Neural Networks for Handwritten Character Recognition

Specification

The goal is to create a neural network (i.e. a multilayered perceptron) that is capable of recognizing handwritten digits. A collection of Training data is provided to train the network to recognize different people's handwriting. The network should then be able to generalize its behavior such that it can correctly classify handwritten characters that it has not seen before, i.e. classify the provided Test set.

Perceptron

The basic processing element of a neural network is the perceptron. It is composed of inputs: xb=1, x1..xd; connection weights: bias, w1..wd; and the Sum-of-Products function σ:

$$\sigma = \sum_{j=1}^{d} w_j x_j + bias$$

A multiple-input perceptron is represented as in the figure below:

The perceptron can be used as a regression or classification operator by using it to implement a linear discriminant (or threshold) function, as in:

$$y = \begin{cases} 1, \mathrm{i}f \sigma > 0 \\ 0, \mathrm{o}therwise \end{cases}$$

This assumes that the classes are linearly separable, representing linear functions. In cases where a non-linear function is required, one whose outputs are a smoothly differentiable function of its inputs, the *sigmoid* (squashing) function can be used:

$$y = sigmoid(\sigma) = \frac{1}{1 + e^{-\sigma}}$$

Multilevel Perceptrons

The multi-layer feedforward network includes an additional intermediate or hidden layer between the input and output layers (see figure below). It is used in conjunction with the sigmoid function to allow the network to approximate non-linear functions. There may be multiple hidden layers, but often there is simply one hidden layer with multiple perceptrons (the number of which may have to be experimentally determined). Generally, each perceptron is connected to every perceptron in the "next" layer. Note that the network may also include a bias input at the hidden layer.

Training the Network

Whether a single perceptron, or a multi-layer network, the training process consists of presenting examples to the system (as in all forms of Supervised learning) and adjusting the weights until the desired outcome is obtained. A process called gradient descent is used to implement the learning, converging iteratively via the form:

$$Update = LearningFactor \dot{(} DesiredOutput - ActualOutput \dot{)} Input$$

The notion of training error (distance from the target value t) is used to determine the direction of steepest descent. Training example error is commonly defined as:

$$E \equiv \frac{1}{2} \sum_{d \in D} (t_d - y_d)^2$$

The error E is computed after all input values have been fed forward. Then each weight is altered proportionally for each edge in order to minimize the error, as in:

$$\Delta w_i = -\eta \frac{\delta E}{\delta w_i}$$

where η is the learning rate (a convergence factor).

Error Backpropagation

Since there is no target value for perceptrons in the hidden layer, error is propagated backwards from the output layer and used for training:

For each training example do:

1. **Feedforward: Propagate the input forward through the network**
   a. Input the instance and calculate the output of each unit
2. **Backpropagate: Compute and propagate error backwards through the network**
   a. For each output unit j, calculate the error
   $E_j = (t_j - y_j) y_j (1 - y_j)$ *// note: derivative of sigmoid function*
   b. For each hidden unit i, calculate the error
   $E_i = h_i (1 - h_i) \sum_k w_{ik} E_k$
3. **Learn: Update each network weight proportionally**
   $w_j = w_j + \eta E_j z_j$ *// connecting hidden to output*
   $w_i = w_i + \eta E_i x_i$ *// connecting input to hidden*

Deciding on a Structure

Neural networks contain many parameters with counts easily getting out of hand with additional hidden layers. Other than the output, each node is associated with an input value, a weight, and an error. To keep these numbers orderly and persist them in a way that makes back propagation efforts more seamless, it is important to consider what the structure of the solution will look like. Starting simple, we can consider the perceptron that takes input and produces the needed output.

To get an output, we just need to calculate the dot product between two vectors where one represents the input values and the other the weights. Moving up a layer in complexity, we can see how to handle the situation where our outputs of the first layer become the inputs of the second.

With the additional layer, we are doing multiple dot products. To be concise, the weights getting us from the input layer to the hidden layer are packaged in a matrix. This structure extrapolates to any number of nodes in the hidden layer. With *n* being the number of inputs, three nodes produces an *n*x3 matrix, while 4 nodes yields one with dimensions *n*x4. This structure also allows for modular code. Once we calculate outputs from an input and the corresponding weights, we have new inputs for another round of calculations.

If all we needed was to do was **feed forward**, a function-based approach would suffice. We could give inputs and let it get transformed down to an output. With the need of a **back propagation** routine, we need these matrices to persist. To associate inputs and their weights, we can build a class that represents a step of the neural network. Shown below, the matrices representing the inputs and the weights to the left of the vertical line will belong to one object while those to the right below to another.

When we first build a layer, we'd need to know how many inputs we expect and the number of outputs needed. From those, we can create a randomly generated matrix of weights with the correct dimensions. This first pass have poor error metrics, but it will be a place to start. Each input layer wll have an aligning error vector. We can initialize this here as well. Note the addition of an input equal to 1 - this represents our bias node and aids in making our ultimate model more generalizable.

As for outputs, we can make the class more robust by giving options with respect to the output type. According to our textbook, *Introduction to Machine Learning*, if we are working with a multiclass classification problems, we should "use softmax to indicate the dependency between classes" (3rd ed., p. 288). Since we need the sigmoid for middle layers, we can set the function to accept a string argument so it knows which output to produce.

In [46]:
```
import numpy as np
import math

class NN_Step:
    '''
    Builds a layer consisting of inputs and weights. Expects a numpy array.
    '''
    def __init__(self, inputs, output_count):
        self.inputs = np.append([1], inputs)
        self.output_count = output_count
        self.weights = self.generate_weights(self.output_count)
        self.errors = np.zeros(inputs.shape)

    def generate_weights(self, _output_count):
        return np.random.rand(len(self.inputs), _output_count)

    def calculate_output(self, type = 'sigmoid'):
        if type == 'sigmoid':
            # need to first vectorize the custom function to apply to our new
 array
            calc_sigmoid_v = np.vectorize(self.calc_sigmoid)
            return calc_sigmoid_v(self.inputs @ self.weights)
        elif type == "softmax":
            inputs_dot_weights = self.inputs @ self.weights
            calc_softmax_v = np.vectorize(self.calc_softmax, excluded = ['all_
values'])
            return calc_softmax_v(inputs_dot_weights, all_values = inputs_dot_
weights)

    def calc_sigmoid(self, value):
        return 1 / (1 + math.exp(-value))

    def calc_softmax(self, value, all_values):
        all_values_exp_sum = np.sum(np.exp(all_values))
        value_exp = np.exp(value)
        return(value_exp / all_values_exp_sum)
```

Single Perceptron

With a base case, we just need an input layer and 1 output.

