STAT606
Computing for Data Science and Statistics

Lecture 23: TensorFlow

TensorFlow

Open source symbolic math library

- Popular in ML, especially for neural networks

Developed by GoogleBrain

- Google’s AI/Deep learning division
- You may recall their major computer vision triumph circa 2012:
- http://www.nytimes.com/2012/06/26/technology/in-a-big-network-of-computers-evidence-of-machine-learning.html

TensorFlow is **not** new, and **not** very special:

- Many other symbolic math programs predate it!
- **TensorFlow is unique in how quickly it gained so much market share**
- Open-sourced in 2015…
- ...and almost immediately became the dominant framework for NNs

TensorFlow: Installation

Easiest: `pip install tensorflow`

Also easy: install in anaconda

More information: https://www.tensorflow.org/install/

Note: if you want to do fancier things (e.g., run on GPU instead of CPU), installation and setup gets a lot harder. For this course, we're not going to worry about it. In general, for running on a GPU, if you don't have access to a cluster with existing TF installation, you should consider paying for Amazon/GoogleCloud instances.

Aside: TensorFlow, Versions and Upgrading

In 2019, TensorFlow made a major change from version 1.X to 2.X

This new version of TensorFlow made some fundamental changes

- Added built-in support for Keras https://en.wikipedia.org/wiki/Keras
- Added tricks for computational speedups such as eager execution
  - https://en.wikipedia.org/wiki/Eager_evaluation
- Streamlined code for running models (more on this soon)

These changes are all good, but the changes hide some of the most interesting stuff that TensorFlow can do! I recommend that you at least look at the old TensorFlow, which you can install with

```
pip install tensorflow==1.15
```

Note: TF v1 documentation is archived at: https://www.tensorflow.org/versions/r1.15/api_docs/python/tf

Fundamental concepts of TensorFlow

Tensor

Recall that a tensor is really just an array of numbers
"Rank" of a tensor is the number of dimensions it has
So, a matrix is a rank-2 tensor, vector is rank 1, scalar rank 0
A cube of numbers is a 3-tensor, and so on

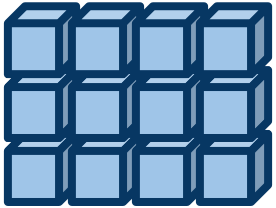
Computational graph

Directed graph that captures the "flow" of data through the program
Nodes are operations (i.e., computations)
Edges represent data sent between operations

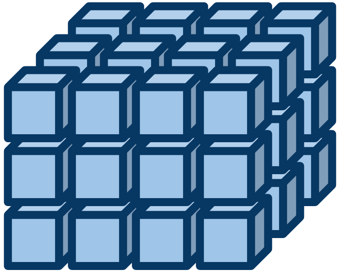
Tensors

0-tensor (scalar)

1-tensor (vector)



2-tensor (matrix)



3-tensor

Note: most things you read will call this dimension the *rank* of the tensor, but you should know that some mathematicians use *rank* to mean the tensor generalization of linear algebraic rank. These people will usually use the term *order* instead.

Tensors: `tf.Tensor` objects

Tensors are represented in TensorFlow as `tf.Tensor` objects

Every `tf.Tensor` object has:

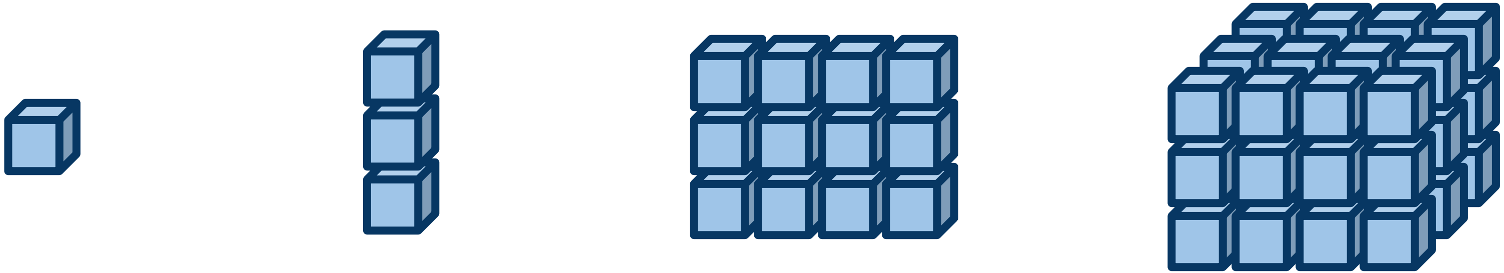
data type (e.g., int, float, string, …)

shape (e.g., 2-by-3-by-5, 5-by-5, 1-by-1, etc)

Shape encodes both rank **and** ‘length’ of each dimension

`tf.Tensor` objects are immutable

with slight exceptions, which we’ll talk about soon



Special `tf.Tensor()` objects

`tf.constant`: will not change its value during your program.

- Like an immutable tensor

`tf.placeholder`: gets its value from elsewhere in your program

- E.g., from training data or from results of other Tensor computations
- **Note:** this was removed in TensorFlow v2; now handled by `tf.function` (in a few slides!)

`tf.Variable`: represents a tensor whose value may change during execution

- Unlike above `tf.Tensor` types, `tf.Variables` are **mutable**
- Useful for ML, because we want to update parameters during training

`tf.SparseTensor`: most entries of a `SparseTensor` will be zero

- TF stores this differently; saves on memory
- Useful for applications where data is sparse, such as networks

Special `tf.Tensor()` objects

`tf.constant`: will not change its value during your program.

- Like an immutable tensor

`tf.placeholder`: gets its value from elsewhere in your program

- E.g., from training data or from results of other Tensor computations
- **Note:** this was removed in TensorFlow v2; now handled by `tf.function` (in a few slides!)

`tf.Variable`: represents a tensor whose value may change during execution

- Unlike above `tf.Tensor` types, `tf.Variables` are **mutable**
- Useful for ML, because we want to update parameters during training

`tf.SparseTensor`: most entries of a SparseTensor will be zero

For now, these three are the important ones.

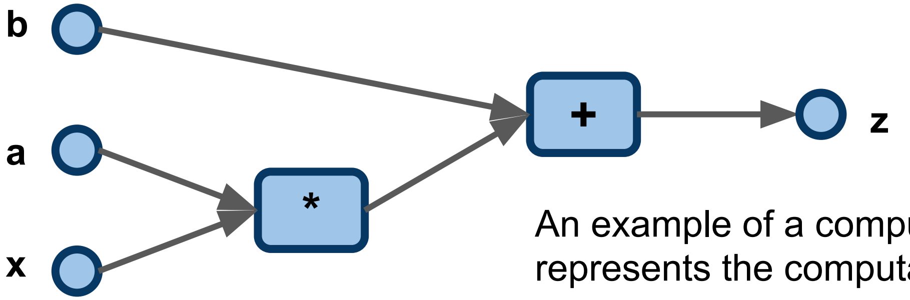
Computational Graph

From the “Getting Started” guide: “A **computational graph** is a series of TensorFlow operations arranged into a graph of nodes.”

Every node takes zero or more tensors as input and outputs one or more tensors.

A TensorFlow program consists, essentially, of two sections:

1) Building the computational graph
2) Running the computational graph



An example of a computational graph that represents the computation `z = a*x + b`.

TF as Dataflow

Dataflow is a term for frameworks in which computation is concerned with the **pipeline** by which the data is processed

Data transformed and combined via a series of operations

This view makes it clear when parallelization is possible…

...because dependence between operations can be read off the graph

https://en.wikipedia.org/wiki/Dataflow

https://en.wikipedia.org/wiki/Stream_processing

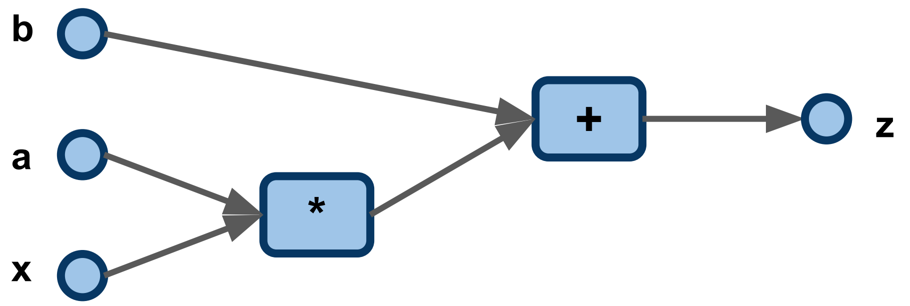
This should sound familiar from PySpark!

Building the Computational Graph

```
a = tf.constant(2, dtype=tf.float32)
b = tf.constant(3, dtype=tf.float32)
x = tf.constant(4, dtype=tf.float32)
z = a*x + b
```

Here’s a snippet of a TF program in which we define a computational graph.

Equivalent computational graph:

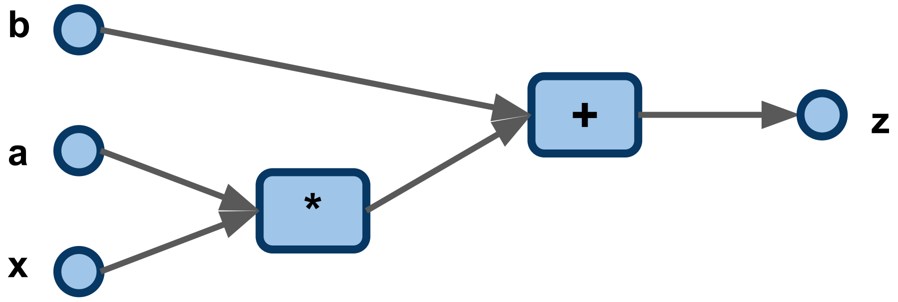


Note: strictly speaking, we haven’t actually built this graph, yet. For that, we need to create a `tf.Graph` object, but working with this object directly is deprecated in TF version 2.

Building the Computational Graph

```
a = tf.constant(2, dtype=tf.float32)
b = tf.constant(3, dtype=tf.float32)
x = tf.constant(4, dtype=tf.float32)
z = a*x + b
```

Equivalent computational graph:



a, b and x here are constants, meaning they're fixed for the duration of our program. Really, we want to, say, let x take values from a data set and let a and b be parameters that we can tune to fit that data. We'll come back to this point.

Data types in TensorFlow

Every `tf.Tensor()` object has a data type, accessed through the `dtype` attribute.

