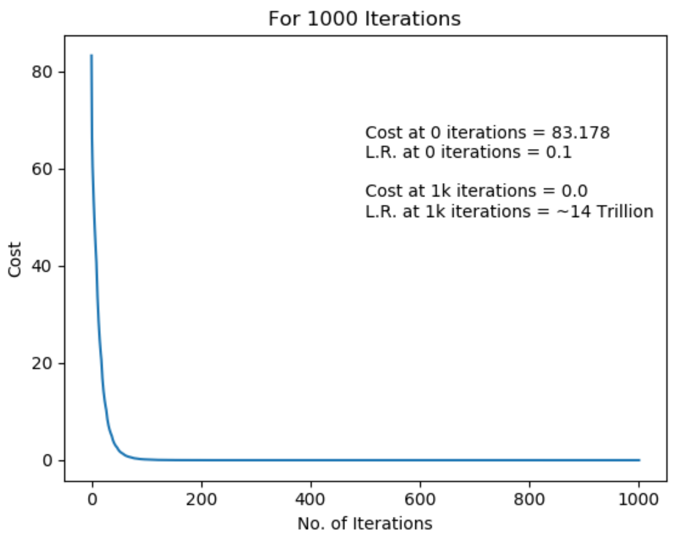


Figure 8: Falling Cost (with S.F = 2)

The following tables compare the three values of S.F with each other.

For the sake of generalization, the previous experiment can be said to have S.F=∞

S.F	α after 100 iter.	Cost after 100 iter.
∞	0.1	6.88
10	1.0	3.002
5	2.5	1.007
2	51.2	0.142

S.F	No of iterations till Cost <0.1	No of iterations till Cost <0.01
∞	N.A	N.A
10	286	444
5	178	270
2	103	162

The graph of Cost vs No. of Iterations was plotted for the first 100 iterations to compare the standard Logistic algorithm with the 3 versions proposed here:

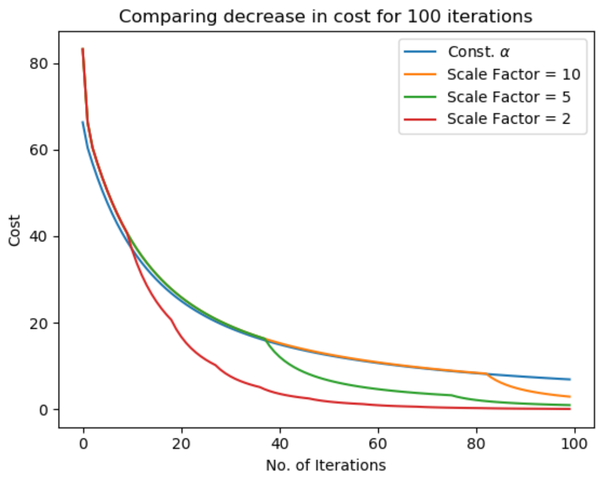


Figure 9:

Since the intent behind this experiment was to negate the update step becoming negligible in the later stages, the following graph was plotted which depicts the decrease in cost for 50 update steps after 500 iterations have been completed, comparing these values for the original Logistic algorithm and the 3 others proposed here.

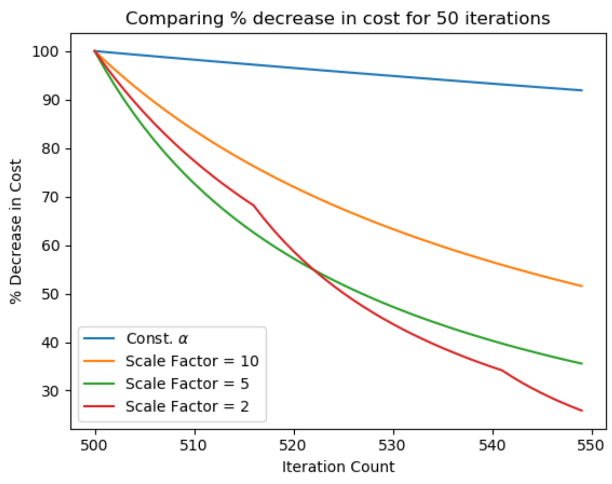


Figure 10:

6.4 Conclusions

* In all cases, the cost fell faster than in Experiment 1, where α was a constant. Hence, these changes have led to an improved Logistic Regression algorithm, as was expected in the Ideation section of this experiment.
* Introducing any scaling factor leads to dramatic improvements over the standard Logistic algorithm.
* From Figure 9, the smaller the value of the Scale Factor, the better the performance of the algorithm.
* From Figure 10, there were dramatic improvements to the algorithms performance in later stages of training. Lower S.F performed better here.

****End of Logistic Regression (Part 1)****

PART II
Neural Networks

DATASETS USED: Iris Flower Dataset, MNIST Dataset

1 Basics of Neural Networks

An Artificial Neural Network is a machine learning model loosely based on the structure of interconnected neurons in the human brain. Since the biological brain is well-adapted to handling complex tasks, the inspiration for Artificial Neural Networks is that a model that mimics the brain's structure and basic functioning would prove most capable at solving difficult machine learning problems.