In [12]:
```
my_inputs = np.array([1, 2, 3, 4, 5])

perceptron = NN_Step(my_inputs, 1)

perceptron.calculate_output()
```

Out[12]:
```
array([1, 1, 2, 3, 4, 5])
```

Multi Layer Network

By feeding the output of one layer to the next, we can build a neural network with an arbitrary number of layers. In this case, we end up with a single value, but this can generalize to a multiple outputs by using the `softmax` functionality set up in our `NN_Step` class. In fact, we could generalize any classification problem by translating binary problems into a 2 node output.

In [51]:
```
layer_1 = NN_Step(my_inputs, 3)
layer_1_output = layer_1.calculate_output()

layer_1_output
```

Out[51]:
```
array([0.99995737, 0.99980679, 0.9512608 ])
```

In [53]:
```
layer_2 = NN_Step(layer_1_output, 1)
layer_2.calculate_output()
```

Out[53]:
```
array([0.68872041])
```

Associating Layers into a Neural Network

With our `NN_Step` building block set, we can add another class level that encapsulates the entire network. For our case, we can build a structure that has a single hidden layer by passing the initial `NN_Step` and the final output nodes expected. This class will be handling the back propagation with its own methods and references, so we will also initialize it with the learning rate. The class also figures out which output function to use based on the number of final output nodes expected. In scenarios where our output is more than 1, it is useful to convert the array scores to an array of 0s and a 1. To maintain the scores, we have a seperate attribute that stores the simplified index, `classed` . Note the alignment in the printed `output` and `classed` attributes.

In [55]:
```
class NN_One_Hidden:
    def __init__(self, first_step, output_count, learning_rate):
        self.first_step = first_step
        self.output_count = output_count
        self.hidden_step = NN_Step(self.first_step.calculate_output(), self.ou
tput_count)
        if output_count < 2:
            self.output = self.hidden_step.calculate_output(type = 'sigmoid')
        else:
            self.output = self.hidden_step.calculate_output(type = 'softmax')
        self.classed = self.classify(self.output)
        self.system_error = 0

    def classify(self, _array):
        # grab max output node to classify a record
        bool_array = _array == np.max(_array)
        return bool_array.astype(int)
```

In [56]:
```
my_nn = NN_One_Hidden(layer_1, 10, .1)
```

In [59]:
```
print(my_nn.output)

print(my_nn.classed)
```

```
[0.03318158 0.09488264 0.02358347 0.03762676 0.05899777 0.29821783
 0.11926546 0.08133112 0.15172745 0.10118592]
[0 0 0 0 0 1 0 0 0 0]
```

Slight Detour for Data Processing

Now that we have the main feed forward infrastructure figured out, we can layer in functionality for our back propagation tasks. However, this type of use is better explored with an actual dataset, so we need to incorporate our data! For development purposes, we can use the `fishing` data set. It contains various categorical metrics on a day and whether or not that day was good for fishing.

In [62]:
```
import pandas as pd

fishing = pd.read_csv("fishingNN.data", header = None, names = ["wind", "water
_temp", "air_temp", "forecast", "target"])
# removing a test record
fishing = fishing.loc[0:13,:]

fishing
```

Out[62]:

	wind	water_temp	air_temp	forecast	target
0	Strong	Warm	Warm	Sunny	Yes
1	Weak	Warm	Warm	Sunny	No
2	Strong	Warm	Warm	Cloudy	Yes
3	Strong	Moderate	Warm	Rainy	Yes
4	Strong	Cold	Cool	Rainy	No
5	Weak	Cold	Cool	Rainy	No
6	Weak	Cold	Cool	Sunny	No
7	Strong	Moderate	Warm	Sunny	Yes
8	Strong	Cold	Cool	Sunny	Yes
9	Strong	Moderate	Cool	Rainy	No
10	Weak	Moderate	Cool	Sunny	Yes
11	Weak	Moderate	Warm	Sunny	Yes
12	Strong	Warm	Cool	Sunny	Yes
13	Weak	Moderate	Warm	Rainy	No

In order for this data to function in our neural network, we need to convert our categories into a numeric equivalent. First, we can map each unique entry to an integer. Using `np.unique` with the `return_inverse` argument, we are able to retrieve both the unique array of entries as well as the integer-converted representation as a tuple. It is useful to store both of these if we need to convert back in the future.

In [63]:
```
# return_inverse argument gives an int representation of a string array
# https://stackoverflow.com/questions/3172509/numpy-convert-categorical-string-arrays-to-an-integer-array
np.unique(fishing.wind, return_inverse = True)
```

Out[63]:
```
(array(['Strong', 'Weak'], dtype=object),
 array([0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1], dtype=int64))
```

After converting the strings to integers, we apply a normalization method to get the variables on the same scale for better comparisons. By using the following formula, we can get each column to fit in the range [0, 1].

$$x_{i,normalized} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

We can package these two steps into a function to get any future data ready for the neural network.

In [64]:
```
def preprocess_training_data(df, target):
    '''
    converts all columns based on unique values.
    returns: unique array per column, converted df.
    '''
    col_names = df.columns
    uniques = []
    encoded = []

    for col in df.columns:
        unique_vals, encoded_col = np.unique(df[col], return_inverse = True)
        uniques.append(unique_vals)
        encoded.append(encoded_col)

    encoded_df = pd.DataFrame(np.stack(encoded, axis = 1), columns = col_names
)

    target_vectors = encoded_df[target]

    return uniques, normalize_discrete(encoded_df)

def normalize_discrete(df):
    '''
    Converts discrete numeric columns to be contains in the 0 - 1 range
    '''
    for col in df.columns:
        col_min = np.min(df[col])
        col_max = np.max(df[col])
        df[col] = df[col].apply(lambda x: (x - col_min) / (col_max - col_min))

    return df
```

