TanushreeNori-CS-598-HW1

September 7, 2018

1 Digit Recognition using MNIST dataset

1.0.1 Implementation of a Neural Network from scratch in Python

Tanushree Nori

In [1]:
```
# Import of libraries
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import h5py
import time
import copy
from random import randint
```

In [2]:
```
# Data import
MNIST_data = h5py.File('C:/Users/Tanushree Nori/Downloads/MNISTdata.hdf5', 'r')
x_train = np.float32(MNIST_data['x_train'][:] )
y_train = np.int32(np.array(MNIST_data['y_train'][:,0]))
x_test = np.float32( MNIST_data['x_test'][:] )
y_test = np.int32( np.array( MNIST_data['y_test'][:,0] ) )

# Transpose matrices
x_train= x_train.T
x_test = x_test.T

MNIST_data.close()
```

In [3]:
```
# one-hot encode labels so that data against a particular label can be labeled '1' and

# Encoding and transpose y_train
digits = 10
examples = y_train.shape[0]
y = y_train.reshape(1, examples)
Y_new = np.eye(digits)[y.astype('int32')]
y_train = Y_new.T.reshape(digits, examples)
```

```
# Encoding and transpose y_test
digits = 10
examples = y_test.shape[0]
y = y_test.reshape(1, examples)
Y_new = np.eye(digits)[y.astype('int32')]
y_test = Y_new.T.reshape(digits, examples)
```

In [4]:
```
# Quick summary of the dataset
# Libraries to visualise the data
import mnist
import scipy.misc
import warnings
warnings.filterwarnings('ignore')

# Look at a sample handwritten digit
images = mnist.train_images()
scipy.misc.toimage(scipy.misc.imresize(images[0,:,:] * -1 + 256, 10.))
```

Out[4]:

In [5]:
```
# Checking the dimensions of mnist test and train data
# 60,000 is for training and 10,000 is used as test data
x_train.shape
```

```
Out[5]: (784, 60000)

In [6]: x_test.shape

Out[6]: (784, 10000)

In [7]: y_train.shape

Out[7]: (10, 60000)

In [8]: y_test.shape

Out[8]: (10, 10000)

In [10]: #number of inputs
         num_inputs = 28*28
         #number of neurons
         neurons = 70
         # number of outputs
         digits = 10

         # Constructing the network with weights and biases
         model = {}
         model['W1'] = np.random.randn(neurons,num_inputs) / np.sqrt(num_inputs)
         model['b1'] = np.zeros((neurons,1)) / np.sqrt(num_inputs)
         model['W2'] = np.random.randn(digits,neurons) / np.sqrt(neurons)
         model['b2'] = np.zeros((digits,1)) / np.sqrt(neurons)
```

The code snippet following this is where we have functions for the activation formula, the forward propagation and backward propagation algorithms :

- The forward function, for each layer computes z and a
- The backpropagation function, in very brief, computes the error in the layer and back propagates it across the the network. It essentially computes the error backwards starting from the final layer and returns it. The weights and biases are then updated in the execution portion of the code, below

```
In [11]: # Activation function is the sigmoid function
         def softmax_function(z):
             ZZ = 1. / (1. + np.exp(-z))
             return ZZ

         # Forward feed function - For each layer (l) =2,3,,L compute zl=wlal1+bl and al=(zl).
         def forward(x,model):
             # A temporary storage to store the z and a for each batch
             cache = {}

             cache["Z1"] = np.matmul(model["W1"], x) + model["b1"]
             cache["A1"] = softmax_function(cache["Z1"])
```

```
        cache["Z2"] = np.matmul(model["W2"], cache["A1"]) + model["b2"]
        # The final activation of the last year is different as we replace the final node
        # with a 10-unit layer. Final activation is normalized exponentials of it's z -va
        cache["A2"] = np.exp(cache["Z2"]) / np.sum(np.exp(cache["Z2"]), axis=0)

        return cache

    # Back propagation function
    def backward(x,y,model,temp):
        dZ2 = cache["A2"] - y
        dW2 = (1./m_batch) * np.matmul(dZ2, cache["A1"].T)
        db2 = (1./m_batch) * np.sum(dZ2, axis=1, keepdims=True)

        dA1 = np.matmul(model["W2"].T, dZ2)
        dZ1 = dA1 * softmax_function(cache["Z1"]) * (1 - softmax_function(cache["Z1"]))
        dW1 = (1./m_batch) * np.matmul(dZ1, X.T)
        db1 = (1./m_batch) * np.sum(dZ1, axis=1, keepdims=True)

        grads = {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2}

        return grads
```

Summary of the network and hyper paramters: - Number of neurons in the hidden layer - 70 - Learning rate - 1 - Batch size for mini-batch gradient descent is 100 - Number of Epochs is 20

```
In [14]: # The execution portion
         np.random.seed(138)

         import time
         time1 = time.time()

         learning_rate = 1
         batch_size = 100
         num_batch = (60000//batch_size)

         # train
         for i in range(20):

             for j in range(num_batch):

                 begin = j * batch_size
                 end = min(begin + batch_size, x_train.shape[1] - 1)
                 X = x_train[:, begin:end]
                 Y = y_train[:, begin:end]
                 m_batch = end - begin

                 cache = forward(X, model)
                 grads = backward(X, Y, model, cache)
```

```
            model["W1"] = model["W1"] - learning_rate * grads["dW1"]
            model["b1"] = model["b1"] - learning_rate * grads["db1"]
            model["W2"] = model["W2"] - learning_rate * grads["dW2"]
            model["b2"] = model["b2"] - learning_rate * grads["db2"]

        cache = forward(x_train, model)
        cache = forward(x_test, model)
        print("Epoch {}" .format(i+1))

    print("Done.")

    time2 = time.time()
    print("\n time taken is {} seconds".format(time2-time1))
```

```
Epoch 1
Epoch 2
Epoch 3
Epoch 4
Epoch 5
Epoch 6
Epoch 7
Epoch 8
Epoch 9
Epoch 10
Epoch 11
Epoch 12
Epoch 13
Epoch 14
Epoch 15
Epoch 16
Epoch 17
Epoch 18
Epoch 19
Epoch 20
Done.

 time taken is 41.994760513305664 seconds
```

In [15]:
```
# Calculation of final accuracy

from sklearn.metrics import accuracy_score
cache = forward(x_test, model)
predictions = np.argmax(cache["A2"], axis=0)
labels = np.argmax(y_test, axis=0)

print(accuracy_score(labels, predictions))
```

```
0.9763
```

The final accuracy is 97.63%