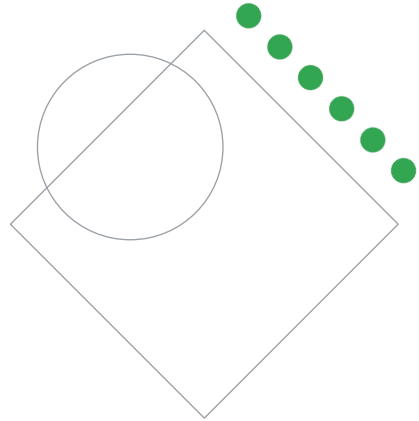


# Introduction to the TensorFlow Ecosystem



Welcome to the **Introduction to the TensorFlow Ecosystem** module.

## In this module, you learn to ...

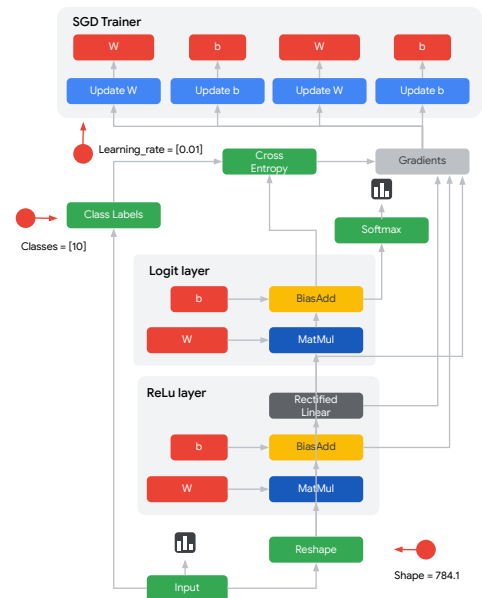
- |    |   |
|----|---|
| 01 | Recall the TensorFlow API hierarchy                             |
| 02 | Understand TensorFlow's building blocks: Tensors and operations |
| 03 | Write low-level TensorFlow programs                             |



In this module, you'll learn about:

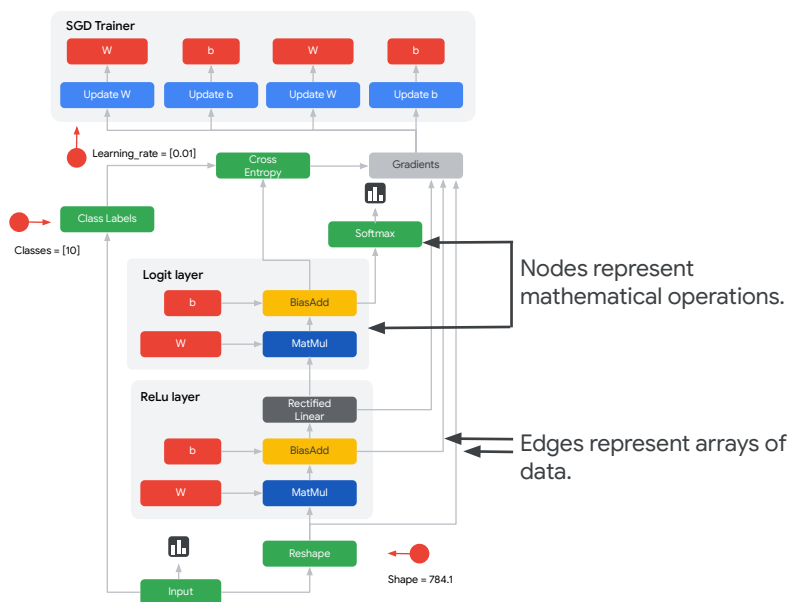
- The TensorFlow API hierarchy
- Building blocks of tensors and operations
- How to write low-level TensorFlow programs

TensorFlow is an  
open-source,  
high-performance library for  
numerical computation that  
uses directed graphs



TensorFlow is an open-source, high-performance library for numerical computation. ANY numeric computation -- Not just about machine learning. In fact, people have used TensorFlow for all kinds of GPU computing; for example, you can use TensorFlow to solve partial differential equations which is super useful in fields like fluid dynamics.