Structurally, a Neural Network is a collection of interconnected units called Perceptrons. The perceptrons are arranged in different layers and the data flows from one layer to the next. This flow of data is controlled by weights that connect adjacent layers, as shown in Figure 1.

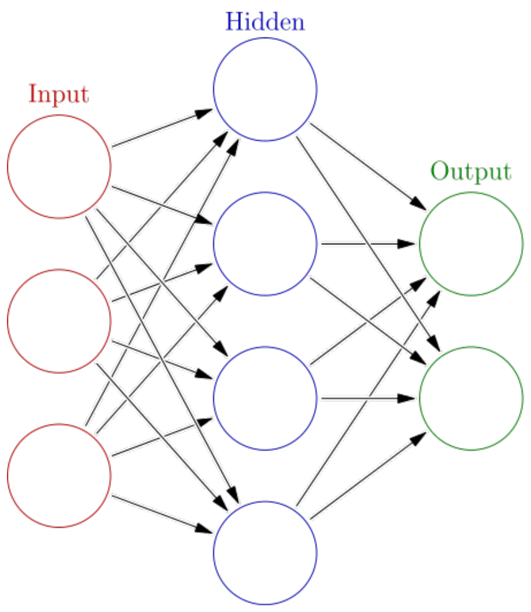


Figure 1: Basic Structure of ANN

In the most common type of neural network, all of the perceptrons in a given layer receive inputs from all the perceptrons in the previous layer. After receiving these inputs, the perceptron computes its value as follows:

If the inputs to the network are X and the weights of the perceptron are W then:

$$X = [x_0, x_1, x_2, \ldots] \tag{1}$$

$$W = [w_0, w_1, w_2, \ldots] \tag{2}$$

The weighted sum of inputs is calculated and a bias is added:

$$Z = X{\cdot}W + b$$
$$\implies Z = b + x_0w_0 + x_1w_1 + \ldots \tag{3}$$

An activation function is used to introduce non-linearity:

$$H = sigmoid(Z) = \frac{1}{1+e^{-Z}} \tag{4}$$

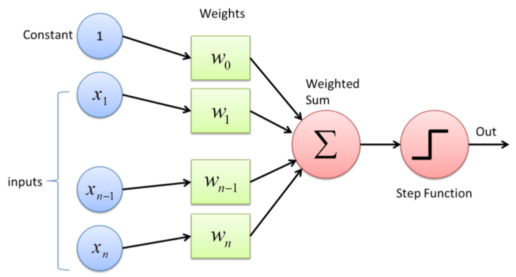


Figure 2: Calculations in a Perceptron

2 Functioning of Neural Networks

2.1 Vectorization

- In a neural network, the collection of weights of every unit in a layer are vectorized into a matrix θ, in which each row holds all the weights of a single perceptron.
- The values of all the units in a given layer are handled together as a vector
- The biases in a given layer are also handled together as a vector.
- The inputs and outputs are given and received in the form of a vector

2.2 Notations Used

$L:$	$The\, Number\, of\, Layers.\, (l = 1,\, 2,\, \ldots,\, L)$
$z_j^i:$	$The\, weighted\, sum\, of\, j^{th}\, unit\, in\, i^{th}\, layer.$
$a_j^{(i)}:$	$the\, activated\, value\, of\, j^{th}\, unit\, in\, i^{th}\, layer$
$(x^{(m)},\, y^{(m)}):$	$input - output\, pair\, for\, the\, m^{th}\, example.$
$\theta^{(l)}:$	$the\, weights\, connecting\, layer\, l\, to\, the\, next.$

2.3 Forward Propagation

Forward propagation uses the parameters of the network to calculate an output vector from an input vector, by stepping through the computations layer by layer as follows:

For a given input vector x, to calculate the prediction vector $\hat{y}$:

Start with:

$$a^{(1)} = x$$

Repeat for l = 2 to L:

$$a^{(l-1)} = [1;\, a^{(l-1)}] \qquad prepend\, 1\, for\, bias.$$

$$z^{(l)} = \theta^{(l-1)} \cdot a^{(l-1)} + b \qquad compute\, the\, weighted\, sum\, z.$$

$$a^{(l)} = sigmoid(z^{(l)}) \qquad apply\, the\, activation\, function.$$

End with:

$$\hat{y} = a^{(L)}$$

2.4 Cost Funciton and Optimization

Although many cost functions are possible, the squared error cost function is most common.

$$J = \sum_{\forall\, j} (\hat{y} - y)^2 \tag{5}$$

For a given dataset with m examples, the cost is averaged over the dataset. Training the neural network is the process of reducing this cost function by adjusting the parametrs θ. Gradient descent is used for this purpose.

Therefore, the gradient of the cost function is computed w.r.t every parameter $\theta_{ij}^{(l)}$ and the the following update step results:

$$\theta_{ij}^{(l)} = \theta_{ij}^{(l)} - \alpha \frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) \tag{6}$$

Gradient Descent is applied on a neural network as per the backpropagation algorithm.

