How To Write Neural Networks From Scratch: Manual

By Ashwin Jeyaseelan

ENGL393 Spring 2017

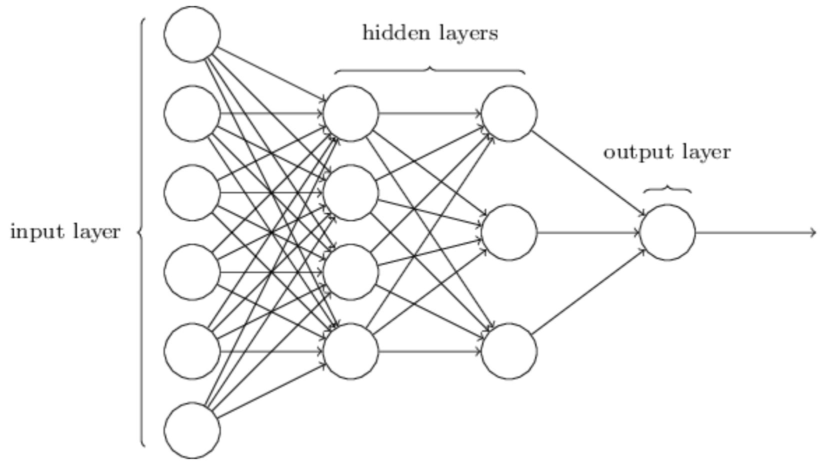
How To Write Neural Networks From Scratch: Manual

Table of Contents:

Introduction

Before building the neural network, it will be helpful to provide a quick review about the overall picture of what we're building, so you can refer back to this page any time you're losing sight of the overall goal. Neural networks are used to predict output based on input. Hence, they are adaptable for a variety of tasks that involve predictions, such as predicting sales of business plans, or predicting the correct/best decision to take in a situation. Think of neural networks as models that learn from examples. When you initially feed data into a neural network, it doesn't know anything, so it spits out random output. Hence, you need to train the network after it makes a prediction. This is done by comparing the output of the neural network to the actual output or answer of the input that you provided. Therefore, in order to adequately build a robust neural network, you'll need lots of examples with inputs and labeled answers. After training the network with many examples and improving its predictions, it becomes better at making predictions.

In short, a neural network looks like network of units that are similar to the concept of neurons in the brain. The units or nodes of a network mimic biological neurons by "taking input data and performing simple operations on the data, selectively passing the results on to other neurons" (Leverington). Each layer of the neural network has units, which are inspired by neurons. Each of the units in the input layer compute their own outputs, based on the input data you feed it. Then, those units pass their output onto the units in the next layer and so on until the last layer releases its final output. But unlike neurons, each unit has a corresponding weight for each incoming input, and uses an activation function on the overall sum of inputs to determine an output. This output is then sent to the units in the next layer, which do the same process.



Remember that our goal is to fine tune the weights of incoming values into each unit so that the final output is more accurate.

The main steps of a neural network that we will focus on are summarized below, as taught by Dr. Perlis in CMSC 421:

Step 1: Choose: training pairs, architecture, learning rate, initial values for weights, activation function, batch size, cost/error functions
Step 2: Shuffle training pairs.
Step 3: Run neural net on each pair in the first minibatch or each pair if your batch is all the samples. (Also known as forward propagation)
Step 4: Calculate cost results
Step 5: Calculate gradient of the cost with respect to weight
Step 6: Update the weights
Step 7: Repeat steps 3-7 on the next minibatch (called 1 epoch)
Step 8: Repeat steps 2-7 for as many epochs as required.

I will approach these steps in an order that makes sense when programming the network, since coding requires a slightly modified order in order to get the neural network to work properly.

Programming the Network: How to create the Architecture

The first step towards building a neural network is to decide on the architecture you will utilize. To simplify this process, we will write a basic neural network template that you can build upon for different tasks.

For those of who you aren’t in/haven’t taken CMSC 421, I will be programming in Python 3. Python has easy syntax and frameworks that make it easy to program the math behind neural networks. (The # symbol represents a comment in python!)

In a neural network, the first layer of units is used to accept the inputs, which will be data that we feed into the network. The layers between the input and output layer are known as hidden layers.

Start off with this template:

```
class NeuralNetwork:
    # initialize the network-
    def __init__(self, units_per_layer_list, iterations, learn_rate):
```

Now that we have a basic template, we need to figure how to program the units so that it will be easy for users to modify the network. Chainer, the framework that you may have used in CMSC 421, already handles this. All the math and unit architecture has already been implemented, and it’s up to the user to build the network like legos using their already programmed API. But in order to get a more intuitive understanding of neural networks, I will provide solutions that can be implemented in scratch. Once you have a basic understanding of the main concepts in neural networks, you should be able to use any neural network framework.

You might be wondering why I have a parameter called 'units_per_layer_list' in the neural network initialize function, instead of separate parameters for units and layers. I did this because an easy way to input the number of layers and units per layer into a neural network is to use a python list, where the length of the list represents the total number of layers, and each value represents the number of units in its layer. For example, the list [3,2,1] represents a neural network with 3 input units in the first layer, 2 hidden units in the second layer, and 1 output unit in the last layer. This will make it easy for you to make changes to your network with input parameters instead of having to rewrite the code in the template every time. At each step I'll be presenting my own design choices for the network. Ultimately the design choices are up to you, but it's important to think of how write the network so that it will be easy to build upon.

How Units Work

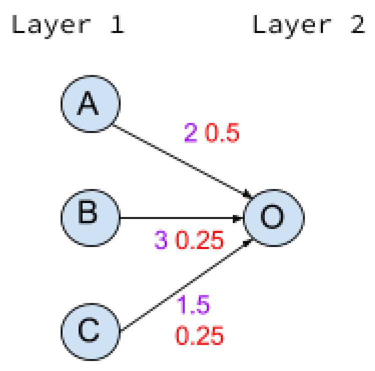
One of the major ideas of a neural network is that each unit in the neural network has corresponding weights for each of its incoming values, including a bias weight for itself. The output of the unit is determined by applying the weighted sum of its inputs as input into a function (Microsoft Build). Therefore, the output of a unit can be mathematically represented as:

activation_function($\sum\limits_{i=1}^{N} a_i w_i + b$)

Where N represents the number of incoming values into a unit, a_i represents the incoming output from the ith unit in the previous layer, w_i represents the weight corresponding to that output value, and b represents the bias node. From CMSC 421, you might remember that each layer has a bias node, whose output value is always 1. But the weight of the bias node is treated just like the weights of the other unit outputs in the fact that its weight will also be updated during training. In CMSC 421, Dr. Perlis stated that the purpose of the bias unit is to modify the threshold instead of solely modifying the architecture. As Texas Tech Professor David Leverington explains, the bias unit "increases the capacity of the network to solve problems by allowing the hyperplanes that separate individual classes to be offset for superior positioning." What David means is that the bias units allow you to position the output function of a neural network better. For example, let the predicted output of your neural network be modeled as the function $f(x) = wx$ that is simply a straight line that goes through the origin in the xy plane. Let the actual output function for making correct predictions be b units above the predicted function. Instead of training and updating the slope or weight of the network (w) to match the actual output function, you could simply move the function up b units by adding a constant to it. Therefore the correct function would be $f(x) = wx + b$, where b in this case represents the bias unit. In this example, the bias node allows the network to predict the correct output function by repositioning it instead of depending on the weights, which wouldn't help since the weights only modify the slope of the predicted function.