Four basic data types:

Strings
Integers
Floats
Complex numbers

```
helloworld = tf.constant('hello world!')
print(helloworld.dtype)
ramanujan = tf.constant(1729, dtype=tf.int16)
print(ramanujan.dtype)
approxpi = tf.constant(3.14159, dtype=tf.float32)
print(approxpi.dtype)
imaginary = tf.constant((0.0,1.0), dtype=tf.complex64)
print(imaginary.dtype)
```

Some flexibility in specifying precision

```
<dtype: 'string'>
<dtype: 'int16'>
<dtype: 'float32'>
<dtype: 'complex64'>
```

Note: if no `dtype` is specified, TF will do its best to figure it out from context, but this doesn't always go as expected, such as when you want a vector of complex numbers. When in doubt, specify!

Creating Tensors

These are all rank-0 tensors. Yes, `tf.string` is a single item, and so is `tf.complex`.

```
helloworld = tf.constant('hello world!', dtype=tf.string)
ramanujan = tf.constant(1729, dtype=tf.int16)
approxpi = tf.constant(3.14159, dtype=tf.float32)
imaginary = tf.constant((0.0,1.0), dtype=tf.complex64)
```

To create a 1-tensor (i.e., a vector), just pass a list of scalars.

```
animals = tf.constant(['cat','dog','bird'], dtype=tf.string)
print(animals)
```

```
tf.Tensor([b'cat' b'dog' b'bird'], shape=(3,), dtype=string)
```

```
animals.shape
```

```
TensorShape([3])
```

Note: all elements of a `tf.Tensor` must be of the same data type. The one sneaky way around this is to serialize objects to strings and store them in a tensor with `dtype=tf.string`.

Creating Tensors

```
onebyonemx = tf.constant([[3.1415]], dtype=tf.float32)
print(onebyonemx)
```

```
tf.Tensor([[3.1415]], shape=(1, 1), dtype=float32)
```

```
onevec = tf.constant([3.1415], dtype=tf.float32)
print(onevec)
```

```
tf.Tensor([3.1415], shape=(1,), dtype=float32)
```

```
scalar = tf.constant(3.1415, dtype=tf.float32)
print(scalar)
```

```
tf.Tensor(3.1415, shape=(), dtype=float32)
```

We can create a 1-by-1 matrix, which is **different** from a 1-vector, which is different from a scalar.

Creating Tensors

```
identity = tf.constant([[1,0,0],[0,1,0],[0,0,1]], dtype=tf.float32)
print(identity)
```

```
tf.Tensor(
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]], shape=(3, 3), dtype=float32)
```

To create a matrix, we can pass a list of its rows.

```
oneThruNine = tf.constant([[1,2,3],[4,5,6],[7,8,9]], dtype=tf.float32)
print(oneThruNine)
```

```
tf.Tensor(
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]], shape=(3, 3), dtype=float32)
```

Matrix populated in row-major order.

Creating Tensors

Create a 5-by-5 matrix of all ones

```
J = tf.ones([5,5])
print(J)
```

```
tf.Tensor(
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]], shape=(5, 5), dtype=float32)
```

```
video = tf.zeros([27,1280,720,3])
```

Create a 4-tensor, which we could use to represent one second of 720p color video (27 frames per second, 1280x720 resolution, 3 colors)

Tensor rank and shape

Rank: number of dimensions

Shape: sizes of the dimensions

```
video = tf.zeros([27,1280,720,3])
print(tf.rank(video))
```

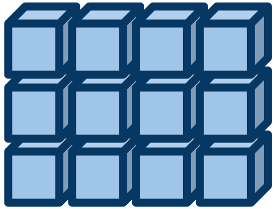
```
tf.Tensor(4, shape=(), dtype=int32)
```

```
print(video.shape)
```

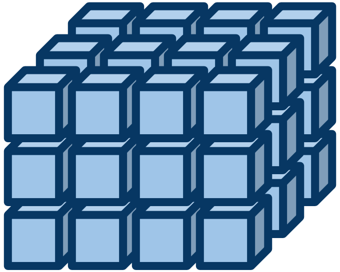
```
(27, 1280, 720, 3)
```

```
video.shape
```

```
TensorShape([27, 1280, 720, 3])
```



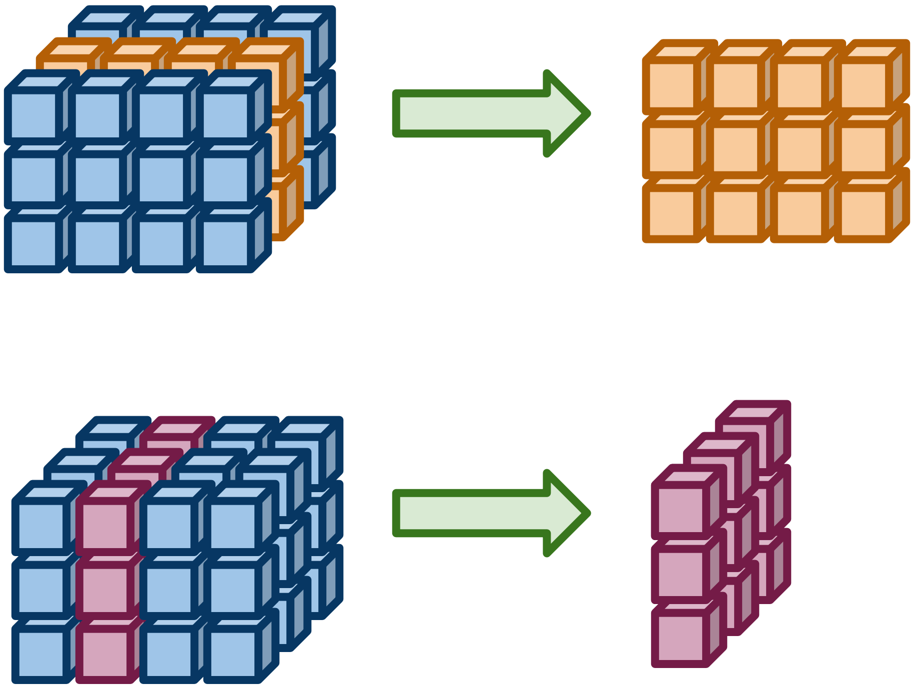
Rank 2, shape 3-by-4



Rank 3, shape 3-by-4-by-3

Note: This looks like a tuple, but it is actually its own special type, `tf.TensorShape`

Tensor Slices



It is often natural to refer to certain subsets of the entries of a tensor. A “subtensor” of a tensor is often called a **slice**, and the operation of picking out a slice is called **slicing** the tensor.

Tensor Indexing

```
fibovec = tf.constant([1,1,2,3,5,8,13,22], tf.int32)
print(fibovec)
```

```
tf.Tensor([ 1  1  2  3  5  8 13 22], shape=(8,), dtype=int32)
```

```
print(fibovec[0])
```

```
tf.Tensor(1, shape=(), dtype=int32)
```

One index is enough to specify a number in a vector (i.e., a 1-tensor)

```
J = tf.ones([3,4])
print(J)
```

```
tf.Tensor(
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]], shape=(3, 4), dtype=float32)
```

```
J[1,2]
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=1.0>
```

Need two indices to pick out an entry of a matrix (i.e., a 2-tensor)

Tensor Slices

Use ':' to pick out all entries along a row or column.

```
J = tf.ones([3,4])
print(J)
```

```
tf.Tensor(
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]], shape=(3, 4), dtype=float32)
```

```
J[1,2]
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=1.0>
```

```
J[1,:]
```

Create a vector from the second (zero-indexing!) row of the matrix.

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([1., 1., 1., 1.], dtype=float32)>
```

```
J[:,2]
```

Create a vector from the third column of the matrix.

```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([1., 1., 1.], dtype=float32)
```

Note: result is a “column vector” regardless of whether we slice a row or a column!

Tensor Slices

Use ':' to pick out all entries along a row or column.

```
J = tf.ones([3,4])
print(J)
```

```
tf.Tensor(
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]], shape=(3, 4), dtype=float32)
```

```
J[1,2]
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=1.0>
```

Sidenote: the data inside a Tensor object is really just a `numpy` array!

```
J[1,:]
```

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([1., 1., 1., 1.], dtype=float32)>
```

```
J[:,2]
```

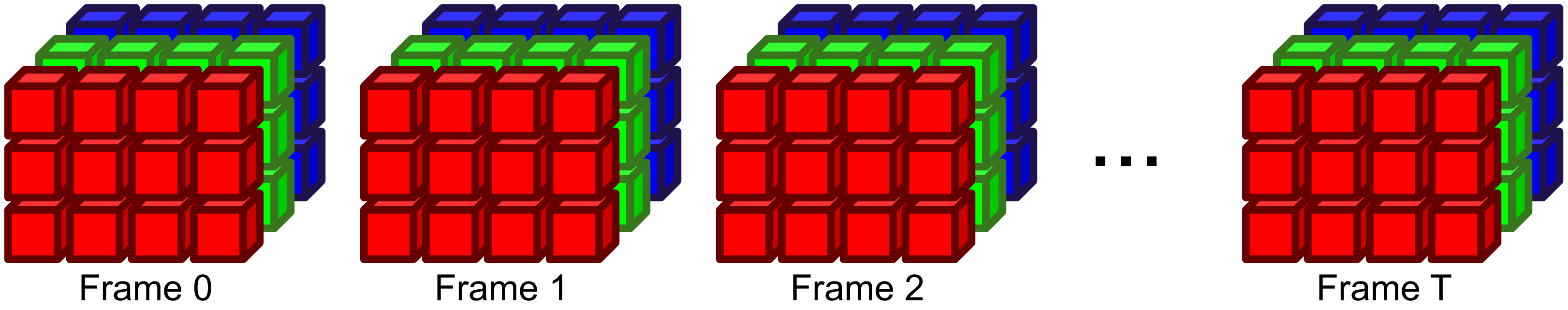
```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([1., 1., 1.], dtype=float32)>
```

Tensor Slices

More complicated example: video processing

Four dimensions:

- Pixels (height-by-width)
- Three colors (RGB)
- Time index (multiple frames)



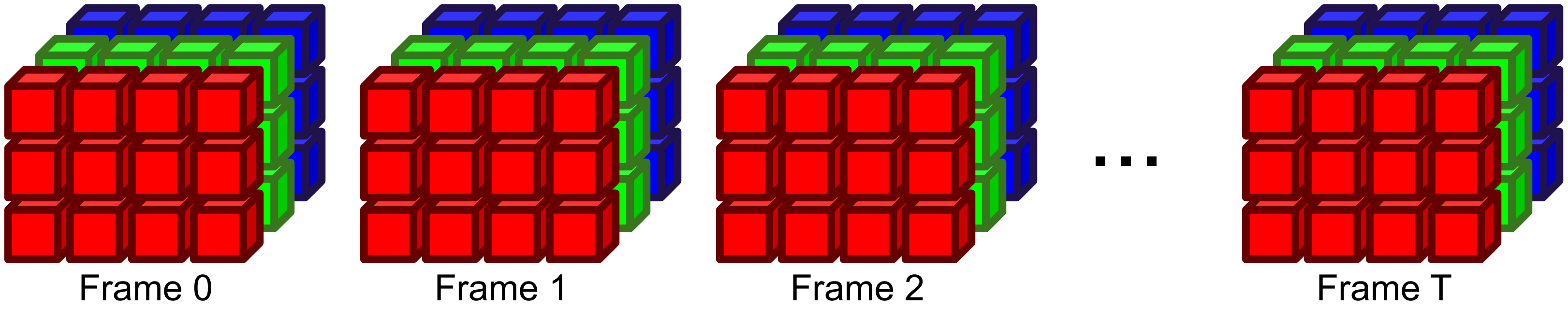
Tensor Slices

More complicated example: video processing

Four dimensions:

- Pixels (height-by-width)
- Three colors (RGB)
- Time index (multiple frames)

Test your understanding:
What is the rank of the "video" tensor below?



