Artificial Intelligence Lab D: Neural Networks

Adapted from work by Dr. Michael J. Garbade, Milo Spencer-Harper, Dr. Lisa Meeden, Konstantinos Kitsios, and James Loy. It draws most heavily on this tutorial by Kitsios.

In a practical environment, you would not generally build your neural network code from scratch. (This is true of almost all machine learning algorithms.) You would use something like TensorFlow, a tool for working with neural networks and other related techniques. The reason that we are spending time learning how things work underneath the hood is that it will build understanding that will enable you to use more powerful tools in a way that is correct.

For this lab, you will be implementing a very basic neural network system. The purpose of this is to get familiar with the basics of how such a system works.

I would strongly encourage you to do this lab as independently as possible, asking for help on Piazza/Email/Slack if necessary, rather than leaning too heavily on copying details from existing implementations. This lab has a very long window of time to work on it to allow for this. This is a solo lab, but you are free to discuss the lab with others - just don't directly share your code. Unlike the first three labs, this lab is brand new for 2020, so please let me know if there are any pain points.

The network you will be building as part of this lab is intentionally extremely simple. We are building something small enough to be understandable, not a full-featured industrial-strength classifier. You should **not** make use of a machine learning package as part of doing this lab, but will use numpy. This package will make it easier to work with the data and do array operations.

This lab is divided into two sections, Part A and Part B. Part A is structured, and is the part that will be graded. Part B is open-ended, but will not be graded. Everybody should complete Part A. If you have the time and ability to do so, I recommend exploring Part B as well, but it will not be collected.

This lab tries to balance being a careful walkthrough with still leaving some things for the student to figure out on their own. Make sure to read all of the instructions carefully, as they in some places instruct you to write code that is not given. It is also up to you to ensure that the overall architecture is correct. Understanding what all of the pieces of data represent is important for getting the whole thing assembled correctly.

Lab D is due **April 24th**. Submit your code, a readme, and a report containing things in the lab that are highlighted in **pink**.

1) Understanding Neural Networks and Matrix Operations

For this lab, we will be building a simple feed-forward neural network that learns using backpropagation. As this is somewhat complex and we don't have the luxury of seeing each other in person regularly, this lab will be structured as more of a walkthrough than some previous labs. However, it is important to understand the basics of how a neural network works before we start. If you feel lost while working on the lab, please go back and review the slides or reach out with questions.

This lab will make a lot more sense if you are comfortable with the idea of the dot product of two matrices. Brush up here if you need to. This lab uses axb and a-by-b interchangeably for an array/matrix with *a* rows and *b* columns.

2) Getting Started

We will be initially building a classifier that recognizes the xor function. This function looks like this:

x	y	output
0	0	0
0	1	1
1	0	1
1	1	0

Xor is the *exclusive or* function. (x xor y) is true if x is true or y is true, but not if both are true. Our neural network will eventually have an **input layer**, a **hidden layer**, and an **output layer**. Because our data has two features (x and y), our input layer will have two nodes. We will also put two nodes in our hidden layer. (This turns out to be the minimum number required to build a simple neural network that handles xor correctly.) As we are only predicting one binary value, our output layer will have a single node. On scratch paper, sketch what you believe the neural network will look like, based on this description. Make sure to draw the arrows. Then go here to check if it matches. (You do not need to submit this.)

Now would be a good time to get xor.txt from Moodle or from ~barbeda/cs365-share/xor.txt on bowie.

3) Building our network - Sigmoid Function

We will start by thinking about the tools that we will need to make a single node in the network work. One of the things we will need is an **activation function**. If you are not clear on the role of an activation function, please reach out with questions before moving on.

For our activation function, we can use a **sigmoid function**. There are many other possible activation functions, many of which are sigmoid functions. We will use a simple and common one.

The sigmoid function we use in a neural network will take one input, x. x is going to be the sum of all of the inputs to the node, plus the bias. There are lots of possible options; for consistency, let's use this, which produces a value between 0 and 1:

$$\frac{1}{1+e^{-x}}$$

e is a mathematical constant. numpy knows about e, so we can define our sigmoid function like this:

```
import numpy as np

def sigmoid(x):
    return 1/(1 + np.e ** (-x))

# Because using e as the base of an exponent is so common, numpy has a
built-in way of doing this. The function below is equivalent to the
one above

def sigmoid(x):
    return 1/(1 + np.exp(-x))
```

(The math library also knows about e.)