A neural network utilizes randomly initialized weights for each unit. Yet why would we want to use random weights? Remember that the whole idea of a neural network is to learn to predict output on its own. At first, the neural network doesn't know the correct weights. These weights are knobs that the network itself will update after comparing its predicted output to the actual output each time. Therefore, we want to easily represent the weights of each unit with a data structure that allows the weights to be changed. This can be accomplished using vectors. For instance, let the vector [1,2,3] represent the incoming values into a unit, and the vector

[0.7,0.2,0.1] represent its corresponding weights. Since the output of the unit before applying the activation function is the dot product of the values and its weights, this can be written using vector multiplication as vector [1,2,3] multiplied the transpose of [0.7,0.2,0.1]. To include bias units, add an extra 1 value into the value vector, and a random weight at the end of weight vector. In fact, this is the same as the sum of the bias weight and the dot product between [1,2,3] and [0.7,0.2,0.1]. Since a layer comprises of multiple units, you could stack each unit's input weights into a matrix and do the same for the bias weights of each input.



<u>Understanding the Matrix Representation of Neural Networks</u>

To demonstrate how you can translate the representation of a neural network into a matrix, I've shown and drawn an example below.

The picture on the left is a neural network with two layers. Assume that input data by a person has already been fed into units of layer 1 and that the units already calculated their outputs. The first layer (Layer 1), has three units labeled A, B, and C. The second layer (Layer 2) only has one unit, labeled O. The arrows show that A, B, and C are connected to unit O, meaning that the outputs of layer 1 units are passed as input to unit O. The purple numbers represent the outputs of the units in layer 1. Unit A has an output of 2, B has an output of 3, and C has an output of 1.5. The red numbers represent the weights of those outputs. This means that for O to make its predicted output, it values incoming information units differently. Since node A has the highest weight of 0.5, its output is valued the highest by unit O. So, the weighted sum of these inputs is 0.5*2 + 0.25* 3 +

0.25 * 1.5. Before adding the bias to the weighted sum and then applying an activation function of our choice, notice the familiarity of the expression. It looks similar to to the dot product. Remember that the dot product between two vectors [a,b,c] and [x,y,z] is ax + by + cz. Therefore, the weighted sum can be represented as the dot product between a vector of the weights and a vector of the corresponding outputs of units in layer 1. So the weight vector is [0.5, 0.25, 0.25] and its matching outputs are [2,3,1.5]. These two vectors are sufficient to represent all the required information for unit O. Remember that a unit computes predictions by applying an activation function to the dot product of incoming values and their corresponding weights. What if we had more than one unit in Layer 2? Then, each of the units in Layer 2 will have both a weight vector and a vector of incoming values (Assuming that all the outputs of the units in layer 1 also get passed into all the other units in Layer 2). To represent the dot product or weighted sum for all the units in layer 2, organize a matrix of weight vectors for all the units in layer 2, and a vector of the outputs of units in layer 1. The weighted sums for all the units in Layer 2 can be represented as the transpose of the weight vector multiplied by layer 1 output vector [2,3,1.5]. We take the transpose of the weight vector so that it can be multiplied properly by the layer 1 output vector. In matrix multiplication, two matrices A and B can only be multiplied by each other if the number of columns in matrix A matches the number of rows in matrix B. Since matrix multiplication involves dot products between each row in matrix A and each column in matrix B, this rule can be extended to a neural network. Vector [2,3,1.5] is a 1 row by 3 column matrix. If layer 2 has n units including unit O, then it's size of the weight vectors for all the units will be n rows by 3 columns. The transpose of the weight matrix will change the dimension to 3 columns by 1 row. Then it can be multiplied by the vector [2,3,1.5].

Alternatively, you could preserve the dimension of the weight matrix and multiply it by the transpose of the vector [2,3,1.5].

Coding the Network

In python, the weighted sum of inputs into units in the same layer can be written as

```
numpy.dot(input, w.T) + b.
```

w.T represents the transpose matrix of all the weights for the units in the same layer. Input is a vector of all the outputs of units from the previous layer. Numpy.dot computes the dot product (or matrix multiplication) between the input and the transpose weight matrix. And b is the bias.

Before we proceed with the computations of a neural network, let's finish the basic architecture. Just as we discussed before, it's helpful to have a matrix of weights for all units in the same layer. Each row of the matrix represents the weight vector of incoming values belonging to a single unit. The number of columns in the weight vector represents the number of incoming values into a unit. Let's represent the weights as a list, where each index has a weight matrix for all the units in the same layer. In python, all the weights can be written as:

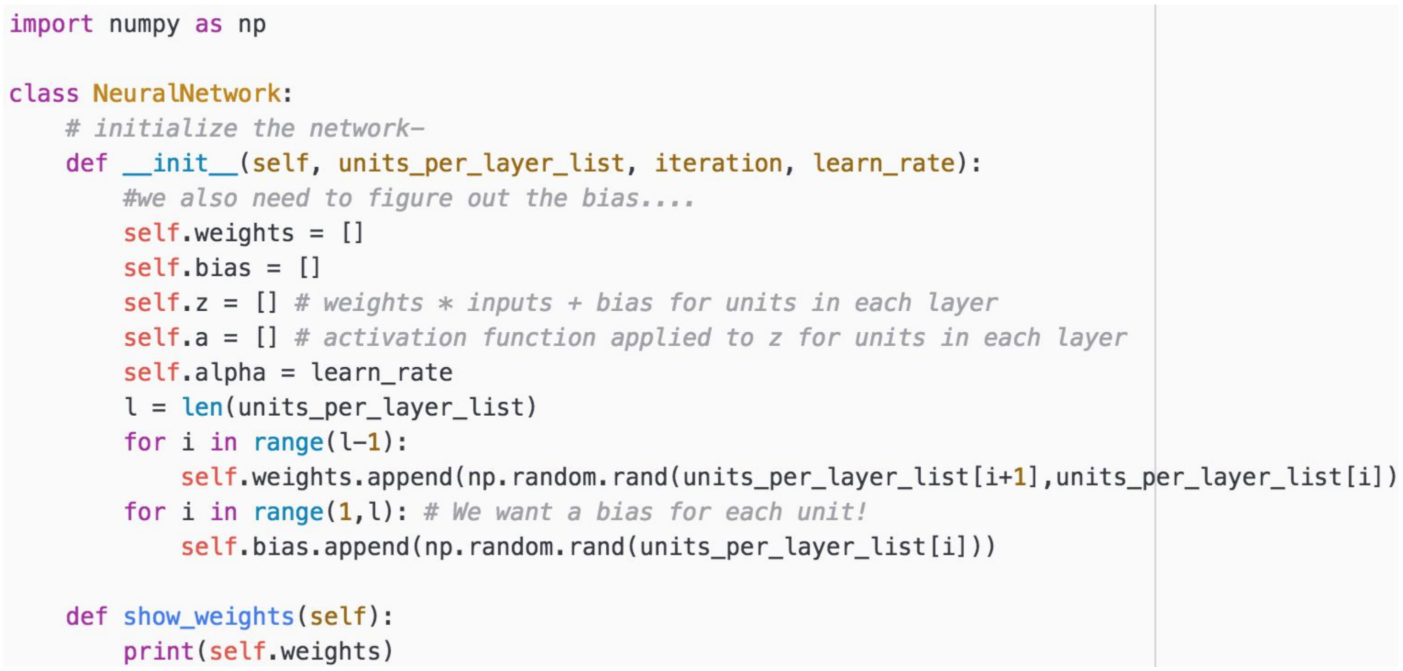
```
self.weights = []
l = len(units_per_layer_list)
for i in range(l-1):
self.weights.append(np.random.rand(units_per_layer_list[i+1],units_per_layer_list[i]))
```

Since the weights are initialized randomly, we use `numpy.random.rand` to generate a matrix of random values for each layer of the network.

Each layer of the network has a bias node. The output of the bias node is passed into each of the units in the next layer. Since each of the units in the next layer have a weight for the incoming bias node's weight value output, then you can simply make a bias for each unit in the neural network. We can iterate through the `units_per_layer_list`, which specifies the number of units in each layer, and assign a bias for each of them using numpy's `random.rand()` function. The `rand(x)` function creates `x` many random numbers. So if `units_per_layer_list[0]` has 5 units in the first layer (layer 0), then we would call `rand(5)`. This will be used to make the random weights for each of the bias units. This can be written in python as:

```
self.bias = [] # Let's have a list of all the bias nodes
l = len(units_per_layer_list)
for i in range(1,l): # We want a bias for each unit!
      self.bias.append(np.random.rand(units_per_layer_list[i]))
```

Now that we have created our weights and bias units, we should place our code into the __init__ function, where the neural network is initialized. The template should look like this now:

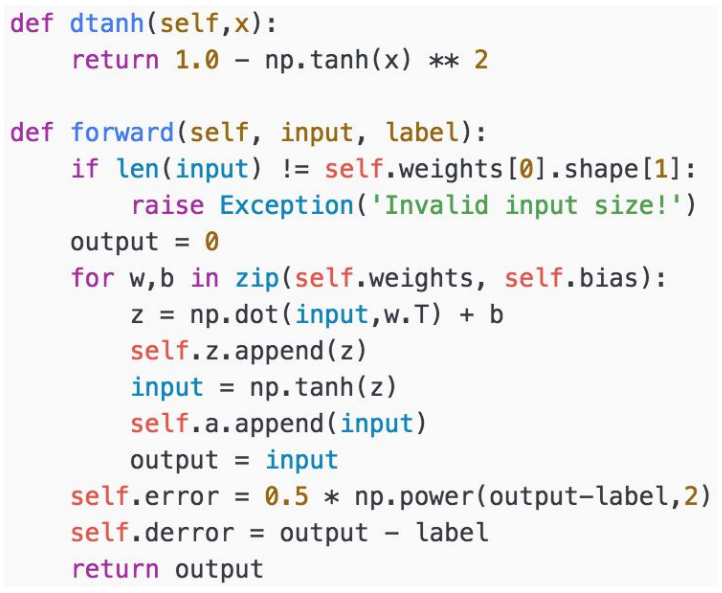


```
import numpy as np

class NeuralNetwork:
    # initialize the network-
    def __init__(self, units_per_layer_list, iteration, learn_rate):
        #we also need to figure out the bias....
        self.weights = []
        self.bias = []
        self.z = [] # weights * inputs + bias for units in each layer
        self.a = [] # activation function applied to z for units in each layer
        self.alpha = learn_rate
        l = len(units_per_layer_list)
        for i in range(l-1):
            self.weights.append(np.random.rand(units_per_layer_list[i+1],units_per_layer_list[i])
        for i in range(1,l): # We want a bias for each unit!
            self.bias.append(np.random.rand(units_per_layer_list[i]))

    def show_weights(self):
        print(self.weights)
```

Ignore the code that we haven’t discussed yet. I added a show_weights function to print out the weights of the neural network.

How to Compute Output in a Neural Network

A neural network computes its output through a process called forward propagation (Ng, et al). A single unit computes its output by applying an activation function to the weighted sum of its inputs. We've already discussed how the dot product calculates this weighted sum. After the activation function is applied to the sum of the weighted sum of inputs and the bias, the value is passed to the units in the next layer. This can be written in python as shown in my picture below. The `dtanh()` function is the derivative of the hyperbolic tanh function. The forward function produces the output of the neural network. It takes in input data, and the correct label for that input. First I check to make sure that the length of the input data vector matches the number of units in the first layer. The code `self.weights[0].shape` returns the number of rows and the number of columns of the weight matrix in the first layer. Remember that the number of columns represents the number of incoming values into each unit in the layer. `Shape[1]` returns just the number of columns, which is the number of units in the input layer. Next, I iterate through both the weight matrices and bias units in parallel. Each iteration represents the computations of outputs of all units in a single layer. In each iteration I compute the dot product between the weights and their inputs from the



```python
def dtanh(self,x):
    return 1.0 - np.tanh(x) ** 2

def forward(self, input, label):
    if len(input) != self.weights[0].shape[1]:
        raise Exception('Invalid input size!')
    output = 0
    for w,b in zip(self.weights, self.bias):
        z = np.dot(input,w.T) + b
        self.z.append(z)
        input = np.tanh(z)
        self.a.append(input)
        output = input
    self.error = 0.5 * np.power(output-label,2)
    self.derror = output - label
    return output
```

units of the previous layer. Then I add it to the bias nodes and save that output in a list called z. I apply an activation function to that output in each iteration. In this case, I'm using the tanh activation function on the output of each unit. After the output of the units in the last layer are computed, I compare them to the label to determine the error. I store the error and then return the final output.