In [66]:
```
preprocess_training_data(fishing, 'target')
```

Out[66]:
```
([array(['Strong', 'Weak'], dtype=object),
  array(['Cold', 'Moderate', 'Warm'], dtype=object),
  array(['Cool', 'Warm'], dtype=object),
  array(['Cloudy', 'Rainy', 'Sunny'], dtype=object),
  array(['No', 'Yes'], dtype=object)],
     wind  water_temp  air_temp  forecast  target
 0    0.0         1.0       1.0       1.0     1.0
 1    1.0         1.0       1.0       1.0     0.0
 2    0.0         1.0       1.0       0.0     1.0
 3    0.0         0.5       1.0       0.5     1.0
 4    0.0         0.0       0.0       0.5     0.0
 5    1.0         0.0       0.0       0.5     0.0
 6    1.0         0.0       0.0       1.0     0.0
 7    0.0         0.5       1.0       1.0     1.0
 8    0.0         0.0       0.0       1.0     1.0
 9    0.0         0.5       0.0       0.5     0.0
 10   1.0         0.5       0.0       1.0     1.0
 11   1.0         0.5       1.0       1.0     1.0
 12   0.0         1.0       0.0       1.0     1.0
 13   1.0         0.5       1.0       0.5     0.0)
```

In [68]:
```
# store
fishing_uniques, fishing_processed = preprocess_training_data(fishing, 'targe
t')
```

Getting Back to Associating Layers

Now that we have a dataset to test against, we can build out the back propagation methods. Each record is an input layer for the network. That means the inputs to our network will be different depending on the record we train on. This pushes us to include a way to update the inputs in an already initialized `NN_Step` object. Similarly, when we begin to calculate weight deltas, we will need to update that attribute.

In [324]:
```
class NN_Step:
    '''
    Builds a layer consisting of inputs and weights. Expects a numpy array.
    '''
    def __init__(self, inputs, output_count):
        self.inputs = np.append([1], inputs)
        self.output_count = output_count
        self.weights = self.generate_weights(self.output_count)
        self.errors = np.zeros(inputs.shape)

    def generate_weights(self, _output_count):
        return np.random.rand(len(self.inputs), _output_count)

    def calculate_output(self, type = 'sigmoid'):
        if type == 'sigmoid':
            # need to first vectorize the custom function to apply to our new
 array
            calc_sigmoid_v = np.vectorize(self.calc_sigmoid)
            return calc_sigmoid_v(self.inputs @ self.weights)
        elif type == "softmax":
            inputs_dot_weights = self.inputs @ self.weights
            calc_softmax_v = np.vectorize(self.calc_softmax, excluded = ['all_
values'])
            return calc_softmax_v(inputs_dot_weights, all_values = inputs_dot_
weights)

    def calc_sigmoid(self, value):
        return 1 / (1 + math.exp(-value))

    def calc_softmax(self, value, all_values):
        all_values_exp_sum = np.sum(np.exp(all_values))
        value_exp = np.exp(value)
        return(value_exp / all_values_exp_sum)

    # New methods to handle new records for training and weight updates
    def update_inputs(self, new_inputs):
        self.inputs = np.append([1], new_inputs)

    def update_weights(self, delta_weights):
        self.weights = self.weights + delta_weights
```

Figuring out the Training Steps

Before we can build a function that will iterate through our data and improve the network parameters, we need to replicate each step ourselves.

First, we need to get a record from the dataframe.

In [69]:
```
train = fishing_processed.iloc[0, 0:4]
target = fishing_processed.iloc[0, 4:]
```

Then, we build a neural network object using the first training set. We know our target variable can take 2 unique values, so we choose 3 nodes for our input layer. This can be added programmatically later.

In [70]:
```
fishing_nn = NN_One_Hidden(NN_Step(train, 3), 2, 0.1)
```

We check our feed forward result by looking at the `classed` attribute of our network. It's important to differentiate the binary values we are seeing. The array output below means that the sigmoid output for the second node was higher. In this scenario, the random weights achieved the correct classification. This means no error will be back propagated, but we will still build the functionality here.

In [71]:
```
print(fishing_nn.output)
print(fishing_nn.classed)
```

```
[0.75089763 0.24910237]
[1 0]
```

To assess our error, we need to convert the target variable into a vector similar to the network classed output. A `1` in target is equivalent to a vector `[0 1]`, while a `0` aligns with `[1 0]`. We can go back to the original set and compare each target variable to the unique values vector. Comparing this vector to the `classed` output from our neural network, we can see an element by element match. To calculate total error for the system, we sum the node-level formula for each potential option.

In [72]:
```
target_vector = fishing_uniques[4] == fishing.iloc[0, 4:][0]

target_vector
```

Out[72]:
```
array([False,  True])
```

From the textbook, we see the error function for a classification problem is:

$$E(W, V|X) = -\sum_i r_i * log(y_i)$$

with the associated update equations from gradient descent being:

$$\Delta v_{hi} = \eta(r_i - y_i)z_h$$
$$\Delta w_{jh} = \eta(\sum_i (r_i - y_i)v_{hi})z_h(1 - z_h)x_j$$

Below, we can calculate the total error by iterating over our target vector and the prediction values. We will take advantage of `numpy` 's vectorized functions.

In [86]:
```
total_error = -np.sum(target_vector * np.log(fishing_nn.output))

total_error
```

Out[86]:
```
1.3898913530658377
```

Drawing out these equations with their associated connections is helpful for finding a well-packaged matrix equation. We took some liberty to clarify notation - `z` is replaced with `h` when we refer to a hidden node value.

First we focus on going from the output nodes to the hidden layer. There are a lot of different equations, but many of them share the same components that can be extracted out.