2.5 The Backpropagation Algorithm

```
Training Set $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j).

For i = 1 to m:
    Set $a^{(1)} = x^{(i)}$
    Perform forward propagation to compute $a^{(l)}$ for l = 2, 3, ..., L
    Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
    Backpropagate $\delta s$: Compute $\delta^{(l)} = (\theta^{(l)})^T \delta^{(l+1)}$ for l=L-1, ..., 2
    $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) := \frac{1}{m} \Delta_{ij}^{(l)}$
```

2.6 One-Hot Coding

To build a Neural Network for multi-class classification, the labels of the data needs to be encoded into a format that the network can output. Most commonly, the data are one-hot coded.

This means that, for an n-class problem, each label is a n-dimesnional vector, with a 1 in the place of the correct output class and 0's everywhere else.

For the Iris Data Set, this coding looks like:

Iris setosa	[1 0 0]
Iris virginica	[0 1 0]
Iris versicolor	[0 0 1]

2.7 Stochastic Gradient Descent

~~Stochastic Gradient Descent~~ is a variation of Gradient Descent where each update step is performed with a single training example. An example is randomly chosen from the dataset, the gradient of the cost function w.r.t this example is computed and the update is also done.

The purpose of ~~S.G.D~~ was to speed up the computation for large datasets, where computing the gradient with all examples before each update step can be very time expensive.

A drawback with this approach is that, since the algorithm sees different data for each update, the optimization never appears to converge to the exact point of minimum. Instead, after many iterations, it continues to stay in random motion in a region surrounding the minimum.

Another variation of Gradient Descent is called mini-batch gradient descent, where a random mini-batch is chosen for computing the gradient. It can be thought of as a middle-ground between batch gradient descent and S.G.D.

3 Experiment 1

3.1 Objective

To build a Neural Network that recognizes handwritten digits by training on the MNIST database using Mini-Batch Gradient Descent.

Use multiple runs successively and tune the hyperparameters manually based on the results of the previous run.

3.2 Implementation

* A class *ANN* was created that handles Neural Networks of any shape. It wrapped up the functions of parameter initialization, Forward Propagation and Backpropagation.
* Created a function *Cost()* that returns the Cost of the network on a given dataset.
* Created a function *fit()* that performs mini-batch gradient descent on a given network. Allows the dataset to be split into *pieces* number of pieces (determines size of a batch) and also sets the given learning rate and no. of epochs.
* Created functions *acc()* and *ConfMat()* that return the accuracy value and the full confusion matrix based on test data.

3.3 Procedure

* Initialized a Neural Network m (using ANN class) with 2 hidden layers of sized h1=183 and h2=42. The sizes were determined by inserting 2 geometric means b/w input size(784) and output size(10)
* Each example in the MNIST dataset was flattened into a 1-dimensional vector and passed into the network.
* **Run1:** Using *fit()*, the network was trained on the dataset for 10 epochs, with batch size chosen as 1000 and learning rate as 0.003
* The graph of Cost for Run1 suggested that the learning rate was too high and/or the batch size was too low.
* **Run2:** Using *fit()*, the network was trained on the dataset for 10 epochs, with batch size chosen as 2000 and learning rate as 0.001
* For Run2, once again, the graph of Cost vs Iterations suggested that the learning rate was too high and/or the batch size was too low.
* **Run3:** Using *fit()*, the network was trained on the dataset for 10 epochs, with batch size chosen as 4000 and learning rate as 0.0003
* After these 3 Runs, the trained network was tested on the 10,000 test examples and a Confusion Matrix was tabulated.

3.4 Results and Analysis

3.4.1 Run 1

batch_size = 1000 $\alpha = 0.003$ epochs = 10

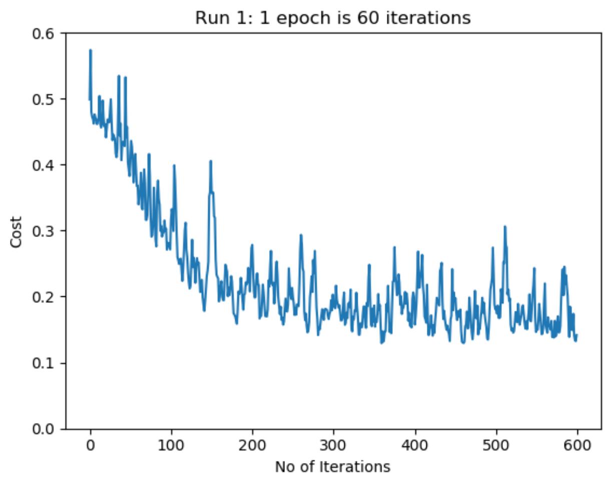


Figure 3: Fall of Cost for Run 1

* The fall of cost plateaued out in the first half of the training itself. There was no meaningful improvement for nearly the latter half of the iterations.
* The curve was very rough and the cost increased and decreased by a significant amount intermittently.
* This trend of roughness in the curve could be due to a step size too large. Hence, α was reduced in the next run.
* This trend might also have arisen because the mini-batch was too small. Hence, batch_size was increased in the next run.