Most neural networks use either the hyperbolic tangent function or the sigmoid function (Goodfellow 195). Sigmoid activations are mainly used to output probabilities (Goodfellow 195). The sigmoid function, as we learned in CMSC 421 is $f(z) = 1/(1+e^{-z})$. It returns values between 0 and 1. Hence, the decimal numbers can represent percentages. Since sigmoid functions are very sensitive to positive and negative values, they make "gradient-based learning very difficult" and are now discouraged (Goodfellow 195). According to Goodfellow, sigmoidal functions are more common in other types of neural networks such as recurrent neural networks. For feedforward neural networks, tanh() is a good option if you want to make a single output that isn't too sensitive to changing weights.

How to Train the Neural Network

Intro

Neural networks are trained with the help of backpropagation. Before computing backpropagation, the output of the neural network is compared to the actual output of the given input through a function known as a loss or cost function. Backpropagation means that the gradient of the loss function with respect to each of the weights in the neural network is computed. The whole idea of backpropagation is to find the weight values that decrease the overall loss function. If the loss function is low in value, it means that the predicted output of the neural network closely matches the actual output. In math, the objective of gradient descent is to find the local or the absolute minimum of a function. This is done by subtracting a product of the slope of the function with respect to each of its variables and a decimal learning parameter from the current coordinates. You might be wondering why gradient descent would even be useful when this could simply be achieved with just the slope. For instance, to find the absolute minimum of a cost function in a 2-dimensional plane, you could just solve for the slope of the function with respect to x and set it to zero, since at a minimum point, the slope is neither rising nor decreasing, hence the value of zero. After solving for the value of x, you simply get the y value of the minimum coordinate by plugging the x value as input into the function. But when many variables are involved, and the function has more than two dimensions as in neural networks, solving for the absolute minimum is time consuming. This is because you will have to solve for more than two coordinates (which are the weights) by setting the slope of the function with respect to each of those coordinates to zero. Gradient descent saves time by approximating this value. The gradient simply means that the slope of more than two variables is solved for.

Instead of solving for the coordinates at which the gradient (slope of each of the variables) is zero, they can be subtracted from the current coordinates to reach a minimum point in the function. The intuition is that traveling down a slope hopefully leads to the minimum point of a function.

Each weight is adjusted by subtracting the learning rate times the gradient of the loss function with respect to the weight, from the previous weight value. This should lead the overall loss closer to zero in each epoch as the weights are optimized (unless the learning rates are too high, which in that case the weights will be chosen past the minimum point). With a higher learning rate (that's not too high!), there will be bigger changes in the weight values. Loss values decrease faster with higher learning rates. A smaller learning rate will result in a longer amount of time to decrease the loss function (since the weights are being adjusted by small values instead of large values in the direction of the minimum point of the loss function).

Math Derivation

Using the computations we discussed earlier, we can represent all of them mathematically. (*The equations below are the same as what we've covered in CMSC 421, but with a slightly easier syntax to follow. These have been derived from former phD researcher Donald R. Tveter.*
)

Note: w_{jk} represents weights of outputs passing from layer *j* to unit *k*. η is the learning rate. o_j is the output of units in layer j . The correct value of output for the output unit o_k is t_k . (We only have one output unit, but to represent more than 1 output unit these formulas are helpful).

1. Weighted input into unit *j:* $net_j = \sum_{i=1,n} w_{ij} o_i$ (Tveter 4)
2. Output of unit j (f represents activation function applied): $o_j = f(net_j)$ (Tveter 4)
3. Error for output unit *k*: $\delta_k = (t_k - o_k) f'(net_k)$ (Tveter 5)
4. Error for hidden unit *j*: $\delta_j = f'(net_j) \sum_k \delta_k w_{jk}$ (Tveter 12)
5. Weight update formula: $w_{jk} \leftarrow w_{jk} - \eta \delta_k o_j$ (Tveter 5)
6. Bias update formula: Since the output of a bias is always 1, it is: $b_j \leftarrow b_j - \eta \delta_j$

Coding the Backpropagation

Even though these computations are for single units, they still work when using matrices to represent units in each layer! Therefore, we can comprehend the all subscripts such as *j,* and *k* to as all the units in the same layer. For instance, with our implementation, w_{jk} represents weights of outputs passing from layer *j* to all the units in layer *k* instead of unit *k*.

Notice that the weight update needs the delta formula, output value of that unit, and the weighted sum of that unit plus the bias. This is why I stored the output values and the weighted sums of units during forward propagation in `self.z` and `self.a`. The delta formula of a unit represents the gradient of the error function with respect to the weights of that unit. Also, the delta value of a unit depends on the delta value of the units in the next layer, which in turn depends on the delta value of units in the next layer and so forth. Hence, we need to iterate backwards from the output unit. Since the delta value of the output unit has a different pattern than the rest of the delta values of units, we will compute it separately outside the loop and then iterate backwards.

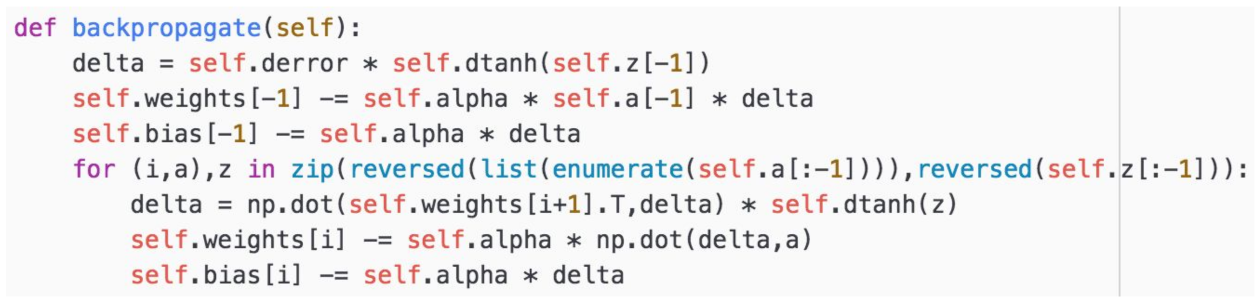
Using `j` to represent the j^th^ layer of units, we can implement equation 1 as: `self.z[j]`. Equation two just means that the derivative of the activation function is applied to the net value of the unit (weighted sum plus the bias). Since we are using the tanh activation function, we just apply the derivative of it, which I already programmed as the function dtanh earlier. So equation 2 is `dtanh(self.z[j])`. In python the index, `-1` represents the last layer or the output layer of our network. Also, we can represent the error $(t_k - o_k)$ as `self.derror`, which was saved in the forward propagation.

