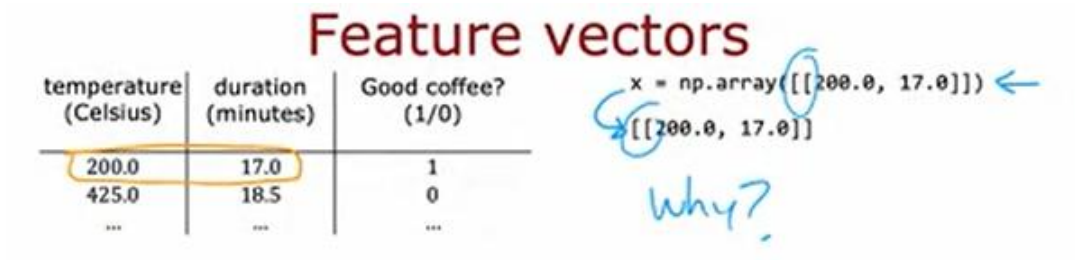
Data in TensorFlow

In this video, I want to step through with you how data is represented in NumPy and in TensorFlow. So that as you're implementing new neural networks, you can have a consistent framework to think about how to represent your data. One of the unfortunate things about the way things are done in code today is that many, many years ago NumPy was first created and became a standard library for linear algebra and Python.

And then much later the Google brain team, the team that I had started and once led created TensorFlow. And so unfortunately there are some inconsistencies between how data is represented in NumPy and in TensorFlow. So it's good to be aware of these conventions so that you can implement correct code and hopefully get things running in your neural networks. Let's start by taking a look at how TensorFlow represents data.

Let's see you have a data set like this from the coffee example. I mentioned that you would write x as follows. So why do you have this double square bracket here? Let's take a look at how NumPy stores vectors and matrices. In case you think matrices and vectors are complicated mathematical concepts don't worry about it.

We'll go through a few concrete examples and you'll be able to do everything you need to do with matrices and vectors in order to implement your networks. Let's start with an example of a matrix.

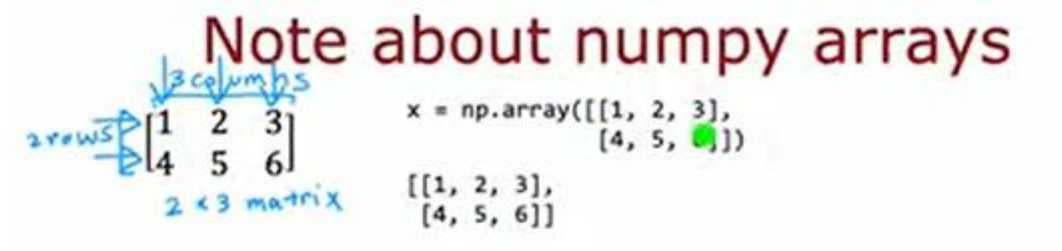


Feature vectors

temperature (Celsius)	duration (minutes)	Good coffee? (1/0)
200.0	17.0	1
425.0	18.5	0
...	...	...

```
x = np.array([[200.0, 17.0]])
[[200.0, 17.0]]
```

Here is a matrix with 2 rows and 3 columns. Notice that there are one, two rows and 1, 2, 3 columns. So we call this a 2 x 3 matrix. And so the convention is the dimension of the matrix is written as the number of rows by the number of columns. So in code to store this matrix, this 2 x 3 matrix, you just write x = np.array of these numbers like these. Where you notice that the square bracket tells you that 1, 2, 3 is the first row of this matrix and 4, 5, 6 is the second row of this matrix. And then this open square bracket groups the first and the second row together.

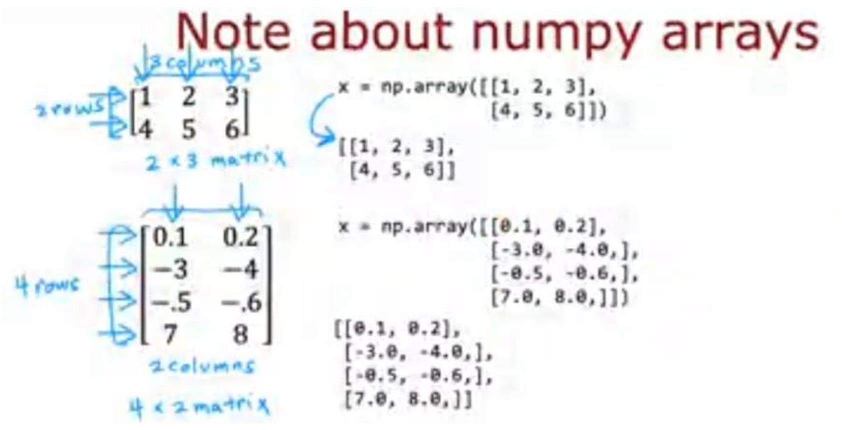


Note about numpy arrays

$$\begin{array}{c|ccc} & \multicolumn{3}{c}{\text{3 columns}} \\ \text{2 rows} & 1 & 2 & 3 \\ & 4 & 5 & 6 \end{array}$$