TensorFlow as a numeric programming library is appealing because you can write your computation code in a high-level language like Python and have it be executed in a fast way at run time.



The way TensorFlow works is that you create a directed graph (a DAG) to represent the computation you want to do.

In this schematic, *the nodes, like those light green circles, represent mathematical operations* -- things like adding, subtracting, and multiplying.

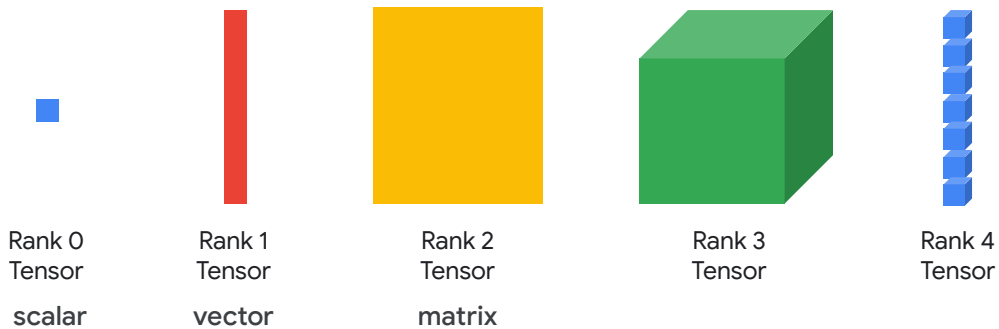
You'll also see some more complex math functions like softmax and matrix-multiplication which are great for machine learning.

Connecting the nodes are the edges which are the input and output of the mathematical operations. *The edges represent arrays of data flowing towards the output.*

Starting from the bottom are arrays of raw input data. Sometimes we will need to reshape it before feeding it into a layers of a neural network (like the ReLu layer here). More on ReLu later. Once inside that ReLu layer the weight is multiplied across the array of data in a MatMul or matrix multiplication. Then a bias term is added and the data flows through to the activation function.

Wow -- ReLu, activation functions? Don't worry, let's start with the basics. I kept mentioning an array of data flowing around -- what exactly does that mean?

## A tensor is an N-dimensional array of data



Well that's actually where TensorFlow gets its name from!

