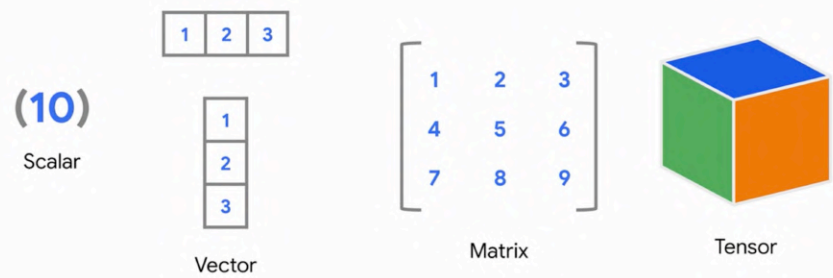
Tensors Revisited



Some types of tensors

Variables
`tf.Variable`

```
tf.Variable("Hello", tf.string)
```

Constants
`tf.constant`

```
tf.constant([1, 2, 3, 4, 5, 6])
```

Characteristics of a tensor


```
tf.Tensor([4 6], shape=(2,), dtype=int32)
```

Characteristics of a tensor

Tensor

Shape | Data type

```
tf.Tensor([4 6], shape=(2,), dtype=int32)
```

Inspect variables of a built-in Keras layer

```
model = tf.keras.Sequential([
            tf.keras.layers.Dense(1, input_shape=(1,))
])

>>> model.variables
[<tf.Variable 'dense_1/kernel:0' shape=(1, 1) dtype=float32,
  numpy=array([[1.4402896]], dtype=float32)>,
 <tf.Variable 'dense_1/bias:0' shape=(1,) dtype=float32,
  numpy=array([0.], dtype=float32)>]
```


Inspect variables of a built-in Keras layer

```
model = tf.keras.Sequential([
            tf.keras.layers.Dense(1, input_shape=(1,))
])

>>> model.variables
[<tf.Variable 'dense_1/kernel:0' shape=(1, 1) dtype=float32,
  numpy=array([[1.4402896]], dtype=float32)>,
 <tf.Variable 'dense_1/bias:0' shape=(1,) dtype=float32,
  numpy=array([0.], dtype=float32)>]
```

Inspect variables of a built-in Keras layer

```
model = tf.keras.Sequential([
            tf.keras.layers.Dense(1, input_shape=(1,))
])

>>> model.variables
[<tf.Variable 'dense_1/kernel:0' shape=(1, 1) dtype=float32,
  numpy=array([[1.4402896]], dtype=float32)>,
 <tf.Variable 'dense_1/bias:0' shape=(1,) dtype=float32,
  numpy=array([0.], dtype=float32)>]
```

Creating Tensors with `tf.Variable`

```
vector = tf.Variable(initial_value = [1,2])
   <tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([1, 2], dtype=int32)>
```

Creating Tensors with `tf.Variable`

```
vector = tf.Variable(initial_value = [1,2])
   <tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([1, 2], dtype=int32)>

vector = tf.Variable([1,2], dtype=tf.float32)

   <tf.Variable 'Variable:0' shape=(2,) dtype=float32, numpy=array([1., 2.], dtype=float32)>
```

Creating Tensors with `tf.Variable`

```
vector = tf.Variable(initial_value = [1,2])
  <tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([1, 2], dtype=int32)>

vector = tf.Variable([1,2], dtype=tf.float32)

  <tf.Variable 'Variable:0' shape=(2,) dtype=float32, numpy=array([1., 2.], dtype=float32)>

vector = tf.Variable([1,2], tf.float32) # don't do please!

  <tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([1, 2], dtype=int32)>
```

Creating Tensors with `tf.Variable`

```
vector = tf.Variable([1,2,3,4])
  <tf.Variable 'Variable:0' shape=(4,) dtype=int32, numpy=array([1, 2, 3, 4], dtype=int32)>
```

Creating Tensors with `tf.Variable`

```
vector = tf.Variable([1,2,3,4])
  <tf.Variable 'Variable:0' shape=(4,) dtype=int32, numpy=array([1, 2, 3, 4], dtype=int32)>
vector = tf.Variable([1,2,3,4], shape=(2,2)) # don't do please!
  ValueError: The initial value's shape ((4,)) is not compatible with the explicitly supplied
  `shape` argument ((2, 2)).
```

Creating Tensors with `tf.Variable`

```
vector = tf.Variable([1,2,3,4])
  <tf.Variable 'Variable:0' shape=(4,) dtype=int32, numpy=array([1, 2, 3, 4], dtype=int32)>
vector = tf.Variable([1,2,3,4], shape=(2,2)) # don't do please!
  ValueError: The initial value's shape ((4,)) is not compatible with the explicitly supplied
  `shape` argument ((2, 2)).
vector = tf.Variable([[1,2],[3,4]])
  <tf.Variable 'Variable:0' shape=(2, 2) dtype=int32, numpy=
  array([[1, 2],
         [3, 4]], dtype=int32)>
```

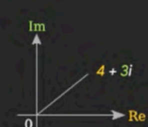

Creating Tensors with `tf.Variable`

```
vector = tf.Variable([1,2,3,4])
  <tf.Variable 'Variable:0' shape=(4,) dtype=int32, numpy=array([1, 2, 3, 4], dtype=int32)>
vector = tf.Variable([1,2,3,4], shape=(2,2)) # don't do please!
  ValueError: The initial value's shape ((4,)) is not compatible with the explicitly supplied
  `shape` argument ((2, 2)).
vector = tf.Variable([[1,2],[3,4]])
  <tf.Variable 'Variable:0' shape=(2, 2) dtype=int32, numpy=
  array([[1, 2],
         [3, 4]], dtype=int32)>
```


```
vector = tf.Variable([1,2,3,4], shape=tf.TensorShape(None))
  <tf.Variable 'Variable:0' shape=<unknown> dtype=int32, numpy=array([1, 2, 3, 4], dtype=int32)>
```

Creating Tensors with `tf.Variable`

```
mammal = tf.Variable("Elephant", dtype=tf.string)

its_complicated = tf.Variable(4 + 3j,
                              dtype=tf.complex64)
```