Initial Cost = 0.498 Final Cost = 0.141

3.4.2 Run 2

batch_size = 2000 $\alpha = 0.001$ epochs = 10

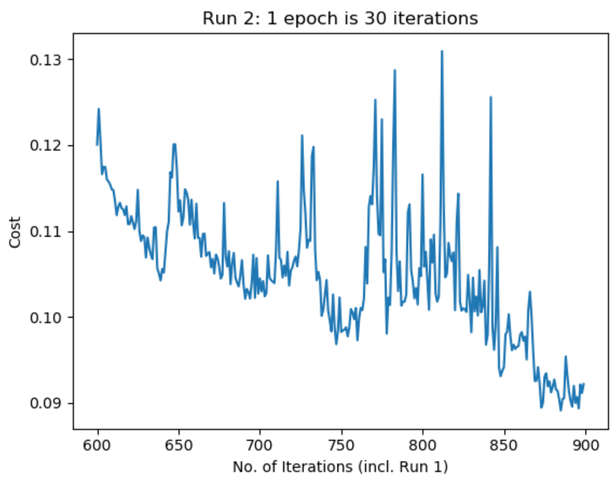


Figure 4: Fall of Cost for Run 2

∗ The overall fall of cost was smaller compared to Run 1.

∗ The trend in the graph was for the cost to increase and decrease drastically in intermittent steps. The cost even exceeded the starting cost for a few iterations.

∗ The reduction of α and increase of batch_size from Run 1 proved effective in the beginning. But by the end of Run 2, the cost had stopped decreasing to the desired degree.

∗ Hence, α was further decreased and batch_size was further increased for the next run.

Initial Cost = 0.120 Final Cost = 0.0921

3.4.3 Run 3

batch_size = 4000 $\alpha = 0.0003$ epochs = 10

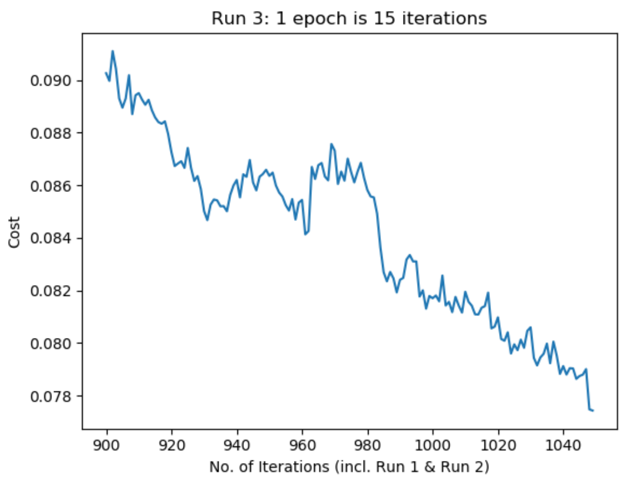


Figure 5: Fall of Cost for Run 3

Initial Cost = 0.0902 Final Cost = 0.0774

A confusion matrix was plotted with the model trained after 3 Runs:

$\frac{Predicted\,Class\longrightarrow}{Actual\,Class}$	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
'0'	949	0	2	3	1	9	8	3	4	1
'1'	0	1107	3	2	1	1	4	3	14	0
'2'	12	5	903	34	17	2	14	19	21	5
'3'	5	2	30	879	1	24	2	15	45	7
'4'	0	0	3	1	856	4	19	1	9	89
'5'	23	2	3	54	13	721	13	8	45	10
'6'	22	3	7	1	12	14	889	1	6	3
'7'	6	20	22	4	9	0	1	936	4	26
'8'	8	2	9	17	11	38	9	6	849	25
'9'	6	5	5	9	36	9	5	15	11	908

Table 1: Summary of the model's classifications on test data

3.5 Conclusions

- If α is too high, the model may not converge close to the minimum. As such, α might need to be lowered as training progresses.

- The higher the batch size, the closer the model can converge to the minimum. As training progresses, the batch size can be increased to enable a convergence closer to the optimum than possible with lower batch sizes.

4 Experiment 2

4.1 Objective

To build a Neural Network that classifies the Iris Data Set - the shape of the Neural Network has to be selected. Find the optimal size of the hidden layer by varying the its size - choose the one with highest classification accuracy.

4.2 Procedure

* An ANN class was created that handles Neural Networks of any shape. It wrapped up the functions of parameter initialization, Forward Propagation and Backpropagation.
* Created a function *basic_fit()* that uses ANN class to train on the Iris Data Set, with 1000 iterations of backpropagation.
* Created a function *rept()* that, with a fixed shape of the hidden layer, trains 100 networks on the data set(using *basic_fit()*) and returns the average accuracy for the 100 trained networks. This step ensured that the results for a given shape are consistent across many runs.
* A function *linsearch()* was created which varies the size of the hidden layer from 1 to 20 and gathers the metrics for each shape using *rept()*.
* Comparing the values of accuracy (on test data), the best Neural Network shape was chosen.

