**Evan Apinis**

**ECE 577 Assignment 1**

**2/18/24**

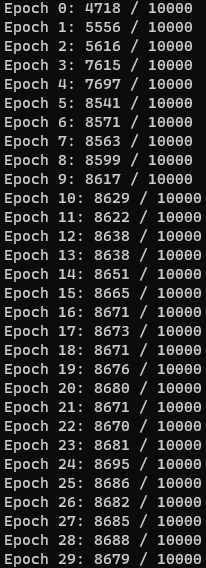
**30 Hidden Neurons:**

**A screen shot of a black screen

Description automatically generated**

Highest Epoch: 95.36% (23)

**100 Hidden Neurons:**

**A screen shot of a black screen

Description automatically generatedA screen shot of a computer screen

Description automatically generated**

The tutorial mentions that with 100 hidden neurons, there is a fair amount of variance between individual runs. I ran it 3 times to demonstrate this result.

Highest Epoch: 96.49% (23 Far Right)

**Code:**

**Main.py**

import mnist\_loader

import network

training\_data, validation\_data, test\_data = mnist\_loader.load\_data\_wrapper()

net = network.Network([784, 100, 10])

net.SGD(training\_data, 30, 10, 3.0, test\_data=test\_data)

**Mnist\_loader.py**

"""

mnist\_loader

~~~~~~~~~~~~

A library to load the MNIST image data.  For details of the data

structures that are returned, see the doc strings for ``load\_data``

and ``load\_data\_wrapper``.  In practice, ``load\_data\_wrapper`` is the

function usually called by our neural network code.

"""

#### Libraries

# Standard library

import pickle

import gzip

# Third-party libraries

import numpy as np

def load\_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.  This is a

    numpy ndarray with 50,000 entries.  Each entry is, in turn, a

    numpy ndarray with 784 values, representing the 28 \* 28 = 784

    pixels in a single MNIST image.

    The second entry in the ``training\_data`` tuple is a numpy ndarray

    containing 50,000 entries.  Those entries are just the digit

    values (0...9) for the corresponding images contained in the first

    entry of the tuple.

    The ``validation\_data`` and ``test\_data`` are similar, except

    each contains only 10,000 images.

    This is a nice data format, but for use in neural networks it's

    helpful to modify the format of the ``training\_data`` a little.

    That's done in the wrapper function ``load\_data\_wrapper()``, see

    below.

    """

    f = gzip.open('../data/mnist.pkl.gz', 'rb')

    training\_data, validation\_data, test\_data = pickle.load(f, encoding="latin1")

    f.close()

    return (training\_data, validation\_data, test\_data)

def load\_data\_wrapper():

    """Return a tuple containing ``(training\_data, validation\_data,

    test\_data)``. Based on ``load\_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)``.  ``x`` is a 784-dimensional numpy.ndarray

    containing the input image.  ``y`` is a 10-dimensional

    numpy.ndarray representing the unit vector corresponding to the

    correct digit for ``x``.

    ``validation\_data`` and ``test\_data`` are lists containing 10,000

    2-tuples ``(x, y)``.  In each case, ``x`` is a 784-dimensional

    numpy.ndarry containing the input image, and ``y`` is the

    corresponding classification, i.e., the digit values (integers)

    corresponding to ``x``.

    Obviously, this means we're using slightly different formats for

    the training data and the validation / test data.  These formats

    turn out to be the most convenient for use in our neural network

    code."""

    tr\_d, va\_d, te\_d = load\_data()

    training\_inputs = [np.reshape(x, (784, 1)) for x in tr\_d[0]]

    training\_results = [vectorized\_result(y) for y in tr\_d[1]]

    training\_data = list(zip(training\_inputs, training\_results))

    validation\_inputs = [np.reshape(x, (784, 1)) for x in va\_d[0]]

    validation\_data = list(zip(validation\_inputs, va\_d[1]))

    test\_inputs = [np.reshape(x, (784, 1)) for x in te\_d[0]]

    test\_data = list(zip(test\_inputs, te\_d[1]))

    return (training\_data, validation\_data, test\_data)

def vectorized\_result(j):

    """Return a 10-dimensional unit vector with a 1.0 in the jth

    position and zeroes elsewhere.  This is used to convert a digit

    (0...9) into a corresponding desired output from the neural

    network."""

    e = np.zeros((10, 1))

    e[j] = 1.0

    return e

**Network.py**

"""

network.py

~~~~~~~~~~

A module to implement the stochastic gradient descent learning

algorithm for a feedforward neural network.  Gradients are calculated

using backpropagation.  Note that I have focused on making the code

simple, easily readable, and easily modifiable.  It is not optimized,

and omits many desirable features.