Tensor Slices

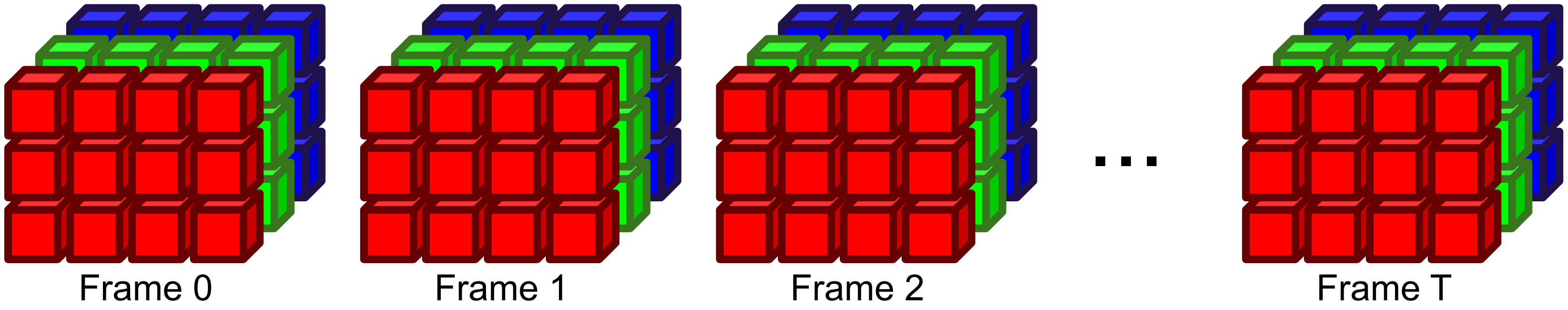
More complicated example: video processing

Four dimensions:

- Pixels (height-by-width)
- Three colors (RGB)
- Time index (multiple frames)

Test your understanding: What is the rank of the "video" tensor below?

Answer: 4, since there are four dimensions; height, width, color and time.



Tensor Slices

```
video = tf.zeros([27,1280,720,3])
video.shape
```

```
TensorShape([27, 1280, 720, 3])
```

Use ':' to pick out all entries along a row or column.

```
firstframe = video[0,:,:,:]
firstframe.shape
```

```
TensorShape([1280, 720, 3])
```

Pick out the 3-color 1280-by-720 image that is the first frame of the video

```
bluevideo = video[:,:,:,2]
bluevideo.shape
```

```
TensorShape([27, 1280, 720])
```

Pick out only the blue channel of the video (see RGB on wikipedia)

```
redvideo = video[:,:,:,0]
redvideo.shape
```

```
TensorShape([27, 1280, 720])
```

Pick out only the red channel of the video

Reshaping tensors

Test your understanding:

Q: I have an x-by-y-by-z tensor. What is its rank?

Reshaping tensors

Test your understanding:

Q: I have an x-by-y-by-z tensor. What is its rank?

A: 3

Q: How many elements are in this x-by-y-by-z 3-tensor?

Reshaping tensors

Test your understanding:

Q: I have an x-by-y-by-z tensor. What is its rank?

A: 3

Q: How many elements are in this x-by-y-by-z 3-tensor?

A: x*y*z

Reshaping tensors

```
mytensor = tf.zeros([10,20,30])
mytensor.shape
```

```
TensorShape([10, 20, 30])
```

Reshape a 3-tensor into a 4-tensor. Note that the shapes are consistent with one another.

```
newtensor = tf.reshape(mytensor, [125,3,2,8])
newtensor.shape
```

```
TensorShape([125, 3, 2, 8])
```

Reshaping to an inconsistent shape results in an error.

```
badtensor = tf.reshape(mytensor, [10,20,40])
```

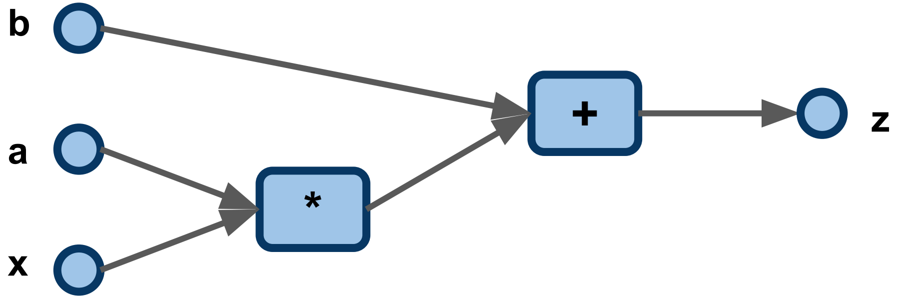
```
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
<ipython-input-71-fecb512dde90> in <module>
----> 1 badtensor = tf.reshape(mytensor, [10,20,40])
```

Incorporating Variable Tensors

```
a = tf.constant(2, dtype=tf.float32)
b = tf.constant(3, dtype=tf.float32)
x = tf.constant(4, dtype=tf.float32)
z = a*x + b
```

In practice, we want to be able to change `a` and `b` to adjust our model. Right now, they're constants, and cannot be changed.

Equivalent computational graph:

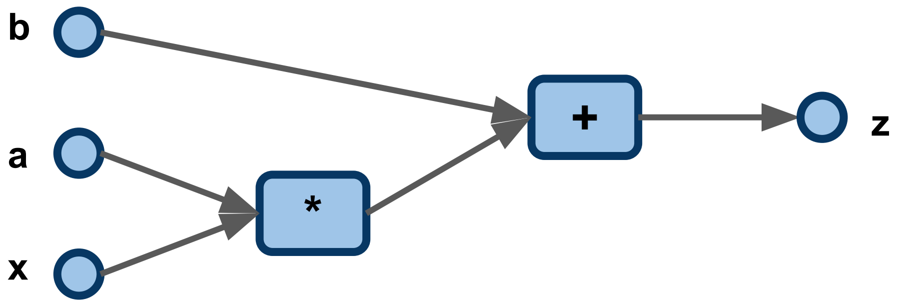


Incorporating Variable Tensors

```
a = tf.constant(2, dtype=tf.float32)
b = tf.constant(3, dtype=tf.float32)
x = tf.constant(4, dtype=tf.float32)
z = a*x + b
```

In practice, we want to be able to change `a` and `b` to adjust our model. Right now, they're constants, and cannot be changed.

Equivalent computational graph:



The solution is to make `a` and `b` Variable tensors.

Incorporating Variable Tensors

```
a = tf.Variable( 2, dtype=tf.float32)
b = tf.Variable( 3, dtype=tf.float32)
x = tf.constant( 1, dtype=tf.float32)
a*x + b
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=5.0>
```

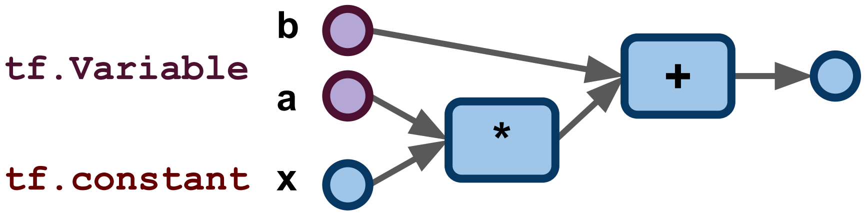
Declare `a` and `b` to be Variable tensors.

```
a.assign(2)
b.assign(1)
a*x + b # x is still 1; 2*1 + 1 = 3
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=3.0>
```

Change values of Variable tensors using the `assign` method.

Equivalent computational graph



Incorporating Variable Tensors

```
a = tf.Variable( 2, dtype=tf.float32)
b = tf.Variable( 3, dtype=tf.float32)
x = tf.constant( 1, dtype=tf.float32)
a*x + b
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=5.0>
```

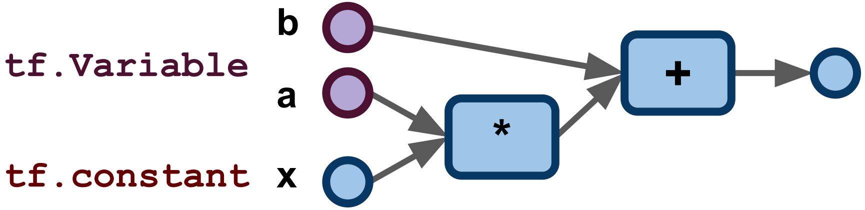
Declare `a` and `b` to be Variable tensors.

```
a.assign(2)
b.assign(1)
a*x + b # x is still 1; 2*1 + 1 = 3
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=3.0>
```

Change values of Variable tensors using the `assign` method.

Equivalent computational graph



Incorporating Variable Tensors

```
a = tf.Variable( 2, dtype=tf.float32)
b = tf.Variable( 3, dtype=tf.float32)
x = tf.constant( 1, dtype=tf.float32)
a*x + b
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=5.0>
```

Declare `a` and `b` to be Variable tensors.

