Dhynasah Cakir

Machine learning

January 29, 2020

Artificial neural network

Overview

This project uses python to create a neural network that can distinguish handwritten digits. Training data is fed through the network. The network uses the data to learn to recognize the digits, and then applies that to the testing data. This network uses sigmoid as the activation fiction and cross entropy for the cost calculation.

Approach

This neural network uses Sigmoid as the activation function because small changes in the weights and bias cause small changes in the output. Therefore, the effect is incremental. The sigmoid function is: $\sigma(z) = \frac{1}{1+e^{-z}}$. Apart for using sigmoid as the activation function, another significant part of this neural network is that is learns with gradient descent. This is an algorithm that helps find a set of weights and biases which make the cost as small as possible. What makes gradient descent Stochastic is that the work is divided into mini sets, or epochs. SGD (stochastic gradient descent) works by minimizing the loss function by updating the weights with each epoch. This occurs by calculating the gradient or taking the derivative of the loss function with respect to the weights in the model. The cost function I am referring to is the

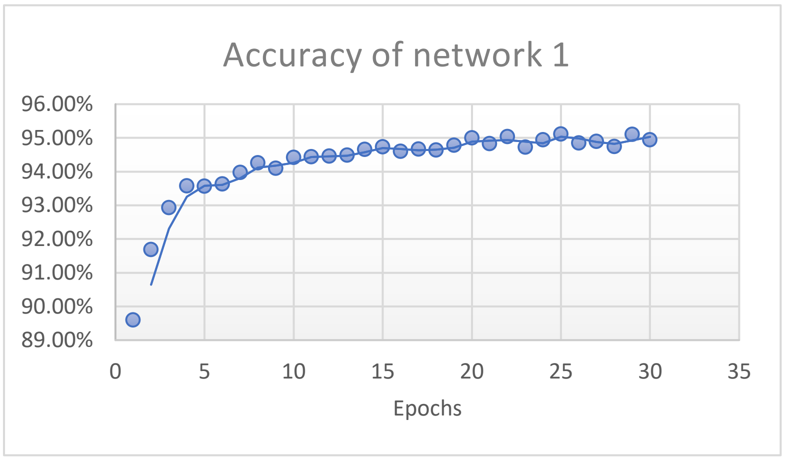
Quadradic cost function: $C = \frac{(\gamma - a)^2}{2}$. *a* is the neurons output, and *y* is the corresponding desired output. This act of calculating the gradients in order to update the weights occurs through the process of backpropagation. The chain rule is used twice in backpropagation. Its first used to calculate the loss with respect to the last layer of weights in the network. Then it's used again to calculate the first term within the derivative, which itself is the derivative of the loss with respect to the activation output. These results depend on components that reside later in the network. Thus, derivatives need to be calculated that depend on components later in the network first, then use those derivatives for calculations for the gradient of the loss with respect to components that come earlier in the network. This is achieved by repeatedly applying the chain rule in a backwards fashion.

Training process: When data is passed through the model the data propagates forward until it reaches the output layer. Each node receives its input from the previous layer. And the input is a weighted sum of the weights at each of the connections. Multiplied by the previous layers output. Then the weighted sum is passed to an activation function. The result from the activation function is an output for that node which is then passed as the input to the next layer in the model. This happens for each layer in the network until the output is reached. Each of the output nodes corresponds to a different answer and whichever node has the highest activation number the network will pick that as the correct answer. Given the result from the output the loss is then calculated. The loss is the difference between what the model predicted, and the actual answer. Gradient decent is supposed to minimize the loss.