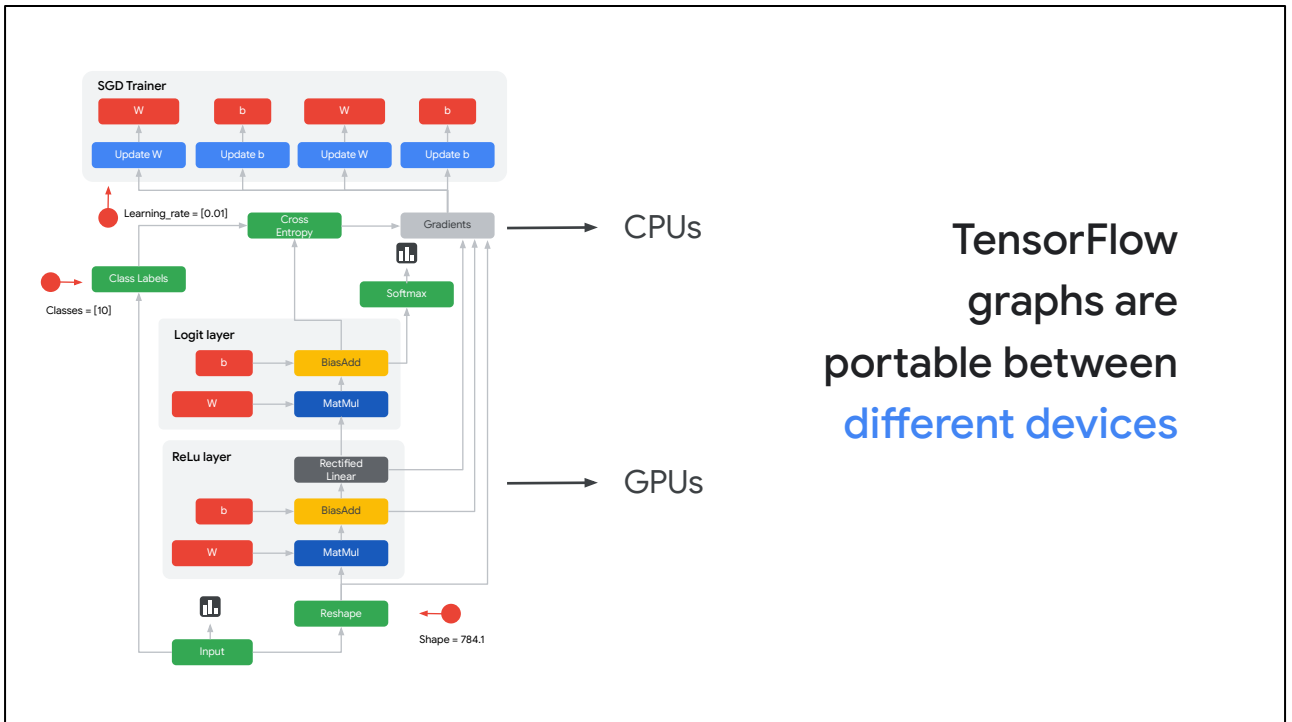
Starting on the far left:

- The simplest piece of data we can have is a **scalar**. That's a number like "3" or "5". It's what we call zero dimensional or Rank 0. We're not going to get very far passing around single numbers in our flow so let's upgrade.
- Rank 1 or 1 dimensional array is a **vector**. In physics, a vector is something with magnitude and direction. But in Computer Science, you use vector to mean 1D arrays like a series of numbers in a list.

Let's keep going.

- A two-dimensional array is a **matrix**.
- A three-dimensional array? We just call it a **3D tensor**. So scalar, vector, matrix, 3D Tensor, 4D Tensor, etc.

A tensor is an n-dimensional array of data. So, your data in TensorFlow are tensors. They flow through the graph. Hence the name TensorFlow.



So, why does TensorFlow use directed graphs to represent computation? The answer is portability.

The directed graph is a language-independent representation of the code in your model. You can build a DAG in Python, store it in a SavedModel, and restore it in a C++ program for low-latency predictions.

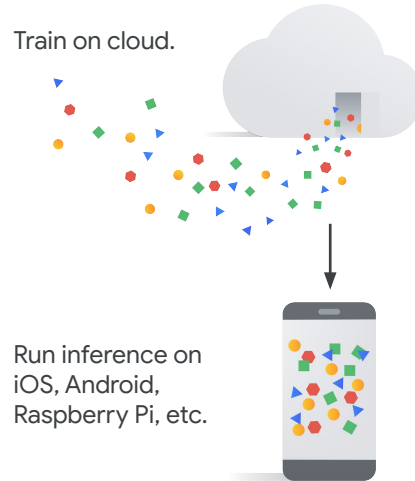
You can use the same Python code and execute it on both CPUs, GPUs, and TPUs. This provides language-and-hardware portability.

In a lot of ways, this is similar to how the Java Virtual Machine (JVM) and its bytecode representation helps the portability of Java code. As a developer, you get to write code in a high-level language (Java) and have it be executed in different platforms by the JVM. The JVM itself is very efficient and targeted toward the exact OS and hardware and written in C or C++.

It's a similar thing with TensorFlow. As a developer, you get to write code in a high-level language (Python) and have it be executed in different platforms by the TensorFlow execution engine. The TensorFlow execution engine is very efficient and targeted toward the exact hardware chip and its capabilities, and is written in C++.