We have most of the above matrix equation stored in our network class! We need to subtract our predicted vector from our target vector to finally get a net change matrix to modify our weights vector. We can look at the `weights` attribute of the `hidden_layer` object in our network to see what dimensions our update matrix should have.

```
In [87]: fishing_nn.hidden_step.weights
```

```
Out[87]: array([[0.93267039, 0.90202447],
                [0.50428784, 0.22403123],
                [0.73943689, 0.08024844],
                [0.86936111, 0.58660732]])
```

We can also peek at our inputs vector.

```
In [88]: fishing_nn.hidden_step.inputs
```

```
Out[88]: array([1.        , 0.72736328, 0.93795486, 0.88636165])
```

In order to properly multiply our vectors, we need to modify the structure slightly. Using matrix multiplication, we can insure each component is correctly aligned. To match our matrix dimensions, we use `np.transpose()`. Finally, we multiply our matrix by the scalar learning rate.

```
In [110]: .01 * (np.transpose(np.asmatrix(fishing_nn.hidden_step.inputs)) @ (np.asmatrix
(target_vector - fishing_nn.output)))
```

```
Out[110]: matrix([[-0.00750898,  0.00750898],
                  [-0.00546175,  0.00546175],
                  [-0.00704308,  0.00704308],
                  [-0.00665567,  0.00665567]])
```

Our dimensions align! Using a 2 output neural network is also proving to be a good sanity check. With 2 outputs, we typically do not need 2 nodes. We can get the same inference with a single node. The usefulness of softmax is more apparent when there are more than 2 classes. However, noticing the we increase the weight for the correct connection with the same magnitude that we decrease the incorrect connection!

Similarly for the weight updates from the hidden nodes to the inputs, we can leverage writing out each equation:

Like the output-hidden weights, we need the actual less our prediction, but we also incorporate the values and weights of the step ahead. We will need to make sure we persist these values while we update in some way. Again, let's show our weights matrix to see what we need our ultimate matrix to look like.

In [252]:
```
fishing_nn.first_step.weights
```

Out[252]:
```
array([[0.37970064, 0.37399591, 0.11710027],
       [0.57597061, 0.50834151, 0.8879831 ],
       [0.26926817, 0.72619893, 0.89585561],
       [0.58671324, 0.07355013, 0.0571626 ],
       [0.40027831, 0.27548051, 0.47452438]])
```

We then make the error vector.

In [295]:
```
error_vector = target_vector - fishing_nn.output

error_vector
```

Out[295]:
```
array([-0.29373077,  0.29373077])
```

Because each input node is multiplied by 1 minus itself, we can multiply them element-wise.

In [296]:
```
h1h = fishing_nn.hidden_step.inputs[1:] * (1 - fishing_nn.hidden_step.inputs[1
:])

h1h
```

Out[296]:
```
array([0.06425012, 0.07246892, 0.05960749])
```

We then matrix multiply the output vector by the `hidden_step` weights. Note we need to exclude the weights for the bias node - this is not linked to the input layer.

In [297]:
```
o_w_h1h = np.multiply(error_vector @ fishing_nn.hidden_step.weights[1:].T, h1h
)

o_w_h1h
```

Out[297]:
```
array([0.00591651, 0.00488742, 0.00307451])
```

In [299]:
```
np.asmatrix(fishing_nn.first_step.inputs).T
```

Out[299]:
```
matrix([[1.],
        [0.],
        [1.],
        [1.],
        [1.]])
```

Finally, we matrix multiply our inputs with the produced vector, producing a `5x3` matrix. These numbers represent the change in the input step's weights. As a sanity check, the second row is showing all `0` s. Remember this particular record had a 0 in that corresponding position and so produces no weight change.

In [300]:
```
.01 * (np.asmatrix(fishing_nn.first_step.inputs).T @ np.asmatrix(o_w_h1h))
```

Out[300]:
```
matrix([[5.91651207e-05, 4.88741885e-05, 3.07450684e-05],
        [0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
        [5.91651207e-05, 4.88741885e-05, 3.07450684e-05],
        [5.91651207e-05, 4.88741885e-05, 3.07450684e-05],
        [5.91651207e-05, 4.88741885e-05, 3.07450684e-05]])
```

Now we have our back propagating steps established. We now need to incorporate them into our network class.

Adding these steps as functions to our `NN_One_Hidden` class

To make these learnings useful, we can fold them into our network class. There is some additional data points we'd like to store with the class, so we need to add additional initialization variables. Note the method `back_propagate` which calls the two layers in order. Because the input layer method needs the weights in the hidden layer, it is important to calculate from left to right.

In [345]:
```
class NN_One_Hidden:
    def __init__(self, first_step, output_count, learning_rate, target, unique
_targets):
        self.first_step = first_step
        self.output_count = output_count
        self.hidden_step = NN_Step(self.first_step.calculate_output(), self.ou
tput_count)
        if output_count < 2:
            self.output = self.hidden_step.calculate_output(type = 'sigmoid')
        else:
            self.output = self.hidden_step.calculate_output(type = 'softmax')
        self.classed = self.classify(self.output)
        self.system_error = 0
        self.target = target
        self.unique_targets = unique_targets
        self.learning_rate = learning_rate

    def classify(self, _array):
        # grab max output node to classify a record
        bool_array = _array == np.max(_array)
        return bool_array.astype(int)

    def make_target_vector(self):
        self.target_vector = self.unique_targets == self.target

    def calculate_network_error(self):
        return -np.sum(np.multiply(self.target_vector, np.log(self.output)))

    def backprop_hidden(self):
        self.hidden_step.update_weights(self.learning_rate * \
                                        (np.transpose(np.asmatrix(self.hidden_
step.inputs)) @ (np.asmatrix(self.target_vector - self.output))))

    def backprop_input(self):
        error_vector = self.target_vector - self.output
        h1h = self.hidden_step.inputs[1:] * (1 - self.hidden_step.inputs[1:])
        o_w_h1h = np.multiply(error_vector @ self.hidden_step.weights[1:].T, h
1h)
        self.first_step.update_weights(.01 * (np.asmatrix(self.first_step.inpu
ts).T @ np.asmatrix(o_w_h1h)))

    # call back propagate steps in correct order to make sure correct weights
 are used
    def back_propagate(self):
        self.backprop_input()
        self.backprop_hidden()
```

In [327]:
```
fishing_nn = NN_One_Hidden(first_step = NN_Step(inputs = train, output_count =
3),
                           output_count = 2,
                           learning_rate = 0.1,
                           target = fishing.iloc[0, 4:][0],
                           unique_targets = fishing_uniques[4])
```

In [328]:
```
print(fishing_nn.first_step.weights)
print(fishing_nn.hidden_step.weights)
```

```
[[0.38151606 0.30564246 0.21582085]
 [0.27645909 0.04705696 0.4487509 ]
 [0.42939427 0.17185722 0.46110925]
 [0.63513151 0.0050518  0.19138874]
 [0.99663461 0.30568833 0.61072953]]
[[0.24438335 0.46183292]
 [0.6999629  0.02736593]
 [0.82071533 0.83750188]
 [0.07001012 0.87726069]]
```

In [329]:
```
fishing_nn.make_target_vector()
fishing_nn.back_propagate()

print(fishing_nn.first_step.weights)
print(fishing_nn.hidden_step.weights)
```

```
[[0.38135024 0.30565751 0.21620108]
 [0.27629328 0.04707201 0.44913113]
 [0.42922845 0.17187226 0.46148948]
 [0.63496569 0.00506685 0.19176897]
 [0.9964688  0.30570338 0.61110976]]
[[0.20189842 0.50431785]
 [0.66010581 0.06722301]
 [0.7910832  0.867134  ]
 [0.03292054 0.91435027]]
```

Training Over The Full Set

Now that we have feed forward and back propagation routines, we can go about training on a full data set. To accomplish, we need to loop over each training record and update the components of our neural network class. To monitor the model, we can store metadata like total error, training rounds, and epochs.

