**Implementing the hidden layer**

**Prerequisites**

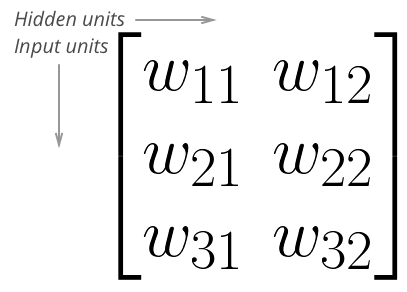
Below, we are going to walk through the math of neural networks in a multilayer perceptron. With multiple perceptron, we are going to move to using vectors and matrices. To brush up, be sure to view the following:

1. Khan Academy's [**introduction to vectors**](https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/vectors/v/vector-introduction-linear-algebra).
2. Khan Academy's [**introduction to matrices**](https://www.khanacademy.org/math/precalculus/precalc-matrices).

**Derivation**

Before, we were dealing with only one output node which made the code straightforward. However now we have multiple input units and multiple hidden units, the weights between them now require two indices: *w*​*ij*​​ where *i* denotes input units and *j* are the hidden units. Before, we were able to write *w*​*i*​​ as an array.

But now, *w*​*ij*​​ is a *matrix*. If we have three input units and two hidden units, the weights matrix looks like this:



Weights matrix for 3 input units and 2 hidden units

To initialize these weights in Numpy, we have to provide the shape of the matrix. If features is a 2D array containing the input data:

*# Number of records and input units*

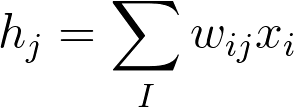
n\_records, n\_inputs = features.shape

*# Number of hidden units*

n\_hidden = 2

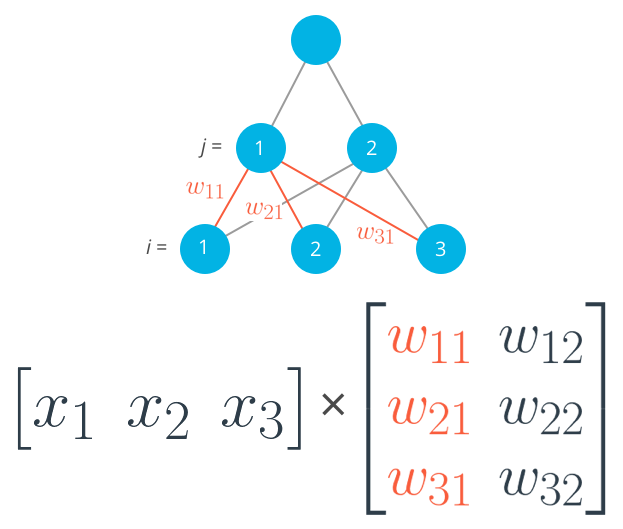
weights = np.random.normal(0, n\_inputs\*\*-0.5, shape=(n\_inputs, n\_hidden))

This creates a 2D array weights with dimensions n\_inputs by n\_hidden. Now, if we want to calculate the inputs to the hidden layer *h*​*j*​​:



we need to use [**matrix multiplication**](https://en.wikipedia.org/wiki/Matrix_multiplication). If your linear algebra is rusty, I suggest taking a look at the suggested resources in the prerequisites section. For this part though, you'll only need to know how to multiply a matrix with a vector.

In this case, we're multiplying the inputs (a row vector here) by the weights. To do this, you take the dot (inner) product of the inputs with each column in the weights matrix. For example, to calculate the input to the first hidden unit, *j*=1, you'd take the dot product of the inputs with the first column of the weights matrix, like so:



Calculating the input to the first hidden unit with the first column of the weights matrix.