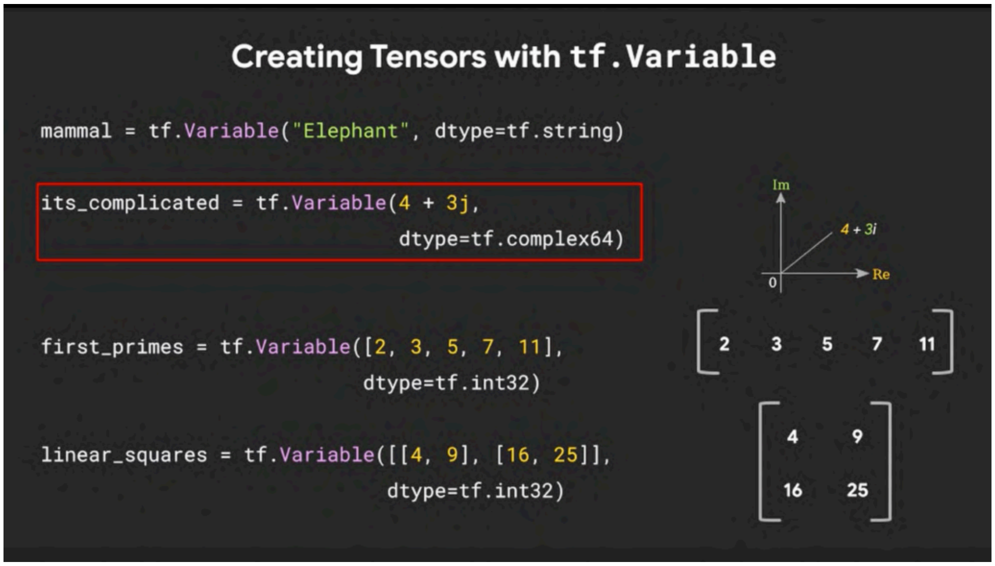


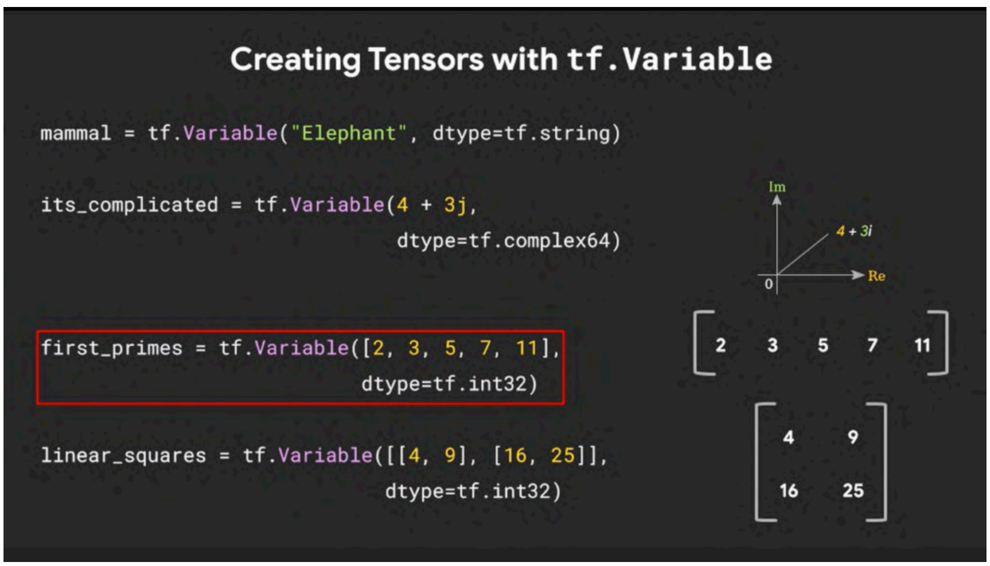

```
first_primes = tf.Variable([2, 3, 5, 7, 11],
                           dtype=tf.int32)
```

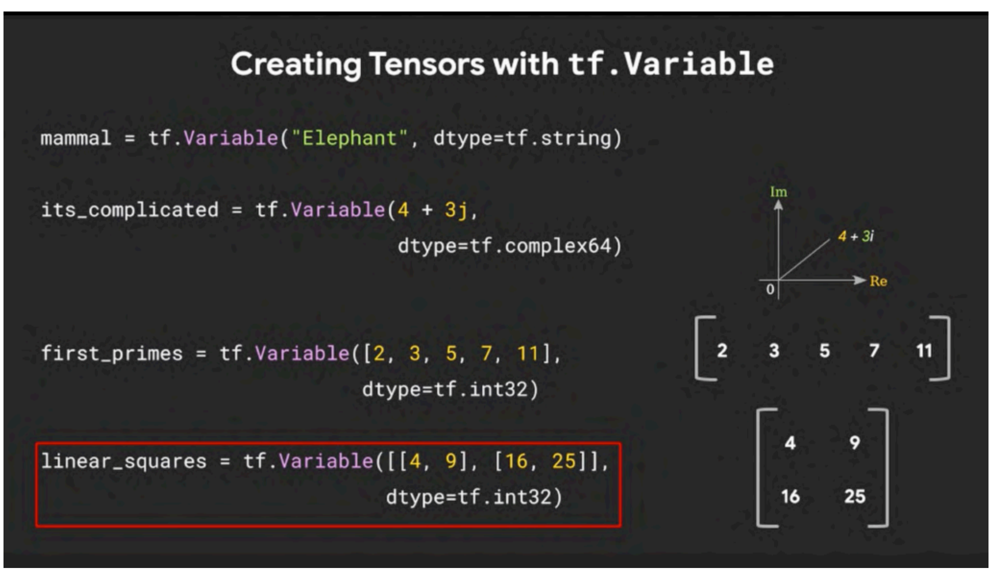
$$\begin{bmatrix} 2 & 3 & 5 & 7 & 11 \end{bmatrix}$$

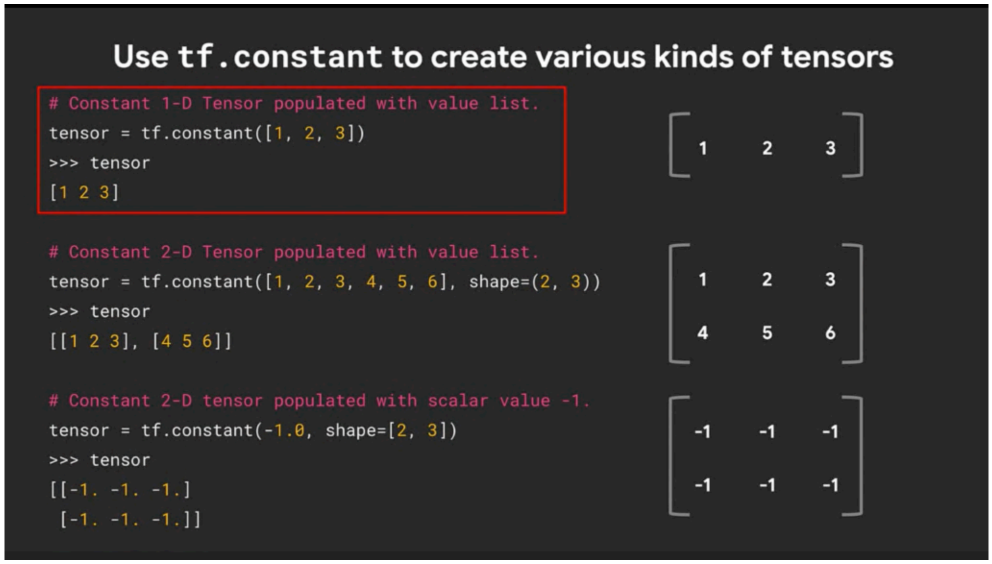
```
linear_squares = tf.Variable([[4, 9], [16, 25]],
                             dtype=tf.int32)
```

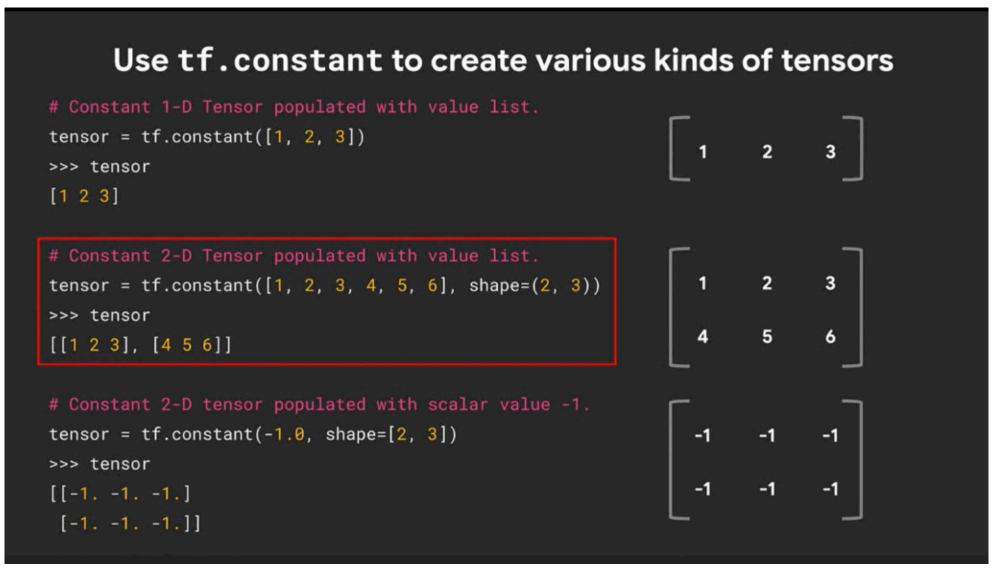
$$\begin{bmatrix} 4 & 9 \\ 16 & 25 \end{bmatrix}$$