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

So this sets x to be this to the array of numbers. So matrix is just a 2D array of numbers. Let's look at one more example, here I've written out another matrix. How many roles and how many columns does this have? Well, you can count this as one, two, three, four rows and it has one, two columns. So this is a number of rows by the number of columns matrix, so it's a 4 x 2 matrix. And so to store this in code, you will write x equals np.array and then this syntax over here to store these four rows of matrix in the variable x.



Note about numpy arrays

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}$$

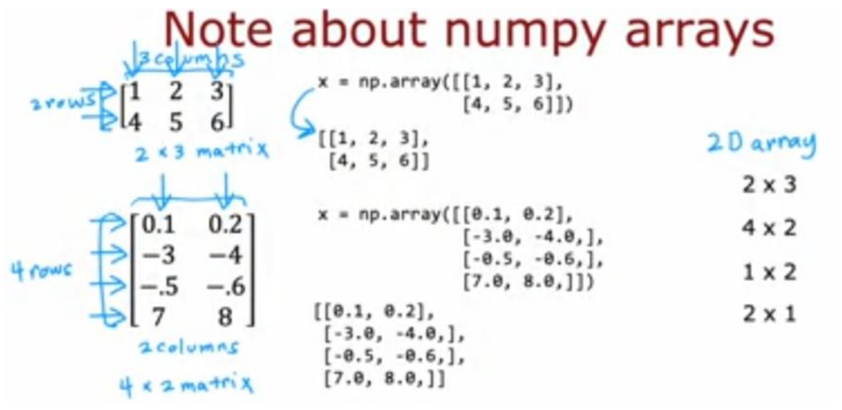
```
x = np.array([[1, 2, 3],
              [4, 5, 6]])
[[1, 2, 3],
 [4, 5, 6]]
```

$$\begin{bmatrix}0.1 & 0.2\\-3 & -4\\-.5 & -.6\\7 & 8\end{bmatrix}$$

```
x = np.array([[0.1, 0.2],
              [-3.0, -4.0,],
              [-0.5, -0.6,],
              [7.0, 8.0,]])
[[0.1, 0.2],
 [-3.0, -4.0,],
 [-0.5, -0.6,],
 [7.0, 8.0,]]
```

So this creates a 2D array of these eight numbers. Matrices can have different dimensions. You saw an example of an 2 x 3 matrix and the 4 x 2 matrix. A matrix can also be other dimensions like 1 x 2 or 2 x 1. And we'll see examples of these on the next slide.

So what we did previously when setting x to be input feature vectors, was set x to be equal to np.array with two square brackets, 200, 17. And what that does is this creates a 1 x 2 matrix, that is just one row and two columns.



Note about numpy arrays

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}$$

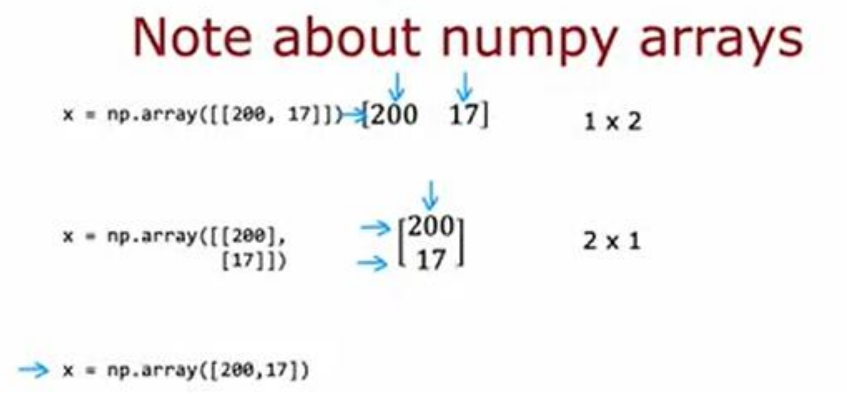
```
x = np.array([[1, 2, 3],
              [4, 5, 6]])
[[1, 2, 3],
 [4, 5, 6]]
```

$$\begin{bmatrix}0.1 & 0.2\\-3 & -4\\-.5 & -.6\\7 & 8\end{bmatrix}$$

```
x = np.array([[0.1, 0.2],
              [-3.0, -4.0,],
              [-0.5, -0.6,],
              [7.0, 8.0,]])
[[0.1, 0.2],
 [-3.0, -4.0,],
 [-0.5, -0.6,],
 [7.0, 8.0,]]
```