Test out your sigmoid function with the following values:
-10; -1; 0; 1; 10

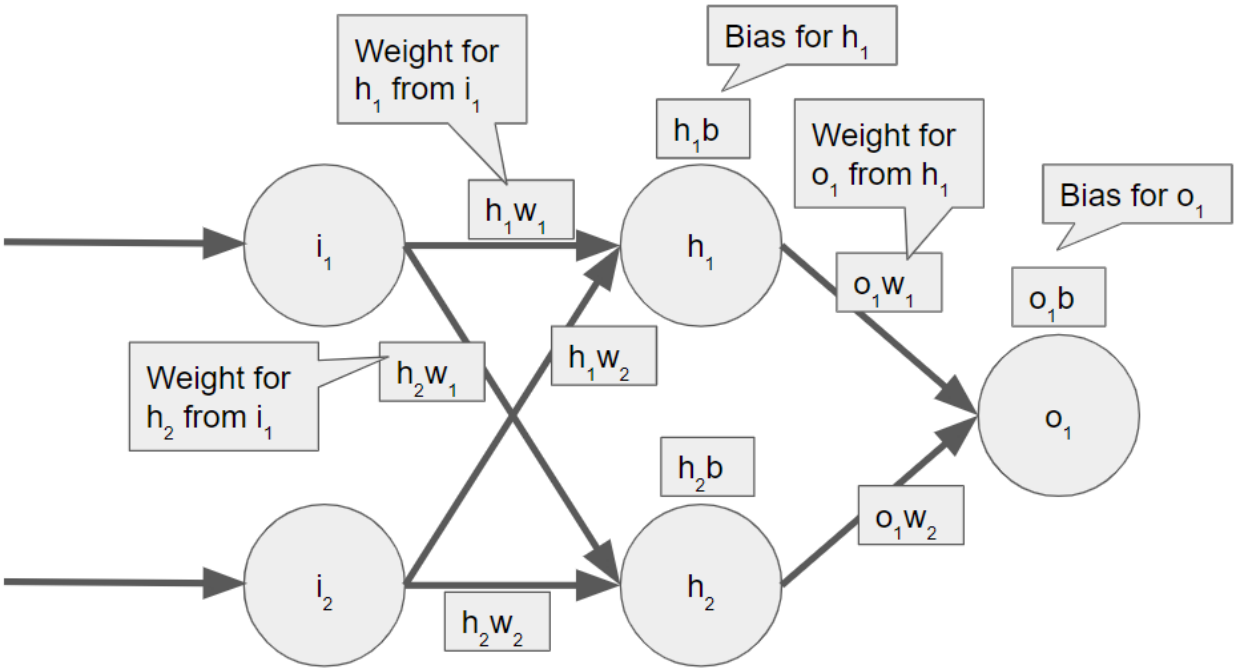
Do you see what you expect? If you're not sure what to expect, review how sigmoid functions work. (You don't need to submit these results.)

4) Building our network - Weights and Biases

While our network will have two nodes in the input layer, two nodes in the hidden layer, and one node in the output layer, we don't want to hard-code those numbers into our architectures. We want to build something that can handle different numbers of features and more or fewer hidden layer nodes.

One possible strategy might be to create a node object (or several types of node objects), and then populate a neural network object using them. However, we do not need to explicitly create

node objects at all, especially with a setup this simple. Instead, we will store the weights and biases in arrays, and update these arrays as we learn. We will need two sets of weights and biases: one set for the hidden layer, and one set for the output layer. Consider that our network will look something like this:



As you can see, the arrows feeding into the hidden layer store a total of (h * i) weights, where i is the size of the input layer, and h is the size of the hidden layer. This is 2*2 = 4 in this case. We also need to store a total of h biases: 2 in this case. Similarly, we store (o * h) weights for the arrows going to the output layer, and o biases. These are 1*2 = 1 and 1, respectively.

To make some later operations simpler, we will store these as two-dimensional arrays. (Notably, we will store them as two-dimensional arrays even if one of the dimensions is 1 - we do this so that we can do matrix operations using them.) Fortunately, numpy provides great support for matrix operations using arrays. Let's start with the biases. We will store each set of biases as a k-by-1 array, where k is the number of nodes in that layer. In our example, that means that the biases for the hidden layer will be stored as a 2x1 array, and the biases for the output layer will be stored as a 1x1 array.