In [379]:
```
# dictionary to store model metrics
train_metrics = {}
index_counter = 0

# going through our entire dataset for 500 epochs
for j in range(500):
    for i in range(len(fishing_processed)):
        # split train and target data
        train = fishing_processed.iloc[i, 0:4]
        target = fishing.iloc[i, 4:][0]

        # if this is our first record, make a new NN object
        if (i == 0) & (j == 0):
            nn = NN_One_Hidden(first_step = NN_Step(inputs = train, output_cou
nt = 3),
                               output_count = 2,
                               learning_rate = 0.1,
                               target = target,
                               unique_targets = fishing_uniques[4])

            nn.make_target_vector()
            train_metrics.update({index_counter : {"epoch" : j, "trial" : i,
                                                   "total_error" : nn.calculat
e_network_error(),
                                                   "target_vector" : nn.target
_vector,
                                                   "output" : nn.output,
                                                   "inputs" : nn.first_step.in
puts,
                                                   "input_weights" : nn.first_
step.weights,
                                                   "hidden_weights" : nn.hidde
n_step.weights}
                                  })
            nn.back_propagate()
            index_counter = index_counter + 1
        else:
            # update train and target values
            nn.first_step.update_inputs(train)
            nn.target = target
            nn.make_target_vector()

            # recalculate first step outputs to incorporate new inputs and bac
kpropagated weights
            # pass these new outputs into the hidden layer's inputs
            nn.hidden_step.update_inputs(nn.first_step.calculate_output(type =
"sigmoid"))

            # recalculate the hidden layer output with the new input and weigh
ts
            nn.output = nn.hidden_step.calculate_output("softmax")

            train_metrics.update({index_counter : {"epoch" : j, "trial" : i,
                                                   "total_error" : nn.calculat
e_network_error(),
                                                   "target_vector" : nn.target
```

```
_vector,
                                                    "output" : nn.output,
                                                    "inputs" : nn.first_step.in
puts,
                                                    "input_weights" : nn.first_
step.weights,
                                                    "hidden_weights" : nn.hidde
n_step.weights}
                                    })
            nn.back_propagate()

            index_counter = index_counter + 1
```

After training, we run through the data set again using the weights the algorithm landed on. This allows us to get a performance metric. Note we loop through the dataset, but do not back propagate errors.

In [387]:
```
test = []

for i in range(len(fishing_processed)):
    # split train and target data
    train = fishing_processed.iloc[i, 0:4]
    target = fishing.iloc[i, 4:][0]

    # update train and target values
    nn.first_step.update_inputs(train)
    nn.target = target
    nn.make_target_vector()

    # recalculate first step outputs to incorporate new inputs and backpropaga
ted weights
    # pass these new outputs into the hidden layer's inputs
    nn.hidden_step.update_inputs(nn.first_step.calculate_output(type = "sigmoi
d"))

    # recalculate the hidden layer output with the new input and weights
    nn.output = nn.hidden_step.calculate_output("softmax")

    test.append(np.sum(nn.classify(nn.output) == nn.target_vector))
```

In [402]:
```
test = np.array(test)

np.mean(test > 0)
```

Out[402]: 0.7857142857142857

Using Our Code With The Hand Writing Dataset

Our focus dataset comes from the Optical Recognition of Handwritten Digits (https://archive.ics.uci.edu/ml/datasets/optical+recognition+of+handwritten+digits). Luckily the site hosts both a training and test set. From the documentation:

> We used preprocessing programs made available by NIST to extract normalized bitmaps of handwritten digits from a preprinted form. From a total of 43 people, 30 contributed to the training set and different 13 to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of 4x4 and the number of on pixels are counted in each block. This generates an input matrix of 8x8 where each element is an integer in the range 0..16. This reduces dimensionality and gives invariance to small distortions.
>
> There are 64 inputs + 1 class attribute. For Each Attribute, all input attributes are integers in the range 0..16. The last attribute is the class code 0..9. Training contains 3823 records, while Test has 1797.

Before feeding this to our algorithm, we need to preprocess.

In [481]:
```
hand_train = pd.read_csv("hand_written\optdigits.tra", header = None)
hand_test = pd.read_csv("hand_written\optdigits.tes", header = None)

hand_train.head()
```

Out[481]:

	0	1	2	3	4	5	6	7	8	9	...	55	56	57	58	59	60	61	62	63	64
0	0	1	6	15	12	1	0	0	0	7	...	0	0	0	6	14	7	1	0	0	0
1	0	0	10	16	6	0	0	0	0	7	...	0	0	0	10	16	15	3	0	0	0
2	0	0	8	15	16	13	0	0	0	1	...	0	0	0	9	14	0	0	0	0	7
3	0	0	0	3	11	16	0	0	0	0	...	0	0	0	0	1	15	2	0	0	4
4	0	0	5	14	4	0	0	0	0	0	...	0	0	0	4	12	14	7	0	0	6

5 rows × 65 columns

While trying to use preprocess the data, there were errors with respect to division by 0. This happens in columns where there is no variance in the inputs. Because these will not help the algorithm learn anything, we can exclude columns with variance = 0. Looking at both the training and test set, those are columns 0, 32, and 39.

