Hidden Layer

An overview of the limitations of a single layer perceptron model.

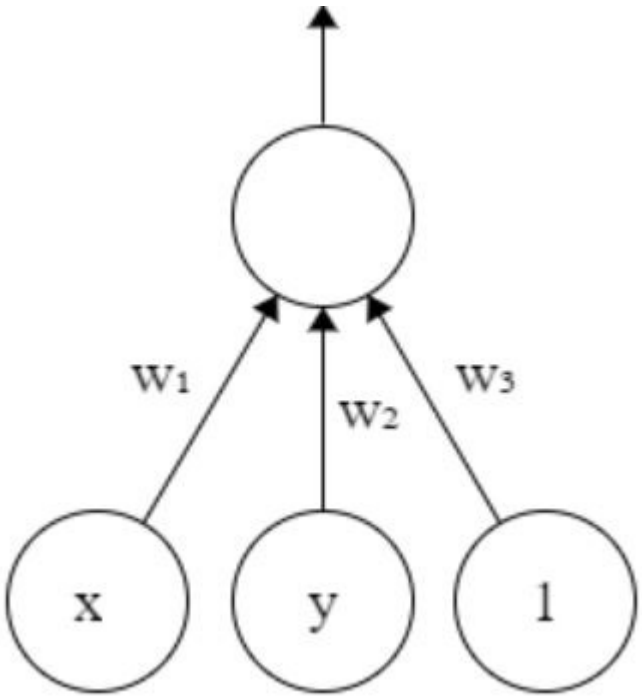
Chapter Goals:

- Add a hidden layer to the model's layers
- Understand the purpose of non-linear activations
- Learn about the ReLU activation function

A. Why a single layer is limited

In the previous chapter we saw that the single layer perceptron was unable to classify points as being inside or outside a circle centered around the origin. This was because the output of the model (the logits) only had connections directly from the input features.

Why exactly is this a limitation? Think about how the connections work in a single layer neural network. The neurons in the output layer have a connection coming in from each of the neurons in the input layer, but the connection weights are all just real numbers.



The single layer neural network architecture. The weight values are denoted w1, w2, and w3.

Based on the diagram, the logits can be calculated as a linear combination of the input layer and weights:

$$logits = w_1 \cdot x + w_2 \cdot y + w_3$$

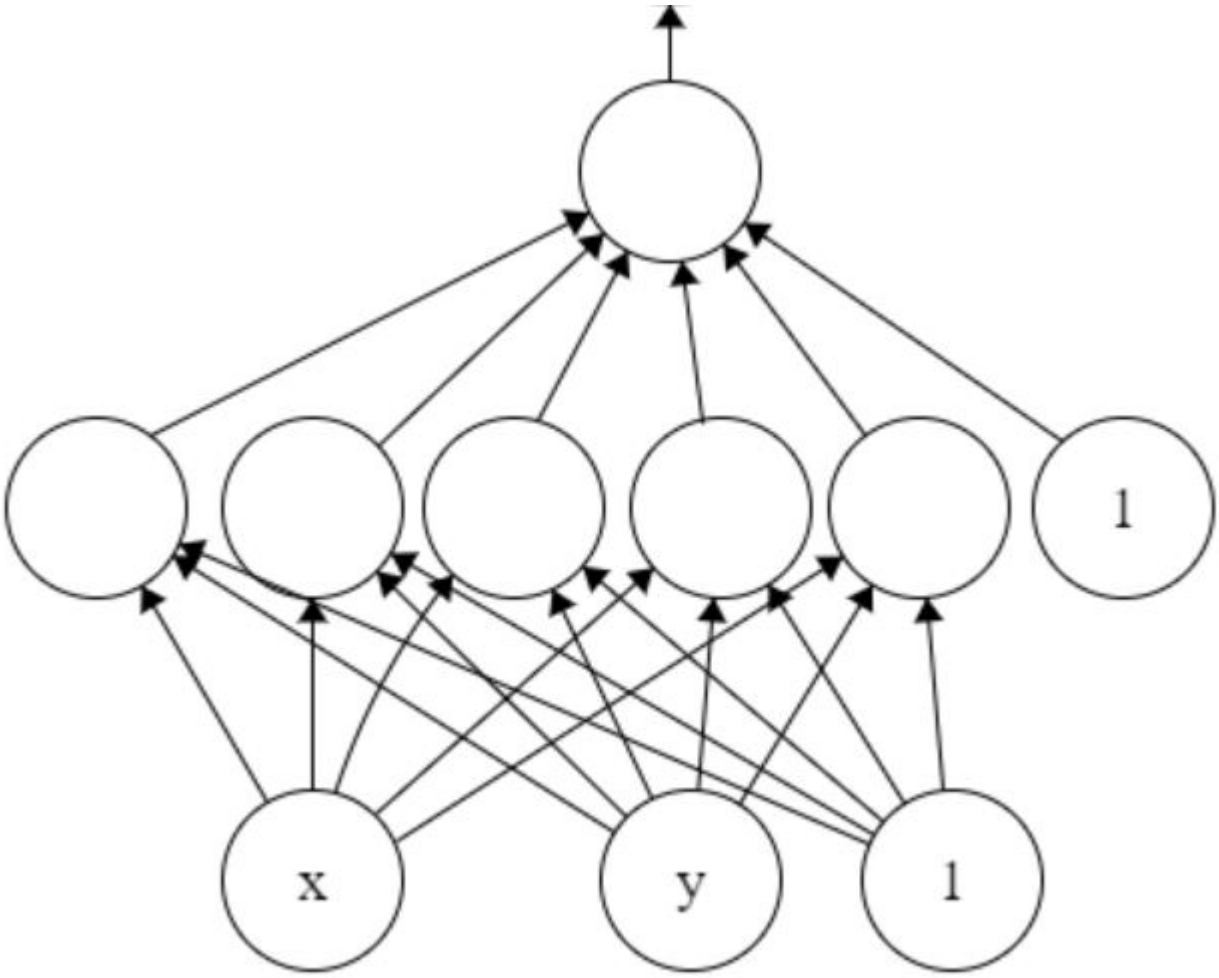
The above linear combination shows the single layer perceptron can model any linear boundary. However, the equation of the circle boundary is:

$$x^2 + y^2 = r^2$$

This is not an equation that can be modeled by a single linear combination.

B. Hidden layers

If a single linear combination doesn't work, what can we do? The answer is to add more linear combinations, as well as *non-linearities*. We add more linear combinations to our model by adding more layers. The single layer perceptron has only an input and output layer. We will now add an additional *hidden layer* between the input and output layers, officially making our model a multilayer perceptron. The hidden layer will have 5 neurons, which means that it will add an additional 5 linear combinations to the model's computation.



The multilayer perceptron architecture.

The 5 neuron hidden layer is more than enough for our circle example. However, for very complex datasets a model could have multiple hidden layers with hundreds of neurons per layer. In the next chapter, we discuss some tips on choosing the number of neurons and hidden layers in a model.

C. Non-linearity

We add non-linearities to our model through *activation functions*. These are non-linear functions that are applied within the neurons of a hidden layer. You've already seen an example of a non-linear activation function, the sigmoid function. We used this after the output layer of our model to convert the logits to probabilities.

The 3 most common activation functions used in deep learning are tanh, ReLU, and the aforementioned sigmoid. Each has its uses in deep learning, so it's normally best to choose activation functions based on the problem. However, the ReLU activation function has been shown to work well in most general-purpose situations, so we'll apply ReLU activation for our hidden

general-purpose situations, so we'll apply ReLU activation for our hidden layer.