We want all of our biases to start out with a value of 0. Numpy allows us to easily make arrays full of zeros. We could do something like this:

```
hidden_biases = np.zeros((2,1))
output_biases = np.zeros((1,1))
# Note that the argument is one 2-tuple, not two integers.
```

However, that locks us into having two hidden nodes and one output node. Instead, we should create a function that takes the number of nodes in each layer and initializes our biases and weights. It might look something like this:

```
def init_weights_biases(num_input_nodes, num_hidden_nodes, num_output_nodes):
    parameter_dictionary = {}
    # Code goes here; it should add
    # “hidden_biases” : np.zeros(num_hidden_nodes,1)
    # to parameter_dictionary, etc.
    return parameter_dictionary
```

This function signature does lock us into having only one layer of hidden nodes, but that will be powerful enough for many purposes. Part B optionally asks you to extend your code to allow an arbitrary number of hidden node layers.

We will store our weights in a similar fashion. However, we want our starting weights to be *random*, not zero. Numpy includes a variety of means for generating random values. The function np.randn(a,b) will generate an a-by-b array of values normally distributed around 0. That will work for our purposes. Execute

```
print(np.random.randn(2, 2))
```

A few times to see what these values look like. Then modify init_weights_biases to add hidden_weights and output_weights to the parameter dictionary. Make sure that they are the correct dimensions - the weights for the hidden nodes should be an (h-by-i) array, and the weights for the output nodes should be an (o-by-h) array, where i, h, and o are the number of input, hidden, and output nodes, respectively.

init_weights_biases should now return a dictionary with four elements. Run it with

```
init_weights_biases(2,2,1)
```

to see if it looks correct for our purposes. If everything is correct, the dictionary should look something like this:

```
{'hidden_biases': array([[0.],
                         [0.]]),
 'output_biases': array([[0.]]),
 'hidden_weights': array([[ 1.33992528, -1.25980713],
                          [-0.85382129,  0.64562339]]),
 'output_weights': array([[-0.74494437, -1.1078568 ]])}
```

Once that's working, we can move on.

5) Setting Up - Reading Our Data
Often, when we're working with data, it will be given to us as a file of some kind. Write a function

```
def read_file_to_array(file_name):
    ...
    return (feature_array, label_array, header_array)
```

That takes a tab-separated file (similar to lab 3) and returns three numpy arrays. If you haven't used numpy before, this is a good opportunity to get acquainted with the basics of creating arrays using it.

The first array should be an (f-by-e) *feature array*, where f is the number of features and e is the number of examples in the input file. (Everything except the last column is a feature.)

The second array should be a (1-by-e) *label array*, where e is the number of examples.

The last array should be an ((f+1)-by-1) *header array* that contains the names of the columns. (The extra one is because this will also have the header of the label column.)

It's important that the feature and label arrays are in the same order, so the first features in the feature array correspond to the first label in the label array. We don't actually need the header array to do any of the calculations, but we should keep track of that information in case we want to use it to make our output easier to understand.

You might find `np.array()` to be a useful function.

This will make a 1-dimensional array out of a python list:

```
np.array(python_list)
```

This will make a 2-dimensional array out of a python list:

```
np.array(python_list, ndmin=2)
```

You can transpose (turn sideways) a two-dimensional array using `array_name.T`
This returns a transposed version of that two-dimensional array.

Depending on your implementation, you may also be able to make use of numpy.append().

If you run using xor.txt, you should get the following. It's important that you get exactly this, not something that's kind of like this but with the numbers reorganized or the wrong number of dimensions.

```
features, labels, headers = read_file_to_array("xor.txt")
print(features)
print(labels)
print(headers)
# The above should produce:

[[0. 0. 1. 1.]
 [0. 1. 0. 1.]]
[[0. 1. 1. 0.]]
[['x']
 ['y']
 ['output']]
```

Note that these are numpy arrays, not Python lists, but numpy arrays are formatted to look sort of like Python lists when you use print(). Make sure that your arrays contain floats or ints, not strings. If they contain ints, they will look like this:

```
[[0 0 1 1]
 [0 1 0 1]]
[[0 1 1 0]]
[['x']
 ['y']
 ['output']]
```

If you're familiar with the pandas package, you can use that to do this step easily, but take this opportunity to get familiar with some numpy basics if you're not.