**TensorFlow Lite provides on-device inference of ML models on mobile devices and is available for a variety of hardware.**

Announcing TensorFlow Lite:  
<https://developers.googleblog.com/2017/11/announcing-tensorflow-lite.html>



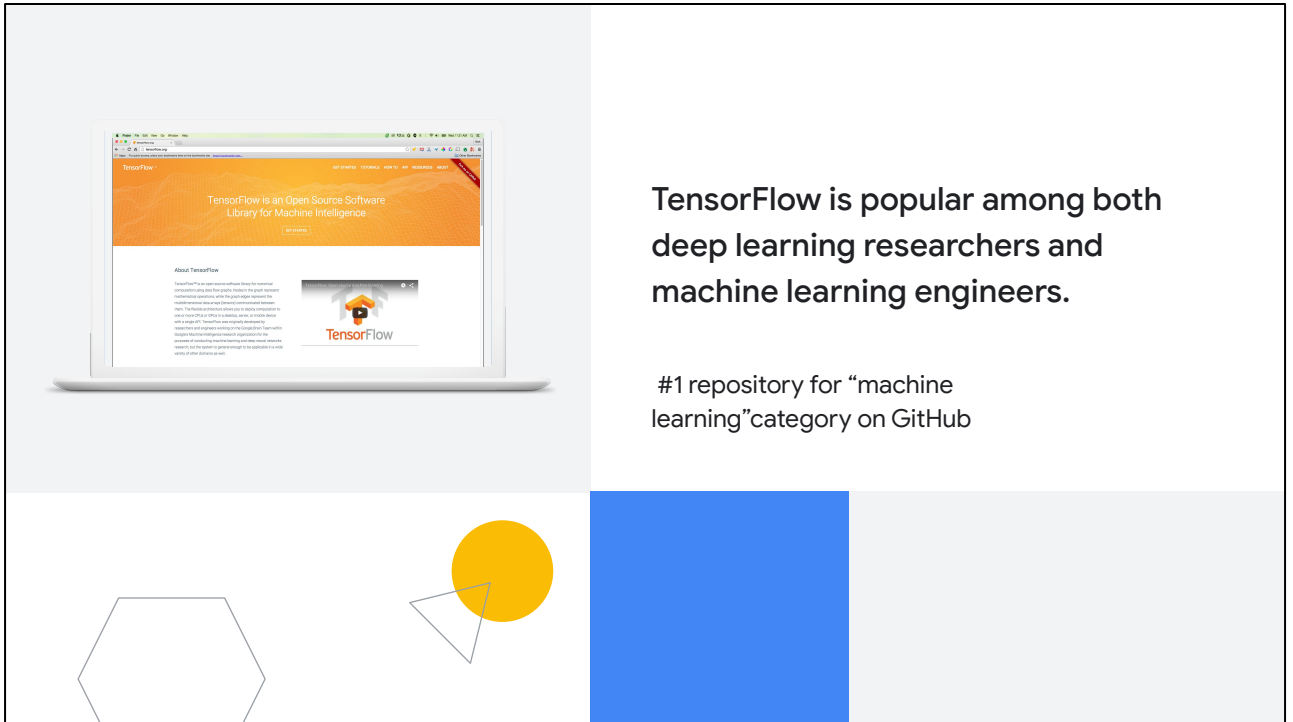
Portability between devices enables a lot of power and flexibility. For example, this is a common pattern. You can train a TensorFlow model on the cloud, on lots and lots of powerful hardware. Then, take the trained model and put it on a device out at the edge -- perhaps a mobile phone or even an embedded chip. You can then do predictions with the model right on that device itself.

Have you had a chance to use the Google Translate app on an Android phone?

That app can work completely offline because the trained translation model is stored on the phone and is available for offline translation.

Due to the limitations of the processing power available on phones, the edge model tends to be a smaller, therefore less powerful than what's on the cloud. However, the fact that TensorFlow allows for models to run on the edge propitiates a much faster response during predictions.