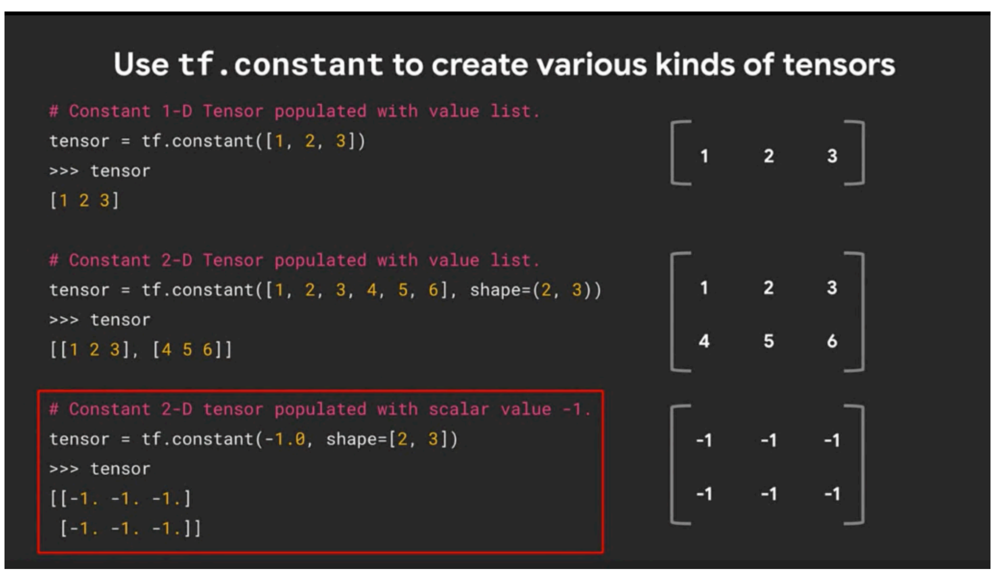

Creating Tensors with tf.Variable
mammal = tf.Variable("Elephant", dtype=tf.string)
its_complicated = tf.Variable(4 + 3j,
dtype=tf.complex64)
first_primes = tf.Variable([2, 3, 5, 7, 11],
dtype=tf.int32)
linear_squares = tf.Variable([[4, 9], [16, 25]],
dtype=tf.int32)
Im
4 + 3i
0
Re
2 3 5 7 11
4 9
16 25


Creating Tensors with tf.Variable
mammal = tf.Variable("Elephant", dtype=tf.string)
its_complicated = tf.Variable(4 + 3j,
dtype=tf.complex64)
first_primes = tf.Variable([2, 3, 5, 7, 11],
dtype=tf.int32)
linear_squares = tf.Variable([[4, 9], [16, 25]],
dtype=tf.int32)
Im
4 + 3i
0
Re
2 3 5 7 11
4 9
16 25


Creating Tensors with tf.Variable
mammal = tf.Variable("Elephant", dtype=tf.string)
its_complicated = tf.Variable(4 + 3j,
dtype=tf.complex64)
first_primes = tf.Variable([2, 3, 5, 7, 11],
dtype=tf.int32)
linear_squares = tf.Variable([[4, 9], [16, 25]],
dtype=tf.int32)
Im
4 + 3i
0
Re
2 3 5 7 11
4 9
16 25


Use tf.constant to create various kinds of tensors
# Constant 1-D Tensor populated with value list.
tensor = tf.constant([1, 2, 3])
>>> tensor
[1 2 3]
# Constant 2-D Tensor populated with value list.
tensor = tf.constant([1, 2, 3, 4, 5, 6], shape=(2, 3))
>>> tensor
[[1 2 3], [4 5 6]]
# Constant 2-D tensor populated with scalar value -1.
tensor = tf.constant(-1.0, shape=[2, 3])
>>> tensor
[[-1. -1. -1.]
[-1. -1. -1.]]
1 2 3
1 2 3
4 5 6
-1 -1 -1
-1 -1 -1


Use tf.constant to create various kinds of tensors
# Constant 1-D Tensor populated with value list.
tensor = tf.constant([1, 2, 3])
>>> tensor
[1 2 3]
# Constant 2-D Tensor populated with value list.
tensor = tf.constant([1, 2, 3, 4, 5, 6], shape=(2, 3))
>>> tensor
[[1 2 3], [4 5 6]]
# Constant 2-D tensor populated with scalar value -1.
tensor = tf.constant(-1.0, shape=[2, 3])
>>> tensor
[[-1. -1. -1.]
[-1. -1. -1.]]
1 2 3
1 2 3
4 5 6
-1 -1 -1
-1 -1 -1


Use tf.constant to create various kinds of tensors
# Constant 1-D Tensor populated with value list.
tensor = tf.constant([1, 2, 3])
>>> tensor
[1 2 3]
# Constant 2-D Tensor populated with value list.
tensor = tf.constant([1, 2, 3, 4, 5, 6], shape=(2, 3))
>>> tensor
[[1 2 3], [4 5 6]]
# Constant 2-D tensor populated with scalar value -1.
tensor = tf.constant(-1.0, shape=[2, 3])
>>> tensor
[[-1. -1. -1.]
[-1. -1. -1.]]
1 2 3
1 2 3
4 5 6
-1 -1 -1
-1 -1 -1

Operations

tf.add

tf.subtract

tf.multiply

...

Applying operations

```
>>> tf.add([1, 2], [3, 4])
tf.Tensor([4 6], shape=(2,), dtype=int32)

>>> tf.square(5)
tf.Tensor(25, shape=(), dtype=int32)

>>> tf.reduce_sum([1, 2, 3])
tf.Tensor(6, shape=(), dtype=int32)

# Operator overloading is also supported
>>> tf.square(2) + tf.square(3)
tf.Tensor(13, shape=(), dtype=int32)
```


Applying operations

```
>>> tf.add([1, 2], [3, 4])
tf.Tensor([4 6], shape=(2,), dtype=int32)

>>> tf.square(5)
tf.Tensor(25, shape=(), dtype=int32)

>>> tf.reduce_sum([1, 2, 3])
tf.Tensor(6, shape=(), dtype=int32)

# Operator overloading is also supported
>>> tf.square(2) + tf.square(3)
tf.Tensor(13, shape=(), dtype=int32)
```

Applying operations

```
>>> tf.add([1, 2], [3, 4])
tf.Tensor([4 6], shape=(2,), dtype=int32)

>>> tf.square(5)
tf.Tensor(25, shape=(), dtype=int32)

>>> tf.reduce_sum([1, 2, 3])
tf.Tensor(6, shape=(), dtype=int32)

# Operator overloading is also supported
>>> tf.square(2) + tf.square(3)
tf.Tensor(13, shape=(), dtype=int32)
```

Applying operations

```
>>> tf.add([1, 2], [3, 4])
tf.Tensor([4 6], shape=(2,), dtype=int32)

>>> tf.square(5)
tf.Tensor(25, shape=(), dtype=int32)

>>> tf.reduce_sum([1, 2, 3])
tf.Tensor(6, shape=(), dtype=int32)

# Operator overloading is also supported
>>> tf.square(2) + tf.square(3)
tf.Tensor(13, shape=(), dtype=int32)
```


Eager execution in TensorFlow

- Evaluate values immediately
- Broadcasting support
- Operator overloading
- NumPy compatibility

Evaluate tensors

```
x = 2
x_squared = tf.square(x)
>>> print("hello, {}".format(x_squared))
hello, 4
```

Broadcast values

```
a = tf.constant([[1, 2],
                 [3, 4]])
>>> tf.add(a, 1)
tf.Tensor(
[[2 3]
 [4 5]], shape=(2, 2), dtype=int32)
```

Overload operators

```
a = tf.constant([[1, 2],
                 [3, 4]])
>>> a ** 2
tf.Tensor(
[[ 1  4]
 [ 9 16]], shape=(2, 2), dtype=int32)
```

NumPy Compatibility

```
import numpy as np
a = tf.constant(5)
b = tf.constant(3)

>>> np.multiply(a, b)
15
```

Numpy interoperability

```
ndarray = np.ones([3, 3])
>>> ndarray
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

```
tensor = tf.multiply(ndarray, 3)
>>> tensor
```

```
tf.Tensor(
[[3. 3. 3.]
 [3. 3. 3.]
 [3. 3. 3.]],
 shape=(3, 3),
 dtype=float64)
```

```
>>> tensor.numpy()
```

```
array([[3., 3., 3.],
       [3., 3., 3.],
       [3., 3., 3.]])
```

Evaluating variables

```
v = tf.Variable(0.0)
>>> v + 1
<tf.Tensor: id=47, shape=(), dtype=float32, numpy=1.0>
```

```
v = tf.Variable(0.0)
>>> v.assign_add(1)
<tf.Variable 'UnreadVariable' shape=() dtype=float32, numpy=1.0>
```

```
v = tf.Variable(0.0)
v.assign_add(1)
>>> v.read_value().numpy()
1.0
```

Evaluating variables