6) Building our network - Forward Propagation

Before we can train our network, we need to be able to try to classify examples. The math is going to get a little tricky here, because we'll be using matrix operations to make things more efficient.

Our forward_propagate() function will take in two arguments:

- A two-dimensional (f-by-e) *feature array*, where f is the number of features and e is the number of examples we're giving the system to classify. The feature array returned by read_file_to_array() should work here.

- A dictionary of our current weights and biases, like the one initialized by init_weights_biases().

Because of the second argument, we have access to the weights and biases for the hidden nodes, and the weights and biases for the output nodes. (A single output node, in our case.)

At this point, one option would be to loop over the columns of the feature array in order to run our examples one at a time. However, we can save time using a matrix operation called the dot product. If you've forgotten how the dot product works since doing part one, go back and review it now. I am writing the exam at the same time as I am writing this lab, and one of the questions on the exam is "Explain why we are to take the dot product of the hidden layer weights and the feature array in Lab D rather than iterating over the array to classify each example one at a time."

To help us understand why taking the dot product of the hidden layer weights and the feature array gives us what we want, let's use a real example.

Suppose our hidden layer weights look like this:

```
[[ 0.28083233 -0.27412689]
 [ 0.06568604 -0.33725368]]
```

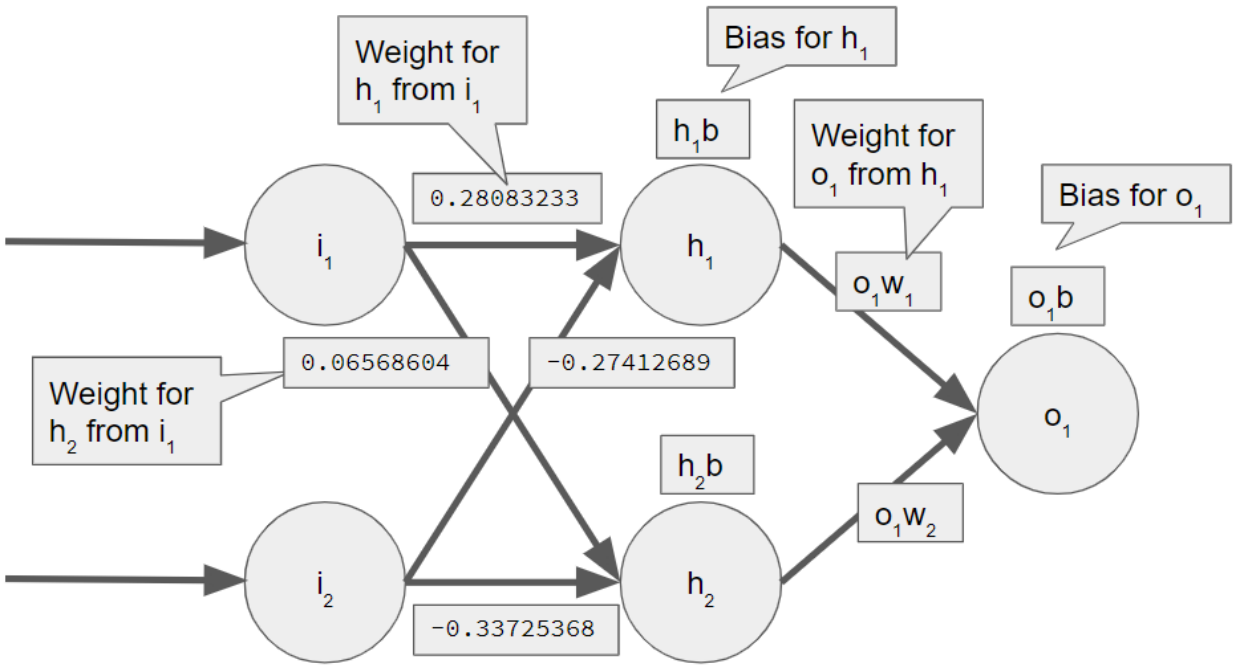
You will have different numbers, because you're generating them randomly, but your hidden layer weights should look something like that.

Suppose our input array looks like this:

```
[[0 0 1 1]
 [0 1 0 1]]
```

If your code that reads from xor.txt is working correctly, that's what your input array should look like. Each column is one of our pairs of features - the first column is the 0,0 case, the second column is the 0,1 case, and so on.