4.3 Results and Analysis

For a 3-layered Neural Network, the following graph shows the variation of accuracy (on test data and training data) vs number of units in the hidden layer:

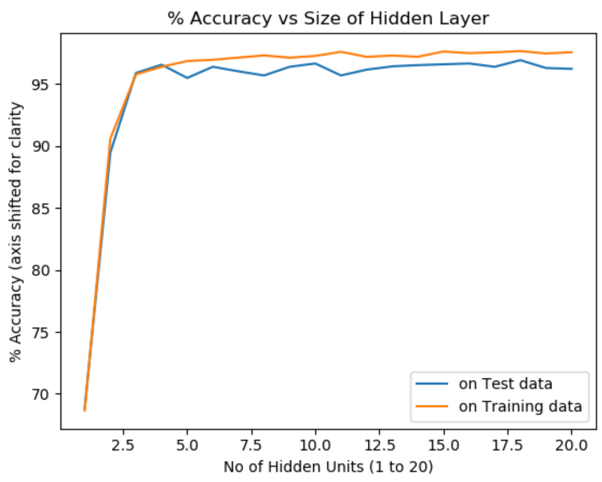


Figure 6: Results of training with various hidden layer sizes

Highest Test Accuracy: 96.93% Hidden Layer Size: 18

∗ Thus, a Hidden Layer with 18 units was found to be optimal.

4.4 Observations Drawn

∗ If the choice of H.L. size were made based on training accuracy, H.L. with 18 units would have been chosen still.

   – This indicates that a H.L. size of 18 is ideally suited to the problem since it outperforms other shapes in fitting to the data (evidenced by Training accuracy) and generalizing to test data.

∗ For the smallest hidden layers, the accuracy is relatively very less.

   – This indicates that these networks are underfitting the data.

∗ For higher values of H.L. size, the testing accuracy begins to fall below the training accuracy.

   – This indicates that these Networks are overfitting the data .

4.5 Concluions

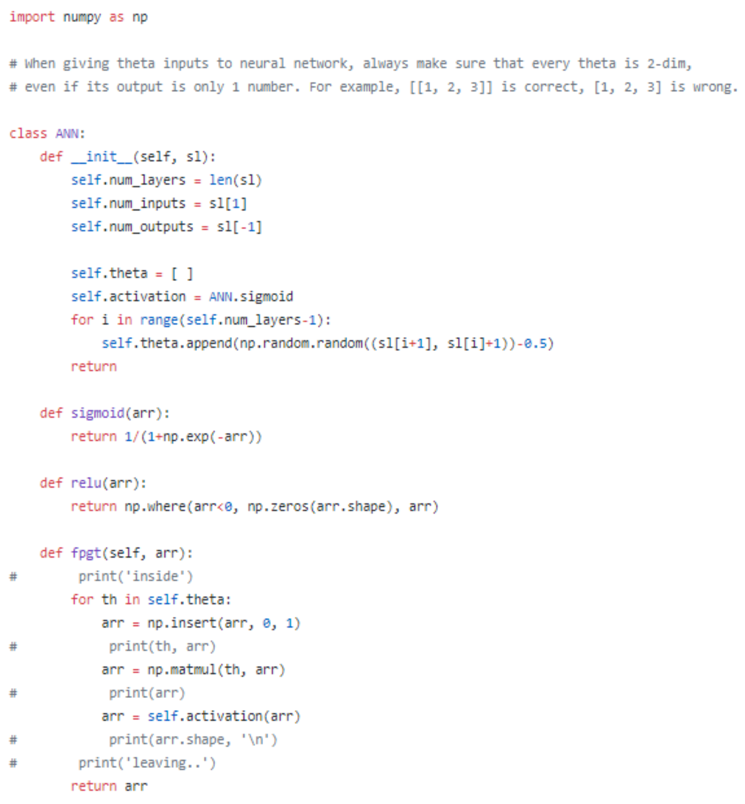
∗ For a given problem, there exists some shape of Neural Network that performs the best. It is worthwhile to search and find the right shape.

∗ If the Network is too complex for the problem, it might overfit the data to some degree.

∗ If the network is not complex enough to learn the patterns in the data, accuracy will be very low and the network underfits.

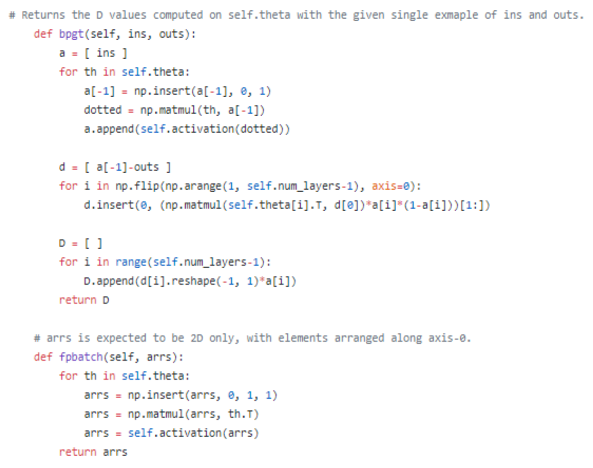
5 Appendix: Python Code

5.1 Class for Neural Networks



```
import numpy as np

# When giving theta inputs to neural network, always make sure that every theta is 2-dim,
# even if its output is only 1 number. For example, [[1, 2, 3]] is correct, [1, 2, 3] is wrong.

class ANN:
    def __init__(self, sl):
        self.num_layers = len(sl)
        self.num_inputs = sl[1]
        self.num_outputs = sl[-1]

        self.theta = [ ]
        self.activation = ANN.sigmoid
        for i in range(self.num_layers-1):
            self.theta.append(np.random.random((sl[i+1], sl[i]+1))-0.5)
        return

    def sigmoid(arr):
        return 1/(1+np.exp(-arr))

    def relu(arr):
        return np.where(arr<0, np.zeros(arr.shape), arr)

    def fpgt(self, arr):
#        print('inside')
        for th in self.theta:
            arr = np.insert(arr, 0, 1)
#            print(th, arr)
            arr = np.matmul(th, arr)
#            print(arr)
            arr = self.activation(arr)
#            print(arr.shape, '\n')
#        print('leaving..')
        return arr
```