```
v = tf.Variable(0.0)
>>> v + 1
<tf.Tensor: id=47, shape=(), dtype=float32, numpy=1.0>

v = tf.Variable(0.0)
>>> v.assign_add(1)
<tf.Variable 'UnreadVariable' shape=() dtype=float32, numpy=1.0>

v = tf.Variable(0.0)
v.assign_add(1)
>>> v.read_value().numpy()
1.0
```

Evaluating variables

```
v = tf.Variable(0.0)
>>> v + 1
<tf.Tensor: id=47, shape=(), dtype=float32, numpy=1.0>

v = tf.Variable(0.0)
>>> v.assign_add(1)
<tf.Variable 'UnreadVariable' shape=() dtype=float32, numpy=1.0>

v = tf.Variable(0.0)
v.assign_add(1)
>>> v.read_value().numpy()
1.0
```

Examine custom layers

```
class MyLayer(tf.keras.layers.Layer):

  def __init__(self):
    super(MyLayer, self).__init__()
    self.my_var = tf.Variable(100)
    self.my_other_var_list = [tf.Variable(x) for x in range(2)]

m = MyLayer()
>>> [variable.numpy() for variable in m.variables]
[100, 0, 1]
```

Examine custom layers

```python
class MyLayer(tf.keras.layers.Layer):

  def __init__(self):
    super(MyLayer, self).__init__()
    self.my_var = tf.Variable(100)
    self.my_other_var_list = [tf.Variable(x) for x in range(2)]

m = MyLayer()
>>> [variable.numpy() for variable in m.variables]
[100, 0, 1]
```

Change data types

```python
tensor = tf.constant([1, 2, 3])
>>> tensor
tf.Tensor([1 2 3], shape=(3,), dtype=int32)

# Cast a constant integer tensor into floating point
tensor = tf.cast(tensor, dtype=tf.float32)
>>> tensor.dtype
tf.float32
```

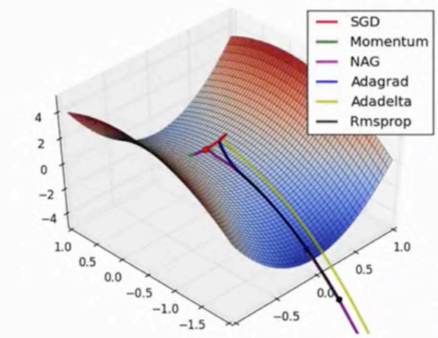
Change data types

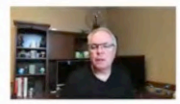
```python
tensor = tf.constant([1, 2, 3])
>>> tensor
tf.Tensor([1 2 3], shape=(3,), dtype=int32)

# Cast a constant integer tensor into floating point
tensor = tf.cast(tensor, dtype=tf.float32)
>>> tensor.dtype
tf.float32
```

At the core of all machine learning are optimizing functions that are used to match features to labels by tweaking parameters. These functions operate on the principle of gradient descent and each optimizing function has a different rate of convergence towards the optimal value.

Over time, they've evolved as researchers have experimented with these algorithms with different scenarios. Intensive flow optimizers are implemented using TensorFlow automatic differentiation API call Gradient Tape.

This API lets you compute and track the gradient of every differentiable TensorFlow operation. As we get into custom training, it's good to understand how the gradients of your learning work, and thus it's good to get an overview of gradient tape.





http://cs231n.github.io/neural-networks-3/

Note that the parameters are set as trainable equals true, so that the gradient tape will keep an eye on the w and b tensors and let us differentiate against them as we optimize the model.

```
# Training data
x_train = np.array([-1.0,  0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
y_train = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

# Trainable variables
w = tf.Variable(random.random(), trainable=True)
b = tf.Variable(random.random(), trainable=True)
```

```
# Loss function
def simple_loss(real_y, pred_y):
    return tf.abs(real_y - pred_y)


# Learning Rate
LEARNING_RATE = 0.001
```

```python
for _ in range(500):
    fit_data(x_train, y_train)

print(f'y ≈ {w.numpy()}x + {b.numpy()}')
```

```python
def fit_data(real_x, real_y):
    with tf.GradientTape(persistent=True) as tape:
        # Make prediction
        pred_y = w * real_x + b
        # Calculate loss
        reg_loss = simple_loss(real_y, pred_y)

    # Calculate gradients
    w_gradient = tape.gradient(reg_loss, w)
    b_gradient = tape.gradient(reg_loss, b)

    # Update variables
    w.assign_sub(w_gradient * LEARNING_RATE)
    b.assign_sub(b_gradient * LEARNING_RATE)
```

Here's where the learning will happen. The fit data function that we'll call 500 times in our loop. To use tf.GradientTape, we start with the keyword "with" to start a with-block. Some code below will then be indented to indicate that it's part of this with-block.

We'll use with tf.GradientTape as a tape, so the variable tape is now an object of type gradients tape, and this can be used later to calculate gradients.

Note that later this week, you'll see an explanation of the gradient tapes function parameter persistent equals true and what it's doing. Don't worry about that for now.

```python
def fit_data(real_x, real_y):
    with tf.GradientTape(persistent=True) as tape:
        # Make prediction
        pred_y = w * real_x + b
        # Calculate loss
        reg_loss = simple_loss(real_y, pred_y)

    # Calculate gradients
    w_gradient = tape.gradient(reg_loss, w)
    b_gradient = tape.gradient(reg_loss, b)

    # Update variables
    w.assign_sub(w_gradient * LEARNING_RATE)
    b.assign_sub(b_gradient * LEARNING_RATE)
```

Inside the with-loop of gradient tape, we usually do two things. First, we'll calculate the prediction. In this case, we'll calculate pred_y based on the current w and b. Then we'll calculate the loss.

You can see here that reg loss stores the return value from calling the function simple loss, which compares the real y with the predicted y.

```python
def fit_data(real_x, real_y):
    with tf.GradientTape(persistent=True) as tape:
        # Make prediction
        pred_y = w * real_x + b
        # Calculate loss
        reg_loss = simple_loss(real_y, pred_y)

    # Calculate gradients
    w_gradient = tape.gradient(reg_loss, w)
    b_gradient = tape.gradient(reg_loss, b)

    # Update variables
    w.assign_sub(w_gradient * LEARNING_RATE)
    b.assign_sub(b_gradient * LEARNING_RATE)
```

To get the gradient for w and b, we differentiate each against the loss value.

The tape.gradient method will give us this functionality. Notice that outside of the with-block, we can still use the tape variable that was declared inside the with-block.

For w gradient, we want to get the derivative of the loss with respect to the weight's w. We'll call tape.gradients for us passing in the loss, reg loss, and then the variable w.

Similarly, b gradient is the derivative of the loss with respect to the bias b. The negative of this gradient will point in the direction of optimal values for w and b. It's forming a very basic optimizer.

```python
def fit_data(real_x, real_y):
    with tf.GradientTape(persistent=True) as tape:
        # Make prediction
        pred_y = w * real_x + b
        # Calculate loss
        reg_loss = simple_loss(real_y, pred_y)

    # Calculate gradients
    w_gradient = tape.gradient(reg_loss, w)
    b_gradient = tape.gradient(reg_loss, b)

    # Update variables
    w.assign_sub(w_gradient * LEARNING_RATE)
    b.assign_sub(b_gradient * LEARNING_RATE)
```

Then for w and b-tensors, we can call assign sub to update them with the gradient times the learning rate to tweak their value.

If you recall the math for back propagation, here we would update the weight w to be w minus the gradient of the loss with respect to w, where the gradient is scaled by the learning rate.

The assign sub does this subtraction and assigns the result back to w.

$$y \approx 1.9902112483978271x + -0.995111882686615$$

Gradient Descent with tf.GradientTape

```python
def train_step(images, labels):
  with tf.GradientTape() as tape:
    logits = model(images, training=True)
    loss_value = loss_object(labels, logits)

  loss_history.append(loss_value.numpy().mean())
  grads = tape.gradient(loss_value, model.trainable_variables)
  optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Here's an approach of how we can implement a training step of a learning algorithm using Tensorflow's GradientTape API.