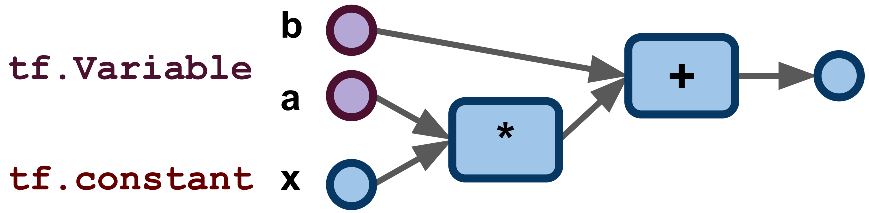
```
a.assign(2)
b.assign(1)
a*x + b # x is still 1; 2*1 + 1 = 3
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=3.0>
```

Change values of Variable tensors using the `assign` method.

Note: in practice, we rarely need to use the `assign` method directly. It is mostly used under the hood by TensorFlow to change our parameters as we are fitting a model.

Equivalent computational graph



Building the computational graph: `tf.function`

```
a = tf.Variable( 2, dtype=tf.float32)
b = tf.Variable( 3, dtype=tf.float32)
x = tf.constant( 1, dtype=tf.float32)
a*x + b
```

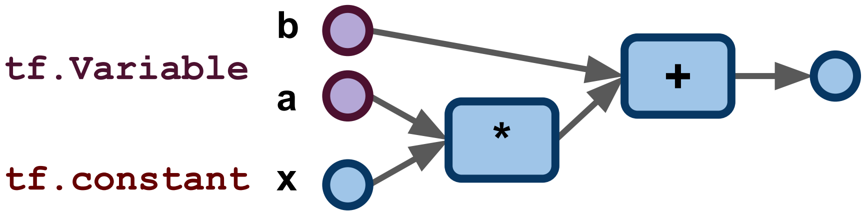
```
<tf.Tensor: shape=(), dtype=float32, numpy=5.0>
```

The predictor `x` is still a constant. What if I want to plug in new data to this linear model?

```
a.assign(2)
b.assign(1)
a*x + b # x is still 1; 2*1 + 1 = 3
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=3.0>
```

Equivalent computational graph



Building the computational graph: `tf.function`

```
def linear(c,d,z):
    return c*z + d
linear_tf = tf.function(linear)
linear_tf
```

```
<tensorflow.python.eager.def_function.Function at 0x7fbb582eab50>
```

We start with a Python function...

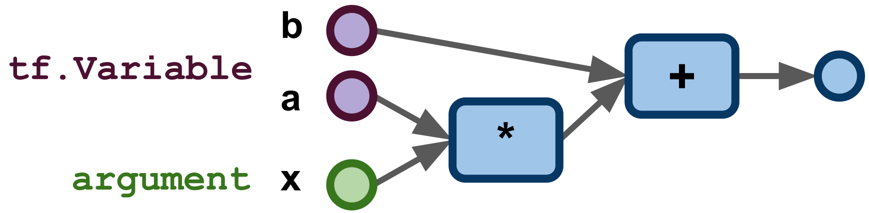
...which we then convert into a `tf.Function` object.

```
a = tf.Variable( 2, dtype=tf.float32)
b = tf.Variable( 3, dtype=tf.float32)
z = tf.constant( 1, dtype=tf.float32)
linear_tf(a,b,z)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=5.0>
```

And now we can evaluate this function on different values.

Equivalent computational graph



Building the computational graph: `tf.function`

```
def linear(c,d,z):
    return c*z + d
linear_tf = tf.function(linear)
linear_tf
```

```
<tensorflow.python.eager.def_function.Function at 0x7fbb582eab50>
```

We start with a Python function...

...which we then convert into a `tf.Function` object.

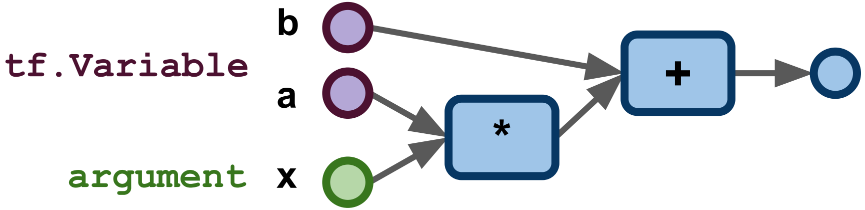
```
a = tf.Variable( 2, dtype=tf.float32)
b = tf.Variable( 3, dtype=tf.float32)
z = tf.constant( 1, dtype=tf.float32)
linear_tf(a,b,z)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=5.0>
```

And now we can evaluate this function on different values.

Note: we might like to have Variable tensors `a` and `b` defined inside `linear_tf`, but this can cause problems with TensorFlow's eager execution.

Equivalent computational graph



Running TensorFlow

```
def lin_comb(x,y):
    # Now we define some variables
    a = tf.constant(2, dtype=tf.float32)
    b = tf.constant(1, dtype=tf.float32)
    return a*x + b*y

linear_combination = tf.function(lin_comb)
linear_combination
```

```
<tensorflow.python.eager.def_function.Function at 0x7fbb58288460>
```

Operations are defined here, but we still haven't actually computed anything, yet...

```
linear_combination(4,2)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=10.0>
```

Evaluate our computational graph with particular values given to x and y.

Running TensorFlow

```
def lin_comb(x,y):
    # Now we define some variables
    a = tf.constant(2, dtype=tf.float32)
    b = tf.constant(1, dtype=tf.float32)
    return a*x + b*y

linear_combination = tf.function(lin_comb)
linear_combination
```

```
<tensorflow.python.eager.def_function.Function at 0x7fbb58288460>
```

```
linear_combination(4,2)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=10.0>
```

Note that we have the constants `a` and `b` defined locally in the function, this time. This is only an issue for Variable tensors.

Running TensorFlow

```
def lin_comb(x,y):
    # Now we define some variables
    a = tf.constant(2, dtype=tf.float32)
    b = tf.constant(1, dtype=tf.float32)
    return a*x + b*y

linear_combination = tf.function(lin_comb)
linear_combination
```

```
<tensorflow.python.eager.def_function.Function at 0x7fbb58288460>
```

Operations are defined here, but we still haven’t actually computed anything, yet...

```
linear_combination(4,2)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=10.0>
```

```
linear_combination([4,3,2,1], [2,3,4,5])
```

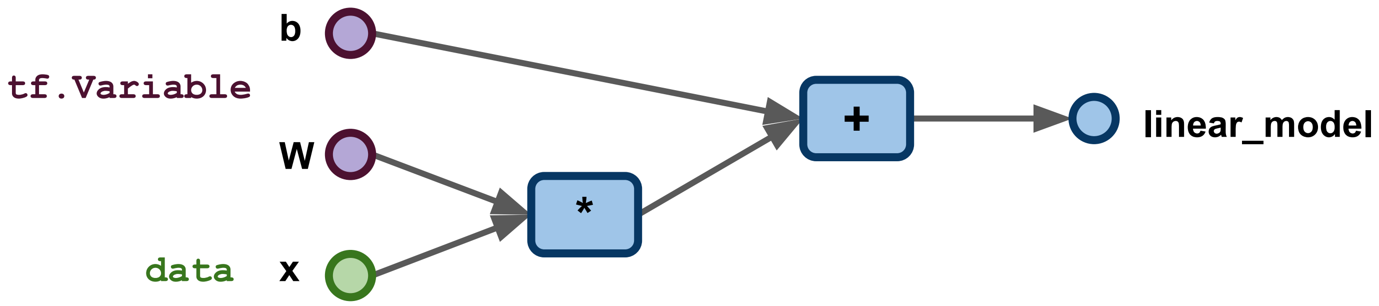
```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([10.,  9.,  8.,  7.], dtype=float32)>
```

Once our `tf.Function` is defined, we can evaluate it on a collection of arguments. For example, we might want to pass in a collection of (x,y) pairs.

Building a Simple Model: Linear Regression

```
def linear_prediction_pyfn(c,d,x):
    return c*x + d
linear_model = tf.function(linear_prediction_pyfn)

W = tf.Variable([0.5], dtype=tf.float32)
b = tf.Variable([-1], dtype=tf.float32)
linear_model(W,b,[0,1,2,3,4])
```

```
<tf.Tensor: shape=(5,), dtype=float32, numpy=array([-1. , -0.5,  0. ,  0.5,  1. ], dtype=float32)>
```



Building a Simple Model: Linear Regression

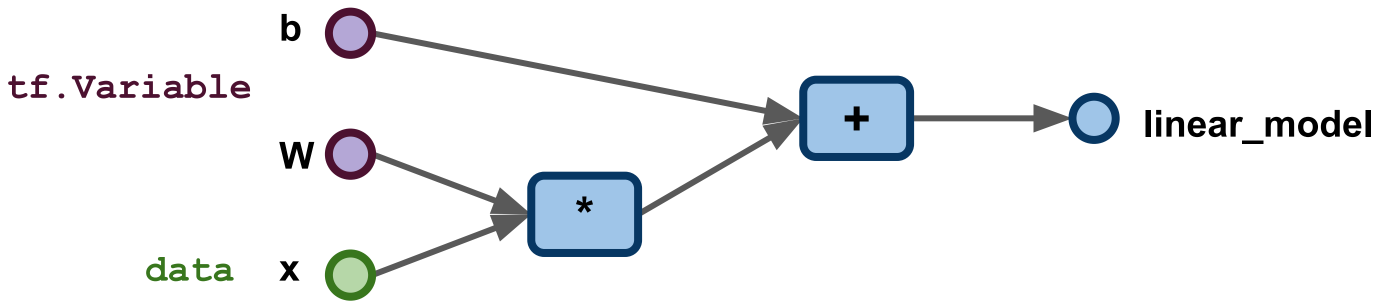
```
def linear_prediction_pyfn(c,d,x):
    return c*x + d
linear_model = tf.function(linear_prediction_pyfn)

W = tf.Variable([0.5], dtype=tf.float32)
b = tf.Variable([-1], dtype=tf.float32)
linear_model(W,b,[0,1,2,3,4])
```

```
<tf.Tensor: shape=(5,), dtype=float32, numpy=array([-1. , -0.5,  0. ,  0.5,  1. ], dtype=float32)>
```

Model: y = Wx + b

We're using `c` and `d` in the function arguments just to avoid confusion with the global variables `W` and `b`.

`W` and `b` are both rank-1 tensors, with values 0.5 and -1, respectively.