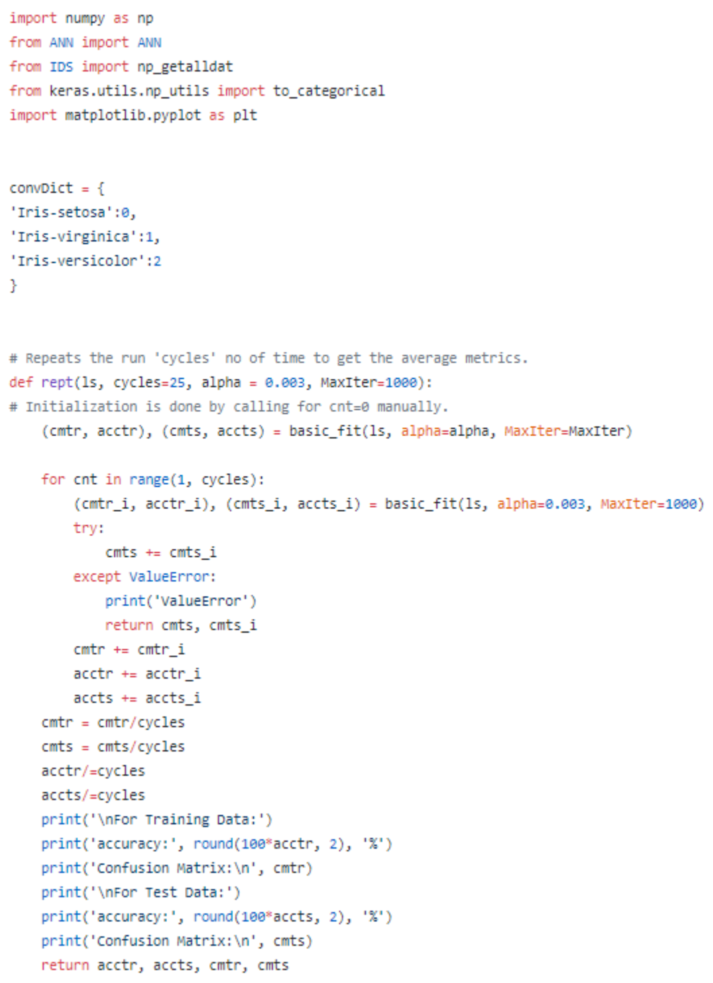
Figure 7: Screenshot



```
# Returns the D values computed on self.theta with the given single exmaple of ins and outs.
    def bpgt(self, ins, outs):
        a = [ ins ]
        for th in self.theta:
            a[-1] = np.insert(a[-1], 0, 1)
            dotted = np.matmul(th, a[-1])
            a.append(self.activation(dotted))

        d = [ a[-1]-outs ]
        for i in np.flip(np.arange(1, self.num_layers-1), axis=0):
            d.insert(0, (np.matmul(self.theta[i].T, d[0])*a[i]*(1-a[i]))[1:])

        D = [ ]
        for i in range(self.num_layers-1):
            D.append(d[i].reshape(-1, 1)*a[i])
        return D

    # arrs is expected to be 2D only, with elements arranged along axis-0.
    def fpbatch(self, arrs):
        for th in self.theta:
            arrs = np.insert(arrs, 0, 1, 1)
            arrs = np.matmul(arrs, th.T)
            arrs = self.activation(arrs)
        return arrs
```