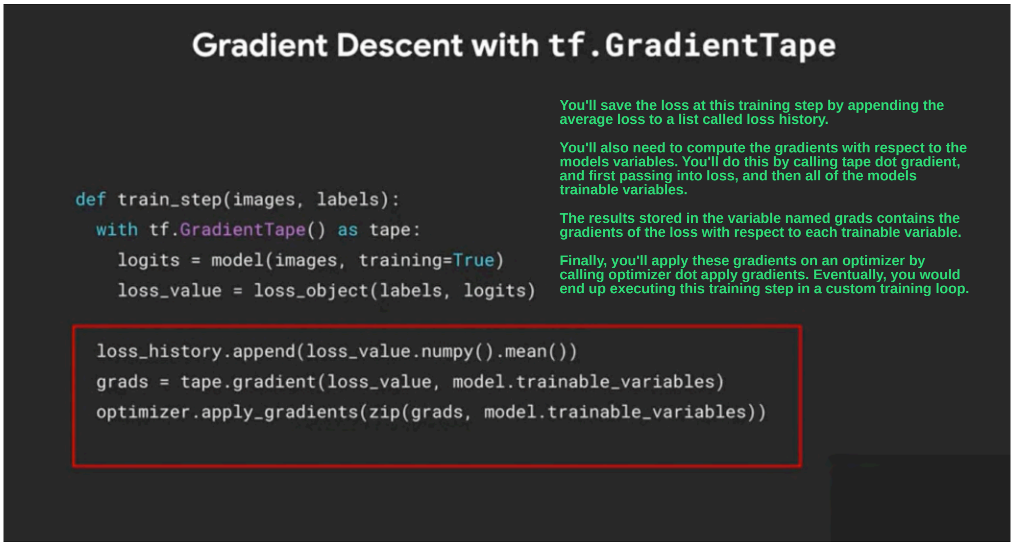
To perform a training step, you need to complete two crucial stages of a learning algorithm inside the current context of a GradientTape.

The first one involves invoking the forward pass of your model. In this example, you invoke the forward pass by calling model and storing the predictions in the variable called logits.

In Machine Learning, logits refer to a vector of raw prediction values for each category and a multi-class classification. These logits are not yet scaled to add up to one, so the logits are normally fed into a softmax function, to turn them into probabilities for each category.

Another thing that you have to calculate is the loss obtained at each forward pass. Here we can do this by calling a function called loss object, which takes in the true labels in the logits to calculate the loss value.

The loss value allows you to update the model in order to reduce the model's prediction errors.

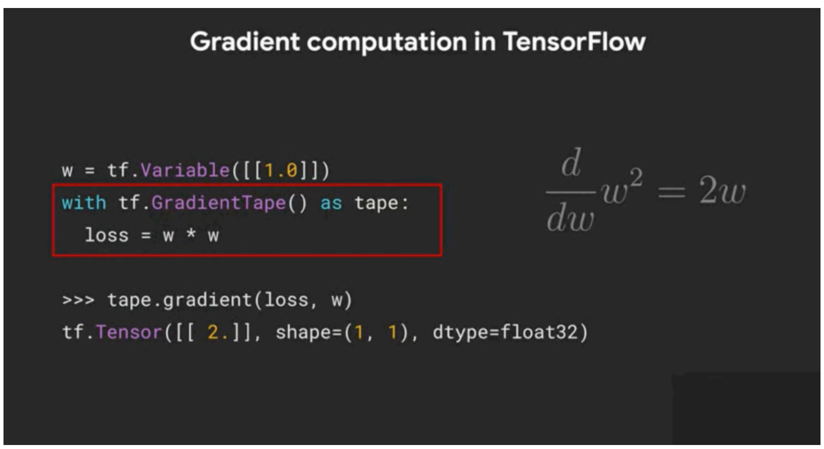


You'll save the loss at this training step by appending the average loss to a list called loss history.

You'll also need to compute the gradients with respect to the models variables. You'll do this by calling tape dot gradient, and first passing into loss, and then all of the models trainable variables.

The results stored in the variable named grads contains the gradients of the loss with respect to each trainable variable.

Finally, you'll apply these gradients on an optimizer by calling optimizer dot apply gradients. Eventually, you would end up executing this training step in a custom training loop.



You'll see how to use custom training loops in a lot more detail later. Now let's see how to use GradientTape to calculate the gradients of a simple equation.

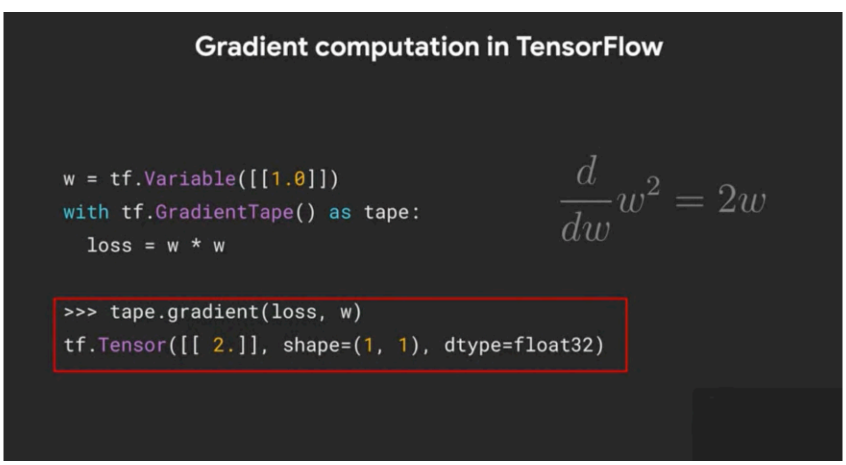
In this case, we're setting the loss equal to W times W, and you may recall from calculus that the derivative of W squared is two times W. So, the gradient of loss with respect to W is two times W.

In the previous example, you had seen how a model's forward pass is computed with GradientTape, we will do the same here, but use our W squared equation instead.

To start, we will have to record values of this operation executed inside the context, using Python's with as syntax and writing with TF dot GradientTape as a tape. We can then get Tensorflow's GradientTape contexts Manager to keep track of all operations executed inside the context onto a tape.

The idea of a tape here, is that the gradients are remembered and stored while they're in the context, a little bit like music on a tape, and they're disposed off once the context is done.

For this scenario, they don't really need to be stored per se, but in advanced scenarios, you might want to differentiate a differential, effectively stacking operations and keeping track of the previous values in that context may be necessary. With that in mind, the context-based objects using Python's with as syntax was chosen for this API.