You may want to put print statements inside of your forward_propagate() function to verify that that's what you're getting when it is called. If we were to put these numbers into a picture of our neural network, it would look like this:



We are taking the dot product of a 2x2 array by a 2x4 array. We expect to get a 2x4 array out. (The size of the dot product of two matrices is the first number of the first one by the second number of the second one.) That's what we want; each of the columns in that result array is one of our four examples, just as it is in the input array. The top row is h_1's value, and the bottom row is h_2's value.

	Example 1	Example 2	Example 3	Example 4
h_1 value				
h_2 value				

Let's first figure out how we will populate the upper-left hand cell. (Don't do any of this in code; we're using the dot product to avoid having to do this in code.) To do this, we will take the numbers in the first row of our first array, multiply them by the numbers in the first column of our second matrix, and add them together. That's `(0*0.28083233 + 0*-0.27412689)`, which is 0.

	Example 1	Example 2	Example 3	Example 4
h_1 value	0			
h_2 value				

That's not the most illustrative example, so let's calculate the h_1 value for example 4. We will take the numbers in the first row of our first array, multiply them by the numbers in the fourth column of our second array, and add them together. That's `(1*0.28083233 + 1*-0.27412689)`, which is 0.00670544.

	Example 1	Example 2	Example 3	Example 4
h_1 value	0			0.00670544
h_2 value				

If you look back at the figure that shows these weights in our network, you can see that we're doing the same operations that we'd do to calculate the value for h_1; we're multiplying the values from the input nodes by the weights on their associated connections, and then adding them together! This means that

```
np.dot(hidden_weights, input_array)
```

Will produce a 2-by-4 array that contains exactly what we want - the values for each of the hidden nodes for each of our four inputs. (Before adding the bias and feeding the result to the activation function.)

At this point, if you get an error like
`shapes (2,1) and (2,4) not aligned: 1 (dim 1) != 2 (dim 0)`
It likely means that one of your arrays has the wrong dimensions, or possibly that your arguments to np.dot are in the wrong order.

Adding the bias is relatively straightforward - we can simply add the correct bias array to our values, like so:

```
np.dot(hidden_weights, input_array) + hidden_biases
```

Note that because we initialized our biases to 0, this won't actually change anything yet. We can then give that entire array to our activation function. The result should be a 2-by-4 array that represents, for each of our four examples, the result that the hidden nodes will pass to the output nodes. This part of your code will look something like this:

```
hidden_layer_values = np.dot(hidden_weights, input_array) +
hidden_biases
hidden_layer_outputs = sigmoid(hidden_layer_values)
```

You should write corresponding code to calculate the output of the output layer - use the dot product to calculate a set of values, then use the sigmoid function to produce the final outputs.

Your output layer will produce a 1x4 array of values between 0 and 1. These are the predictions your network made for each of your four inputs! Mine look like this:

```
[[0.44443625, 0.45453278, 0.44563674, 0.45580192]]
```

You will notice that these aren't very good. We want `[0, 1, 1, 0]`, and we didn't get anything like that. Unless your network got extremely lucky, your output values probably aren't much better. (You can run a few times and see what you get each time - it should vary, as we are initializing the weights randomly.) That's because we haven't trained the network yet. To actually learn, our network needs to understand where it went wrong.

Before we leave our forward_propagate() function, there's one more thing we want to do. Because of how backpropagation works, we're going to want the values that were produced not just by the output layer, but by the hidden layer as well. An easy way to handle this is to return a dictionary that looks something like this:

```
output_vals = {"hidden_layer_outputs": hidden_layer_outputs,
               "output_layer_outputs": output_layer_outputs}
```

7) Training our network - Calculating Our Loss

The loss (or error) is a measure of the total amount by which the network was incorrect in its predictions. There are many ways to produce a value that can be used as the loss. For any of them, we need the `output_layer_outputs` from our forward_propagation() as well as our `label_array` from back when we first read in the data from the file.