2 x 3
4 x 2
1 x 2
2 x 1

So this creates a 2D array of these eight numbers. Matrices can have different dimensions. You saw an example of an 2 x 3 matrix and the 4 x 2 matrix. A matrix can also be other dimensions like 1 x 2 or 2 x 1. And we'll see examples of these on the next slide. So what we did previously when setting x to be input feature vectors, was set x to be equal to np.array with two square brackets, 200, 17. And what that does is this creates a 1 x 2 matrix, that is just one row and two columns.



Note about numpy arrays

```
x = np.array([[200, 17]])
```

$$\begin{bmatrix}200 & 17\end{bmatrix}$$ 1 x 2

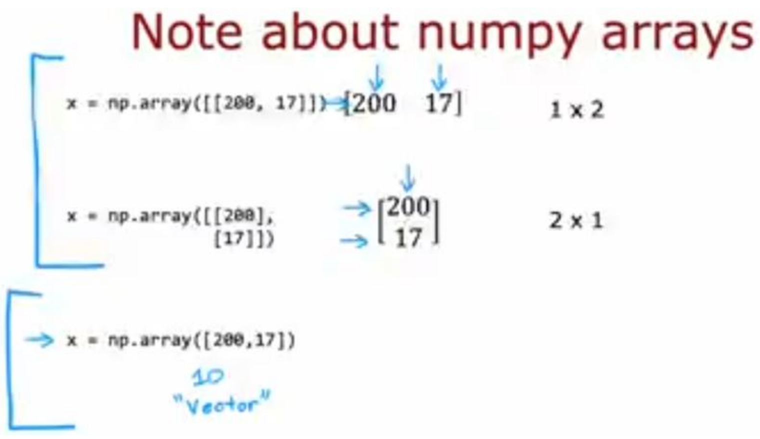
```
x = np.array([[200],
              [17]])
```

$$\begin{bmatrix}200\\17\end{bmatrix}$$ 2 x 1

```
x = np.array([200,17])
```

Let's look at a different example, if you were to define x to be np.array but now written like this, this creates a 2 x 1 matrix that has two rows and one column. Because the first row is just the number 200 and the second row, is just the number 17. And so this has the same numbers but in a 2 x 1 instead of a 1 x 2 matrix.

Enough this example on top is also called a row vector, is a vector that is just a single row. And this example is also called a column vector because this vector that just has a single column. And the difference between using double square brackets like this versus a single square bracket like this, is that whereas the two examples on top of 2D arrays where one of the dimensions happens to be 1. This example results in a 1D vector.

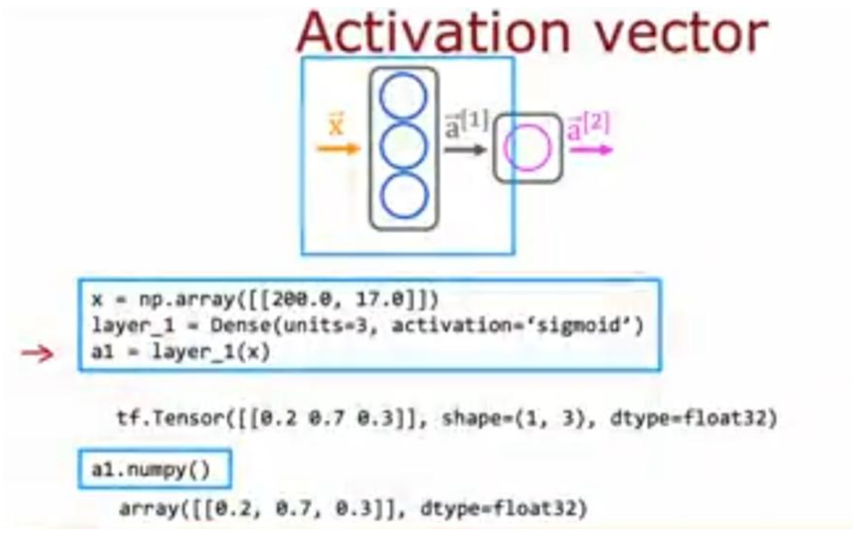


Note about numpy arrays

```
x = np.array([[200, 17]])     [200  17]     1 x 2

x = np.array([[200],          [200]         2 x 1
              [17]])          [ 17]

x = np.array([200,17])
```

So this is just a 1D array that has no rows or columns, although by convention we may right x as a column like this. So on a contrast this with what we had previously done in the first course, which was to write x like this with a single square bracket. And that resulted in what's called in Python, a 1D vector instead of a 2D matrix.