<https://developers.googleblog.com/2017/11/announcing-tensorflow-lite.html>



So, TensorFlow is this portable, powerful, production-ready software to do numerical computing. It's particularly popular for machine learning. The #1 repository for machine learning on GitHub. Why is it so popular?

It's popular among Deep Learning researchers because of the community around it, and the ability to extend it and do new, cool things.

It's popular among Machine Learning engineers because of the ability to productionize models, do things at scale.

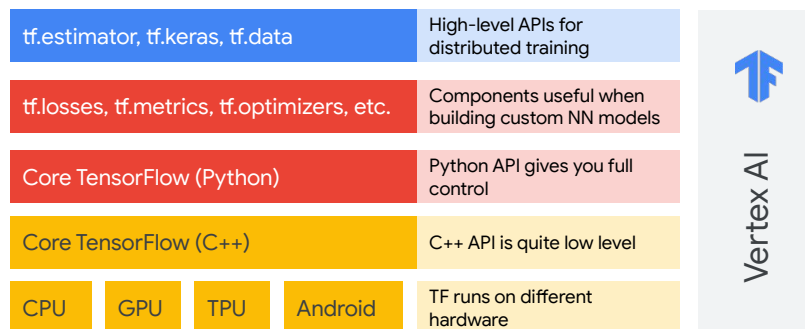


## TensorFlow API hierarchy

Let's take a look at the API hierarchy which will consist of a spectrum of low-level APIs for hardware all the way up to very abstract high-level APIs for super powerful tasks like creating a 128 layer neural network with just a few lines of code written with the Keras API.

Let's start at the bottom.

TensorFlow  
contains  
multiple  
abstraction  
layers.



Run TF at scale with AI Platform.

*Note that AI Platform is now Vertex AI.*

- The lowest level of abstraction is the layer that is implemented to target the different **hardware platforms**. Unless your company makes hardware, it's unlikely that you'll do much at this level but it does exist.
- The next level is the **TensorFlow C++ API**. This is how you can write a custom TensorFlow operation. You would implement the function you want in C++ and register it as a TensorFlow operation. TensorFlow will then give you a Python wrapper that you can use just like you would use an existing function. Assuming you're not an ML researcher, you don't have to do this. But if you ever need to implement your own custom op, you would do it in C++ and it's not too hard. TensorFlow is extensible that way.
- The **Core Python API** is what contains much of the numeric processing code. Add, subtract, divide, matrix multiply, etc. Creating variables, creating tensors, getting the shape or dimension of a tensor. All that core, basic, numeric processing stuff is in the Python API.
- Then, there are sets of Python modules that have high-level representation of useful neural network components.  
Let's say you are interested in creating a new layer of hidden neurons with a ReLU activation function. You can do that by using **tf.layers**.  
If you want to compute the RMSE on data as it comes in, you can use **tf.metrics**.  
To compute cross-entropy-with-logits, for example, which is a common loss

measure in classification problems, you can use **tf.losses**. These modules provide components that are useful when building 'custom' Neural Network models.

Why are 'custom' NN models emphasized? Because you often don't need a custom NN model. Many times, you're quite happy to go with a relatively standard way of training, evaluating and serving models. You don't need to customize the way you train. You're going to use one of the family of gradient-descent-based optimizers and you're going to backpropagate the weights and do this iteratively. In that case, don't write the low-level session loop. Just use an estimator or a high-level API such as Keras!

- The **high-level APIs** allow you to easily do distributed training, data preprocessing and model definition, compilation and training. It knows how to evaluate, how to create a checkpoint, how to save a model, how to set it up for TensorFlow serving. It comes with everything done in a sensible way that fits most ML models in production.