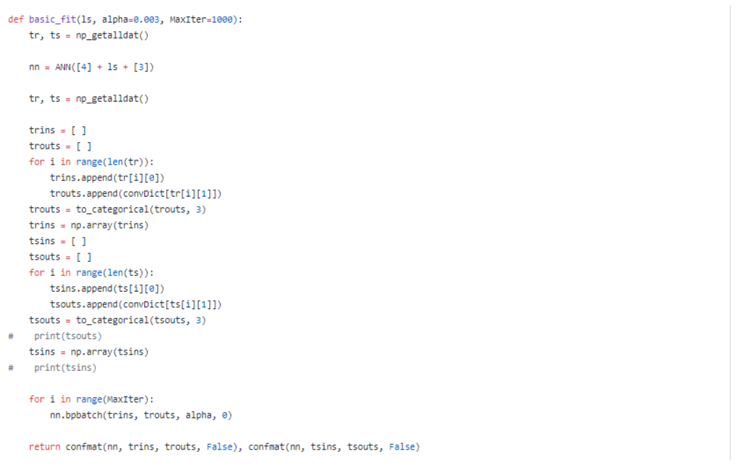
Figure 8: Screenshot

5.2 Functions for Experiment 2



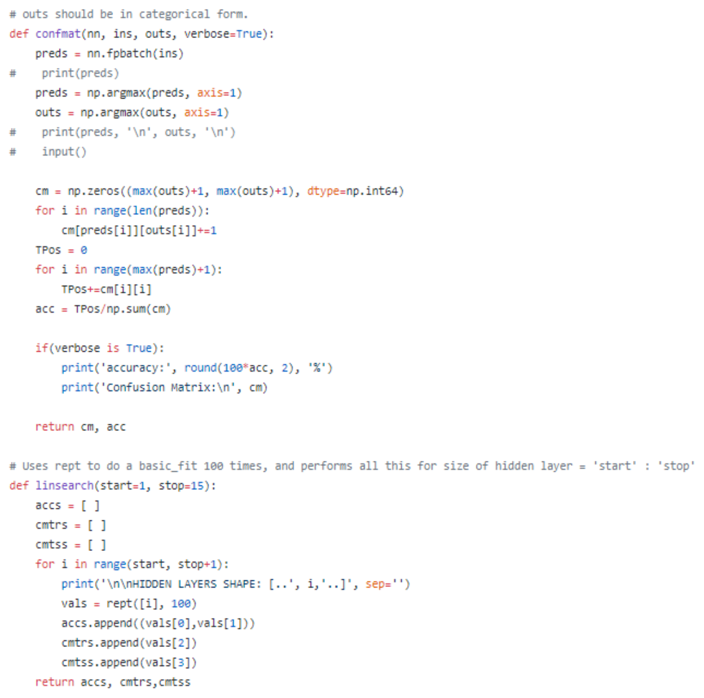
```python
import numpy as np
from ANN import ANN
from IDS import np_getalldat
from keras.utils.np_utils import to_categorical
import matplotlib.pyplot as plt


convDict = {
'Iris-setosa':0,
'Iris-virginica':1,
'Iris-versicolor':2
}


# Repeats the run 'cycles' no of time to get the average metrics.
def rept(ls, cycles=25, alpha = 0.003, MaxIter=1000):
# Initialization is done by calling for cnt=0 manually.
    (cmtr, acctr), (cmts, accts) = basic_fit(ls, alpha=alpha, MaxIter=MaxIter)

    for cnt in range(1, cycles):
        (cmtr_i, acctr_i), (cmts_i, accts_i) = basic_fit(ls, alpha=0.003, MaxIter=1000)
        try:
            cmts += cmts_i
        except ValueError:
            print('ValueError')
            return cmts, cmts_i
        cmtr += cmtr_i
        acctr += acctr_i
        accts += accts_i
    cmtr = cmtr/cycles
    cmts = cmts/cycles
    acctr/=cycles
    accts/=cycles
    print('\nFor Training Data:')
    print('accuracy:', round(100*acctr, 2), '%')
    print('Confusion Matrix:\n', cmtr)
    print('\nFor Test Data:')
    print('accuracy:', round(100*accts, 2), '%')
    print('Confusion Matrix:\n', cmts)
    return acctr, accts, cmtr, cmts
```

Figure 9: Screenshot



```
def basic_fit(ls, alpha=0.003, MaxIter=1000):
    tr, ts = np_getalldat()

    nn = ANN([4] + ls + [3])

    tr, ts = np_getalldat()

    trins = [ ]
    trouts = [ ]
    for i in range(len(tr)):
        trins.append(tr[i][0])
        trouts.append(convDict[tr[i][1]])
    trouts = to_categorical(trouts, 3)
    trins = np.array(trins)
    tsins = [ ]
    tsouts = [ ]
    for i in range(len(ts)):
        tsins.append(ts[i][0])
        tsouts.append(convDict[ts[i][1]])
    tsouts = to_categorical(tsouts, 3)
#    print(tsouts)
    tsins = np.array(tsins)
#    print(tsins)

    for i in range(MaxIter):
        nn.bpbatch(trins, trouts, alpha, 0)

    return confmat(nn, trins, trouts, False), confmat(nn, tsins, tsouts, False)
```

