Neural Networks - My Research Project over the Holidays

Siddhartha Rajeev, May 2021

More out of curiosity, I decided to foray into the much broached, yet mysterious realm of “Artificial Intelligence”, and to be more specific, the realm concerning deep learning using neural networks.

Part of my curiosity entailed the inner workings of a deep learning system and how I could build one myself. After much research and browsing, I was able to understand how a fully connected deep learning neural network functions – with activation layers and activation functions and more importantly the underlying algorithm of forward propagation and backpropagation, the latter including the fascinating idea of “gradient descent” to converge to a local minimum of the network’s “error”.

As we had our holiday homework based on python modules anyway, I thought – why not tackle both projects together? And hence I wrote the following python modules (with the aid of my research on the subject from multiple sources).

The purpose of the following modules is to **classify handwritten digits** from the MNIST dataset**,** which is I learn a common exercise when diving into the subject of neural networks.

(Turn to page 4 for results)

```
# Module “layers”

class Layer(): #Abstract class
    def __init__(self):
        self.input = None
        self.output = None
    def forwardPropagation(self, inputData):
        raise NotImplementedError

    def backwardPropagation(self, outputError, learningRate):
        raise NotImplementedError
```

```
#Module “dense_layers”

import numpy as np
from layers import Layer

class DenseLayer(Layer):
    def __init__(self, inputSize, outputSize):
        self.weights = np.random.rand(outputSize, inputSize) - 0.5
        self.biases = np.random.rand(outputSize, 1) - 0.5

    def forwardPropagation(self, inputData):
        self.input = inputData
        self.output = np.dot(self.weights, self.input) + self.biases
        return self.output

    def backwardPropagation(self, outputError, learningRate): #Note that the outputError,
weightsError, biasesError are actually the value of the partial derivative of E wrt y, W
and b respectively
        inputError = np.dot(self.weights.T, outputError)
        weightsError = np.dot(outputError, self.input.T)
```

```
        biasesError = outputError

        #Time to update the weights and biases for the neuron layer
        self.weights = self.weights - weightsError*learningRate
        self.biases = self.biases - biasesError*learningRate
        return inputError
```

#module “activation_layers”

```
 import numpy as np
from layers import Layer

class ActivationLayer(Layer):
    def __init__(self, activation, activationDerivative):
        self.activation = activation
        self.activationDerivative = activationDerivative

    def forwardPropagation(self, inputData):
        self.input = inputData
        self.output = self.activation(self.input)
        return self.output

    def backwardPropagation(self, outputError, learningRate): #Learning rate not required
for activation layers
        inputError = outputError*self.activationDerivative(self.input) #This is
corresponding position multiplication for numpy arrays of any matching dimensions
        return inputError
```

---

module “activation_functions”

```
import numpy as np

def sigmoid(inputData):
    return 1/(1+np.exp(-1*inputData))
def sigmoidDerivative(inputData):
    numerator = np.exp(-1*inputData)
    denominator = np.power(1+np.exp(-1*inputData), 2)
    return(numerator/denominator)
def tanh(inputData):
    return np.tanh(inputData)
def tanhDerivative(inputData):
    return 1-np.power(np.tanh(inputData), 2)
```

---

module “losses”

```
import numpy as np

def MSE(y_true, y):
    return np.mean(np.power(y_true-y, 2)) #We calculate the mean of all the elements in the
column vector of errors for that sample
def MSEDerivative(y_true, y):
    return( (2*(y-y_true))/y_true.shape[0])
```

---

module “networks”

```
import numpy as np
class Network():
    def __init__(self):
        self.layers = []
```

```
        self.loss = None
        self.lossDerivative = None

    def addLayer(self, layerToAdd):
        self.layers.append(layerToAdd)

    def useLoss(self, lossToUse, lossDerivativeToUse):
        self.loss = lossToUse
        self.lossDerivative = lossDerivativeToUse

    def predict(self, inputData): #Input data will be a set of input vectors (sampleCount x
inputSize x 1)
        sampleCount = inputData.shape[0]
        result = []

        for i in range(sampleCount):
            output = inputData[i]
            for layer in self.layers:
                output = layer.forwardPropagation(output)

            result.append(output)

        return result

    def fit(self, x_train, y_train, epochs, learningRate): #Epochs are the number of passes
through the whole dataset
        for i in range(epochs):
            displayError = 0
            for j in range(x_train.shape[0]):
                output = x_train[j]
                for layer in self.layers:
                    output = layer.forwardPropagation(output)
                displayError +=  self.loss(y_train[j], output)

                error = self.lossDerivative(y_train[j], output)
                for layer in reversed(self.layers):
                    error = layer.backwardPropagation(error, learningRate)

            displayError /= x_train.shape[0]
            print("Epoch {0}/{1} complete. Error = {2}".format(i+1, epochs, displayError))
```