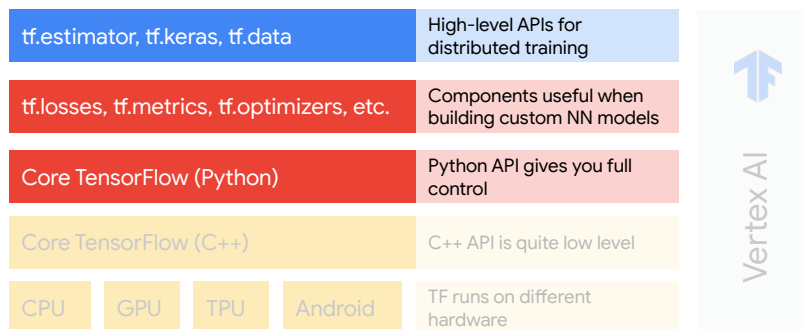
If you see example TensorFlow code out on the internet that does not use the Estimator API, ignore that code. Walk away. It's not worth it. You'll have to write a lot of code to do device placement, and memory management, and distribution.

Let the high-level API do it for you.

So, those are the TensorFlow levels of abstraction. Vertex AI Platform is orthogonal to this hierarchy -- meaning it cuts across all low-level and high-level APIs.

Regardless of which abstraction level you are writing your TensorFlow code at, CAIP (AI Platform) gives you a managed service. It's fully hosted TensorFlow. So you can run TF on the cloud on a cluster of machines without having to install any software or manage any servers.

## TensorFlow toolkit hierarchy



For the rest of this module we will be largely working with these top 3 level APIs here.

Before we start writing API code and showing you the syntax for building machine learning models however -- we first need to really understand the pieces of data that we're working with.

Much like in regular computer science classes where you start with variables and their definitions before moving on to advanced topics like classes, methods, and functions -- that is exactly how we're going to start learning TF components next.






## TensorFlow **tensors and variables**

Now let's go over tensors and variables in TensorFlow.

It's time to see some code!

How can we bring life to each dimension of a tensor that we learned about earlier?

## A tensor is an N-dimensional array of data

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	<code>x = tf.constant(3)</code>	<code>()</code>
	Vector	1	<code>x = tf.constant([3, 5, 7])</code>	<code>(3,)</code>
	Matrix	2	<code>x = tf.constant([[3, 5, 7], [4, 6, 8]])</code>	<code>(2, 3)</code>
	3D Tensor	3	<code>tf.constant([[[3, 5, 7], [4, 6, 8]], [[1, 2, 3], [4, 5, 6]] ])</code>	<code>(2, 2, 3)</code>
	nD Tensor	n	<code>x1 = tf.constant([2, 3, 4]) x2 = tf.stack([x1, x1]) x3 = tf.stack([x2, x2, x2]) x4 = tf.stack([x3, x3]) ...</code>	<code>(3,)</code> <code>(2, 3)</code> <code>(4, 2, 3)</code> <code>(2, 4, 2, 3)</code>

- A tensor is an N-dimensional array of data. When you create a tensor, you will specify its SHAPE. Occasionally, you'll not specify the shape completely. For example, the first element of the shape could be variable, but that special case will be ignored for now.  
Understanding the SHAPE of your data -- or often times the shape it SHOULD be -- is the first essential part of your machine learning flow.  
Here, you're going to create a `tf.constant(3)`. This is 0-rank tensor. Just a number. **A scalar**. The shape when you look at the Tensor debug output will be simply open-parenthesis close-parenthesis. It's zero-rank. To better understand why there isn't a number in those parentheses let's upgrade to the next level.
- If you passed in a bracketed list like 3, 5, 7 to `tf.constant` instead -- you would now be the proud owner of a one-dimensional tensor (otherwise known as a vector). Now, you have a one-dimensional tensor. **A vector**.  
It's grow horizontally (think like on the X-Axis) by three units. Nothing on the Y Axis yet since we're still in 1 dimension here. That's why the shape is (3, nothing).
- Level up! Now we have **a matrix** of numbers or a two dimensional array. Take a look at the shape (2,3) that means we have two rows and three columns of data. The first row being that original vector of [3,5,7] which also has three elements in length (that's where the three columns of data comes from).  
You can think of a matrix as essentially a stack of 1 D tensors. The first tensor

is the vector [3,5,7], and the second 1D tensor that is being stacked is the vector [4,6,8]. Okay so we've got height and width. Let's get more complex!