Some loss functions are good for when we are trying to predict a value (regression). Others are good for when we are trying to do classification. We will use a loss function that is known to be a good starting place for binary classification - that is, trying to produce a 0 or a 1, as we are doing here. The loss function we will use is called Binary Cross-Entropy Loss, or sometimes just cross-entropy for short. You can learn about this loss function here. The important thing to know is that this produces a value that is **higher** if more of our predictions are **further away** from what they should be, which is a desirable feature for a loss function to have. If you are feeling ambitious, you can try to build a calculate_loss() function that implements this, but to save time I have provided one here:

```
# Use binary cross-entropy to calculate the loss
def find_loss(output_layer_outputs, labels):
    # The number of examples is the number of columns in labels
    num_examples = labels.shape[1]
    loss = (-1 / num_examples) * np.sum(np.multiply(labels, np.log(output_layer_outputs)) +
                                        np.multiply(1-labels, np.log(1-output_layer_outputs)))
    return loss
```

8) Training our network - Backpropagation

The basic idea behind backpropagation is that we look at our error, figure out which weights and biases are most responsible for it, and adjust those the most. We also want to make sure that we're adjusting things in the right direction. We say that we are attempting to calculate the *gradients* on the weights and biases. The full mathematics for how this works involves partial derivatives, which this course does not mandate a background in, so we will be covering this at a somewhat higher level in the lecture notes. If your confidence in your calculus is high, you may want to check out this resource.

You can make use of this function for backpropagation. Step through it carefully to get a basic idea of what it is doing:

```
def backprop(feature_array, labels, output_vals, weights_biases_dict, verbose=False):
    if verbose:
        print()
    # We get the number of examples by looking at how many total
    # labels there are. (Each example has a label.)
    num_examples = labels.shape[1]

    # These are the outputs that were calculated by each
    # of our two layers of nodes that calculate outputs.
    hidden_layer_outputs = output_vals["hidden_layer_outputs"]
    output_layer_outputs = output_vals["output_layer_outputs"]

    # These are the weights of the arrows coming into our output
    # node from each of the hidden nodes. We need these to know
    # how much blame to place on each hidden node.
    output_weights = weights_biases_dict["output_weights"]

    # This is how wrong we were on each of our examples, and in
    # what direction. If we have four training examples, there
    # will be four of these.
    # This calculation works because we are using binary cross-entropy,
    # which produces a fairly simply calculation here.
    raw_error = output_layer_outputs - labels
    if verbose:
        print("raw_error", raw_error)

    # This is where we calculate our gradient for each of the
    # weights on arrows coming into our output.
    output_weights_gradient = np.dot(raw_error, hidden_layer_outputs.T)/num_examples
    if verbose:
        print("output_weights_gradient", output_weights_gradient)

    # This is our gradient on the bias. It is simply the
    # mean of our errors.
    output_bias_gradient = np.sum(raw_error, axis=1, keepdims=True)/num_examples
    if verbose:
```

```
        print("output_bias_gradient", output_bias_gradient)

    # We now calculate the amount of error to propagate back to our hidden nodes.
    # First, we find the dot product of our output weights and the error
    # on each of four training examples. This allows us to figure out how much,
    # for each of our training examples, each hidden node contributed to our
    # getting things wrong.
    blame_array = np.dot(output_weights.T, raw_error)
    if verbose:
        print("blame_array", blame_array)

    # hidden_layer_outputs is the actual values output by our hidden layer for
    # each of the four training examples. We square each of these values.
    hidden_outputs_squared = np.power(hidden_layer_outputs, 2)
    if verbose:
        print("hidden_layer_outputs", hidden_layer_outputs)
        print("hidden_outputs_squared", hidden_outputs_squared)

    # We now multiply our blame array by 1 minus the squares of the hidden layer's
    # outputs.
    propagated_error = np.multiply(blame_array, 1-hidden_outputs_squared)
    if verbose:
        print("propagated_error", propagated_error)

    # Finally, we compute the magnitude and direction in which we
    # should adjust our weights and biases for the hidden node.
    hidden_weights_gradient = np.dot(propagated_error, feature_array.T)/num_examples
    hidden_bias_gradient = np.sum(propagated_error, axis=1,
keepdims=True)/num_examples
    if verbose:
        print("hidden_weights_gradient", hidden_weights_gradient)
        print("hidden_bias_gradient", hidden_bias_gradient)

    # A dictionary that stores all of the gradients
    # These are values that track which direction and by
    # how much each of our weights and biases should move
    gradients = {"hidden_weights_gradient": hidden_weights_gradient,
                 "hidden_bias_gradient": hidden_bias_gradient,
                 "output_weights_gradient": output_weights_gradient,
                 "output_bias_gradient": output_bias_gradient}

    return gradients
```

9) Training our network - Updating our weights

Our backpropagation function returns a set of *gradients* that we can use to update our weights and biases. (The biases are essentially just additional weights.)