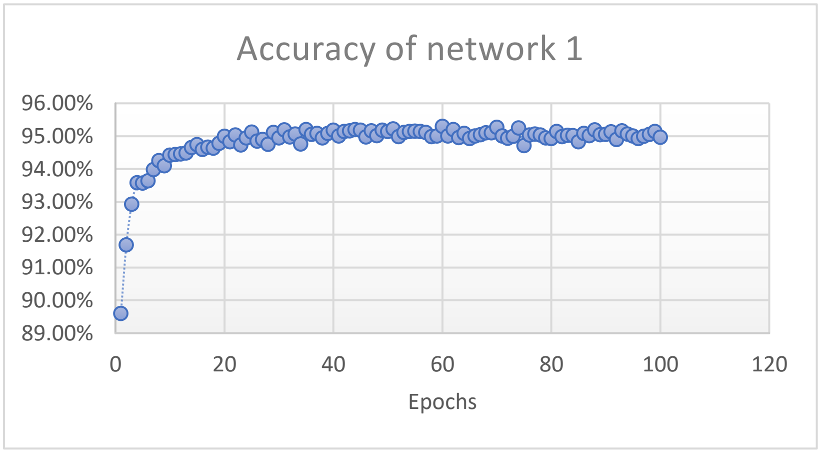
Results

For my first attempt I just got the network to run. It ran for 30 epochs, with the input layer chosen as 784 nodes, and a learning rate of 3. Epoch 18 had a 95.24% accuracy. But this was after taking some time to find the best learning rate. One of the downfalls of this network is that if the right learning rate is not chosen the network might seem to be preforming badly, and the user might not know where the problem lies, its hard to isolate the fact that this is caused by the learning rate. But overall this is an effective network which still got over 90% accuracy. That was the goal and it was achieved.

Graph 1 shows that as the epoch number increased the rate of accuracy increased for the most part, there are sometimes where the network backtracks in accuracy, but overall the trend is upward with a learning rate of 3.0. Graph 2 is the same, except with 100 epochs instead of 30.



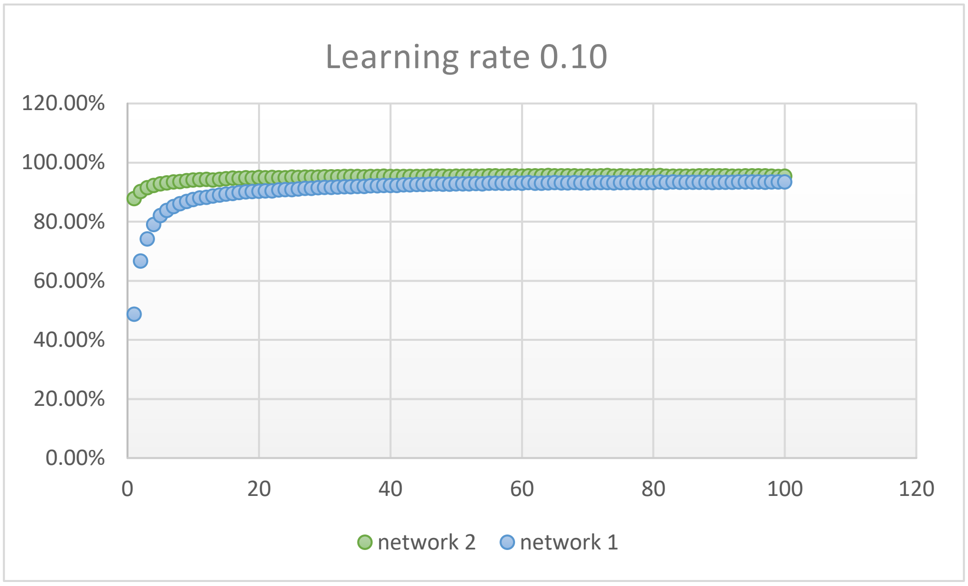
Graph 1



Graph 2

On improvement made was the cost function. This time the function used was cross entropy. Cross entropy is used to improve the learning rate. $\frac{\partial c}{\partial b} = \frac{1}{n}\sum_x(\sigma(z) - y)$. At Some point with the quadratic cost function, if the learning rate is not chosen correctly, the network can have a learning slowdown. When it is a little wrong it can correct itself, but when it is extremely wrong it takes a longer time to improve and to correct itself. Therefore, the cross- entropy function is used to resolve this issue. It still functions in the beneficial way the quadratic cost function does by moving towards zero as the network gets better at computing the desired output, *y*, for all the training inputs. The way it works differently than the Quadratic cost function is that the rate at which the weights learn is controlled by the error in the output. The larger the error the faster the neuron will learn. The same is true for the biases.