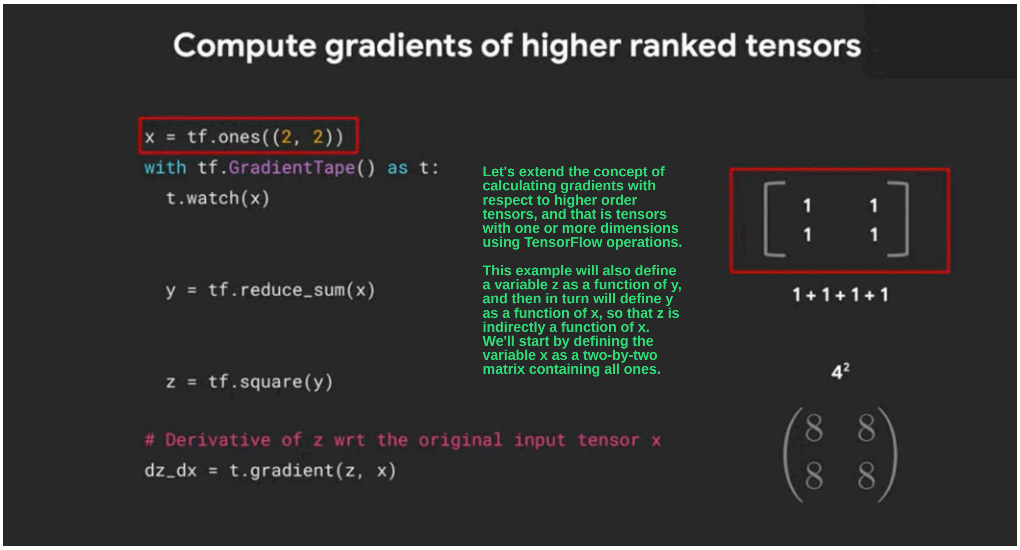


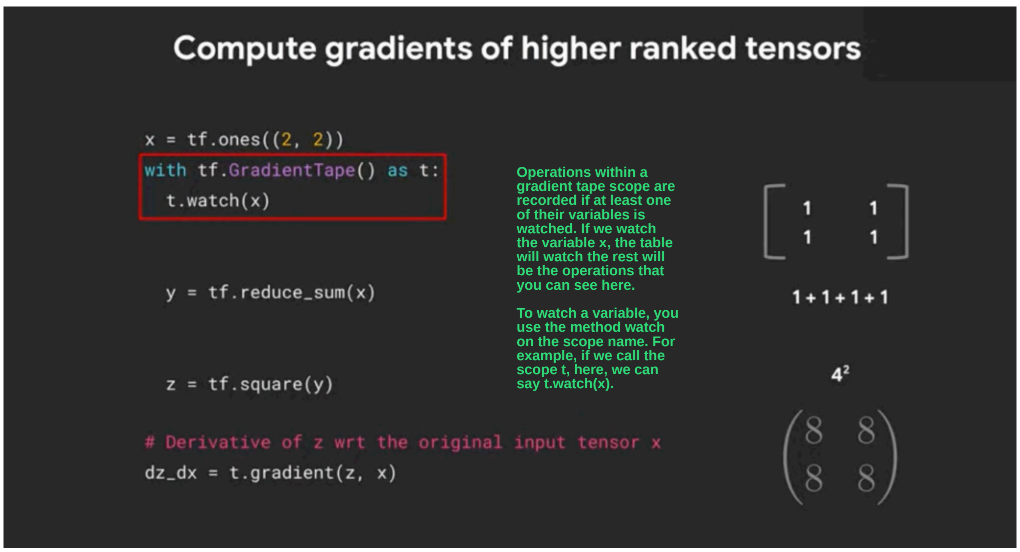
Next, similar to how you calculate derivative of functions in calculus, you can call tape.gradients to compute the gradient of loss with respect to the input value W.

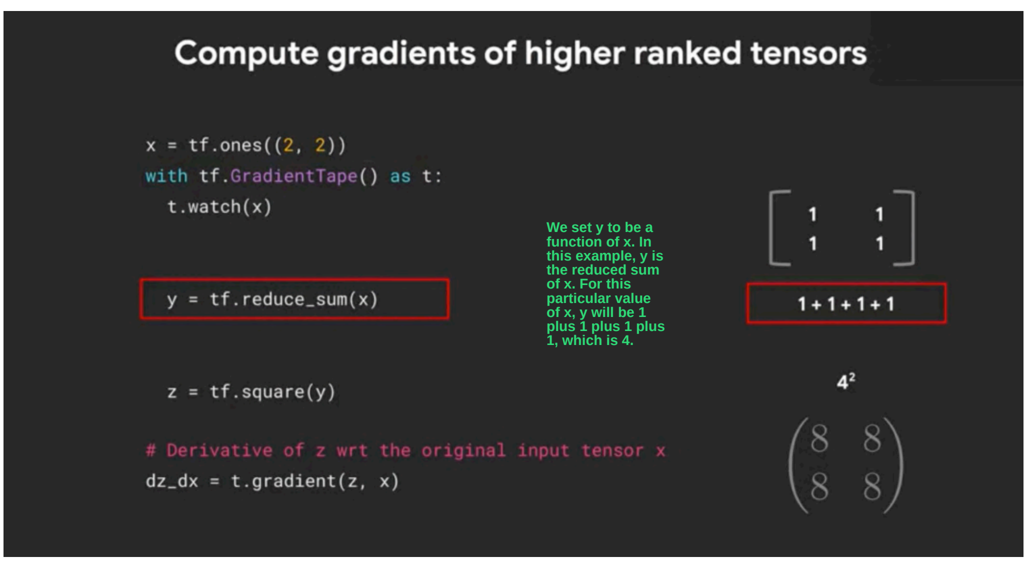
We purposely chose a simple expression for the loss, so that we could calculate the gradient by hand and compare it with the gradient calculated by GradientTape. If the loss is W squared, then its gradient with respect to W is two times W. When we set W equal to one, then the gradient is two times one, which is two.

Using GradientTape to do the same thing, defining W to be a tensor with the value one and calling tape dot gradient passing in the loss and W, we will get back a tensor containing two.

When working with real loss functions, that can be more complicated, and you won't want to calculate the gradient by hand, so you can just use GradientTape to calculate it for you.


Compute gradients of higher ranked tensors
x = tf.ones((2, 2))
with tf.GradientTape() as t:
t.watch(x)
y = tf.reduce_sum(x)
z = tf.square(y)
# Derivative of z wrt the original input tensor x
dz_dx = t.gradient(z, x)
Let's extend the concept of calculating gradients with respect to higher order tensors, and that is tensors with one or more dimensions using TensorFlow operations.
This example will also define a variable z as a function of y, and then in turn will define y as a function of x, so that z is indirectly a function of x. We'll start by defining the variable x as a two-by-two matrix containing all ones.
1 1
1 1
1 + 1 + 1 + 1
4²
8 8
8 8


Compute gradients of higher ranked tensors
x = tf.ones((2, 2))
with tf.GradientTape() as t:
t.watch(x)
y = tf.reduce_sum(x)
z = tf.square(y)
# Derivative of z wrt the original input tensor x
dz_dx = t.gradient(z, x)
Operations within a gradient tape scope are recorded if at least one of their variables is watched. If we watch the variable x, the table will watch the rest will be the operations that you can see here.
To watch a variable, you use the method watch on the scope name. For example, if we call the scope t, here, we can say t.watch(x).
1 1
1 1
1 + 1 + 1 + 1
4²
8 8
8 8


Compute gradients of higher ranked tensors
x = tf.ones((2, 2))
with tf.GradientTape() as t:
t.watch(x)
y = tf.reduce_sum(x)
z = tf.square(y)
# Derivative of z wrt the original input tensor x
dz_dx = t.gradient(z, x)
We set y to be a function of x. In this example, y is the reduced sum of x. For this particular value of x, y will be 1 plus 1 plus 1 plus 1, which is 4.
1 1
1 1
1 + 1 + 1 + 1
4²
8 8
8 8

Compute gradients of higher ranked tensors

```
x = tf.ones((2, 2))
with tf.GradientTape() as t:
  t.watch(x)

  y = tf.reduce_sum(x)

  z = tf.square(y)

# Derivative of z wrt the original input tensor x
dz_dx = t.gradient(z, x)
```

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

1 + 1 + 1 + 1

$\boxed{4^2}$

$$\begin{pmatrix} 8 & 8 \\ 8 & 8 \end{pmatrix}$$

Then, we set z to be a function of y. Here z is y squared. For this particular input of x, y is 4, z is 4 squared or 16.

Compute gradients of higher ranked tensors

```
x = tf.ones((2, 2))
with tf.GradientTape() as t:
  t.watch(x)

  y = tf.reduce_sum(x)

  z = tf.square(y)

# Derivative of z wrt the original input tensor x
dz_dx = t.gradient(z, x)
```

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

1 + 1 + 1 + 1

4^2

$$\boxed{\begin{pmatrix} 8 & 8 \\ 8 & 8 \end{pmatrix}}$$

Then, if we differentiate z with respect to the original tensor x, let's see what happens. Z is 4 to the power of 2, so the differential is 2 times 4, which is 8. As we're differentiating with respect to the original two-by-two matrix, it gets populated with eights.

$$x = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix}$$

Our goal is to calculate the gradient of z with respect to x. According to the chain rule, you can first calculate the gradient of z with respect to y, and then calculate the gradient of y with respect to x. Multiply these two out and you'll get the gradient of z with respect to x.

$$y = x_{1,1} + x_{1,2} + x_{2,1} + x_{2,2}$$ “reduce sum”

$$z = y^2$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x}$$

$$x = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix}$$

$$y = x_{1,1} + x_{1,2} + x_{2,1} + x_{2,2}$$ "reduce sum"

$$z = y^2$$

$$\frac{\partial z}{\partial y} = 2 \times y \qquad \frac{\partial y}{\partial x_{1,1}} = 1 \quad \frac{\partial y}{\partial x_{1,2}} = 1 \quad \frac{\partial y}{\partial x_{2,1}} = 1 \quad \frac{\partial y}{\partial x_{2,2}} = 1$$

The gradient of z with respect to y is 2 times y because z equals y squared. The gradient of y with respect to x is actually the gradient of y with respect to each of the four x variables.

When you take the derivative of y with respect to x_1,1 is just one. The three other x's are treated as constants because you're taking the gradient with respect to just x_1,1.