We want to write a function

```
def update_weights_biases(parameter_dictionary, gradients, learning_rate):
    ...
    return updated_parameters
```

That calculates a new set of weight and bias parameters, based on the gradients, and returns it as a dictionary in the same format as parameter_dictionary. To do this, we will be making use of a new constant we call the *learning rate*. The learning rate is a measure of how quickly our network will change itself. If the learning rate is high, then it will change itself quickly, and if the learning rate is low, it will change itself slowly. We might think that we should set the learning rate as high as possible, so that it will learn fast, but a learning rate that is too high will cause the system to have a hard time converging. (See the lecture notes for more details.) The learning rate is between 0.0 and 1.0.

Our new values for each of our weight and bias parameters can be calculated like this:

```
new_hidden_weights = hidden_weights - learning_rate*hidden_weights_gradient
```

You will need to write corresponding lines of code for the other weight and bias values, and then store them in a dictionary that is returned.

Test your update code by:

1. Using read_file_to_array() to generate feature and label arrays
2. Using init_weights_biases() to generate an initial sets of weights and biases, stored as a dictionary
3. Using forward_propagate() to generate a set of output values, stored as another dictionary.
4. Using backprop() to generate a set of gradients, stored as yet another dictionary.
5. Using the parameter dictionary from step 2 and the gradients dictionary from step 4, along with a learning rate of 0.3, call update_weights_biases(). It should return a dictionary of parameters similar to the dictionary that was passed into it, but adjusted slightly.

10) Training our network - Putting it all together

We've now built all of the code we need for our neural network. All that's left to do now is to build a piece of code that will updates the weights over and over again, so that it can continue to improve.

The basic flow should work by:

1. Using read_file_to_array() to generate feature and label arrays

2. Using init_weights_biases() to generate an initial sets of weights and biases, stored as a dictionary
3. Repeating steps 4-7 some number of times. (This number should be a parameter that is passed into the function that does this.) We call this parameter the number of *epochs*.
4. Using forward_propagate() to generate a set of output values, stored as another dictionary.
5. Using find_loss(), report the current loss, along with the number of our current epoch.
    a. In practice, we may want to set things up so that this is only reported every 100 epochs or every 1000 epochs.
6. Using backprop() to generate a set of gradients, stored as yet another dictionary.
7. Using the parameter dictionary from step 2 and the gradients dictionary from step 4, along with a learning rate of 0.3, call update_weights_biases(). It should return a dictionary of parameters similar to the dictionary that was passed into it, but adjusted slightly.
8. Return the final set of weights and biases, which can be inspected and used to classify new examples.

An example signature might be:

```
def model_file(file_name, num_inputs, num_hiddens, num_outputs, epochs, learning_rate)
```

11) Testing things out - Hyperparameters

It's not obvious what we should set our learning rate to be, and it's not obvious how many iterations it will take to start to converge on a set of weights and biases. These two things are closely related - if our learning rate is small, it will take more iterations to converge. Start by using a learning rate of 0.3 and training for 1000 epochs. This shouldn't take very long, but you may notice that the loss isn't yet close to 0. (There's randomness in the initial weights, so it might be close to zero, but probably isn't.) Raise the number of epochs to see what you see. How many epochs does it take for the loss to fall to near zero (e.g., <0.01) with a learning rate of 0.3? Try it a few times, to get a sense of how much variability there is here. Does it always actually converge, or can it get stuck? Don't be afraid to run for a very large number of epochs - training can take a while.

Repeat this experiment with learning rates of 0.6, 0.1, and 0.001.

In your **report**, write up what you see here.

If your code from the previous step is working, it should be producing a set of weights and biases that you can use to draw what your system looks like after training. Draw that neural network, with the weights your system learned, in your **report**. Does this network seem correct?

Part B

There are many things you can do to extend your neural network program. Some of these may work best if you redesign the overall architecture. Note that all of these ideas are optional, and some are very difficult. Not all of these have been tested.

- Experiment with adding additional hidden nodes to your network's hidden node layer. You shouldn't need to modify your existing code much, if at all, to do this.
- Experiment with different sets of data. Can your network learn other logical operators? Are they faster to learn? Slower?
- Extend your code to allow an arbitrary number of hidden node layers. (This one is quite a bit of work, and will require re-architecting large portions of the code.)