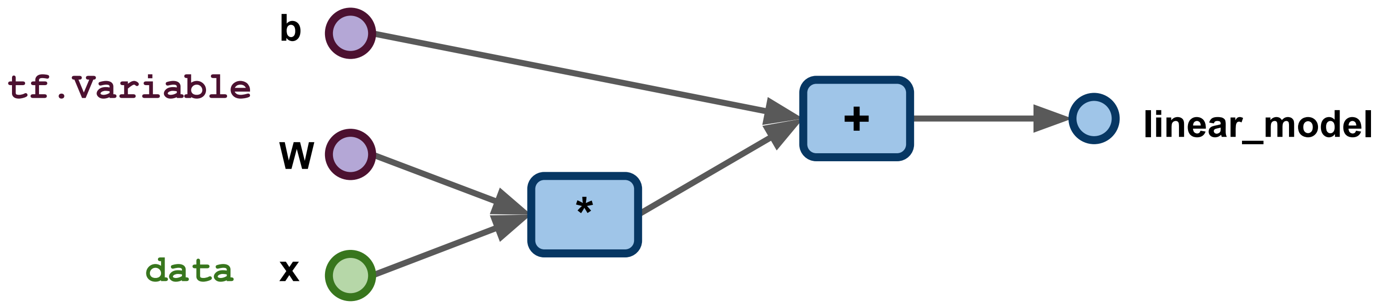


Building a Simple Model: Linear Regression

```
def linear_prediction_pyfn(c,d,x):
    return c*x + d
linear_model = tf.function(linear_prediction_pyfn)

W = tf.Variable([0.5], dtype=tf.float32)
b = tf.Variable([-1], dtype=tf.float32)
linear_model(W,b,[0,1,2,3,4])
```

```
<tf.Tensor: shape=(5,), dtype=float32, numpy
```

A `tf.Variable` can be a Tensor of any type and shape. The type and shape of the variable are specified by the initialization. After construction, the value can be changed using the **assign method** that we saw earlier.

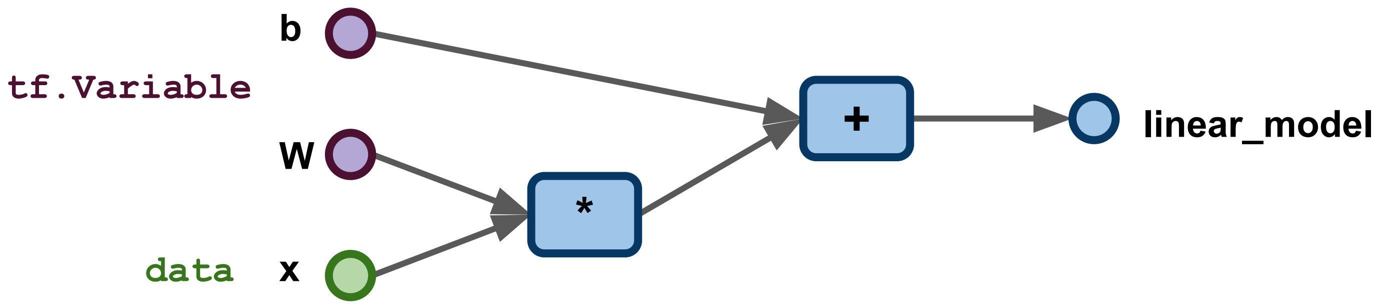


Building a Simple Model: Linear Regression

```python
def linear_prediction_pyfn(c,d,x):
    return c*x + d
linear_model = tf.function(linear_prediction_pyfn)

W = tf.Variable([0.5], dtype=tf.float32)
b = tf.Variable([-1], dtype=tf.float32)
linear_model(W,b,[0,1,2,3,4])
```

```
<tf.Tensor: shape=(5,), dtype=float32, numpy=array([-1. , -0.5,  0. ,  0.5,  1. ], dtype=float32)>
```

Evaluate the model with different values of `x`.

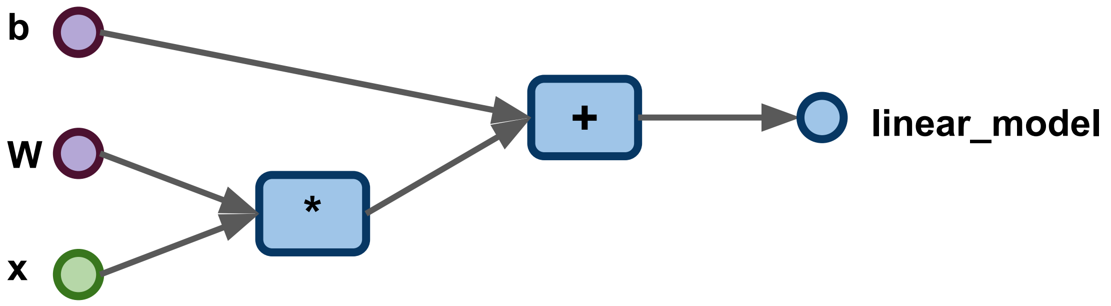


Building a Simple Model: Linear Regression

So far, we have a circuit that computes a linear regression estimate

To train our model, we need:

1) A loss function
2) An argument y for the training data dependent values



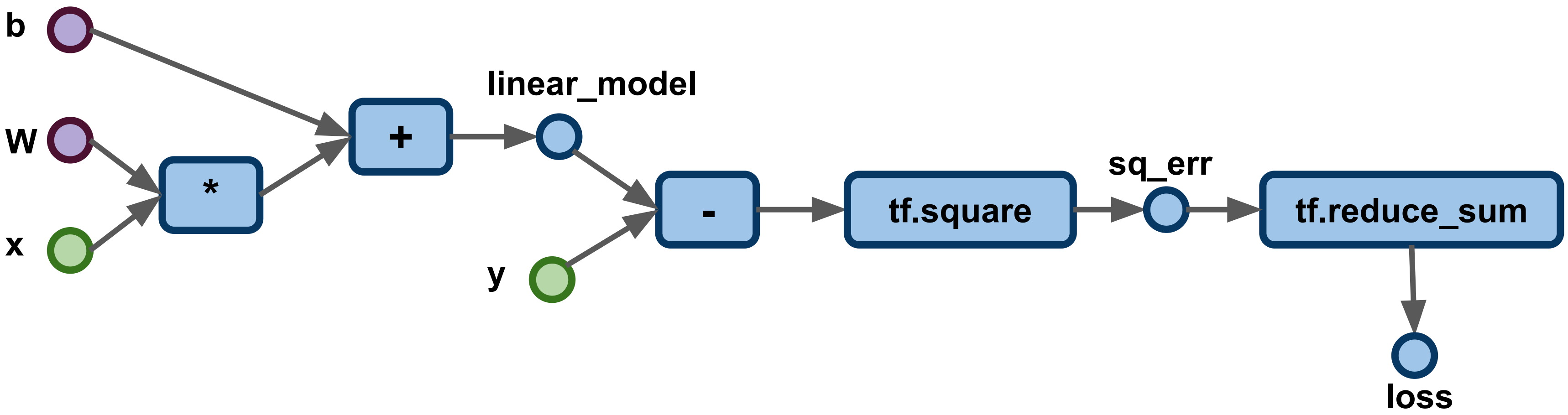
Building a Simple Model: Linear Regression

```
y = tf.placeholder(tf.float32)
sq_err = tf.square(linear_model - y)
loss = tf.reduce_sum(sq_err)
print(sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]}))
```

```
23.66
```

Warning: this code was run in TensorFlow version 1. It will not run in TF v2.

In TF v1, we could just keep building out this graph to define a loss function.



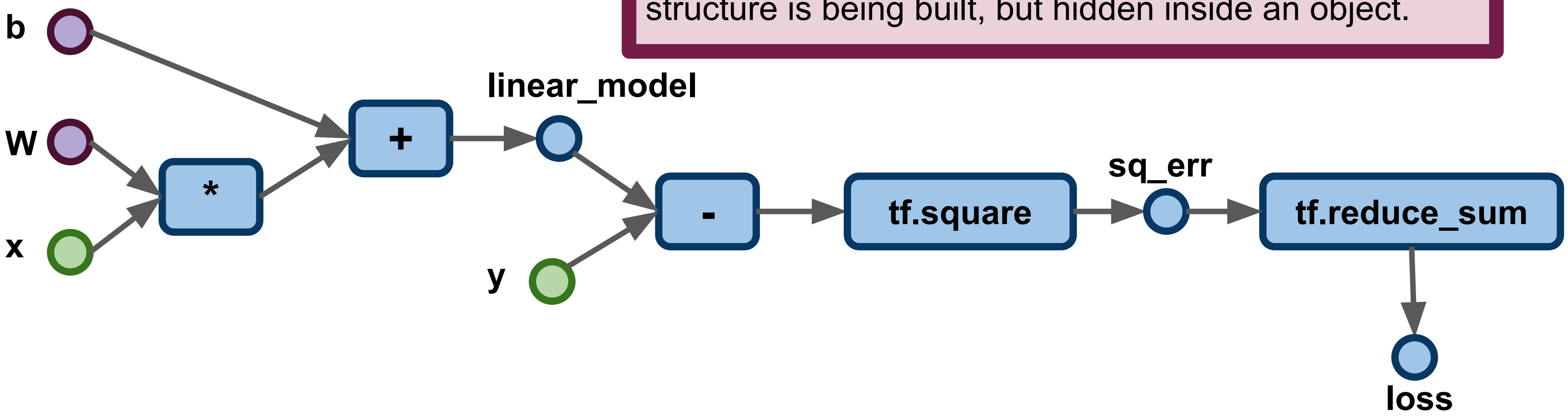
Building a Simple Model: Linear Regression

```
y = tf.placeholder(tf.float32)
sq_err = tf.square(linear_model - y)
loss = tf.reduce_sum(sq_err)
print(sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]}))
```

```
23.66
```

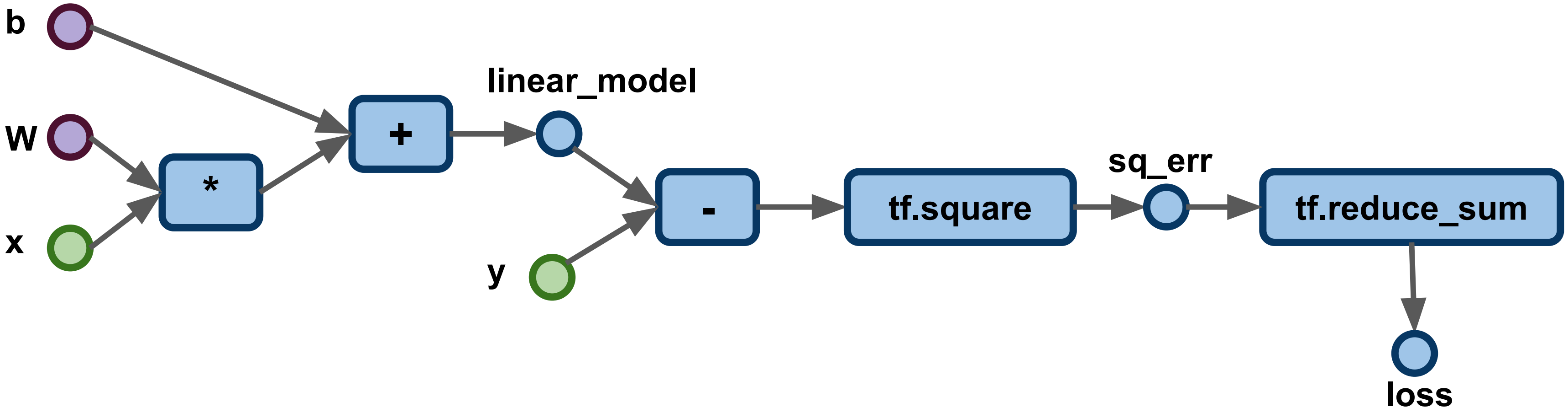
Warning: this code was run in TensorFlow version 1. It will not run in TF v2.