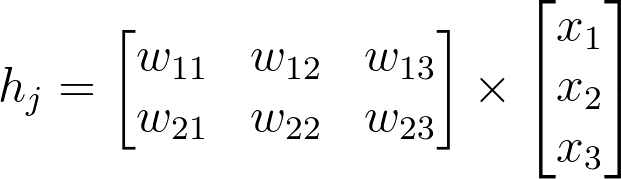
https://d17h27t6h515a5.cloudfront.net/topher/2017/January/588ae392_codecogseqn-2/codecogseqn-2.png

And for the second hidden layer input, you calculate the dot product of the inputs with the second column. And so on and so forth.

In Numpy, again, you do this with np.dot

hidden\_inputs = np.dot(inputs, weights\_input\_to\_hidden)

You could also define your weights matrix such that it has dimensions n\_hidden by n\_inputs then multiply like so where the inputs form a *column vector*:



The important thing is that *the dimensions match*. For the matrix multiplication to work, there has to be the same number of elements in the dot products. In the first example, there are three columns in the input vector, and three rows in the weights matrix. In the second example, there are three columns in the weights matrix and three rows in the input vector. If the dimensions don't match, you'll get this:

# Same weights and features as above, but swapped the order

hidden\_inputs = np.dot(weights\_in\_hidden, features)

---------------------------------------------------------------------------

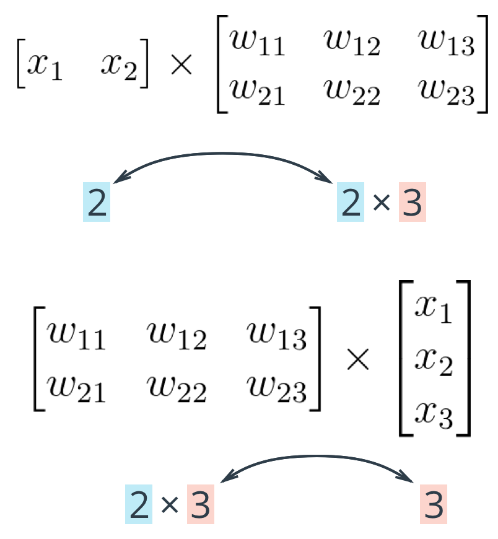
ValueError Traceback (most recent call last)

<ipython-input-11-1bfa0f615c45> in <module>()

----> 1 hidden\_in = np.dot(weights\_in\_hidden, X)

ValueError: shapes (3,2) and (3,) not aligned: 2 (dim 1) != 3 (dim 0)

The dot product can't be computed for a 3x2 matrix and 3-element array. That's because the 2 columns in the matrix don't match the number of elements in the array. Some of the dimensions that could work would be the following:



The rule is that if you're multiplying an array from the left, the array must have the same number of elements as there are rows in the matrix. And if you're multiplying the *matrix* from the left, the number of columns in the matrix must equal the number of elements in the array on the right.

**Making a column vector**

You see above that sometimes you'll want a column vector, even though by default Numpy arrays work like row vectors. It's possible to get the transpose of an array like so arr.T, but for a 1D array, the transpose will return a row vector. Instead, use arr[:,None] to create a column vector:

print(features)

> array([ 0.49671415, -0.1382643 , 0.64768854])

print(features.T)

> array([ 0.49671415, -0.1382643 , 0.64768854])

print(features[:, **None**])

> array([[ 0.49671415],

[-0.1382643 ],

[ 0.64768854]])

Alternatively, you can create arrays with two dimensions. Then, you can use arr.T to get the column vector.

np.array(features, ndmin=2)

> array([[ 0.49671415, -0.1382643 , 0.64768854]])

np.array(features, ndmin=2).T

> array([[ 0.49671415],

[-0.1382643 ],

[ 0.64768854]])

I personally prefer keeping all vectors as 1D arrays, it just works better in my head.

**Programming quiz**

Below, you'll implement a forward pass through a 4x3x2 network, with sigmoid activation functions for both layers.

Things to do:

* Calculate the input to the hidden layer.
* Calculate the hidden layer output.
* Calculate the input to the output layer.
* Calculate the output of the network.