D. ReLU

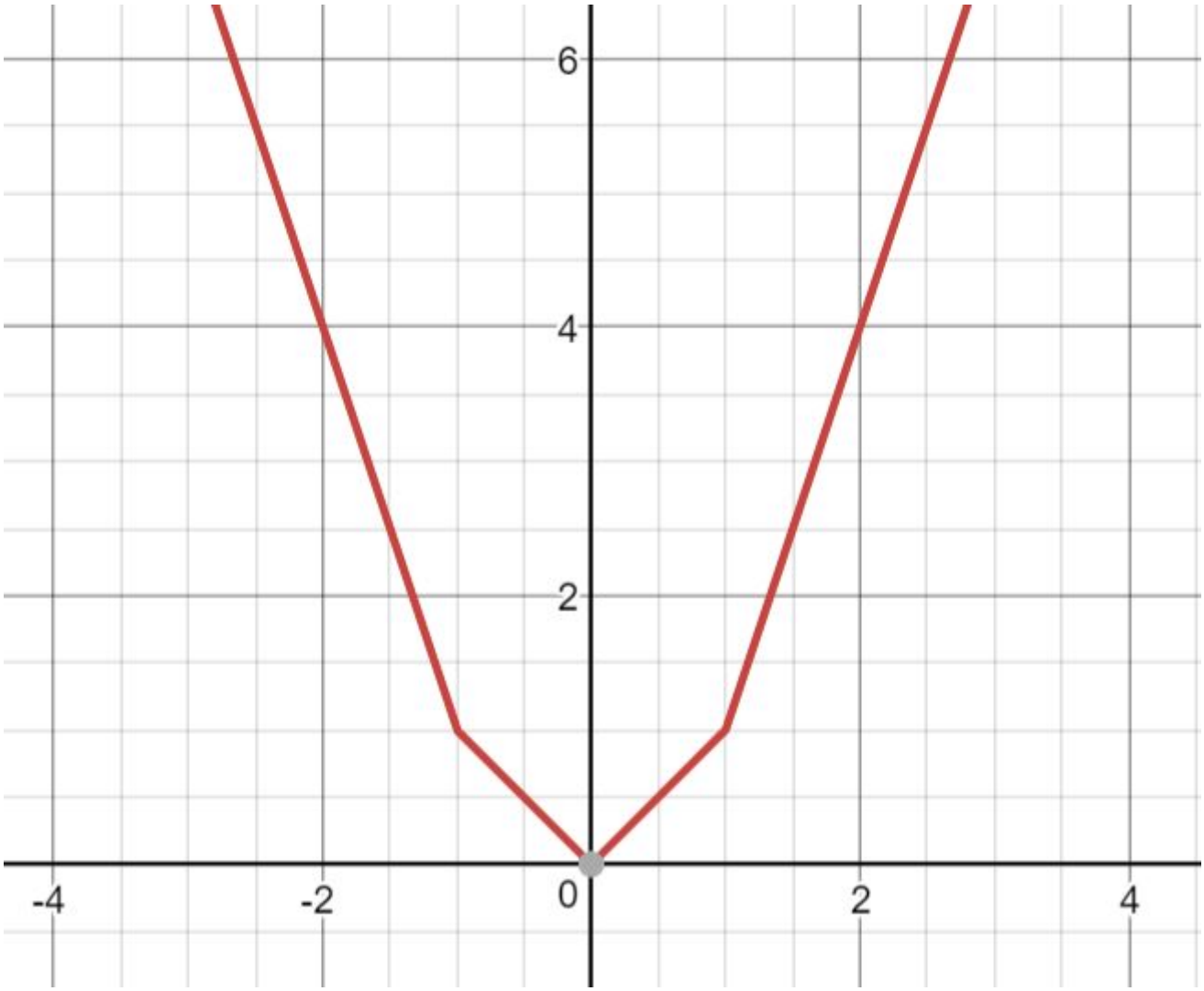
The equation for ReLU is very simple:

$$ReLU(x) = max(0, x)$$

You might wonder why ReLU even works. While tanh and sigmoid are both inherently non-linear, the ReLU function seems pretty linear (it's just $f(x) = 0$ for $x < 0$ and $f(x) = x$ for $x \geq 0$). However, let's take a look at the following function:

$$f(x) = ReLU(x) + ReLU(-x) + ReLU(2x - 2) + ReLU(-2x - 2)$$

This is just a linear combination of ReLU. However, the graph it produces looks like this:



Though a bit rough on the edges, it looks somewhat like the quadratic function, $f(x) = x^2$. In fact, with enough linear combinations and ReLU

function, $f(x) = x^2$. In fact, with enough linear combinations and ReLU

activations, a model can easily learn the quadratic transformation. This is exactly how our multilayer perceptron can learn the circle decision boundary.

