Creating a Neural Network from Scratch

We will be using Python 3 along with NumPy (a linear algebra package) to create a neural network for handwritten digit recognition using the MNIST dataset.

Consider writing a computer program to recognize digits. While we may have some intuitions about the general shape of each symbol, the variation in handwriting samples makes the task very difficult to achieve algorithmically.

Handwriting recognition is a good tool to introduce neural networks, as it is a relatively straightforward problem for a neural net to solve, and does not require tremendous computational power.

The Perceptron

While the sigmoid neuron is the typical neuron model used in neural networks, it will help to first understand a simpler model -- the perceptron.

- A perceptron takes several **binary inputs** and produces a single binary output ((draw sample perceptron with x1, x2, x3))
- A perceptron uses a set of **weights** on the inputs to determine the values fed in to the perceptron
- A perceptron has a **threshold** that determines the minimum value that will cause the perceptron to output 1
- This threshold can also be expressed as a **bias**, which is the negative threshold

See the code below for an example of how a perceptron works

In [ ]:
```
def calculate_perceptron(inputs, weights, bias):
    total = 0

    for single_input, paired_weight in zip(inputs, weights):
        total += single_input * paired_weight

    if total + bias > 0: # if total > threshold
        return 1
    else:
        return 0
```

Sigmoid Neurons

We want to devise a system where a small change in a weight will only cause a small corresponding change in the output. Using a perceptron, a small change in the input can potentially cause the output of a perceptron to flip from 0 to 1 or vice versa.

Just like the perceptron, the sigmoid neuron has **inputs**, **weights**, and an overall **bias**. But the output is a number between 0 and 1, expressed as σ(wx + b), where σ is called the sigmoid function, and is defined as follows:

```
σ(z) = 1 / (1 + e^-z)
```

Although we won't go in to detail here, the sigmoid function is helpful in that it is relatively easy to calculate the derivative which is needed for training the model. In fact, your activation function **must** be differentiable in order to perform gradient descent.

See an example of a sigmoid neuron below

In [ ]:
```
import math

def sigmoid_function(weighted_sum):
    return 1 / (1 + exp(-weighted_sum)) # exp(x) performs e^x

def calculate_sigmoid_neuron(inputs, weights, bias):
    total = 0

    for single_input, paired_weight in zip(inputs, weights):
        total += single_input * paired_weight

    return sigmoid_function(total + bias)
```

Neural Network Architecture

- The leftmost layer is known as the **input** layer
- The rightmost layer is known as the **output** layer
- The middle layers are called **hidden** layers
    - Hidden here simply means 'not an input or an output'

Such networks are sometimes called *multilayer perceptrons* or MLPs, despite being made up of sigmoid neurons, not perceptrons.

For our handwritten digit network using 28 x 28 images, how many input neurons are there? How many output neurons?

The design of input and output layers is relatively straightforward, but there is no simple way to determine the architecture of the hidden layers. There are many ways to make such decisions, but are outside the scope of this lesson

We will be using a network called a *feedforward neural network* where information is always fed forward.

There are also models which allow feedback loops, which is called a recurrent neural network.

Let's take a look at how we will generate a network below

In [5]:
```
#Note that in practice, we will implement this using a class object

#np.random.randn(m, n) creates an m x n numpy matrix of
#random numbers with mean 0 and variance 1

import numpy as np

def network(sizes):
    '''The list 'sizes' contains the number of neurons in each
    layer of the network. For example [2, 3, 1] would describe
    a network with 2 input neurons, 3 neurons in the hidden
    layer, and 1 output neuron'''
    num_layers = len(sizes)

    '''Biases are initialized randomly. The input layer does
    not need to include a bias so it will be omitted'''
    biases = [np.random.randn(x, 1) for x in sizes[1:]]

    print(biases)

    '''Describes a set of weights for the connections between
    layers. In the [2, 3, 1] example above, it would create a
    2x3 set of weights followed by a 3x1 set of weights.'''
    weights = [np.random.randn(x, y)
               for x, y in zip(sizes[:-1], sizes[1:])]

    print(weights)

# Example
network([2, 3, 1])
```

```
[array([[0.36694371],
       [1.46929057],
       [2.16794526]]), array([[-0.73983206]])]
[array([[ 0.90551488, -0.08059904,  1.12724755],
       [-0.06150455,  0.56386351, -0.43554712]]), array([[ 0.9257386 ],
       [ 0.64996471],
       [-0.96203509]])]
```

Learning with Gradient Descent

- Given x is a 784-dimensional vector (why is this?)
- Given y is a 10-dimensional vector
- Given a is the output of the network when x is input
- Given ||x|| is the length of vector x

We would like an algorithm which lets us find weights and biases so that the output from the network approximates y(x) for all training inputs x. To quantify how well we are doing this we define a *cost function*

$C(w, b) = (1/2n) \sum||y(x) - a||^2$

C thus represents the **mean squared error** (MSE)

As C(w, b) approaches 0, y(x) becomes approximately equal to output a for all training inputs x. So we want to find a set of weights and biases that minimize the cost C(w, b). We will do this using an algorithm called **gradient descent**.