---

module “MNIST_implementation”

```
import numpy as np
from networks import Network
from layers import Layer
from dense_layers import DenseLayer
from activation_layers import ActivationLayer
from losses import MSE, MSEDerivative
from activation_functions import tanh, tanhDerivative
from preprocessing import toGreyscale

from keras.datasets import mnist
from keras.utils import np_utils

#Loading the mnist database from server
(x_train, y_train), (x_test, y_test) = mnist.load_data()

#x_train, x_test of shape (60000, 28, 28), (10000, 28, 28) and y_train, y_test of shape
(60000,), (10000,)
#Now we reshape and normalise:

x_train = x_train.reshape(x_train.shape[0], 784, 1)
x_train = x_train.astype('float32')
```

```
x_train /= 255


y_train = np_utils.to_categorical(y_train)  #Adds an extra dimension by converting the number 3 for example to [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] Shape change from (60000,) to 60000, 10)
y_train = y_train.reshape(y_train.shape[0], y_train.shape[1], 1)


x_test = x_test.reshape(x_test.shape[0], 784, 1)
x_test = x_test.astype('float32')
x_test /= 255

y_test = np_utils.to_categorical(y_test)
y_test = y_test.reshape(y_test.shape[0], y_test.shape[1], 1)


#Now we establish the network

net = Network()
net.addLayer(DenseLayer(784, 100))
net.addLayer(ActivationLayer(tanh, tanhDerivative))
net.addLayer(DenseLayer(100, 50))
net.addLayer(ActivationLayer(tanh, tanhDerivative))
net.addLayer(DenseLayer(50, 10))
net.addLayer(ActivationLayer(tanh, tanhDerivative))

#We shall train on 1000 samples only
net.useLoss(MSE, MSEDerivative)
net.fit(x_train[5000:6500], y_train[5000:6500], epochs = 100, learningRate = 0.1)

#Now we shall load a single handwritten digit to test the model on. It must be a 1 x 784 x 1 array.
testData = toGreyscale()
output = np.array(net.predict(testData))
print(output)
print(np.argmax(output[0]))
```

And done! The following number 3 was drawn by me on a 28x28 grid in MS Paint. On training the network over 100 cycles (or epochs), the network **recognised the digit drawn as a 3!**

```
Epoch 1/20 complete. Error = 0.12381156133414206
Epoch 2/20 complete. Error = 0.06723525189147256
Epoch 3/20 complete. Error = 0.04821336309219638
Epoch 4/20 complete. Error = 0.038722577663221354
Epoch 5/20 complete. Error = 0.03258935481758163
Epoch 6/20 complete. Error = 0.02855318936994717
Epoch 7/20 complete. Error = 0.02547731052402534
Epoch 8/20 complete. Error = 0.02314707779903862
Epoch 9/20 complete. Error = 0.02120204647833404
Epoch 10/20 complete. Error = 0.019411719353047606
Epoch 11/20 complete. Error = 0.018067561444162952
Epoch 12/20 complete. Error = 0.01700979277462541
Epoch 13/20 complete. Error = 0.015873438300134983
Epoch 14/20 complete. Error = 0.01506280504289223
Epoch 15/20 complete. Error = 0.01420895534548682
Epoch 16/20 complete. Error = 0.013791923952497624
Epoch 17/20 complete. Error = 0.01307691184364708
Epoch 18/20 complete. Error = 0.012920041190345339
Epoch 19/20 complete. Error = 0.012548202675187168
Epoch 20/20 complete. Error = 0.012313762374278852
[[[ 3.77165010e-03]
  [ 5.16649889e-01]
  [ 1.53719712e-04]
  [ 9.57525993e-01]
  [ 3.92376424e-01]
  [ 1.46951436e-02]
  [ 4.58989164e-03]
  [-1.20148784e-01]
  [-1.43148877e-01]
  [-2.56573348e-01]]]
The number input is a:  3
>>> 
```

My hand-drawn ‘number three’