graph 3 shows the accuracy when a bad learning rate is chosen. we can see that the second edition with cross-entropy learns a lot faster but also its initial accuracy is a lot higher. The minimum accuracy for network 2 is 87.79% whereas the minimum accuracy for network 1 is 48.65%. And network 1 doesn't reach the same level of accuracy with 100 epochs as network 2 does. Its highest accuracy is 93.49% compared to network 2 at 95.59%. Although network 1 can reach 95% with a learning rate of 3.0. It doesn't with poorly chosen learning rates.   Lastly, I did run the network with the validation data set to ensure the network would function the same with a new set of inputs, and to ensure that it wasn't overfitting. As you can see the network preforms well and even better after 300 epochs. I didn't do anymore epochs after that to avoid overfitting.


Learning rate 0.10
120.00%
100.00%
80.00%
60.00%
40.00%
20.00%
0.00%
0
20
40
60
80
100
120
network 2
network 1

```python
# -*- coding: utf-8 -*-
"""
Created on Wed Apr  1 12:41:49 2020

@author: Mauri
"""

"""
mnist_data
~~~~~~~~~~~~
A library to load the MNIST image data.  load_data_wrapper is the
function called by our neural network code.
"""

#### Libraries
import pickle
import gzip
import numpy as np

def data():
    """Return the MNIST data as a tuple containing the training data,
    the validation data, and the test data.
    The training_data is returned as a tuple with two entries.
    The first entry contains the actual training images.
    The second entry in the training_data tuple is a numpy ndarray
    containing 50,000 entries.
    """
    f = gzip.open('mnist.pkl.gz', 'rb')
    train_data, val_data, test_data = pickle.load(f, encoding="latin1")
    f.close()
    return (train_data, val_data, test_data)

def data_loader():
    """Return a tuple containing (training_data, validation_data,
    test_data). Based on ``data, but the format is more
    convenient for use in our implementation of neural networks.
    In particular, training_data is a list containing 50,000
    2-tuples (x, y).
    validation_data and test_data are lists containing 10,000
    2-tuples (x, y)."""
    tr_d, va_d, te_d = data()
    train_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
    train_results = [vector_result(y) for y in tr_d[1]]
    train_data = zip(train_inputs, train_results)
    val_in = [np.reshape(x, (784, 1)) for x in va_d[0]]
    val_data = zip(val_in, va_d[1])
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = zip(test_inputs, te_d[1])
    return (train_data, val_data, test_data)