In TF v2, this gets wrapped in the `tf.Module` class, which is borrowed from Keras. The same basic structure is being built, but hidden inside an object.



Building a Simple Model: Linear Regression

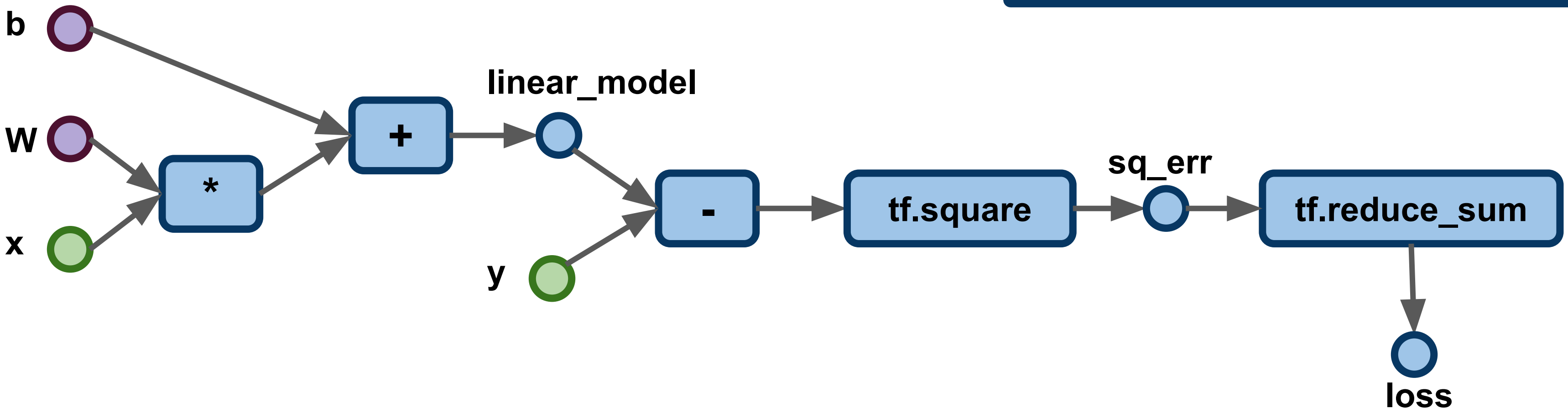
```
class LinearModel(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)
        self.W = tf.Variable([0.5], dtype=tf.float32, name="slope")
        self.b = tf.Variable([-1.0], dtype=tf.float32, name="intercept")
    def __call__(self, x):
        return self.W * x + self.b

def loss(y_observed, y_predicted):
    return tf.reduce_sum(tf.square(y_observed - y_predicted))
```



Building a Simple Model: Linear Regression

```python
class LinearModel(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)
        self.W = tf.Variable([0.5], dtype=tf.float32, name="slope")
        self.b = tf.Variable([-1.0], dtype=tf.float32, name="intercept")
    def __call__(self, x):
        return self.W * x + self.b

def loss(y_observed, y_predicted):
    return tf.reduce_sum(tf.square(y_observed - y_predicted))
```

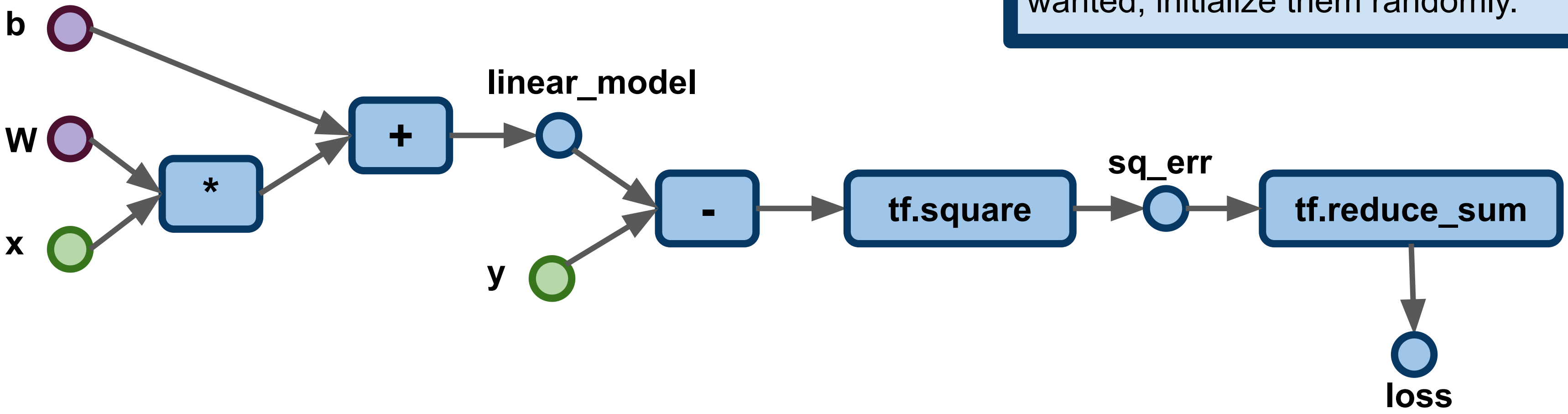
The Python `super()` function accesses the parent class (i.e., the class we are inheriting from).



Building a Simple Model: Linear Regression

```python
class LinearModel(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)
        self.W = tf.Variable([0.5], dtype=tf.float32, name="slope")
        self.b = tf.Variable([-1.0], dtype=tf.float32, name="intercept")
    def __call__(self, x):
        return self.W * x + self.b

def loss(y_observed, y_predicted):
    return tf.reduce_sum(tf.square(y_observed - y_predicted))
```

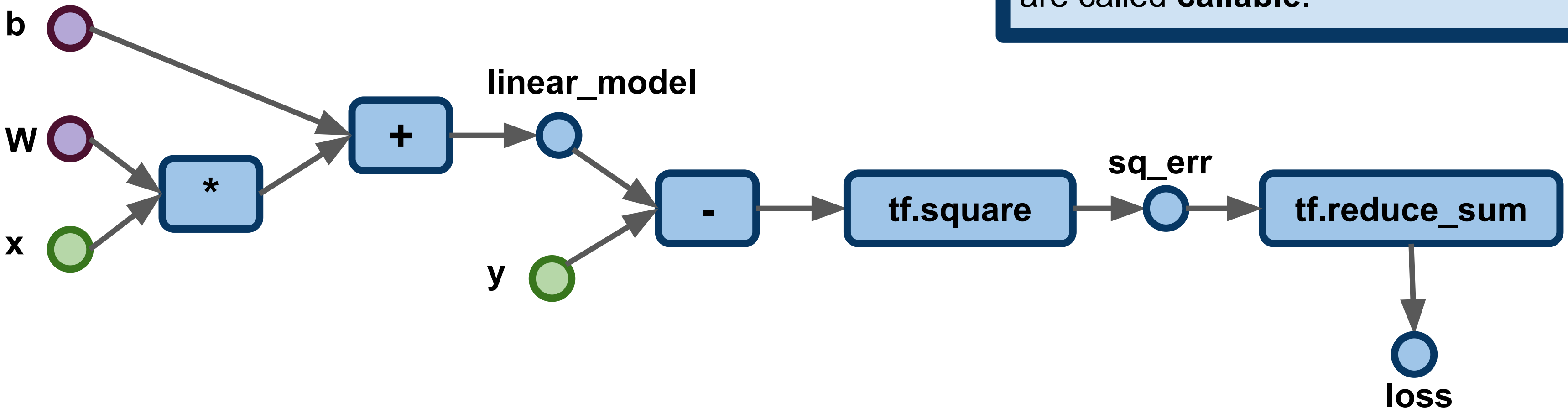
The model parameters are `tf.Variable` tensors stored as instance attributes. We could, if we wanted, initialize them randomly.



Building a Simple Model: Linear Regression

```
class LinearModel(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)
        self.W = tf.Variable([0.5], dtype=tf.float32, name="slope")
        self.b = tf.Variable([-1.0], dtype=tf.float32, name="intercept")
    def __call__(self, x):
        return self.W * x + self.b

def loss(y_observed, y_predicted):
    return tf.reduce_sum(tf.square(y_observed - y_predicted))
```

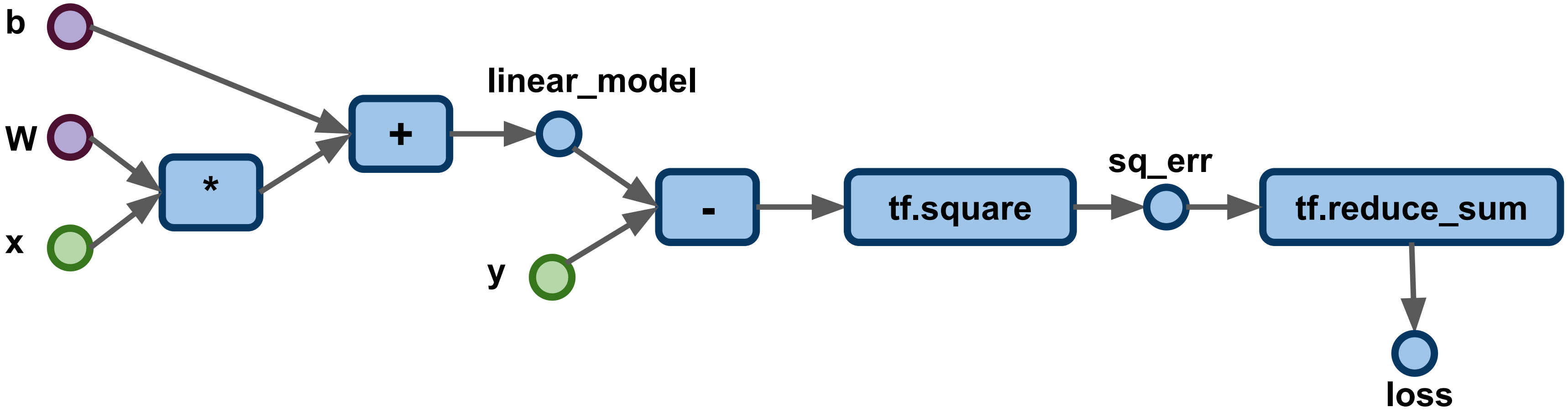
Defining a `__call__` method makes it so we can treat an instance of this class like a function. Objects like this are called **callable**.