We've shown that ReLU is capable of being a non-linear activation function. So what makes it work well in general purpose situations? Its aforementioned simplicity. The simplicity of ReLU, specifically with respect to its gradient, allows it to avoid the vanishing gradient problem. Furthermore, the fact that it maps all negative values to 0 actually helps the model train faster and avoid overfitting (discussed in the next chapter).

Time to Code!

The coding exercise for this chapter involves modifying the `model_layers` function from Chapter 3. You will be adding an additional hidden layer to the model, to change it from a single layer to a multilayer perceptron.

The additional hidden layer will go directly before the `'logits'` layer.

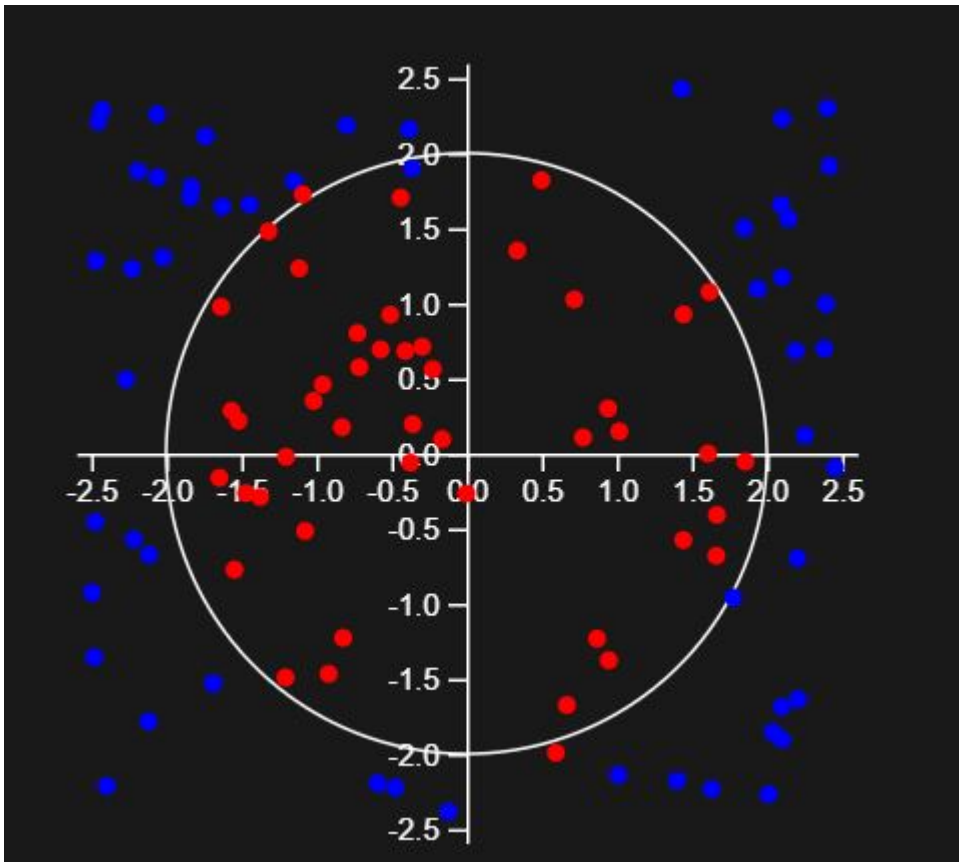
Set `hidden1` equal to `tf.layers.dense` with required arguments `inputs` and `5`, as well as keyword arguments `activation=tf.nn.relu` and `name='hidden1'`.

We also need to update the layer which produces the logits, so that it takes in `hidden1` as input.

Change the first argument of `tf.layers.dense` for `logits` to be `hidden1`.

```
def model_layers(inputs, output_size):
  logits = tf.layers.dense(inputs, output_size,
                           name='logits')
  return logits
```

After adding in the hidden layer to the model, the multilayer perceptron will be able to classify the 2-D circle dataset. Specifically, the classification plot will now look like:



Blue represents points the model thinks is outside the circle and red represents points the model thinks is inside. As you can see, the model is a lot more accurate now.