"""

#### Libraries

# Standard library

import random

# Third-party libraries

import numpy as np

import mnist\_loader

# import network

class Network(object):

    def \_\_init\_\_(self, sizes):

        """The list ``sizes`` contains the number of neurons in the

        respective layers of the network.  For example, if the list

        was [2, 3, 1] then it would be a three-layer network, with the

        first layer containing 2 neurons, the second layer 3 neurons,

        and the third layer 1 neuron.  The biases and weights for the

        network are initialized randomly, using a Gaussian

        distribution with mean 0, and variance 1.  Note that the first

        layer is assumed to be an input layer, and by convention we

        won't set any biases for those neurons, since biases are only

        ever used in computing the outputs from later layers."""

        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:])]

    def feedforward(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

    def SGD(self, training\_data, epochs, mini\_batch\_size, eta,

            test\_data=None):

        """Train the neural network using mini-batch stochastic

        gradient descent.  The ``training\_data`` is a list of tuples

        ``(x, y)`` representing the training inputs and the desired

        outputs.  The other non-optional parameters are

        self-explanatory.  If ``test\_data`` is provided then the

        network will be evaluated against the test data after each

        epoch, and partial progress printed out.  This is useful for

        tracking progress, but slows things down substantially."""

        if test\_data: n\_test = len(test\_data)

        n = len(training\_data)

        for j in range(epochs):

            random.shuffle(training\_data)

            mini\_batches = [

                training\_data[k:k+mini\_batch\_size]

                for k in range(0, n, mini\_batch\_size)]

            for mini\_batch in mini\_batches:

                self.update\_mini\_batch(mini\_batch, eta)

            if test\_data:

                print("Epoch {0}: {1} / {2}".format(

                    j, self.evaluate(test\_data), n\_test))

            else:

                print("Epoch {0} complete".format(j))

    def update\_mini\_batch(self, mini\_batch, eta):

        """Update the network's weights and biases by applying

        gradient descent using backpropagation to a single mini batch.

        The ``mini\_batch`` is a list of tuples ``(x, y)``, and ``eta``

        is the learning rate."""

        nabla\_b = [np.zeros(b.shape) for b in self.biases]

        nabla\_w = [np.zeros(w.shape) for w in self.weights]

        for x, y in mini\_batch:

            delta\_nabla\_b, delta\_nabla\_w = self.backprop(x, y)

            nabla\_b = [nb+dnb for nb, dnb in zip(nabla\_b, delta\_nabla\_b)]

            nabla\_w = [nw+dnw for nw, dnw in zip(nabla\_w, delta\_nabla\_w)]

        self.weights = [w-(eta/len(mini\_batch))\*nw

                        for w, nw in zip(self.weights, nabla\_w)]

        self.biases = [b-(eta/len(mini\_batch))\*nb

                       for b, nb in zip(self.biases, nabla\_b)]

    def backprop(self, x, y):

        """Return a tuple ``(nabla\_b, nabla\_w)`` representing the

        gradient for the cost function C\_x.  ``nabla\_b`` and

        ``nabla\_w`` are layer-by-layer lists of numpy arrays, similar

        to ``self.biases`` and ``self.weights``."""

        nabla\_b = [np.zeros(b.shape) for b in self.biases]

        nabla\_w = [np.zeros(w.shape) for w in self.weights]

        # feedforward

        activation = x

        activations = [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, activation)+b

            zs.append(z)

            activation = sigmoid(z)

            activations.append(activation)

        # backward pass

        delta = self.cost\_derivative(activations[-1], y) \* \

            sigmoid\_prime(zs[-1])

        nabla\_b[-1] = delta

        nabla\_w[-1] = np.dot(delta, activations[-2].transpose())

        # Note that the variable l in the loop below is used a little

        # differently to the notation in Chapter 2 of the book.  Here,

        # l = 1 means the last layer of neurons, l = 2 is the

        # second-last layer, and so on.  It's a renumbering of the

        # scheme in the book, used here to take advantage of the fact

        # that Python can use negative indices in lists.

        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

            nabla\_b[-l] = delta

            nabla\_w[-l] = np.dot(delta, activations[-l-1].transpose())

        return (nabla\_b, nabla\_w)

    def evaluate(self, test\_data):

        """Return the number of test inputs for which the neural

        network outputs the correct result. Note that the neural

        network's output is assumed to be the index of whichever

        neuron in the final layer has the highest activation."""

        test\_results = [(np.argmax(self.feedforward(x)), y)

                        for (x, y) in test\_data]

        return sum(int(x == y) for (x, y) in test\_results)

    def cost\_derivative(self, output\_activations, y):

        """Return the vector of partial derivatives \partial C\_x /

        \partial a for the output activations."""

        return (output\_activations-y)

#### Miscellaneous functions

def sigmoid(z):

    """The sigmoid function."""

    return 1.0/(1.0+np.exp(-z))

def sigmoid\_prime(z):

    """Derivative of the sigmoid function."""

    return sigmoid(z)\*(1-sigmoid(z))