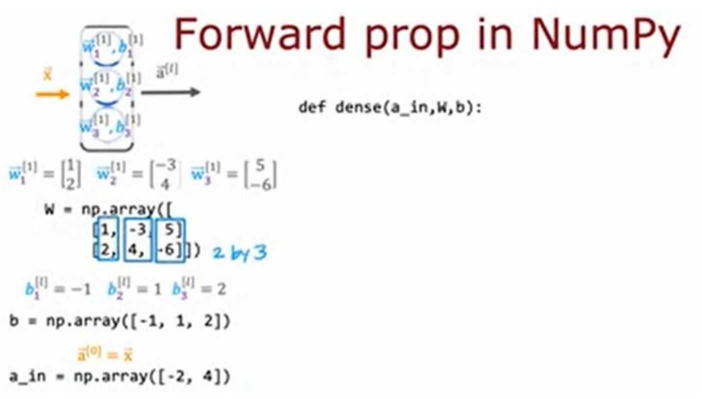
General implementation of forward propagation

In the last video, you saw how to implement forward prop in Python, but by hard coding lines of code for every single neuron. Let's now take a look at the more general implementation of forward prop in Python. Similar to the previous video, my goal in this video is to show you the code so that when you see it again in their practice lab, in the optional labs, you know how to interpret it.

As we walk through this example, don't worry about taking notes on every single line of code. If you can read through the code and understand it, that's definitely enough. What you can do is write a function to implement a dense layer, that is a single layer of a neural network. I'm going to define the dense function, which takes as input the activation from the previous layer, as well as the parameters w and b for the neurons in a given layer.


Forward prop in NumPy

$\vec{w}_1^{[1]} = \begin{bmatrix}1\\2\end{bmatrix}$ $\vec{w}_2^{[1]} = \begin{bmatrix}-3\\4\end{bmatrix}$ $\vec{w}_3^{[1]} = \begin{bmatrix}5\\-6\end{bmatrix}$

```
W = np.array([
    [1, -3, 5]
    [2, 4, -6]])
```

2 by 3

$b_1^{[l]} = -1$ $b_2^{[l]} = 1$ $b_3^{[l]} = 2$

```
b = np.array([-1, 1, 2])
```

$\vec{a}^{[0]} = \vec{x}$

```
a_in = np.array([-2, 4])
```

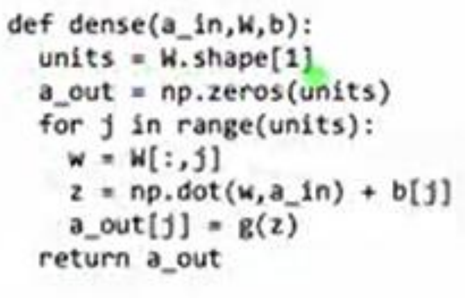
```
def dense(a_in,W,b):
```

$\vec{x}$
$\vec{w}_1^{[1]}, b_1^{[1]}$
$\vec{w}_2^{[1]}, b_2^{[1]}$
$\vec{w}_3^{[1]}, b_3^{[1]}$
$\vec{a}^{[l]}$

Using the example from the previous video, if layer 1 has three neurons, and if w_1 and w_2 and w_3 are these, then what we'll do is stack all of these wave vectors into a matrix. This is going to be a two by three matrix, where the first column is the parameter w_1,1 the second column is the parameter w_1, 2, and the third column is the parameter w_1,3.

Then in a similar way, if you have parameters be, b_1,1 equals negative one, b_1,2 equals one, and so on, then we're going to stack these three numbers into a 1D array b as follows, negative one, one, two. What the dense function will do is take as inputs the activation from the previous layer, and a here could be a_0, which is equal to x, or the activation from a later layer, as well as the w parameters stacked in columns, like shown on the right, as well as the b parameters also stacked into a 1D array, like shown to the left over there.

What this function would do is input a to activation from the previous layer and will output the activations from the current layer. Let's step through the code for doing this. Here's the code. First, units equals W.shape,1.