In [433]:
```
train_variances = np.array(hand_train.var(axis = 0))
test_variances = np.array(hand_test.var(axis = 0))

print(np.where(test_variances == 0))
print(np.where(train_variances == 0))
```

```
(array([ 0, 32, 39], dtype=int64),)
(array([ 0, 39], dtype=int64),)
```

In [482]:
```
hand_train = hand_train.drop([0, 32, 39], axis = 1)
hand_test = hand_test.drop([0, 32, 39], axis = 1)

hand_uniques, hand_processed_tr = preprocess_training_data(hand_train, 64)
hand_uniques_test, hand_processed_te = preprocess_training_data(hand_test, 64)
```

In [440]:
```
hand_processed_tr.head()
```

Out[440]:

	1	2	3	4	5	6	7	8	9	10	...	55	56	57	5
0	0.125	0.3750	0.9375	0.7500	0.0625	0.0	0.0	0.0	0.466667	1.0000	...	0.0	0.0	0.0	0.375
1	0.000	0.6250	1.0000	0.3750	0.0000	0.0	0.0	0.0	0.466667	1.0000	...	0.0	0.0	0.0	0.625
2	0.000	0.5000	0.9375	1.0000	0.8125	0.0	0.0	0.0	0.066667	0.6875	...	0.0	0.0	0.0	0.562
3	0.000	0.0000	0.1875	0.6875	1.0000	0.0	0.0	0.0	0.000000	0.3125	...	0.0	0.0	0.0	0.000
4	0.000	0.3125	0.8750	0.2500	0.0000	0.0	0.0	0.0	0.000000	0.8125	...	0.0	0.0	0.0	0.250

5 rows × 62 columns

Now that the data is preprocessed, we can implement our embedded loop solution. Because we have 10 output nodes and 64 inputs, we will go for the average of the 2 and make a hidden layer with 37 nodes.

In [473]:
```
# dictionary to store model metrics
train_metrics = {}
index_counter = 0

# going through our entire dataset for 500 epochs
for j in range(500):
    for i in range(len(hand_processed_tr)):
        # split train and target data
        train = hand_processed_tr.loc[i, :63]
        target = hand_train.loc[i, 64]

        # if this is our first record, make a new NN object
        if (i == 0) & (j == 0):
            nn = NN_One_Hidden(first_step = NN_Step(inputs = train, output_cou
nt = 37),
                               output_count = 10,
                               learning_rate = 0.1,
                               target = target,
                               unique_targets = hand_uniques[-1])

            nn.make_target_vector()
            train_metrics.update({index_counter : {"epoch" : j, "trial" : i,
                                                   "total_error" : nn.calculat
e_network_error(),
                                                   "target_vector" : nn.target
_vector,
                                                   "output" : nn.output,
                                                   "inputs" : nn.first_step.in
puts,
                                                   "input_weights" : nn.first_
step.weights,
                                                   "hidden_weights" : nn.hidde
n_step.weights}
                                  })
            nn.back_propagate()
            index_counter = index_counter + 1
        else:
            # update train and target values
            nn.first_step.update_inputs(train)
            nn.target = target
            nn.make_target_vector()

            # recalculate first step outputs to incorporate new inputs and bac
kpropagated weights
            # pass these new outputs into the hidden layer's inputs
            nn.hidden_step.update_inputs(nn.first_step.calculate_output(type =
"sigmoid"))

            # recalculate the hidden layer output with the new input and weigh
ts
            nn.output = nn.hidden_step.calculate_output("softmax")

            train_metrics.update({index_counter : {"epoch" : j, "trial" : i,
                                                   "total_error" : nn.calculat
e_network_error(),
                                                   "target_vector" : nn.target
```

```
_vector,
                                                    "output" : nn.output,
                                                    "inputs" : nn.first_step.in
puts,
                                                    "input_weights" : nn.first_
step.weights,
                                                    "hidden_weights" : nn.hidde
n_step.weights}
                                    })
            nn.back_propagate()

            index_counter = index_counter + 1
```