Now, let's compute the delta of the output unit (Equation 3) and update its weights inside a function:

```
def backpropagate(self):
      delta = self.derror * self.dtanh(self.z[-1])
      self.weights[-1] -= self.alpha * self.a[-1] * delta
      Self.bias[-1] -= self.alpha * 1 * delta
```

Following this, we need a for loop that will iterate backwards through both `self.a` and `self.z`. We can use `reversed()` to reverse the two lists. So the two lists we iterate through, excluding the data in the output layer would be: `reversed(self.a[:-1])` and `reversed(self.z[:-1])`. Iterate through both of them in parallel with the function `zip()`. Since we need to use indices represent the index of layers in each iteration, we will use `enumerate()`, which returns a number representing how many times that the for loop has occurred.

Now, after translating the remaining math equations and coding our implementations, we can code the full backpropagation algorithm shown on the next page:



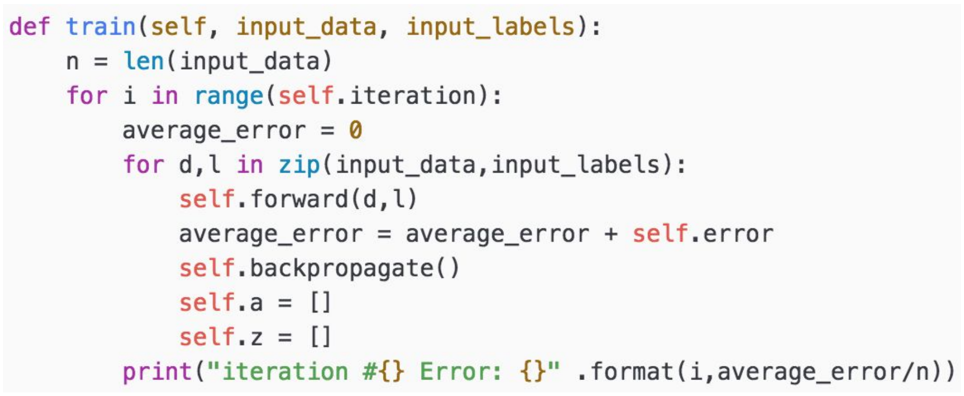
```
def backpropagate(self):
    delta = self.derror * self.dtanh(self.z[-1])
    self.weights[-1] -= self.alpha * self.a[-1] * delta
    self.bias[-1] -= self.alpha * delta
    for (i,a),z in zip(reversed(list(enumerate(self.a[:-1]))),reversed(self.z[:-1])):
        delta = np.dot(self.weights[i+1].T,delta) * self.dtanh(z)
        self.weights[i] -= self.alpha * np.dot(delta,a)
        self.bias[i] -= self.alpha * delta
```

This algorithm only updates weights after a single example. Updating weights after every example or a small number of examples is known as stochastic or online gradient descent (Tveter 9). Stochastic gradient descent is actually easier to implement in our model, which is why I used it instead of batch gradient descent. Batch gradient descent uses the "full training set to compute the next update to parameters at each iteration tend to converge very well to local optima" (Ng, et al). Common issues with batch gradient descent are that "computing the cost and gradient for the entire training set can be very slow and sometimes intractable on a single machine if the dataset is too big to fit in main memory" (Ng, et al). So the larger your dataset is, the longer it will take to update your weights with batch gradient descent. Since stochastic gradient descent updates the weights after each example, it usually runs faster.

Training the Network

Now that we have both backpropagation and forward propagation programmed, we can fully implement the training phase of the neural network. If you look back at the steps in the introduction, steps 5 and 6 are implemented in the backpropagation function. Notice that forward propagation and backwards propagation occur in a loop, which is step 7. The number of times this loop occurs for each example is dictated in step 8, which is known as `iteration` in our code. The weights are updated each time forward propagation is run on inputs in the data. This process is known as an epoch. In step 8, an epoch is run for a number of times specified by `iteration`.

We can program this process below in a function called train:



```
def train(self, input_data, input_labels):
    n = len(input_data)
    for i in range(self.iteration):
        average_error = 0
        for d,l in zip(input_data,input_labels):
            self.forward(d,l)
            average_error = average_error + self.error
            self.backpropagate()
            self.a = []
            self.z = []
        print("iteration #{} Error: {}" .format(i,average_error/n))
```

We will assume that the user inputs data and the corresponding correct outputs/labels as lists. After each iteration/epoch, I print out the average error of all the input data. I also have to manually clear the activations (`self.a`) of units in each layer and weighted sum (`self.z`) list after running the network on a single input, or else old data from previous inputs will corrupt the backpropagation algorithm.