If you do similar for x_1,2, x_2,1, and x_2,2 the gradient of y with respect to each is one.

$$\frac{\partial z}{\partial x} = \begin{pmatrix} \frac{\partial z}{\partial x_{1,1}} & \frac{\partial z}{\partial x_{1,2}} \\ \frac{\partial z}{\partial x_{2,1}} & \frac{\partial z}{\partial x_{2,2}} \end{pmatrix}$$

$$\frac{\partial z}{\partial x_{1,1}} = \frac{\partial z}{\partial y} \times \frac{\partial dy}{\partial x_{1,1}} \qquad \frac{\partial z}{\partial x_{1,2}} = \frac{\partial z}{\partial y} \times \frac{\partial dy}{\partial x_{1,2}}$$

$$\frac{\partial z}{\partial x_{2,1}} = \frac{\partial z}{\partial y} \times \frac{\partial dy}{\partial x_{2,1}} \qquad \frac{\partial z}{\partial x_{2,2}} = \frac{\partial z}{\partial y} \times \frac{\partial dy}{\partial x_{2,2}}$$

$$x = \begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix} \qquad x = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

$$y = x_{1,1} + x_{1,2} + x_{2,1} + x_{2,2} \qquad y = 1 + 1 + 1 + 1 = 4$$

$$x = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \qquad \frac{\partial z}{\partial y} = 2 \times y = 2 \times 4 \qquad \frac{\partial y}{\partial x_{1,1}} = 1 \quad \frac{\partial y}{\partial x_{1,2}} = 1 \quad \frac{\partial y}{\partial x_{2,1}} = 1 \quad \frac{\partial y}{\partial x_{2,2}} = 1$$

$$y = 4$$

$$\frac{\partial z}{\partial x_{1,1}} = 2 \times 4 \times 1 = 8 \qquad \frac{\partial z}{\partial x_{1,2}} = 2 \times 4 \times 1 = 8$$

$$\frac{\partial z}{\partial x_{2,1}} = 2 \times 4 \times 1 = 8 \qquad \frac{\partial z}{\partial x_{2,2}} = 2 \times 4 \times 1 = 8$$

$$\frac{\partial z}{\partial x} = \begin{pmatrix} 8 & 8 \\ 8 & 8 \end{pmatrix}$$

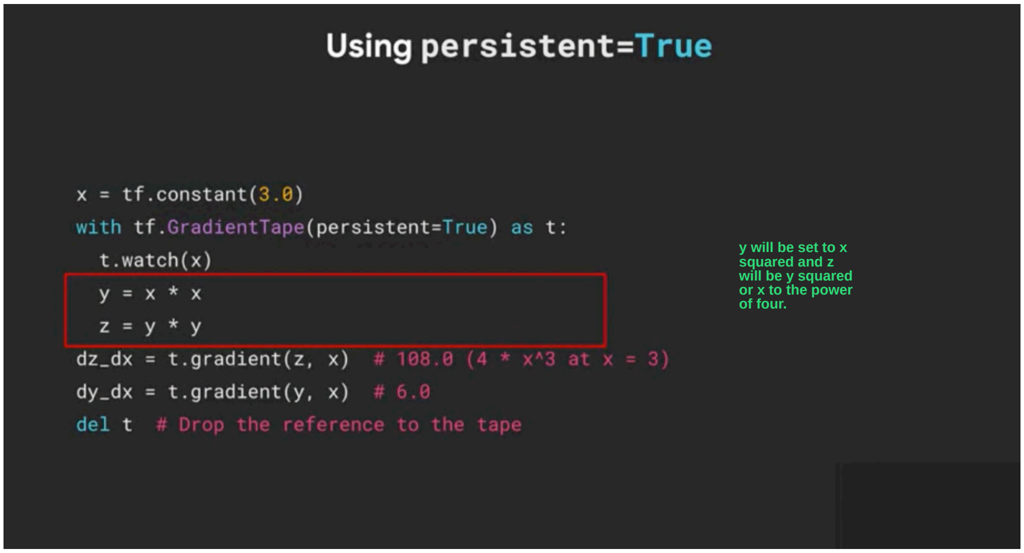
```
Same as:
dz_dx = t.gradient(z, x)
```

Using persistent=True

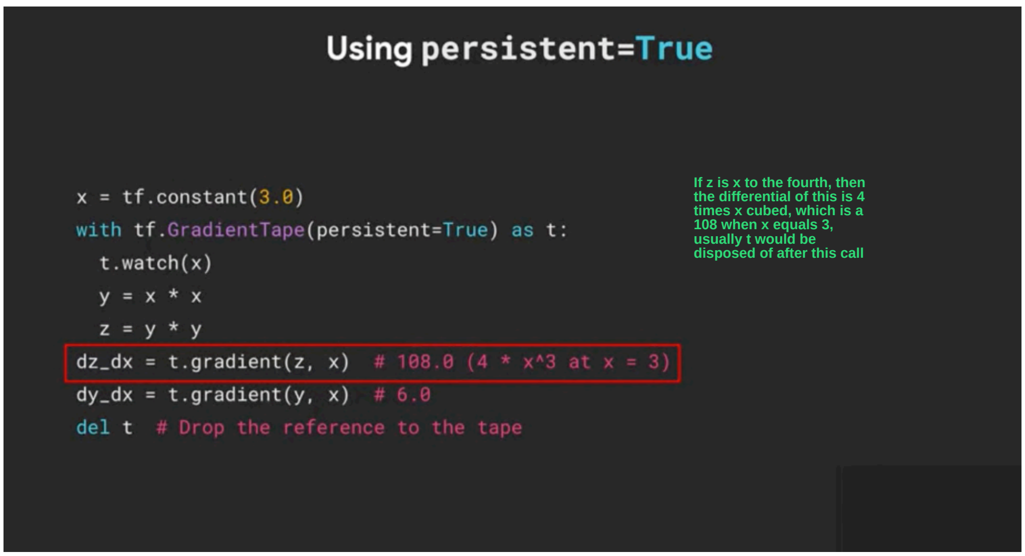
```
x = tf.constant(3.0)
with tf.GradientTape(persistent=True) as t:
  t.watch(x)
  y = x * x
  z = y * y
dz_dx = t.gradient(z, x)  # 108.0 (4 * x^3 at x = 3)
dy_dx = t.gradient(y, x)  # 6.0
del t  # Drop the reference to the tape
```

When you use a gradient tape, the resources held by it are released as soon as the gradient method is called. If you want to use them again, you can set the gradient tape to be persistent by setting persistent equals true as a parameter, for example, in this case we'll call the gradient twice.

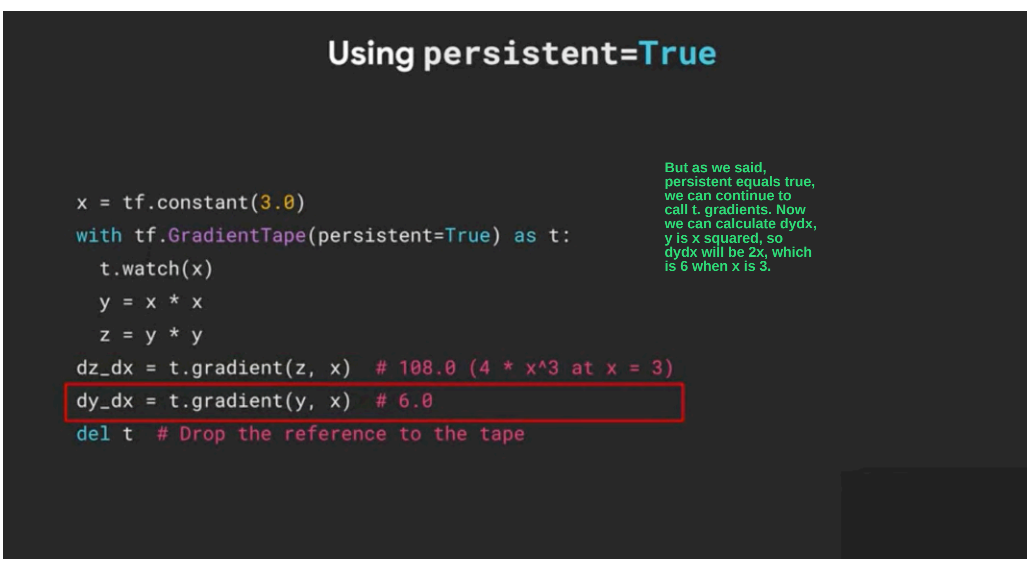
Usually after the first call, the gradient would be disposed of, but this time it isn't because we're setting persistent equals true, so we can use the tape multiple times, for example, here, I'm setting x to be a constant 3.0. My gradient tape will be persistent and called t. I'll ask the tape to watch x