In [489]:
```
train_metrics[1911499]
```

Out[489]:
```
{'epoch': 499,
 'trial': 3822,
 'total_error': 7.771561172376126e-15,
 'target_vector': array([False, False, False, False, False, False, False,  Tr
ue, False,
        False]),
 'output': matrix([[1.14337004e-23, 1.17792910e-15, 9.07167998e-19, 6.4956529
0e-15,
          6.12914888e-30, 7.13216791e-18, 3.82884493e-37, 1.00000000e+00,
          3.53437539e-18, 2.69999440e-16]]),
 'inputs': array([1.        , 0.        , 0.125     , 0.9375    , 1.        ,
       0.8125    , 0.0625    , 0.        , 0.        , 0.        ,
       0.1875    , 0.4375    , 0.625     , 1.        , 0.625     ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.6875    , 0.6875    , 0.        , 0.        ,
       0.        , 0.        , 0.125     , 0.5       , 0.9375    ,
       0.3125    , 0.        , 0.        , 0.        , 0.5625    ,
       1.        , 1.        , 0.57142857, 0.        , 0.        ,
       0.        , 0.125     , 1.        , 0.3125    , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.75      ,
       0.4375    , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.25      , 0.875     , 0.0625    , 0.        ,
       0.        , 0.        ]),
 'input_weights': matrix([[ 0.16483157,  0.69829445,  0.55633123, ..., -0.309
41984,
          0.57731489, -1.63660476],
        [ 0.16439025,  0.45194033,  0.88110001, ...,  1.57704803,
          0.32092523, -0.76458772],
        [ 0.89063022,  0.44900194,  0.9643662 , ..., -1.74624188,
          0.55968184, -0.44727303],
        ...,
        [ 0.73372815,  0.97956446,  0.28078426, ..., -1.87707273,
          0.03353477, -0.85385719],
        [ 0.72853235,  0.91674845,  0.86010429, ...,  1.82745132,
          0.80104936,  0.06845786],
        [ 0.88082177,  0.60360443,  0.12235725, ..., -6.19411157,
          0.39057376,  1.16890781]]),
 'hidden_weights': matrix([[-4.06161731e-01,  6.56104047e-01,  1.20286690e+0
0,
          6.91829964e-01, -2.31434086e-01,  1.05670826e+00,
          8.16861325e-01,  5.02754583e-02,  5.24182789e-01,
          4.16804067e-01],
        [-3.14221587e-01,  1.09070647e+00,  8.47802311e-01,
          1.32168858e+00, -5.83744462e-01,  8.51682940e-01,
          2.93390835e-01, -2.48996068e-01,  6.04028525e-01,
          5.43531183e-01],
        [-1.40731276e-01,  1.12105165e+00,  7.51772904e-01,
          9.42631612e-01, -1.04750179e-01,  1.01855025e+00,
          4.50890842e-01, -5.37950372e-01, -1.75356680e-01,
          3.85869460e-01],
        [ 9.51025773e-02,  9.24575347e-01,  1.11825685e+00,
          5.40576510e-01, -1.36461314e-02,  1.39580382e+00,
          4.60594708e-01,  8.22357370e-01,  2.15906509e-01,
         -3.33033353e-01],
        [-1.02342670e-01,  9.79056621e-01,  4.42110707e-01,
          2.06060154e-01, -4.32075088e-02,  1.03080250e+00,
          4.07459072e-01,  3.06756347e-01, -2.11680597e-01,
```

```
 -3.11962553e-01],
[ 4.40222809e-01,  7.70262552e-01,  6.63182549e-01,
  4.08282334e-01, -3.81769510e-01,  9.04870333e-01,
  1.80563199e-01,  5.42817895e-01,  1.56963898e-01,
  1.38666607e-01],
[-1.21199842e-01,  7.26213618e-01,  4.59493267e-01,
  1.07816570e+00, -4.17562238e-01,  1.29477978e+00,
  8.30028263e-01,  9.90216202e-02, -5.97672474e-02,
  2.66381307e-01],
[ 1.76177863e+01, -5.80153049e+00, -2.61688472e+01,
 -4.37187133e+00,  1.21651090e+01, -3.65270500e+00,
 -2.95364505e+01,  3.88785339e+01, -5.65239779e+00,
  1.07970425e+01],
[-3.79291564e-01,  1.20225487e+00,  1.22960399e+00,
  5.30637105e-01, -5.09957704e-01,  1.47748096e+00,
  9.58949999e-01,  4.84178755e-02, -1.89528676e-01,
 -2.87978740e-02],
[ 4.46289823e-01,  2.94227796e-01,  1.31937593e+00,
  5.70669960e-01, -2.75178368e-01,  1.10419050e+00,
  5.39945320e-01, -3.55163993e-01,  6.93603630e-02,
  1.24223536e-01],
[-1.91381859e+00,  8.24092427e+00, -2.40087383e+01,
  1.23303973e+01, -5.73043108e+00,  1.21766306e+01,
 -4.10787041e+00, -1.86749480e+01,  1.62551154e+00,
  2.38500241e+01],
[-3.31396502e-01,  5.52042514e-01,  8.85105741e-01,
  8.77539125e-01, -3.79665237e-01,  9.69603368e-01,
  6.79200613e-01, -7.70811649e-02,  6.83112865e-01,
 -3.45949292e-01],
[ 7.15632628e-01,  9.80234314e-01,  1.21736529e+00,
  2.17350612e-01, -5.47997883e-01,  1.59490032e+00,
  1.32080008e-01,  4.27557632e-01,  8.44506280e-01,
 -2.03204338e-01],
[-1.23304264e+01,  1.14089380e+00,  1.11717128e+01,
  8.57701143e+00, -1.01534901e+01, -1.82009744e+01,
 -1.46225438e+01,  1.53544301e+01,  1.47556206e+01,
  8.94934745e+00],
[-4.37026695e-01,  1.01946395e+00,  7.58029938e-01,
  8.42321453e-01, -5.73851995e-01,  1.34805999e+00,
  9.54272376e-01,  5.97398571e-01, -2.94243412e-01,
  1.51050586e-01],
[-4.30766442e-01,  1.02118169e+00,  6.67921221e-01,
  8.19972981e-01, -6.00457968e-01,  8.34036669e-01,
  4.69418530e-01,  5.35168043e-02,  5.84456642e-01,
 -4.78525491e-02],
[-3.10171939e-01,  5.84810540e-01,  8.11608298e-01,
  2.28586801e-01, -4.44334991e-01,  7.99437645e-01,
 -1.01574476e-01,  6.98781997e-01,  5.85834689e-01,
  4.51570270e-01],
[ 6.06695808e-01,  8.19076450e-01,  7.55886896e-01,
  7.97408677e-02, -2.35268931e-01,  1.57904303e+00,
  6.19954208e-01,  6.16230194e-01,  2.14842271e-01,
 -1.34681514e-01],
[-1.12573956e-01,  7.40751352e-01,  1.40867037e+00,
  5.27629091e-01,  3.15890243e-01,  1.12128134e+00,
  4.63224361e-01,  6.41371329e-01, -1.09772088e-01,
  1.88602854e-01],
```

```
[ 3.92295507e-01,  5.94480293e-01,  6.34336278e-01,
  1.06717955e+00, -4.17059272e-01,  1.52517521e+00,
  5.32869923e-01,  4.50550327e-01,  4.72360024e-01,
  4.80329270e-01],
[ 3.98116632e-01,  1.29633220e+00,  8.58808433e-01,
  8.81631496e-01,  1.72351769e-01,  1.47835757e+00,
  1.12201611e+00, -2.97666626e-01, -1.01326903e-01,
 -1.59903944e-01],
[ 3.02926354e-01,  1.17127935e+00,  1.25596094e+00,
  7.36616080e-01,  8.47350785e-02,  1.50898614e+00,
  6.95792660e-01, -5.85705531e-03, -1.17992568e-01,
  2.55319589e-01],
[ 3.49872807e-01,  1.58926281e+00,  6.35344415e-01,
  1.29231908e+00, -7.77561451e-01,  1.35518841e+00,
  4.01361494e-01, -1.04532607e-01,  4.14101842e-01,
 -2.77862562e-01],
[-7.49195278e-02,  1.22677845e+00,  8.16050166e-01,
  2.12846985e-01, -8.87702133e-02,  1.54242001e+00,
  1.09383426e+00,  6.71322397e-02,  3.84447848e-01,
 -3.05487916e-03],
[-2.56240210e-01,  1.33235129e+00,  9.76446529e-01,
  7.29235917e-01, -6.30762619e-02,  1.36220671e+00,
  3.60493866e-01,  8.56100669e-01,  6.90422563e-01,
  1.73497760e-01],
[-3.78478622e-01,  8.08542504e-01,  1.12238396e+00,
  8.83294454e-01, -2.28474858e-01,  8.76554994e-01,
  5.08257988e-01,  2.84217851e-01, -1.36069410e-01,
  7.03809899e-02],
[ 3.51631237e-01,  4.67096078e-01,  1.15400936e+00,
  9.84035834e-01, -1.26268397e-01,  1.61875839e+00,
  4.07829245e-01,  1.47330785e-01,  5.99239875e-02,
 -2.82217603e-01],
[-3.48706578e-01,  8.65256735e-01,  1.25342608e+00,
  5.68207914e-01, -5.57520677e-01,  1.67101144e+00,
  6.74164415e-01,  2.51545429e-01, -7.19598036e-02,
  3.44957630e-01],
[-4.87549861e-02,  6.87684496e-01,  9.32184180e-01,
  5.60077006e-01,  9.84524423e-02,  8.55085474e-01,
  5.84860024e-01,  4.13366704e-01, -1.78064862e-01,
 -1.78200891e-02],
[-2.34337036e-01,  9.82044690e-01,  6.14733119e-01,
  3.63439630e-01, -2.00095889e-01,  1.66137174e+00,
  8.20993596e-02,  5.11719992e-01,  7.69334830e-01,
  2.06831163e-01],
[ 5.22487297e-02,  1.38380434e+00,  1.17061951e+00,
  8.70480499e-01, -1.72314044e-01,  1.34852544e+00,
  1.27482922e+00, -1.25499386e-01,  6.62323089e-01,
  5.44598813e-01],
[ 5.11156936e-01,  3.15467978e-01,  1.33028429e+00,
  1.18113384e+00, -5.88221175e-01,  6.24276677e-01,
  2.14097601e-01,  5.04805667e-01,  3.74292823e-01,
  2.07378982e-02],
[ 5.13033400e-01,  9.44857939e-01,  1.05917443e+00,
  6.76327242e-01,  2.13400029e-01,  1.64280082e+00,
  8.95660268e-01,  6.69635508e-01,  5.60953453e-01,
  4.23799308e-01],
[-3.92739690e-01,  1.00883832e+00,  1.07093045e+00,
```

```
         5.02529576e-01, -4.08092341e-01,  1.53496681e+00,
         3.73785409e-01,  2.78302204e-01,  5.21197072e-01,
         3.71264982e-01],
        [-2.00697514e-01,  9.30638305e-01,  5.65909740e-01,
         3.56563342e-01, -1.04749358e-01,  1.10676677e+00,
         1.92076844e-01,  4.69986383e-02,  5.55423820e-02,
        -2.91144546e-01],
        [ 1.86011663e+01, -1.13504549e+01,  1.77917195e+01,
         9.72032279e+00, -2.24577453e+01,  1.15713293e+00,
        -5.24178373e+00, -1.38251436e+01,  9.59210752e+00,
         3.08764533e+00],
        [-1.47512566e-01,  7.50938131e-01,  8.49654021e-01,
         1.07384501e-01, -2.39789930e-01,  1.21569941e+00,
         8.60470916e-01,  4.53383055e-01,  8.07882705e-01,
         8.14187438e-02],
        [ 1.25510949e+01,  1.09041847e+01, -1.94712592e+01,
        -6.86031886e+01,  6.10658725e+01, -2.47206613e+01,
         3.05875885e+01, -8.13024307e+00,  1.31821263e+01,
        -2.93944135e+00]])}
```