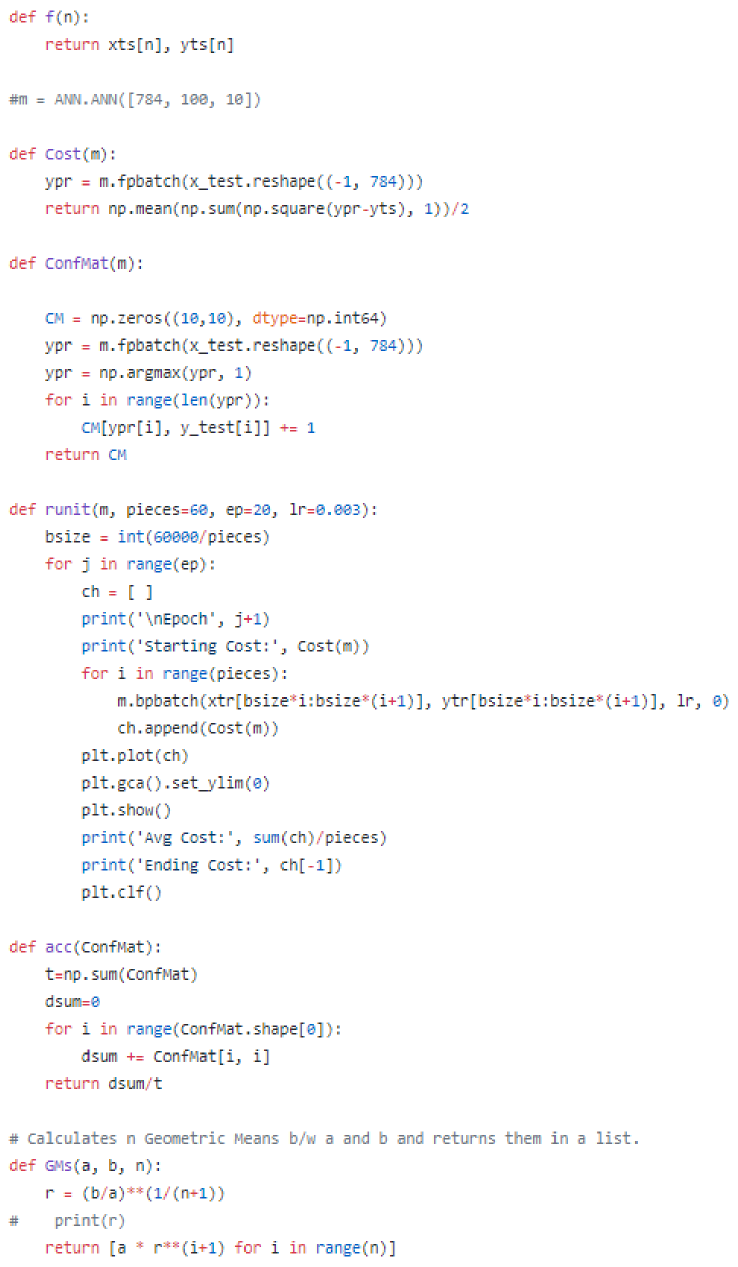
Figure 10: Screenshot



```
# outs should be in categorical form.
def confmat(nn, ins, outs, verbose=True):
    preds = nn.fpbatch(ins)
#    print(preds)
    preds = np.argmax(preds, axis=1)
    outs = np.argmax(outs, axis=1)
#    print(preds, '\n', outs, '\n')
#    input()

    cm = np.zeros((max(outs)+1, max(outs)+1), dtype=np.int64)
    for i in range(len(preds)):
        cm[preds[i]][outs[i]]+=1
    TPos = 0
    for i in range(max(preds)+1):
        TPos+=cm[i][i]
    acc = TPos/np.sum(cm)

    if(verbose is True):
        print('accuracy:', round(100*acc, 2), '%')
        print('Confusion Matrix:\n', cm)

    return cm, acc

# Uses rept to do a basic_fit 100 times, and performs all this for size of hidden layer = 'start' : 'stop'
def linsearch(start=1, stop=15):
    accs = [ ]
    cmtrs = [ ]
    cmtss = [ ]
    for i in range(start, stop+1):
        print('\n\nHIDDEN LAYERS SHAPE: [..', i,'..]', sep='')
        vals = rept([i], 100)
        accs.append((vals[0],vals[1]))
        cmtrs.append(vals[2])
        cmtss.append(vals[3])
    return accs, cmtrs,cmtss
```

Figure 11: Screenshot

5.3 Functions for Experiment 1



```
def f(n):
    return xts[n], yts[n]

#m = ANN.ANN([784, 100, 10])

def Cost(m):
    ypr = m.fpbatch(x_test.reshape((-1, 784)))
    return np.mean(np.sum(np.square(ypr-yts), 1))/2

def ConfMat(m):

    CM = np.zeros((10,10), dtype=np.int64)
    ypr = m.fpbatch(x_test.reshape((-1, 784)))
    ypr = np.argmax(ypr, 1)
    for i in range(len(ypr)):
        CM[ypr[i], y_test[i]] += 1
    return CM

def runit(m, pieces=60, ep=20, lr=0.003):
    bsize = int(60000/pieces)
    for j in range(ep):
        ch = [ ]
        print('\nEpoch', j+1)
        print('Starting Cost:', Cost(m))
        for i in range(pieces):
            m.bpbatch(xtr[bsize*i:bsize*(i+1)], ytr[bsize*i:bsize*(i+1)], lr, 0)
            ch.append(Cost(m))
        plt.plot(ch)
        plt.gca().set_ylim(0)
        plt.show()
        print('Avg Cost:', sum(ch)/pieces)
        print('Ending Cost:', ch[-1])
        plt.clf()

def acc(ConfMat):
    t=np.sum(ConfMat)
    dsum=0
    for i in range(ConfMat.shape[0]):
        dsum += ConfMat[i, i]
    return dsum/t

# Calculates n Geometric Means b/w a and b and returns them in a list.
def GMs(a, b, n):
    r = (b/a)**(1/(n+1))
#    print(r)
    return [a * r**(i+1) for i in range(n)]
```

Figure 12: Screenshot

****End of Neural Networks (Part 2)****