- What does **3D** look like? It's a 2D tensor with another 2D tensor on top of it. Here you can see we're stacking the 3,5,7 matrix on top of the 1,2,3 matrix. We started with two 2x3 matrices so our resulting shape of the 3D tensor is now 2 comma 2 comma 3.
- Of course, you can do the stacking in code itself instead of counting parentheses. Take the example here:
  - Our x1 variable is a tf constant constructed from a simple list [2,3,4]. That makes it a vector of length 3.
  - X2 is constructed by stacking x1 on top of x1. So, that makes it a 2x3 matrix.
  - X3 is constructed by stacking 4 x2s on top of each other. Since each x2 was a 2x3 matrix, this makes X3 a 3D tensor of shape 4x2x3.
  - X4 is constructed by stacking x3 on top of x3. That makes it two 4x2x3 tensors, or the final result which is a 4D tensor of shape 2 comma 4 comma 2 comma 3.

## A tensor is an N-dimensional array of data

They behave like numpy n-dimensional arrays except that:

- `tf.constant` produces constant tensors
- `tf.Variable` produces tensors that can be modified

If you've worked with arrays of data before like with Numpy they're similar expect for these two points.

`tf.constant` will produce tensors with constant values and `tf.variable` produces tensors with variable values (or ones that can be modified). This will prove super useful later when we need to adjust the model weights during the training phase of our ML project. Those weights can simply be a modifiable tensor array.

Let's take a look at the syntax for each as you will become a ninja in combining, slicing, and reshaping tensors as you see fit.



## tf.constant produces constant tensors...

```
import tensorflow as tf

x = tf.constant([
    [3, 5, 7],
    [4, 6, 8]
])
```

Here's a constant tensor produced by --- well tf.constant of course!

Remember that 3,5,7 is our 1D vector and we've just stacked it here to be a 2D matrix.

Pop quiz -- what is the shape of X? Hint: How many rows of data (or stacks) and then how many columns do you see?

Answer: This shape is 2x3 or 2 rows and 3 columns. When you're coding you can also invoke `tf.shape()` which is quite handy when debugging.

Okay -- much like you can stack tensors to get to higher dimensions you can also SLICE them down too.

## Tensors can be sliced

```
import tensorflow as tf

x = tf.constant([
    [3, 5, 7],
    [4, 6, 8]
])

y = x[:, 1]

[5, 6]
```

Now, take a look at this code for y. It's 'slicing' x. Is it slicing rows, columns, or both?

The syntax is let Y be the result of taking X and take ALL rows (that's the COLON) and just the first column. Keeping in mind that Python is zero indexed when it comes to arrays what would the result be?

Remember we're going down from 3D to 2D here so your answer should only be a single bracketed list of numbers.

Answer? [5, 6]. Again -- take all rows and only the first indexed column. 3,4 is the 0 index and 5,6 is the first index which is what gets printed out.

Make sense?

Don't worry you'll get plenty of practice.

## Tensors can be reshaped

```
import tensorflow as tf

x = tf.constant([
    [3, 5, 7],
    [4, 6, 8]
])

y = tf.reshape(x, [3, 2])

[[3 5]
 [7 4]
 [6 8]
]
```

So we've seen stacking and slicing -- next let's talk about reshaping with `tf.reshape`.

Let's use the same 2D tensor or matrix of values that is X. What's the shape again? Think rows then columns. If you said 2x3 you're right.

NOW what if I reshaped X as [3,2] or three rows and two columns, what would happen?

Essentially Python will read the input tensor row-by-row and put numbers into the output tensor. It will pick the first two values and put it into the first row. So, you get 3 and 5.