We start by thinking of our function as a kind of a valley, and imagining a ball rolling down the slope of the valley. Eventually, the ball will roll to the bottom of the valley. By **computing the derivative** of the shape of the valley, we can **determine the slope** of the valley and therefore which way the ball should roll.

Intuition

Consider a model with 2 parameters, v1 and v2. C(v1, v2) can then be represented as a 3d surface just as f(x) can be represented as a 2d line.

Think of what happens when we move the ball a small amount $\Delta v1$ in the v1 direction, and a small amount $\Delta v2$ in the v2 direction. Calculus tells us that C changes as follows:

$\Delta C \approx (\partial C/\partial v1)\Delta v1 + (\partial C/\partial v2)\Delta v2$

We will make ΔC negative, so the ball rolls downhill, and denote the gradient vector by ∇C:

$\nabla C = (\partial C/\partial v1, \partial C/\partial v2)^T$

We can now rewrite the change ΔC in terms of ∇C and Δv:

$\Delta C = \nabla C \cdot \Delta v$

Finally, we can choose a Δv that makes ΔC negative:

$\Delta v = -\eta \nabla C$

where η is a small, positive parameter known as the learning rate

Substituting for the formula above, we now have:

$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$

This guarantees that ΔC is negative.

This gives us our update rule:

$v \rightarrow v' = v - \eta \nabla C$

We will use this update rule over and over, making moves and decreasing C until we reach a global minimum.

Backpropagation

Whereas the error at the output layer is clear, the error at the hidden layers seems mysterious because the training data do not say what value the hidden nodes should have. Fortunately, it turns out that we can **back-propagate** the error from the output layer to the hidden layers. The back-propagation process emerges directly from a derivation of the overall error gradient.

At the output layer, we can define the update rule as follows:

$\Delta k = Err(k) * sigmoid'(input(k))$

We can then propagate backwards, defining the update rule as:

$\Delta j = sigmoid'(input(k)) * \sum(weights(j,k)\ \Delta k)$

Note

We will be using a process called **stochastic gradient descent**, which speeds up the training process by computing the gradient for only a small sample of randomly chosen training inputs at each step. By averaging over many samples we can get a good estimate of the true gradient while speeding up training substantially.

This small sample will be referred to as a **mini batch**.

Implementing Our Network

If you haven't downloaded the MNIST data, you can get it like so:

In [1]:
```
!git clone https://github.com/mnielsen/neural-networks-and-deep-learning.git
```

```
'git' is not recognized as an internal or external command,
operable program or batch file.
```

Note: omit the '!' if not using Jupyter Notebook -or- Download the dataset here:
https://github.com/mnielsen/neural-networks-and-deep-learning/archive/master.zip
(https://github.com/mnielsen/neural-networks-and-deep-learning/archive/master.zip)

MNIST Loader

This simply loads the dataset and prepares it for feeding in to the network

In [7]:
```
import pickle
import gzip

import numpy as np

def load_data():
    f = gzip.open('data/mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = pickle.load(f, encoding='iso-8
859-1')
    f.close()
    return (training_data, validation_data, test_data)

def load_data_wrapper():
    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):
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

The Network

This code puts together everything we have worked on today and implements it as a class we can create instances of in order to generate networks

In [8]:
```
#https://pastebin.com/fJ5cHGt5

import random
import numpy as np

class Network(object):

    def __init__(self, sizes):
        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): # Returns the output of the network with a as th
e 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 = (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):
            delta = np.dot(self.weights[-l+1].transpose(), delta) * \
                sigmoid_prime(zs[-l])
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def evaluate(self, test_data):
        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)

# 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))
```

The Training

In [16]:
```
training_data, validation_data, test_data = \
    load_data_wrapper()

net = Network([784, 60, 10])

net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

```
Epoch 0: 8368 / 10000
Epoch 1: 9349 / 10000
Epoch 2: 9450 / 10000
Epoch 3: 9452 / 10000
Epoch 4: 9521 / 10000
Epoch 5: 9495 / 10000
Epoch 6: 9556 / 10000
Epoch 7: 9553 / 10000
Epoch 8: 9590 / 10000
Epoch 9: 9591 / 10000
Epoch 10: 9606 / 10000
Epoch 11: 9602 / 10000
Epoch 12: 9590 / 10000
Epoch 13: 9609 / 10000
Epoch 14: 9624 / 10000
Epoch 15: 9623 / 10000
Epoch 16: 9624 / 10000
Epoch 17: 9584 / 10000
Epoch 18: 9615 / 10000
Epoch 19: 9609 / 10000
Epoch 20: 9595 / 10000
Epoch 21: 9621 / 10000
Epoch 22: 9627 / 10000
Epoch 23: 9628 / 10000
Epoch 24: 9618 / 10000
Epoch 25: 9638 / 10000
Epoch 26: 9626 / 10000
Epoch 27: 9626 / 10000
Epoch 28: 9630 / 10000
Epoch 29: 9623 / 10000
```

In [ ]:
```
```