Building a Simple Model: Linear Regression

```
class LinearModel(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)
        self.W = tf.Variable([0.5], dtype=tf.float32, name="slope")
        self.b = tf.Variable([-1.0], dtype=tf.float32, name="intercept")
    def __call__(self, x):
        return self.W * x + self.b

def loss(y_observed, y_predicted):
    return tf.reduce_sum(tf.square(y_observed - y_predicted))
```

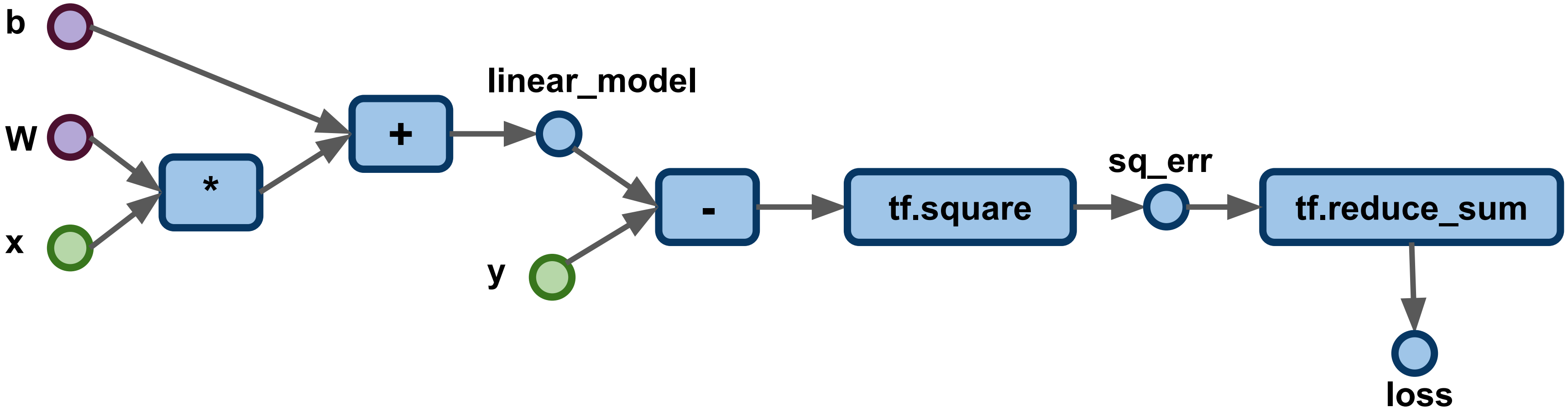
Note: `tf.reduce_sum` does just what you think it does!



Building a Simple Model: Linear Regression

Note: As you can see, the computational graph can get very complicated very quickly. TensorFlow has a set of built-in tools, collectively called **TensorBoard**, for handling some of this complexity:
https://www.tensorflow.org/tensorboard/graphs

```
    return tf.reduce_sum(tf.square(y_observed - y_predicted))
```



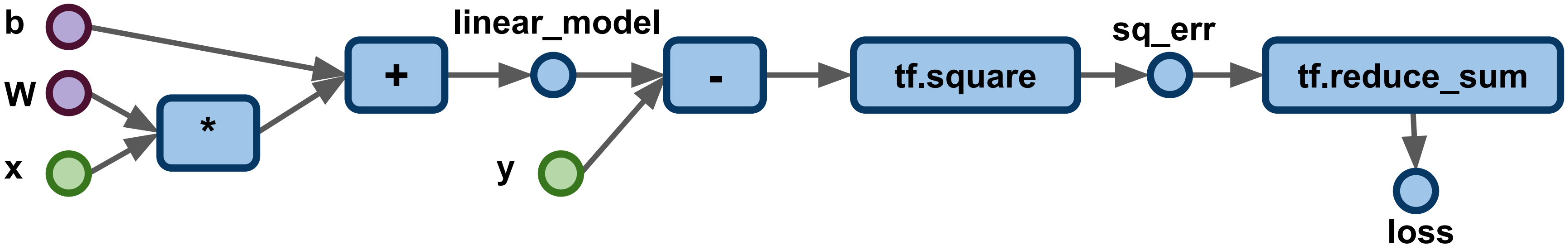
Building a Simple Model: Linear Regression

```
class LinearModel(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)
        self.W = tf.Variable([0.5], dtype=tf.float32, name="slope")
        self.b = tf.Variable([-1.0], dtype=tf.float32, name="intercept")
    def __call__(self, x):
        return self.W * x + self.b

def loss(y_observed, y_predicted):
    return tf.reduce_sum(tf.square(y_observed - y_predicted))
```

```
x = tf.constant( [1,2,3,4], dtype=tf.float32 )
y = tf.constant( [0,-1,-2,-3], dtype=tf.float32 )
loss( linear_model(x), y ).numpy()
```

```
23.5
```

Give (x,y) values to the model, evaluate its ability to replicate the observed y values.



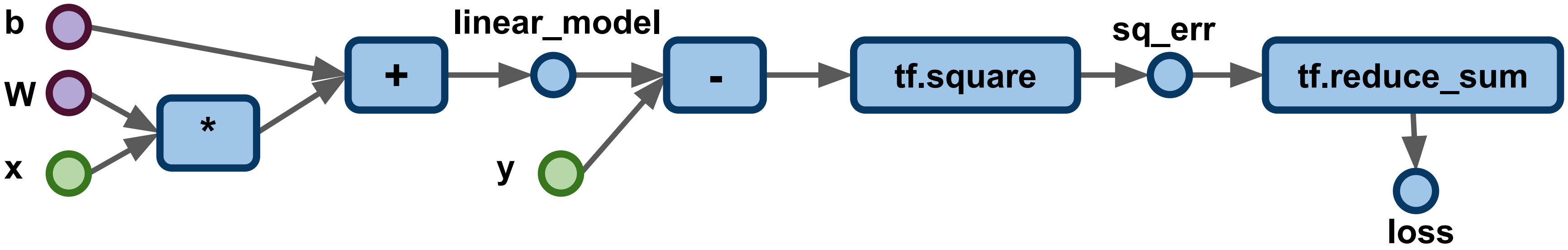
Building a Simple Model: Linear Regression

```
class LinearModel(tf.Module):
    def __init__(self, name=None):
        super().__init__(name=name)
        self.W = tf.Variable([0.5], dtype=tf.float32, name="slope")
        self.b = tf.Variable([-1.0], dtype=tf.float32, name="intercept")
    def __call__(self, x):
        return self.W * x + self.b

def loss(y_observed, y_predicted):
    return tf.reduce_sum(tf.square(y_observed - y_predicted))
```

```
x = tf.constant( [1,2,3,4], dtype=tf.float32 )
y = tf.constant( [0,-1,-2,-3], dtype=tf.float32 )
loss( linear_model(x), y ).numpy()
```

```
23.5
```

The `numpy()` method retrieves the actual numpy object from inside the `tf.Tensor`.



Building a Simple Model: Linear Regression

```
x = tf.constant( [1,2,3,4], dtype=tf.float32 )
y = tf.constant( [0,-1,-2,-3], dtype=tf.float32 )
loss( linear_model(x), y ).numpy()
```

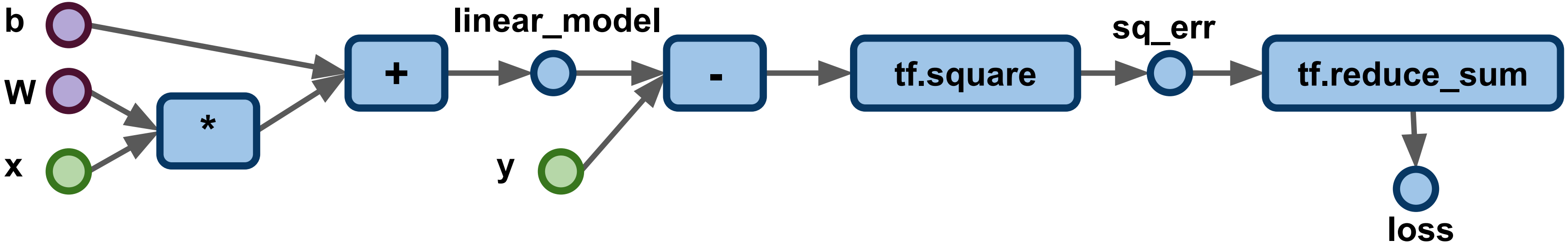
```
23.5
```

How can we improve (i.e., decrease) this loss?

Option 1: set `w` and `b` manually.

We know `W=-1, b=1` is the correct answer

Change values of `tf.Variables` using `assign` method



Building a Simple Model: Linear Regression

```
x = tf.constant( [1,2,3,4], dtype=tf.float32 )
y = tf.constant( [0,-1,-2,-3], dtype=tf.float32 )
loss( linear_model(x), y ).numpy()
```

```
23.5
```

How can we improve (i.e., decrease) this loss?

Option 1: set `w` and `b` manually.

We know `W=-1, b=1` is the correct answer

Change values of `tf.Variables` using `assign` method

```
linear_model.W.assign([-1])
linear_model.b.assign([1])
loss( linear_model(x), y ).numpy()
```

```
0.0
```

Update the slope and intercept in the model to the correct values.

Building a Simple Model: Linear Regression

```
x = tf.constant( [1,2,3,4], dtype=tf.float32 )
y = tf.constant( [0,-1,-2,-3], dtype=tf.float32 )
loss( linear_model(x), y ).numpy()
```

```
23.5
```

How can we improve (i.e., decrease) this loss?

Option 1: set `w` and `b` manually.

We know `W=-1, b=1` is the correct answer

Change values of `tf.Variables` using `assign` method

```
linear_model.W.assign([-1])
linear_model.b.assign([1])
loss( linear_model(x), y ).numpy()
```

```
0.0
```

Update the slope and intercept in the model to the correct values.