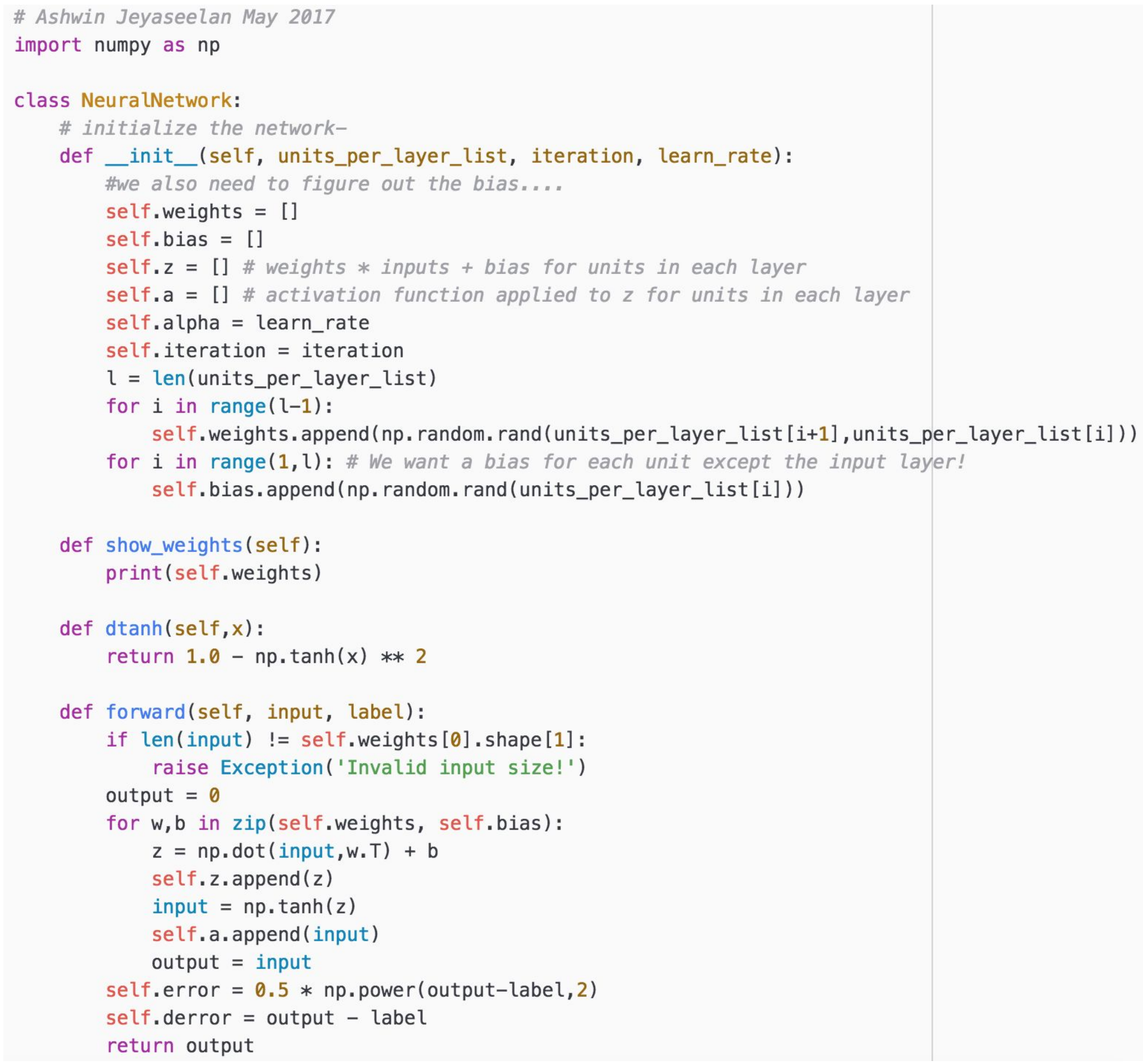
<u>Further Reading:</u>

If you still want more practice with backpropagation, I highly suggest that you check out Dr. Tveter's notes. He uses multiple examples with numbers to derive all the backpropagation equations! Here is the link: http://www.cim.mcgill.ca/~jer/courses/ai/readings/backprop.pdf. If you enjoy his notes, definitely read his other chapters on neural networks.

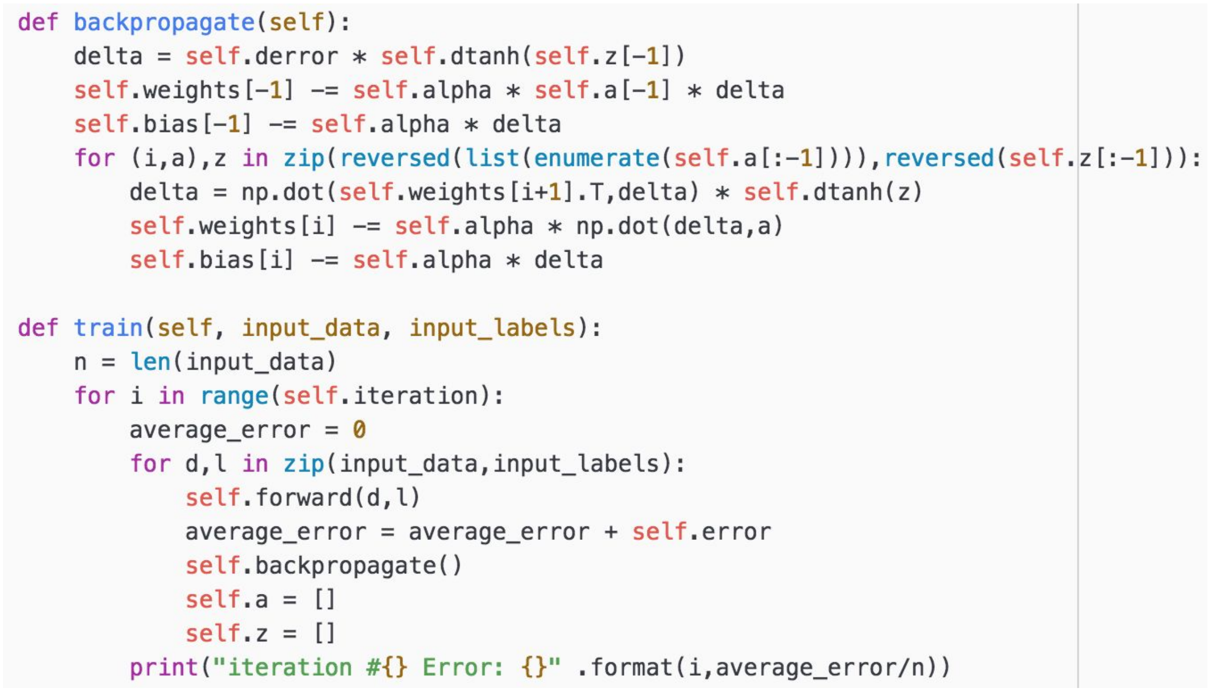
Backpropagation is actually one of three widely used techniques for training neural networks (Microsoft Build). The other two popular techniques are genetic algorithms and particle swarm optimization. Genetic algorithms are the slowest technique, but can sometimes be highly effective (Microsoft Build). They sound pretty interesting too. That’s because genetic algorithms are based off of the concept of evolution (Montana & Davis 763). If you want to learn more about genetic algorithms, I suggest you start with this old but introductory paper: https://www.ijcai.org/Proceedings/89-1/Papers/122.pdf by David J Montana. There actually exist a bunch of evolutionary methods to train neural networks, described on wikipedia with direct paper links here: https://en.wikipedia.org/wiki/Neuroevolution. One of the most popular evolutionary algorithms implemented in neural networks is a recent breakthrough called NEAT. The link is also on the wikipedia page I provided.