The next two values, 7 and 4, go into the second row. And the last two values, 6 and 8, go into the third row.

That's what reshaping does.

Well that's it for constants -- not too bad right?

Next up are variable tensors.

## A variable is a tensor whose value can be changed...

```
import tensorflow as tf

# x <- 2
x = tf.Variable(2.0, dtype=tf.float32,
name='my_variable')
```

The `Variable()` constructor requires an initial value for the variable, which can be a tensor of any type and shape.

This initial value defines the type and shape of the variable. After construction, the type and shape of the variable are fixed.

## A variable is a tensor whose value can be changed...

tf.Variable will typically hold model weights that need to be updated in a training loop.

```
import tensorflow as tf

# x <- 2
x = tf.Variable(2.0, dtype=tf.float32, name='my_variable')

# x <- 48.5
x.assign(45.8)

# x <- x + 4
x.assign_add(4)

# x <- x - 3
x.assign_sub(3)
```

The value can be changed using one of the assign methods (assign, assign\_add, assign\_sub).

As we mentioned before, tf.variables are generally used for values that are modified during training (such as model weights).

## A variable is a tensor whose value can be changed...

```
import tensorflow as tf

# w * x
w = tf.Variable([[1.], [2.]])
x = tf.constant([[3., 4.]])
tf.matmul(w, x)
```

Just like any tensor, variables created with `Variable()` can be used as inputs to operations.

Additionally, all the operators overloaded for the `Tensor` class are carried over to variables.

## GradientTape records operations for automatic differentiation

TensorFlow can compute the derivative of a function with respect to any parameter.

- the computation is recorded with [GradientTape](#)
- the function is expressed with [TensorFlow ops only!](#)

TensorFlow has the ability to calculate the partial derivative of any function with respect to any variable.

We know that, during training, weights are updated by using the partial derivative of the loss with respect to each individual weight.

To differentiate automatically, TensorFlow needs to remember what operations happen in what order during the forward pass.

Then, during the backward pass, TensorFlow traverses this list of operations in reverse order to compute gradients.

GradientTape is a context manager in which these partial differentiations are calculated.

The functions have to be expressed within TensorFlow operations only. But, since most basic operations like addition, multiplication and subtraction, are overloaded by TensorFlow ops, these happen seamlessly.

## GradientTape records operations for automatic differentiation

```
def compute_gradients(X, Y, w0, w1):  
    with tf.GradientTape() as tape:  
        loss = loss_mse(X, Y, w0, w1)  
    return tape.gradient(loss, [w0, w1])
```

Record the computation with GradientTape when it's executed (not when it's defined!)

```
w0 = tf.Variable(0.0)  
w1 = tf.Variable(0.0)  
  
dw0, dw1 = compute_gradients(X, Y, w0, w1)
```

Let's say that we want to compute a loss gradient.

TensorFlow "records" all operations executed inside the context of a `tf.GradientTape` onto a "tape".

Then, it uses that tape and the gradients associated with each recorded operation to compute the gradients of a "recorded" computation using reverse mode differentiation.



## GradientTape records operations for automatic differentiation

```
def compute_gradients(X, Y, w0, w1):  
    with tf.GradientTape() as tape:  
        loss = loss_mse(X, Y, w0, w1)  
        return tape.gradient(loss, [w0, w1])
```

Specify the function (loss) as well as the parameters you want to take the gradient with respect to ([w0, w1])

```
w0 = tf.Variable(0.0)  
w1 = tf.Variable(0.0)
```

```
dw0, dw1 = compute_gradients(X, Y, w0, w1)
```

There are cases where you may want to control exactly how gradients are calculated rather than using the default.

These cases can be when the default calculations are numerically unstable or you wish to cache an expensive computation from the forward pass, among others.

For such scenarios, you can use Custom Gradient functions to write a new operation or to modify the calculation of the differentiation.