Note: because `W` and `b` are rank-1 tensors, we have to pass their new values as length-1 lists, not scalars. `linear_model.W.assign(-1)` would result in an error.

Building a Simple Model: Linear Regression

```
x = tf.constant( [1,2,3,4], dtype=tf.float32 )
y = tf.constant( [0,-1,-2,-3], dtype=tf.float32 )
loss( linear_model(x), y ).numpy()
```

```
23.5
```

How can we improve (i.e., decrease) this loss?

Option 1: set `w` and `b` manually.

We know `W=-1, b=1` is the correct answer

Change values of `tf.Variables` using `assign` method

Option 2: use closed-form solution for loss-minimizing `W` and `b`.

...but then what happens when we have a model with no closed-form solution?

Building a Simple Model: Linear Regression

```
x = tf.constant( [1,2,3,4], dtype=tf.float32 )
y = tf.constant( [0,-1,-2,-3], dtype=tf.float32 )
loss( linear_model(x), y ).numpy()
```

```
23.5
```

How can we improve (i.e., decrease) this loss?

Option 1: set `w` and `b` manually.
We know `W=-1, b=1` is the correct answer
Change values of `tf.Variables` using `assign` method

Option 2: use closed-form solution for loss-minimizing `W` and `b`.
...but then what happens when we have a model with no closed-form solution?

Option 3: take advantage of **automatic differentiation**
Allows easy implementation of **gradient descent** and related techniques

Building a Simple Model: Linear Regression

```
x = tf.constant( [1,2,3,4], dtype=tf.float32 )
y = tf.constant( [0,-1,-2,-3], dtype=tf.float32 )
loss( linear_model(x), y ).numpy()
```

```
23.5
```

How can we improve (i.e., decrease) this loss?

Option 1: set `w` and `b` manually.

We know `W=-1, b=1` is the correct answer

Change values of `tf.Variables` using `assign` method

Option 2: use close **This is why we use TensorFlow!** `W` and `b`.

...but then what with no closed-form solution?

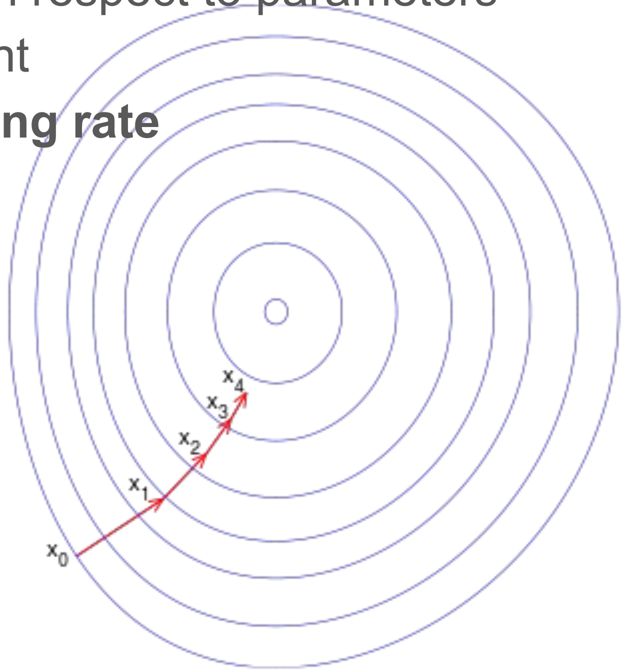
Option 3: take advantage of **automatic differentiation**

Allows easy implementation of **gradient descent** and related techniques

Gradient Descent: Crash Course

Iterative optimization method for minimizing a function

- At location (w, b), take gradient of loss function with respect to parameters
- Take a **gradient step** in the direction of the gradient
- Size of step changes over time according to **learning rate**



Gradient Descent: Crash Course

Iterative optimization method for minimizing a function

- At location (w, b), take gradient of loss function with respect to parameters
- Take a **gradient step** in the direction of the gradient
- Size of step changes over time according to **learning rate**

In short, gradient descent is a method for minimizing a function, provided we can compute its gradient (i.e., derivative). It's enough for this course to treat this as a black box.

For more information:

S. P. Boyd and L. Vandenberghe (2004). *Convex Optimization*. Cambridge University Press.

J. Nocedal and S. J. Wright (2006). *Numerical Optimization*. Springer.

Training a Simple Model: Linear Regression

```
def train(model, x, y, learning_rate):
    with tf.GradientTape() as t:
        current_loss = loss(y, model(x))

    (dW, db) = t.gradient(current_loss, [model.W, model.b])

    model.W.assign_sub( learning_rate*dW )
    model.b.assign_sub( learning_rate*db )
```

Define a `train` function, which takes a single gradient step with respect to a `model`'s performance with respect to a loss function on data `x` and `y`, with a given `learning rate`.

Training a Simple Model: Linear Regression

```
def train(model, x, y, learning_rate):
    with tf.GradientTape() as t:
        current_loss = loss(y, model(x))

    (dW, db) = t.gradient(current_loss, [model.W, model.b])

    model.W.assign_sub( learning_rate*dW )
    model.b.assign_sub( learning_rate*db )
```

The `tf.GradientTape` object keeps track of our gradients. This is especially useful when we want to check whether or not our estimates have converged. Here, we just need it to do automatic differentiation for us.

Training a Simple Model: Linear Regression

```
def train(model, x, y, learning_rate):
    with tf.GradientTape() as t:
        current_loss = loss(y, model(x))

    (dW, db) = t.gradient(current_loss, [model.W, model.b])

    model.W.assign_sub( learning_rate*dW )
    model.b.assign_sub( learning_rate*db )
```

Use the `tf.GradientTape` to compute the gradient of the loss with respect to the current model parameters.

Caution: notice that `loss` is a Python function, and `current_loss` is a `tf.Tensor` that is output by that function. Breaking this pattern is a common source of bugs.

Training a Simple Model: Linear Regression

```
def train(model, x, y, learning_rate):
    with tf.GradientTape() as t:
        current_loss = loss(y, model(x))

    (dW, db) = t.gradient(current_loss, [model.W, model.b])

    model.W.assign_sub( learning_rate*dW )
    model.b.assign_sub( learning_rate*db )
```

Update the parameters. `assign_sub` is the `tf.Variable` analogue of writing `x = x-dx`.

Training a Simple Model: Linear Regression

```
def train(model, x, y, learning_rate):
    with tf.GradientTape() as t:
        current_loss = loss(y, model(x))

    (dW, db) = t.gradient(current_loss, [model.W, model.b])

    model.W.assign_sub( learning_rate*dW )
    model.b.assign_sub( learning_rate*db )
```

```
x = tf.constant( [1,2,3,4], dtype=tf.float32 )
y = tf.constant( [0,-1,-2,-3], dtype=tf.float32 )
```

```
linear_model.W.assign(tf.random.normal(shape=[1]))
linear_model.b.assign(tf.random.normal(shape=[1]))
(linear_model.W.numpy(), linear_model.b.numpy())
```

```
(array([-0.49148715], dtype=float32), array([0.48135814], dtype=float32))
```

Initialize the model parameters randomly. `tf.random.normal` is similar to `numpy/scipy` RNGs. Note that we need our random variables to be `shape=[1]` to match the shapes of `W` and `b`.

Training a Simple Model: Linear Regression

```
def train(model, x, y, learning_rate):
    with tf.GradientTape() as t:
        current_loss = loss(y, model(x))

    (dW, db) = t.gradient(current_loss, [model.W, model.b])

    model.W.assign_sub( learning_rate*dW )
    model.b.assign_sub( learning_rate*db )
```

```
x = tf.constant( [1,2,3,4], dtype=tf.float32 )
y = tf.constant( [0,-1,-2,-3], dtype=tf.float32 )

linear_model.W.assign(tf.random.normal(shape=[1]))
linear_model.b.assign(tf.random.normal(shape=[1]))
(linear_model.W.numpy(), linear_model.b.numpy())
```

```
(array([-0.49148715], dtype=float32), array([0.48135814], dtype=f
```

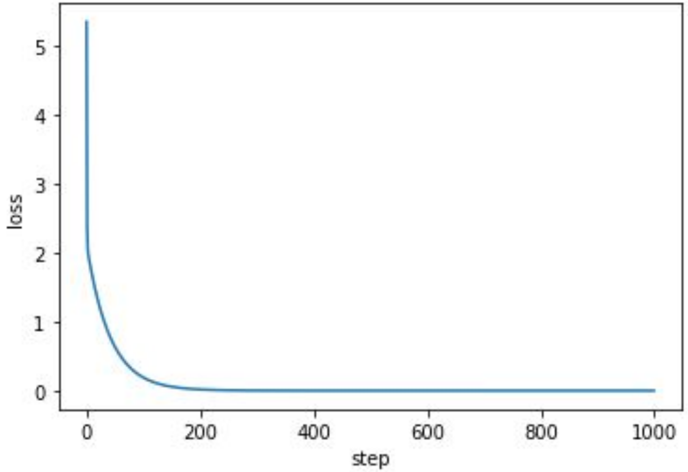
```
for i in range(1000):
    train(linear_model, x, y, learning_rate=0.01)
(linear_model.W.numpy(), linear_model.b.numpy())
```

```
(array([-0.9999991], dtype=float32), array([0.99999744], dtype=float32))
```

Each iteration of this loop computes one gradient step and updates the variables accordingly.

Training a Simple Model: Linear Regression

```
x = tf.constant( [1,2,3,4], dtype=tf.float32 )
y = tf.constant( [0,-1,-2,-3], dtype=tf.float32 )

linear_model.W.assign([0.5])
linear_model.b.assign([-0.5])
(linear_model.W.numpy(), linear_model.b.numpy())

losses = list(range(1000))
for i in range(1000):
    train(linear_model, x, y, learning_rate=0.01)
    losses[i] = loss(linear_model(x), y)
plt.xlabel('step'); plt.ylabel('loss'); _ = plt.plot(losses);
```



Note: TensorBoard includes a set of tools for visualization, including for tracking loss, but the approach here is quicker and easier for our purposes.

TensorFlow Estimators API: `tf.estimators`

`tf.estimators` is a TF module that simplifies model training and evaluation

Module allows one to run models on CPU or GPU, local or on GCP, etc

Simplifies much of the work of building the graph and estimating parameters

More information:

https://www.tensorflow.org/guide/estimator

Note: Keras in TensorFlow v2 serves similar purpose for specifying neural nets

https://www.tensorflow.org/guide/keras