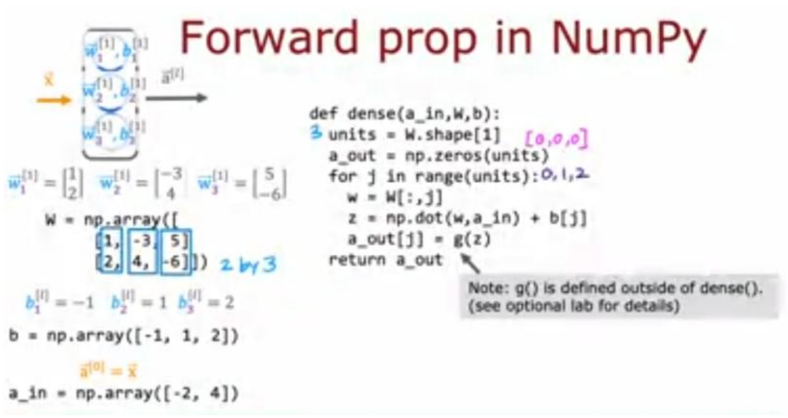
```
def dense(a_in,W,b):
  units = W.shape[1]
  a_out = np.zeros(units)
  for j in range(units):
    w = W[:,j]
    z = np.dot(w,a_in) + b[j]
    a_out[j] = g(z)
  return a_out
```

W here is a two-by-three matrix, and so the number of columns is three. That's equal to the number of units in this layer. Here, units would be equal to three. Looking at the shape of w, is just a way of pulling out the number of hidden units or the number of units in this layer.

Next, we set a to be an array of zeros with as many elements as there are units. In this example, we need to output three activation values, so this just initializes a to be zero, zero, zero, an array of three zeros. Next, we go through a for loop to compute the first, second, and third elements of a.

For j in range units, so j goes from zero to units minus one. It goes from 0, 1, 2 indexing from zero and Python as usual. This command w equals W colon comma j, this is how you pull out the jth column of a matrix in Python. The first time through this loop, this will pull the first column of w, and so will pull out w_1,1.

The second time through this loop, when you're computing the activation of the second unit, will pull out the second column corresponding to w_1, 2, and so on for the third time through this loop. Then you compute z using the usual formula, is a dot product between that parameter w and the activation that you have received, plus b, j. And then you compute the activation a, j, equals g sigmoid function applied to z.



Forward prop in NumPy

$\vec{w}_1^{[1]} = \begin{bmatrix}1\\2\end{bmatrix}$ $\vec{w}_2^{[1]} = \begin{bmatrix}-3\\4\end{bmatrix}$ $\vec{w}_3^{[1]} = \begin{bmatrix}5\\-6\end{bmatrix}$

```
W = np.array([
        [1, -3, 5]
        [2, 4, -6]])  2 by 3
```

$b_1^{[1]} = -1$ $b_2^{[1]} = 1$ $b_3^{[1]} = 2$

```
b = np.array([-1, 1, 2])
```

$\vec{a}^{[0]} = \vec{x}$

```
a_in = np.array([-2, 4])
```

```
def dense(a_in,W,b):
3 units = W.shape[1]   [0,0,0]
  a_out = np.zeros(units)
  for j in range(units):0,1,2
    w = W[:,j]
    z = np.dot(w,a_in) + b[j]
    a_out[j] = g(z)
  return a_out
```

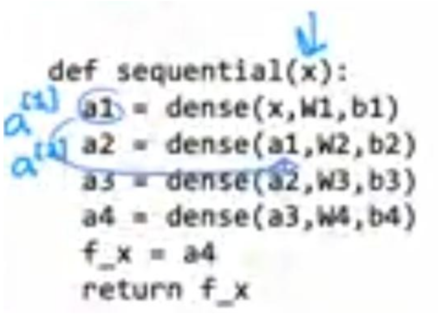
Note: g() is defined outside of dense().
(see optional lab for details)

$\vec{x}$
$\vec{w}_1^{[1]}, b_1^{[1]}$
$\vec{w}_2^{[1]}, b_2^{[1]}$
$\vec{w}_3^{[1]}, b_3^{[1]}$
$\vec{a}^{[1]}$

Three times through this loop and you compute it, the values for all three values of this vector of activation is a. Then finally you return a. What the dense function does is it inputs the activations from the previous layer, and given the parameters for the current layer, it returns the activations for the next layer.

Given the dense function, here's how you can string together a few dense layers sequentially, in order to implement forward prop in the neural network. Given the input features x, you can then compute the activations a_1 to be a_1 equals dense of x, w_1, b_1, where here w_1, b_1 are the parameters, sometimes also called the weights of the first hidden layer.