We can see the logs for each trial. It includes `total_error`, the `target_vector`, and the `inputs` and two `weights` associated with the layers. This allows us to go back and use weights at any given point. And now to use our calculated weights on the test set! Note that an incorrect prediction will have a vector sum of 8 - the correct one is classified as a 0 while one incorrect one is classified as a 1. If the test vector equals 10, then all were correctly identified.

In [496]:
```
def run_testing(nn):
    test = []

    for i in range(len(hand_processed_te)):
        # split train and target data
        train = hand_processed_te.loc[i, :63]
        target = hand_test.loc[i, 64]

    # update train and target values
        nn.first_step.update_inputs(train)
        nn.target = target
        nn.make_target_vector()

        # recalculate first step outputs to incorporate new inputs and backpro
pagated weights
        # pass these new outputs into the hidden layer's inputs
        nn.hidden_step.update_inputs(nn.first_step.calculate_output(type = "si
gmoid"))

        # recalculate the hidden layer output with the new input and weights
        nn.output = nn.hidden_step.calculate_output("softmax")

        test.append(np.sum(nn.classify(nn.output) == nn.target_vector))

    return np.array(test)
```

In [497]:
```
test = run_testing(nn)

np.mean(test == 10)
```

Out[497]:
```
0.9048414023372288
```

Even without implementing an early stopping criteria of some kind, we were able to get over 90% accuracy. It's likely that we began to overfit the training set with the number of trials we ran. We can use our logs to find past trials and use those weights. Let test on an earlier set of weights to see if our accuracy improves. This is by no means an exhaustive method, but it is important to show the point. We picked another trial and adopted those weights into a new neural network configuration. By trying different values, we were able to squeeze out about a ~.5% improvement in accuracy on the test set.

In [519]:
```
nn_tester = nn

trial_num = 1909499
new_input_wts = train_metrics[trial_num]['input_weights']
new_hidden_wts = train_metrics[trial_num]['hidden_weights']

nn_tester.first_step.weights = new_input_wts
nn_tester.hidden_step.weights = new_hidden_wts

test = run_testing(nn_tester)

np.mean(test == 10)
```

Out[519]:
```
0.9115191986644408
```

Discussion

Neural Networks can get cumbersome very quickly if you do not go into the problem with a design plan in place. With just a single hidden layer, we needed quite a few lines of code. This structure gets more complicated quickly once we incorporate additional layers and methods to prevent overfitting. Personally, I've got a much deeper appreciation for deep learning libraries such as TensorFlow and PyTorch.

For improvements, I think a validation set would have been most useful. We would split off part of the training data and then test it every epoch or so. This would allow us to store an accuracy measure and implement an early stopping criteria. Once we see that accuracy isn't improving by much or is degrading, we can stop and then use the weights as they stand with the test set.

Of course, the actual training methods could also be incorporated into the class methods. The project took quite some time and so my will to implement a class-first architecture began to wane. I switched to functional programming, but I think it would not be too complicated to introduce the additional lines of code into the neural network class.