def vector_result(j):
    """Return a 10-dimensional unit vector with a 1.0 in the jth
    position and zeroes elsewhere. """
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

```
"""
Created on Mon Apr 13 16:12:15 2020

@author: Dhynasah Cakir
"""


from random import shuffle
import numpy as np
import mnist_data_configure



class CrossEntropyCost(object):

    @staticmethod
    def fn(a, y):
        """Return the cost associated with an output. np.nan_to_num is used to
        ensure numerical stability. If both a and y have a 1 then this fuction
        returns nan. and nan is converted to (0.0).
        """
        return np.sum(np.nan_to_num(-y*np.log(a)-(1-y)*np.log(1-a)))

    @staticmethod
    def delta(z, a, y):
        """Return the error delta from the output layer.
        """
        return (a-y)


# the list sizes contains the number of neurons in the respective layers
#the biases and weights are initalized with random numbers
#this assumes the first layer of neurons in an input layer
class Network(object):
    def __init__(self,sizes, cost=CrossEntropyCost):
        self.num_layers= len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y,1) for y in sizes[1:]]
        self.weights =[np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
        self.cost = cost


    def ff(self, a):
        #return the output of the network if a is input
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w,a)+b)
        return a

    #stochastic gradient descent
    #lrate is the learning rate
    # train_data is a list of tuples x as inputs and y as desired

    def SG_Descent(self, train_data, epochs, tiny_batch_size, lrate,
            lmbda = 0.0,
            evaluation_data=None):
```

```
    train_data = list(train_data)
    n= len(train_data)

    if evaluation_data:
        evaluation_data = list(evaluation_data)
        n_eval = len(evaluation_data)
    for j in range(epochs):
        shuffle(train_data)
        tiny_batches = [
            train_data[i:i+tiny_batch_size]
            for i in range(0, n, tiny_batch_size)]
        for batch in tiny_batches:
            self.update_tiny_batch(
                batch, lrate, lmbda, len(train_data))
        percent = (self.evalu(evaluation_data)/n_eval)*100
        print("{}%".format(percent))



def update_tiny_batch(self, tiny_batch, lrate,lmbda, n):
    grad_b = [np.zeros(b.shape) for b in self.biases]
    grad_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in tiny_batch:
        delta_grad_b, delta_grad_w = self.backprop(x, y)
        grad_b = [nb+dnb for nb, dnb in zip(grad_b, delta_grad_b)]
        grad_w = [nw+dnw for nw, dnw in zip(grad_w, delta_grad_w)]
    self.weights = [(1-lrate*(lmbda/n))*w-(lrate/len(tiny_batch))*nw
                    for w, nw in zip(self.weights, grad_w)]
    self.biases = [b-(lrate/len(tiny_batch))*nb
                    for b, nb in zip(self.biases, grad_b)]

def backprop(self, x, y):
    """Return a tuple ``(grad_b, grad_w)`` representing the
    gradient for the cost function C_x.  ``grad_b`` and
    ``grad_w`` are layer-by-layer lists of numpy arrays."""
    grad_b = [np.zeros(b.shape) for b in self.biases]
    grad_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activ = x
    activs = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activ)+b
        zs.append(z)
        activ = sigmoid(z)
        activs.append(activ)
    # backward pass
    delta = (self.cost).delta(zs[-1], activs[-1], y)
    grad_b[-1] = delta
    grad_w[-1] = np.dot(delta, activs[-2].transpose())
    # l = 1 means the last layer of neurons, l = 2 is the
    # second-last layer, and so on.
    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        grad_b[-l] = delta
```

```
            grad_w[-l] = np.dot(delta, activs[-l-1].transpose())
        return (grad_b, grad_w)

    def total_cost(self, data, lmbda, convert=False):
        """Return the total cost for the data set ``data``.  The flag
        ``convert`` should be set to False if the data set is the
        training data (the usual case), and to True if the data set is
        the validation or test data.
        """
        cost = 0.0
        for x, y in data:
            a = self.ff(x)
            data_list= list(data)
            if convert: y = vect_res(y)
            cost += self.cost.fn(a, y)/len(data_list)
        cost += 0.5*(lmbda/len(data_list))*sum(
            np.linalg.norm(w)**2 for w in self.weights)
        return cost

    def accuracy(self, data, convert=False):
        """Return the number of inputs in ``data`` for which the neural
        network outputs the correct result.
        The flag ``convert`` should be set to False if the data set is
        validation or test data (the usual case), and to True if the
        data set is the training data.
        """
        if convert:
            results = [(np.argmax(self.ff(x)), np.argmax(y))
                        for (x, y) in data]
        else:
            results = [(np.argmax(self.ff(x)), y)
                         for (x, y) in data]
        return sum(int(x == y) for (x, y) in results)

    def evalu(self, test_data):
        """Return the number of test inputs for which the neural
        network outputs the correct result. """
        test_results = [(np.argmax(self.ff(x)), y)
                         for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

def vect_res(j):
    """Return a 10-dimensional unit vector with a 1.0 in the j'th position
    and zeroes elsewhere.
    """
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e

def sigmoid(z):
    s = 1 / (1 + np.exp(-z))
    return s

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

```
def main():

  training_data, validation_data, test_data = mnist_data_configure.data_loader()
  net = Network([784,30,10],cost =CrossEntropyCost)
  net.SG_Descent(training_data,100,10,0.10, evaluation_data=test_data)




if __name__ == "__main__":
    main()
```