Running the Network

I’ve attached a snapshot of the full code that we’ve implemented so far:



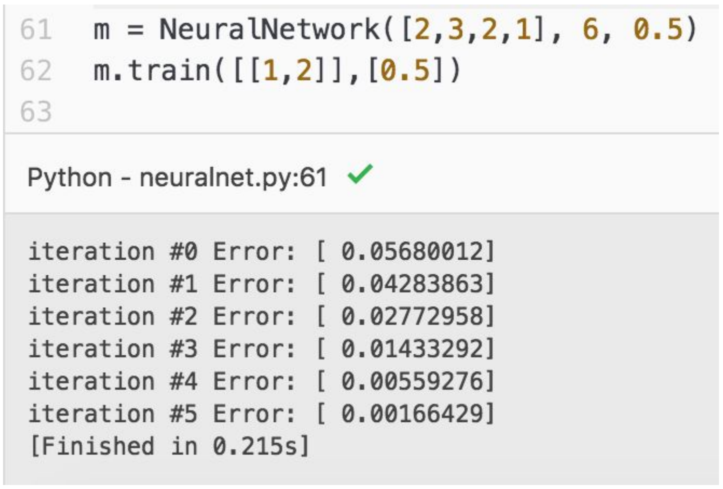
```python
# Ashwin Jeyaseelan May 2017
import numpy as np

class NeuralNetwork:
    # initialize the network-
    def __init__(self, units_per_layer_list, iteration, learn_rate):
        #we also need to figure out the bias....
        self.weights = []
        self.bias = []
        self.z = [] # weights * inputs + bias for units in each layer
        self.a = [] # activation function applied to z for units in each layer
        self.alpha = learn_rate
        self.iteration = iteration
        l = len(units_per_layer_list)
        for i in range(l-1):
            self.weights.append(np.random.rand(units_per_layer_list[i+1],units_per_layer_list[i]))
        for i in range(1,l): # We want a bias for each unit except the input layer!
            self.bias.append(np.random.rand(units_per_layer_list[i]))

    def show_weights(self):
        print(self.weights)

    def dtanh(self,x):
        return 1.0 - np.tanh(x) ** 2

    def forward(self, input, label):
        if len(input) != self.weights[0].shape[1]:
            raise Exception('Invalid input size!')
        output = 0
        for w,b in zip(self.weights, self.bias):
            z = np.dot(input,w.T) + b
            self.z.append(z)
            input = np.tanh(z)
            self.a.append(input)
            output = input
        self.error = 0.5 * np.power(output-label,2)
        self.derror = output - label
        return output
```



```python
def backpropagate(self):
    delta = self.derror * self.dtanh(self.z[-1])
    self.weights[-1] -= self.alpha * self.a[-1] * delta
    self.bias[-1] -= self.alpha * delta
    for (i,a),z in zip(reversed(list(enumerate(self.a[:-1]))),reversed(self.z[:-1])):
        delta = np.dot(self.weights[i+1].T,delta) * self.dtanh(z)
        self.weights[i] -= self.alpha * np.dot(delta,a)
        self.bias[i] -= self.alpha * delta

def train(self, input_data, input_labels):
    n = len(input_data)
    for i in range(self.iteration):
        average_error = 0
        for d,l in zip(input_data,input_labels):
            self.forward(d,l)
            average_error = average_error + self.error
            self.backpropagate()
            self.a = []
            self.z = []
        print("iteration #{} Error: {}" .format(i,average_error/n))
```

Now for the fun part! Let's run our neural network.

Using the code we've programmed, I've created a neural network object (In the picture I named it m). Feel free to test your code with various learning rates, network designs in the unit_per_layer_list, and different numbers of iterations. After running it with the train method you can view the results. In this simple example, the only input data I passed is an array of [1,2] with a correct output of 0.5. Notice that after 6 iterations, the neural network's prediction error gets closer to zero! Our network clearly gets better at predicting the correct output over time. Experiment with different input data and values for the variables in our code. Trying separating some examples with outputs as training data, and then



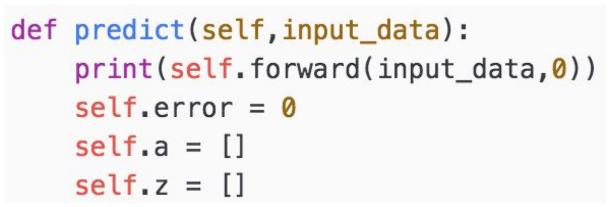
```python
m = NeuralNetwork([2,3,2,1], 6, 0.5)
m.train([[1,2]],[0.5])
```

Python - neuralnet.py:61 ✓

```
iteration #0 Error: [ 0.05680012]
iteration #1 Error: [ 0.04283863]
iteration #2 Error: [ 0.02772958]
iteration #3 Error: [ 0.01433292]
iteration #4 Error: [ 0.00559276]
iteration #5 Error: [ 0.00166429]
[Finished in 0.215s]
```

evaluate your model on separate testing data. What do you notice about the testing accuracy? One question that you may still have after an AI class such as CMSC 421 is how to intuitively decide on the number of hidden layers and units in your neural network. Unfortunately, there is no correct answer and you must experiment by trial and error (Manjunatha). If you're feeling confident enough, try to test our model on more complex data such as handwritten data. Since we haven't coded preprocessing of input data, it's up to you to convert it into a list of inputs and labels.

Last but not least, we can code a prediction function for the neural network to print out its prediction based on our input data. Since we already implemented forward propagation to spit out the output, all we need to do is call it in our predict function. I've coded the prediction function below:



```
def predict(self,input_data):
    print(self.forward(input_data,0))
    self.error = 0
    self.a = []
    self.z = []
```

Again, I need to clear the activations and weighted sums lists since I won't be using them. Remember that we only want to utilize the two lists during backpropagation in our training phase. Therefore, so that the lists aren't filled with junk data the next time we train our network, clear `self.a` and `self.z`. In our implementation, I'm assuming that prediction will only be called on input_data that doesn't have labeled answers (i.e. we're using the neural network to assist us on finding the output. That's why I passed in a random label of 0. Nonetheless, this is a case where the implementation of a function depends on you. If you want to print the error and use actual labels, then go ahead.

What's Next

Now that we've implemented the basic implementation of a feedforward neural network, I encourage you to continue learning more advanced methods in deep learning and to update our neural network program as you gain more knowledge. UMD Graduate student Abishek Sharma suggest that you "Develop mathematical skills in linear algebra, calculus, read some neuroscience as interesting reads" and to learn as much material as you can. I myself am trying to learn as much as I can about the subject, since it's such a fascinating and new area in the field of Computer Science.

Also, I posted the full code on my github repository at:
https://github.com/Techro-Raj/Neural-Network/blob/master/neuralnet.py.
I'll be updating my code in my spare time to accommodate more advanced techniques and other implementations. If you want to see the progress of my code, follow the repository and feel free to submit issues and suggestions on my Github.

Overall, it was a very fun project, and I hope that you've enjoyed this manual. The next two pages include interview transcripts of interviews that I conducted with two graduate machine learning students at the University of Maryland College Park. Their advice and remarks should leave you feeling inspired as you end this manual.

There are many other options to improve a neural network, but backpropagation and forward propagation are the two main ingredients to build a neural network. In order to learn more advanced neural network models, I suggest that you check out the paper ""Neural Network Models for Software Development Effort Estimation: A Comparative Study" by Nassif Bou.