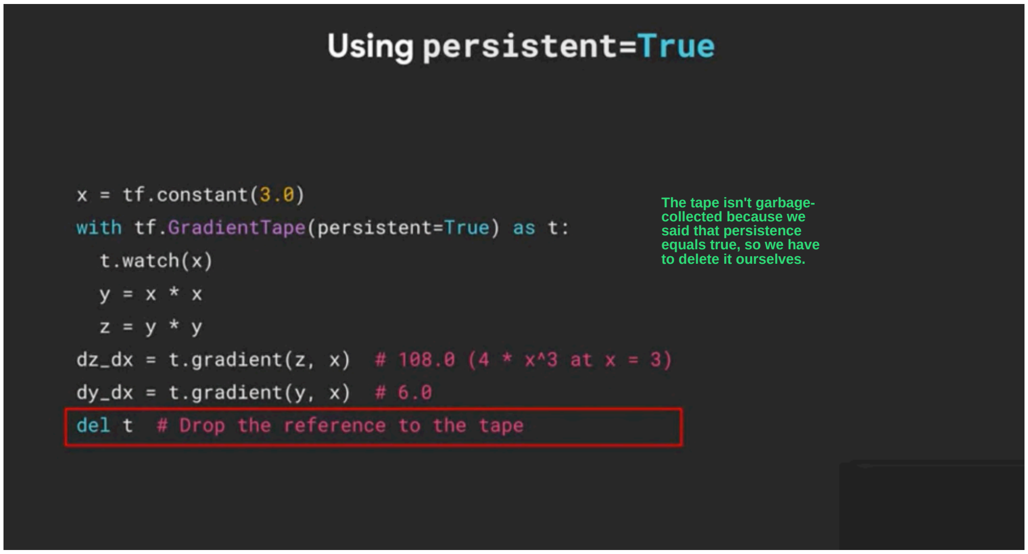
y will be set to x squared and z will be y squared or x to the power of four.



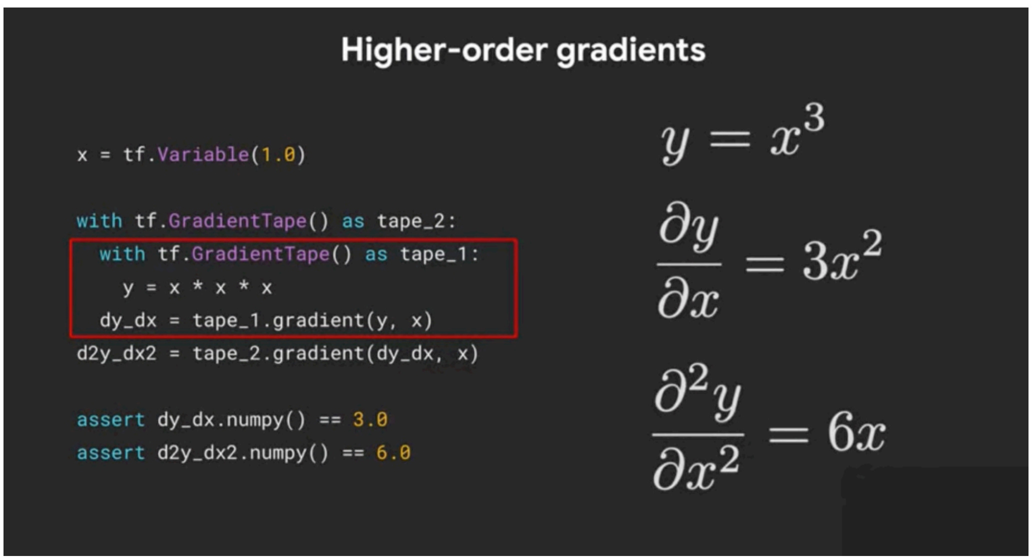
If z is x to the fourth, then the differential of this is 4 times x cubed, which is a 108 when x equals 3, usually t would be disposed of after this call

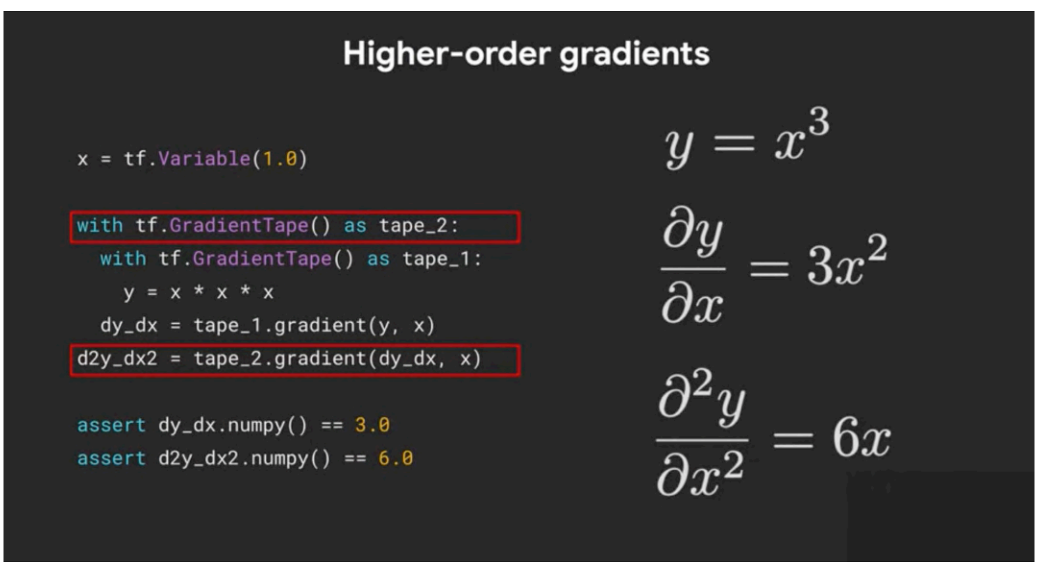


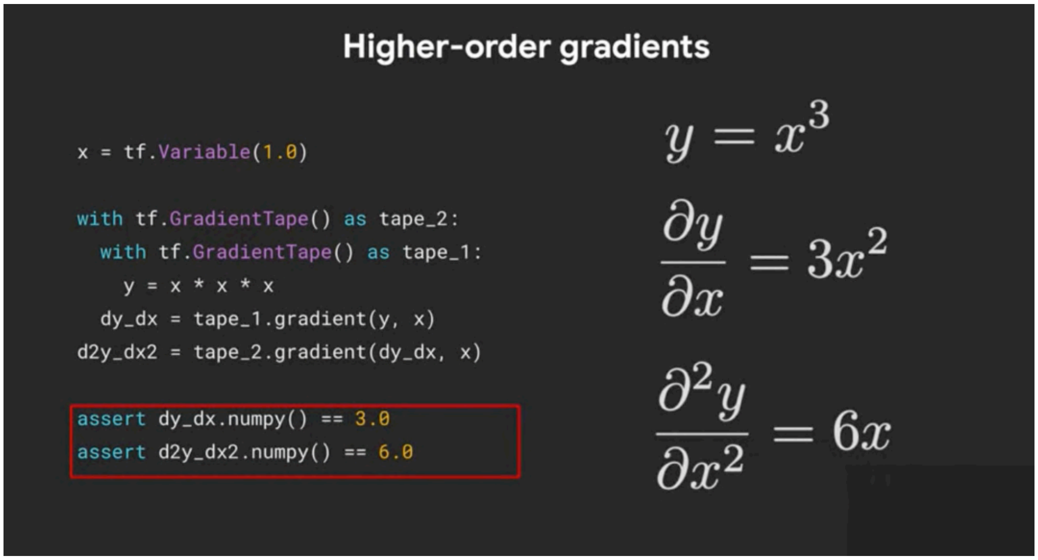
But as we said, persistent equals true, we can continue to call t. gradients. Now we can calculate dydx, y is x squared, so dydx will be 2x, which is 6 when x is 3.



The tape isn't garbage-collected because we said that persistence equals true, so we have to delete it ourselves.







Convolutional Neural Network for Visual Recognition

- https://cs231n.github.io/neural-networks-3/