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.

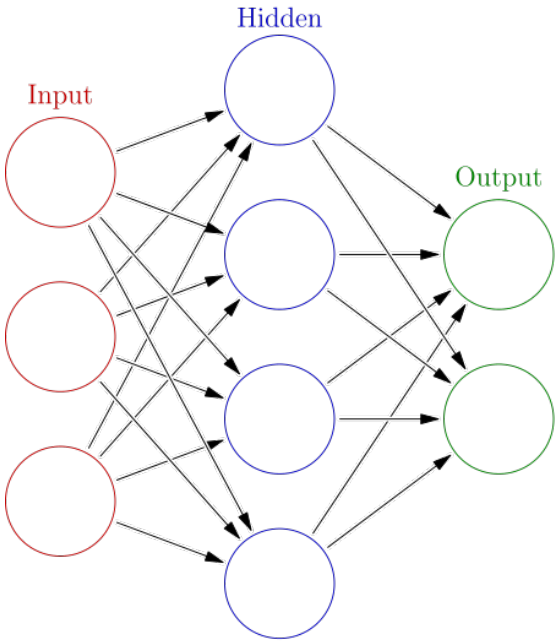

Figure 1: Structure of a Neural Network with 1 Hidden Layer, by Glosser.ca, Wikipedia, Public Domain

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:

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

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

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

$$Z = X \cdot W + b$$
$$\implies Z = b + x_0 w_0 + x_1 w_1 + \cdots + x_n w_n \tag{3}$$

An activation function is used to introduce non-linearity:

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

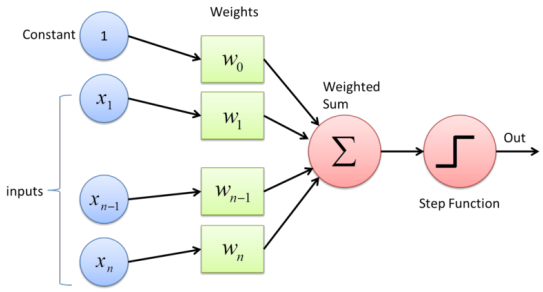

Figure 2: Calculations in a Perceptron, by unknown, Wikipedia, public domain

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 for this section

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 Function 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 δs: 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-dimensional 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 (SGD) 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 SGD 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 SGD.

3 Experiment 1

3.1 Objective

To build a Neural Network that recognizes handwritten digits by training on the MNIST database [4], 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 sizes h1 and h2. Their values were chosen as h1=183, h2=42 These 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

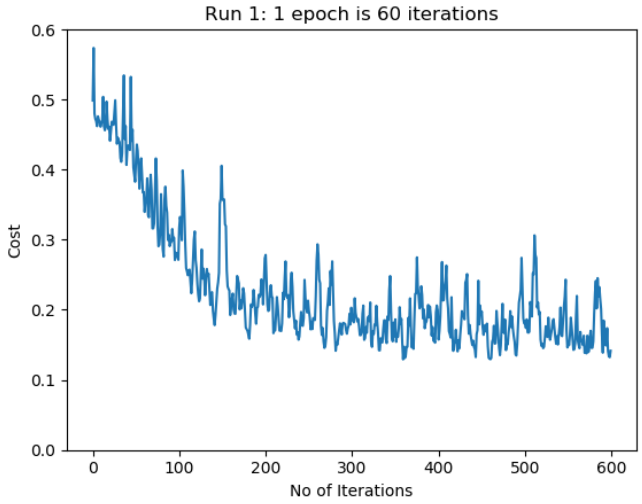

Figure 3: Fall of Cost for Run 1

* 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

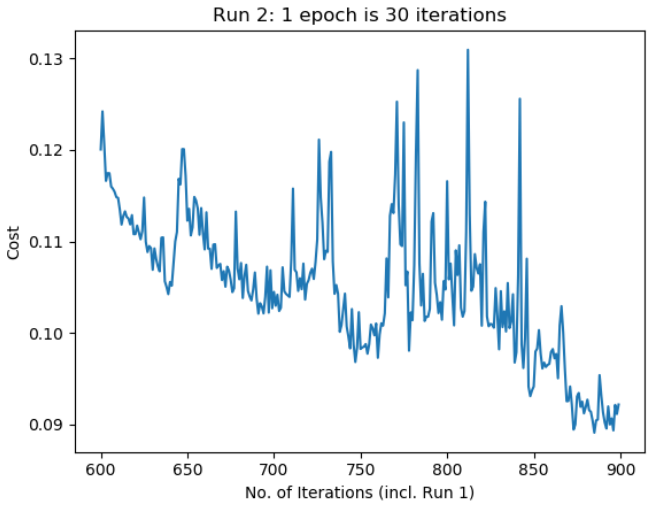

Figure 4: Fall of Cost for Run 2

* 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

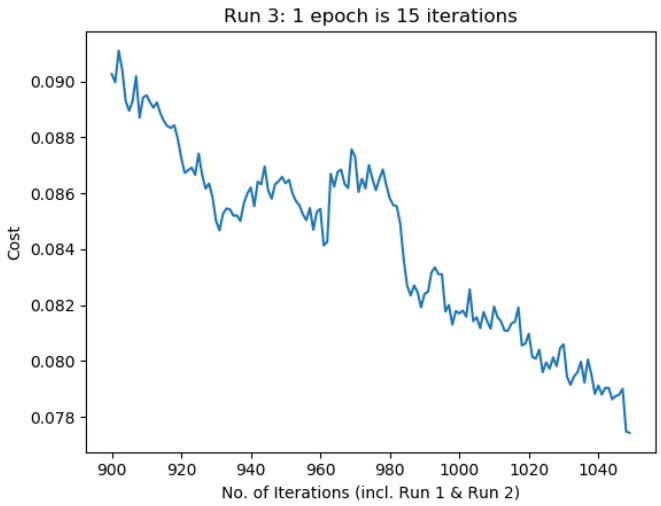

Figure 5: Fall of Cost for Run 3

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:

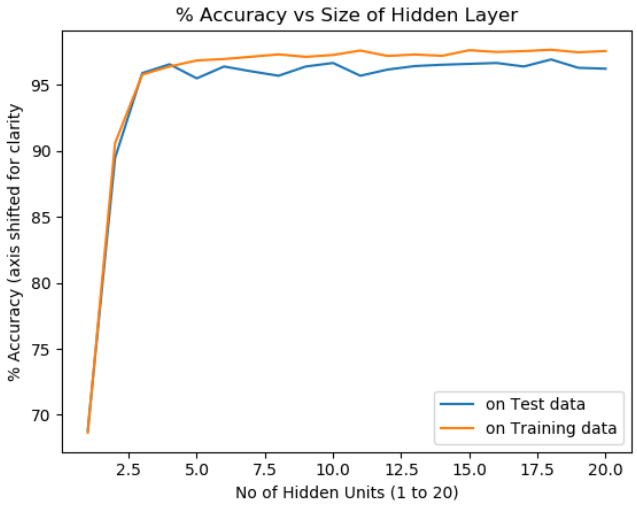

Figure 6: Results of training with various hidden layer sizes

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

```
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)****