Then you can compute a_2 as dense of now a_1, which you just computed above. W_2, b-2 which are the parameters or weights of this second hidden layer.



```
def sequential(x):
    a1 = dense(x,W1,b1)
    a2 = dense(a1,W2,b2)
    a3 = dense(a2,W3,b3)
    a4 = dense(a3,W4,b4)
    f_x = a4
    return f_x
```

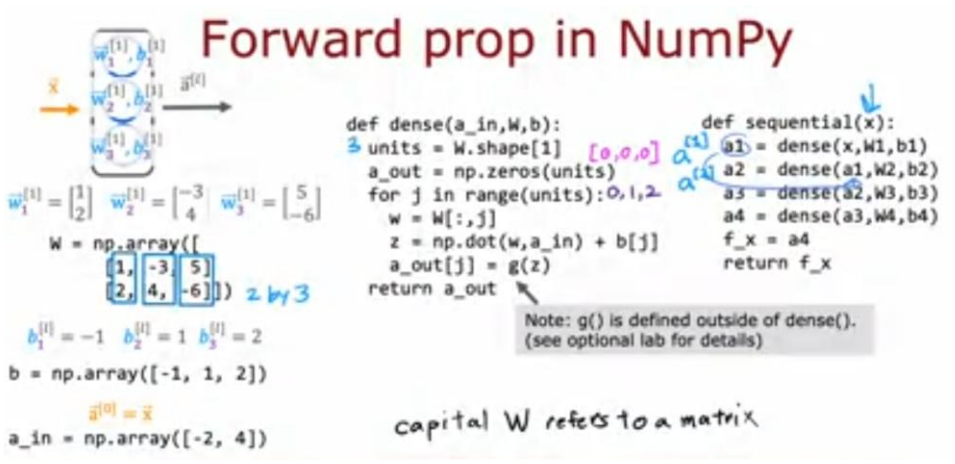
Then compute a_3 and a_4. If this is a neural network with four layers, then define the output f of x is just equal to a_4, and so you return f of x. Notice that here I'm using W, because under the notational conventions from linear algebra is to use uppercase or a capital alphabet is when it's referring to a matrix and lowercase refer to vectors and scalars.

So because it's a matrix, this is W. That's it. You now know how to implement forward prop yourself from scratch. You get to see all this code and run it and practice it yourself in the practice lab coming off to this as well.

I think that even when you're using powerful libraries like TensorFlow, it's helpful to know how it works under the hood. Because in case something goes wrong, in case something runs really slowly, or you have a strange result, or it looks like there's a bug, your ability to understand what's actually going on will make you much more effective when debugging your code.

When I run machine learning algorithms a lot of the time, frankly, it doesn't work. Sophie, not the first time. I find that my ability to debug my code to be a TensorFlow code or something else, is really important to being an effective machine learning engineer.

Even when you're using TensorFlow or some other framework, I hope that you find this deeper understanding useful for your own applications and for debugging your own machine learning algorithms as well. That's it. That's the last required video of this week with code in it.



Forward prop in NumPy

$\vec{w}_1^{[1]} = \begin{bmatrix}1\\2\end{bmatrix}$ $\vec{w}_2^{[1]} = \begin{bmatrix}-3\\4\end{bmatrix}$ $\vec{w}_3^{[1]} = \begin{bmatrix}5\\-6\end{bmatrix}$

```
W = np.array([
        [1, -3, 5]
        [2, 4, -6]])
```

2 by 3

$b_1^{[l]} = -1$ $b_2^{[l]} = 1$ $b_3^{[l]} = 2$

```
b = np.array([-1, 1, 2])
```

$\vec{a}^{[0]} = \vec{x}$

```
a_in = np.array([-2, 4])
```

```
def dense(a_in,W,b):
  units = W.shape[1]
  a_out = np.zeros(units)
  for j in range(units):
    w = W[:,j]
    z = np.dot(w,a_in) + b[j]
    a_out[j] = g(z)
  return a_out
```

```
def sequential(x):
  a1 = dense(x,W1,b1)
  a2 = dense(a1,W2,b2)
  a3 = dense(a2,W3,b3)
  a4 = dense(a3,W4,b4)
  f_x = a4
  return f_x
```

Note: g() is defined outside of dense().
(see optional lab for details)

capital W refers to a matrix

$\vec{x}$
$\vec{w}_1^{[1]}, b_1^{[1]}$
$\vec{w}_2^{[1]}, b_2^{[1]}$
$\vec{w}_3^{[1]}, b_3^{[1]}$
$\vec{a}^{[l]}$
3
[0,0,0]
0, 1, 2
$a^{[1]}$
$a^{[2]}$

In the next video, I'd like to dive into what I think is a fun and fascinating topic, which is, what is the relationship between neural networks and AI or AGI, artificial general intelligence? This is a controversial topic, but because it's been so widely discussed, I want to share with you some thoughts on this. When you are asked, are neural networks at all on the path to human level intelligence? You have a framework for thinking about that question. Let's go take a look at that fun topic, I think, in the next video.