And this technically is not 1 x 2 or 2 x 1, is just a linear array with no rows or no columns, but it's just a list of numbers. So whereas in course one when we're working with linear regression and logistic regression, we use these 1D vectors to represent the input features x. With TensorFlow the convention is to use matrices to represent the data. And why is there this switching conventions?

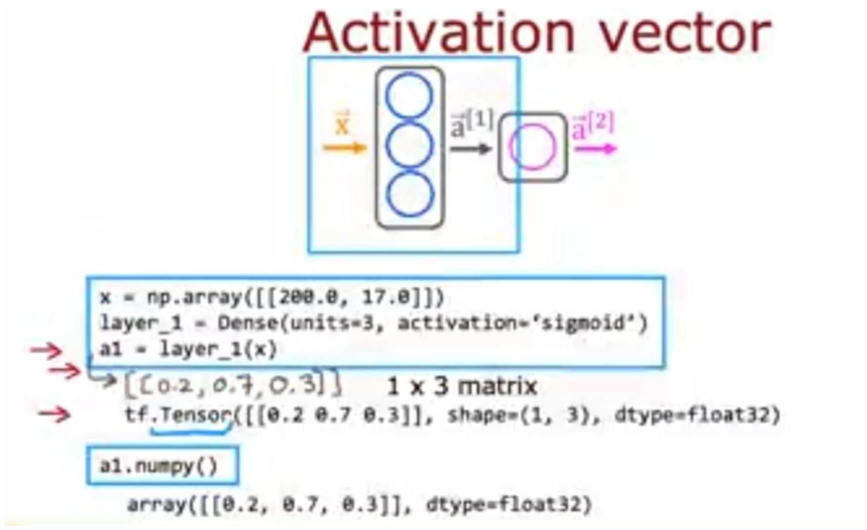


Activation vector

```
x = np.array([[200.0, 17.0]])
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)

tf.Tensor([[0.2 0.7 0.3]], shape=(1, 3), dtype=float32)

a1.numpy()

array([[0.2, 0.7, 0.3]], dtype=float32)
```

Well it turns out that TensorFlow was designed to handle very large datasets and by representing the data in matrices instead of 1D arrays, it lets TensorFlow be a bit more computationally efficient internally. So going back to our original example for the first training, example in this dataset with features 200°C in 17 minutes, we were represented like this.

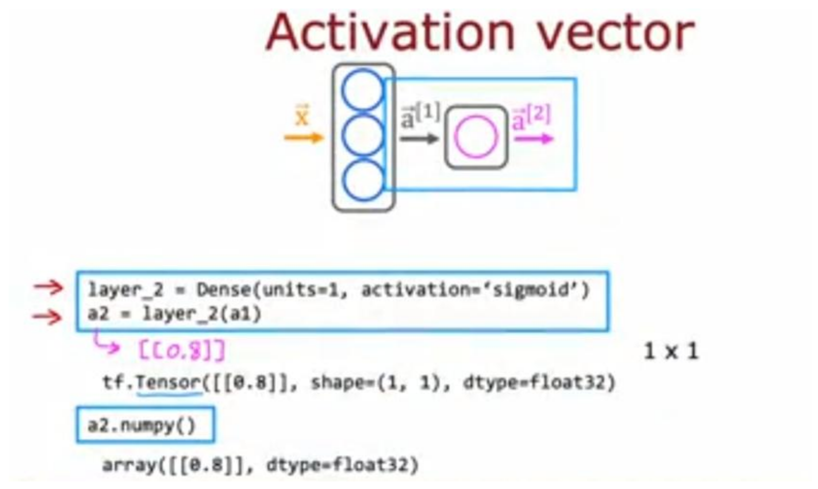
And so this is actually a 1 x 2 matrix that happens to have one row and two columns to store the numbers 217. And in case this seems like a lot of details and really complicated conventions, don't worry about it all of this will become clearer. And you get to see the concrete implementations of the code yourself in the optional labs and in the practice labs. Going back to the code for carrying out for propagation or influence in the neural network. When you compute a1 equals layer 1 applied to x, what is a1?



```
x = np.array([[200.0, 17.0]])
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)
  [[0.2, 0.7, 0.3]]
  tf.Tensor([[0.2 0.7 0.3]], shape=(1, 3), dtype=float32)
a1.numpy()
  array([[0.2, 0.7, 0.3]], dtype=float32)
```

Well, a1 is actually going to be because the three numbers, is actually going to be a 1 x 3 matrix. And if you print out a1 you will get something like this is tf.tensor 0.2, 0.7, 0.3 as a shape of 1 x 3, 1, 3 refers to that this is a 1 x 3 matrix. And this is TensorFlow's way of saying that this is a floating point number meaning that it's a number that can have a decimal point represented using 32 bits of memory in your computer, that's where the float 32 is.