To advance your knowledge on neural networks check out Dr. Goodfellow's new Deep Learning book: http://www.deeplearningbook.org/. If you find those too difficult, then read Donald Tveter's notes about neural networks first.

Abishek Sharma is a graduate student in Computer Science at UMD College Park who has recently become interested in deep learning.

Interview with Abishek Sharma: conducted by Ashwin Jeyaseelan on April 15, 2017

1. What advice would you give to undergraduate students hoping to study deep learning?
Develop mathematical skills in linear algebra, calculus, read some neuroscience as interesting reads. On the programming side, learn python and C++ in-depth. Start early with research, start with some online courses, lectures from famous deep-learning professors to familiarize yourself with the concept.

2. What's one of the coolest things you've seen that utilized neural networks?
There are many. I am afraid of choosing one, but face recognition is one of them.

3. Not all undergraduate machine-learning courses give assignments that force students to write neural networks from scratch or implement the math behind the techniques. Do you think that full implementations in code from scratch are better for a student to understand the concepts better? Why?
Absolutely. Full stack implementation for some basic neural network is essential for practice of putting the neural network pertaining software design into action and mathematical maturity. Writing equations on a piece of paper and putting it in the form of an efficient code are two different things and they can only be made better by actually trying to do it.

4. What are the most important or main heuristics that you decide on when building feedforward networks?

These days the field has been standardized and sanitized of dark magic tricks and getting common networks to work is rather straightforward out-of-the-box. Data preparation is often where people make mistakes and then comes the issue of novel loss functions or architectures that might need fiddling with the learning rates, proper regularizations to keep away the trivial solutions away etc.

5. How do you decide on the number of hidden layers in a neural network?

It depends on the task and there are different ways, I prefer to start with a point where I can fit the training data easily w/o bothering about validation accuracy then I work backwards and reduce the complexity while optimizing for validation accuracy and stop when I have achieved the best validation accuracy.

Varun Manjunatha is a 5th (2012-present) PhD student in Computer Science at UMD College Park. He studies computer vision and machine learning.

Interview with Varun Manjunatha: conducted by Ashwin Jeyaseelan on April 16, 2017

1. What's your background, and how did you get involved within the field of machine learning?
My route was EE to Signal Processing to Image Processing to Computer Vision to Machine Learning

2. What advice would you give to undergraduate students hoping to study deep learning?
Spend $600 on a GPU - it is worth it. If you're a gamer, you probably have one. Work on your own projects. It's really easy to come up with a nice project, because they all have the form : Download _____ from the internet and train a Deep NN to classify/generate _______. Eg : Download lyrics from various musical artists and can you get an Deep NN to generate lyrics ?

3. What's one of the coolest things you've seen that utilized neural networks?
I think if we can get self driving cars to work on a large scale in uncertain environments, that would be the coolest thing. Handwriting recognition and Face Recognition are pretty much solved.

4. If you have taken any machine learning courses are there any common issues you've come across in undergraduate machine learning courses, or concepts that you feel should be emphasized more?
I haven't taken an ML course at undergraduate level, but making students proficient at linear algebra and matrix calculus is absolutely critical. You'll need it a lot more than say, a Database or HCI course. I would also recommend studying Statistical Mechanics, because that would help you understand Boltzmann Machines (i.e., NNs of previous generations).

5. Not all undergraduate machine-learning courses give assignments that force students to write neural networks from scratch or implement the math behind the techniques. Do you think that full implementations in code from scratch are better for a student to understand the concepts better? Why?
Yes and No. I think it's a bit like learning about recursion in an algorithms class. You need to practice coding it a few times until the concept sticks in your brain.

6. What are the most important or main heuristics that you decide on when building feedforward networks?

There are a ton of tricks you can use to extract an extra few % points in performance. Study someone's state of the art code on github to learn more.

7. How do you decide on the number of hidden layers in a neural network?

Mostly trial and error.

8. Do you have any advice for students hoping to work at artificial intelligence companies?
Specific advice : Be adaptive. If tomorrow, a superior method/library comes out, be prepared to abandon everything you've done so far and start working on that.
Long-term advice : Progress in Deep Learning in the last 5 years has happened due to a handful (10-12) of good ideas. A "good idea" is like a good chess move. It's very hard to discover, but once discovered, it seems very obvious. Make it your life's aim to discover one such idea.

Works Consulted

Beckett, Jamie. “Why Enrollment Is Surging in Machine Learning Classes.” *Nvidia*, 24 Feb. 2016, https://blogs.nvidia.com/blog/2016/02/24/enrollment-in-machine-learning/. Accessed 25 Mar. 2017.

Chainer. Preferred Networks, http://chainer.org/.

“Developing Neural Networks Using Visual Studio.” *Youtube*, uploaded by Microsoft Build, 22 April 2014, https://www.youtube.com/watch?v=9aHJ-FAzQaE.

Goodfellow, Ian, et al. *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org/. Accessed 20 Mar. 2017.

Leverington, David. “A Basic Introduction to Feedforward Backpropagation Neural Networks.” *David William Leverington*. Texas Tech University, Department of Geosciences, 2009, http://www.webpages.ttu.edu/dleverin/neural_network/neural_networks.html. Accessed 27 Mar. 2017.

Mandel, Travis and Mache, Jens. “Developing a short undergraduate introduction to online machine learning.” *Journal of Computing Sciences in Colleges*, vol. 32, no. 1, 10 Jan. 2016, pp 144-150. *ACM Digital Library*. Accessed 26 Mar. 2017.

Manjunatha, Varun. Personal Interview. 16 April 2017.

Montana, David J. and Lawrence Davis. *Training Feedforward Neural Networks Using Genetic Algorithms*. 1989, https://www.ijcai.org/Proceedings/89-1/Papers/122.pdf.

Nassif Bou Ali, et al. "Neural Network Models for Software Development Effort Estimation: A

Comparative Study." *arXiv.org. Cornell University Library*, doi: 10.10007/s00521-015-2121-1.

Ng, Andrew, et al. *UFLDL Tutorial*. Stanford University, http://ufldl.stanford.edu/tutorial/. Accessed 20 Mar. 2017.

Nielsen, Michael. *Neural Networks and Deep Learning*, http://neuralnetworksanddeeplearning.com/images/tikz11.png. Accessed April 15, 2017.

Sharma, Abishek. Personal interview. 15 April 2017.

Tveter, Donald R. *Chapter 2 The Backprop Algorithm*. 1995, http://www.cim.mcgill.ca/~jer/courses/ai/readings/backprop.pdf.