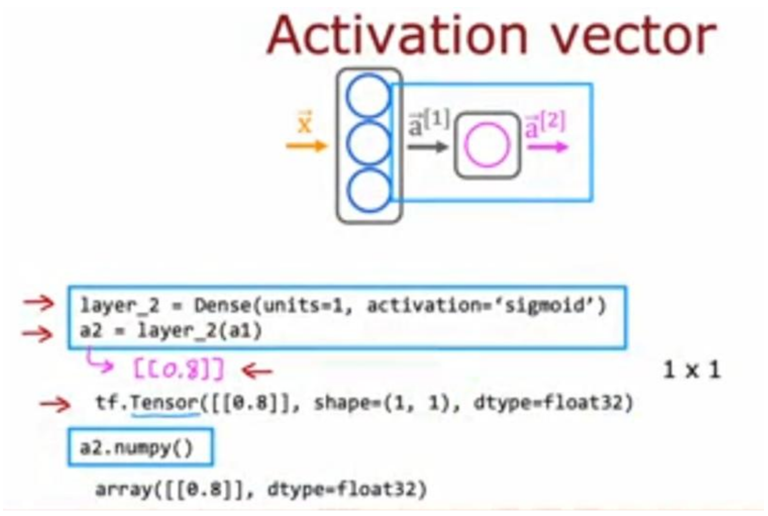
And what is the tensor? A tensor here is a data type that the TensorFlow team had created in order to store and carry out computations on matrices efficiently. So whenever you see tensor just think of that matrix on these few slides. Technically a tensor is a little bit more general than the matrix but for the purposes of this course, think of tensor as just a way of representing matrices.



```
layer_2 = Dense(units=1, activation='sigmoid')
a2 = layer_2(a1)
  [[0.8]]
  tf.Tensor([[0.8]], shape=(1, 1), dtype=float32)
a2.numpy()
  array([[0.8]], dtype=float32)
```

So remember I said at the start of this video that there's the TensorFlow way of representing the matrix and the NumPy way of representing matrix. This is an artifact of the history of how NumPy and TensorFlow were created and unfortunately there are two ways of representing a matrix that have been baked into these systems. And in fact if you want to take a1 which is a tensor and want to convert it back to NumPy array, you can do so with this function a1.numpy.

And it will take the same data and return it in the form of a NumPy array rather than in the form of a TensorFlow array or TensorFlow matrix. Now let's take a look at what the activations output the second layer would look like. Here's the code that we had from before, layer 2 is a dense layer with one unit and sigmoid activation and a2 is computed by taking layer 2 and applying it to a1 so what is a2? A2, maybe a number like 0.8 and technically this is a 1 x 1 matrix is a 2D array with one row and one column and so it's equal to this number 0.8.



```
layer_2 = Dense(units=1, activation='sigmoid')
a2 = layer_2(a1)

tf.Tensor([[0.8]], shape=(1, 1), dtype=float32)
a2.numpy()
array([[0.8]], dtype=float32)
```

And if you print out a2, you see that it is a TensorFlow tensor with just one element one number 0.8 and it is a 1 x 1 matrix. And again it is a float32, decimal points number taking up 32 bits in computer memory. Once again you can convert from a tensorflow tensor to a NumPy matrix using a2.numpy and that will turn this back into a NumPy array that looks like this.

So that hopefully gives you a sense of how data is represented in TensorFlow and in NumPy. I'm used to loading data and manipulating data in NumPy, but when you pass a NumPy array into TensorFlow, TensorFlow likes to convert it to its own internal format. The tensor and then operate efficiently using tensors. And when you read the data back out you can keep it as a tensor or convert it back to a NumPy array.

I think it's a bit unfortunate that the history of how these library evolved has let us have to do this extra conversion work when actually the two libraries can work quite well together. But when you convert back and forth, whether you're using a NumPy array or a tensor, it's just something to be aware of when you're writing code. Next let's take what we've learned and put it together to actually build a neural network.