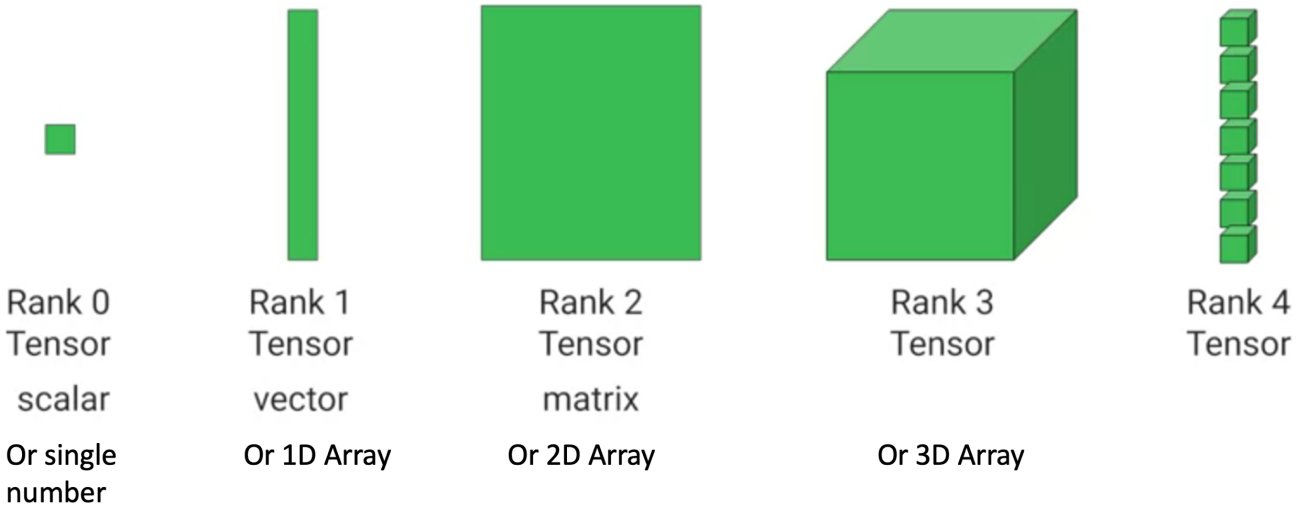
Tensorflow and Keras

Table of Contents

1 Introduction

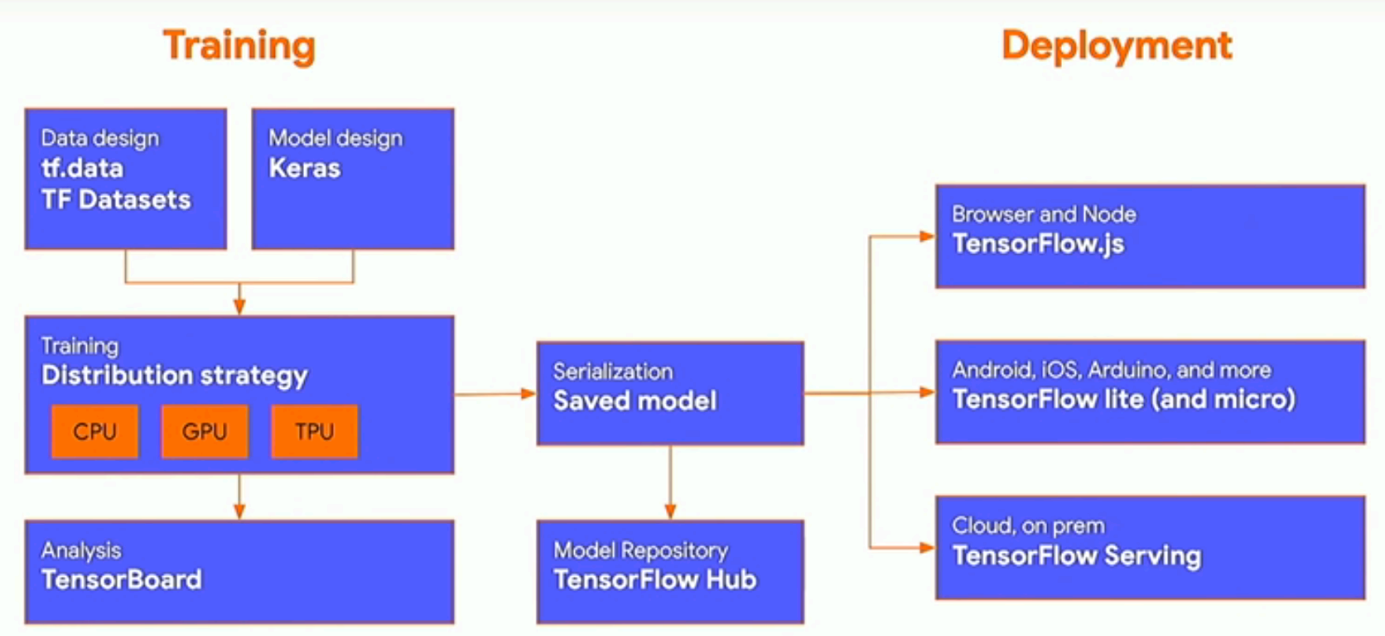
- TensorFlow is an open-source, high-performance library for any numerical computation, not just for machine learning.
- TensorFlow works by creating a directed graph or a DAG to represent the computation that you want to do.
- A tensor is an N-dimensional array. The building blocks are represented below:



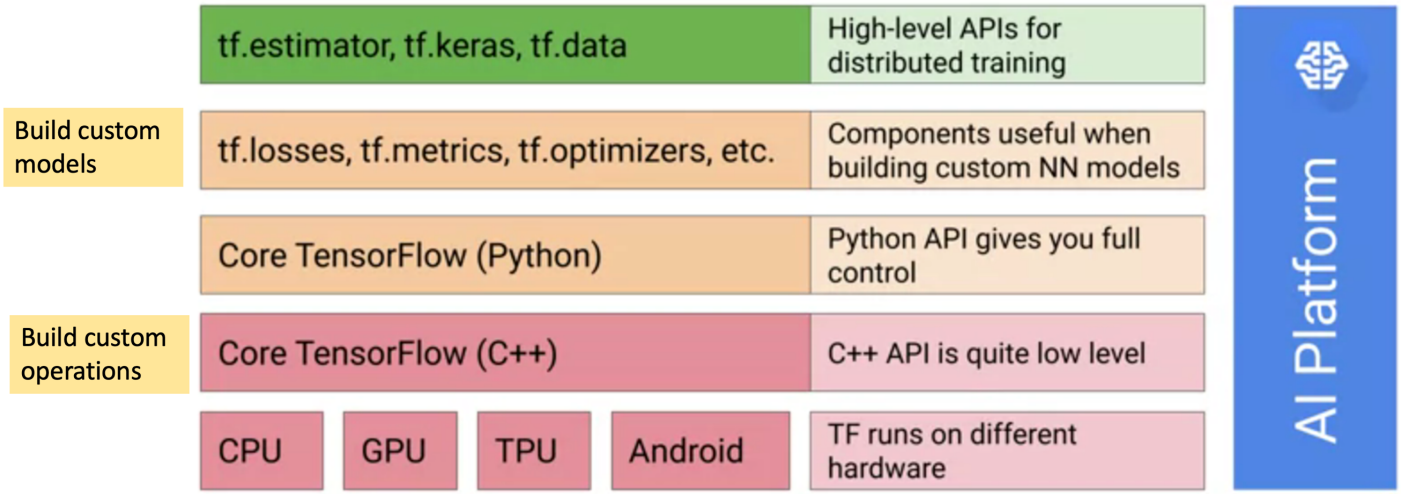
- The data are called tensors and since these flow through a graph, this is called Tensor Flow.
- Why does tensorflow use DAGs?
  - It is for portability. (both language and hardware)
  - The graph is a language independent representation of the code in your model.
  - Model can be written in python and saved. It can then be restored and run in a C environment.
  - Models can be executed on CPU or GPU.
- In TensorFlow, tensors are multi-dimensional arrays with a uniform type (called a dtype). All tensors are immutable like Python numbers and strings: you can never update the contents of a tensor, only create a new one.

- A tf.Variable represents a tensor whose value can be changed by running operations (ops) on it

- The tensorflow framework is as depicted below



- Tensorflow contains multiple abstraction layers:



- Tensors and variables

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)
	Matrix	2	`x = tf.constant([[3, 5, 7],` `[4, 6, 8]])`	(2, 3)

	3D Tensor	3	`tf.constant([[[3, 5, 7],[4, 6, 8]],`
			`[[1, 2, 3],[4, 5, 6]]`
			`])`
				(2, 2, 3)

	nD Tensor	n	`x1 = tf.constant([2, 3, 4])`
			`x2 = tf.stack([x1, x1])`
			`x3 = tf.stack([x2, x2, x2, x2])`
			`x4 = tf.stack([x3, x3])`
			`...`
				(3,)
				(2, 3)
				(4, 2, 3)
				(2, 4, 2, 3)

They behave like numpy n-dimensional arrays except that
- tf.constant produces constant tensors
- tf.Variable produces tensors that can be modified

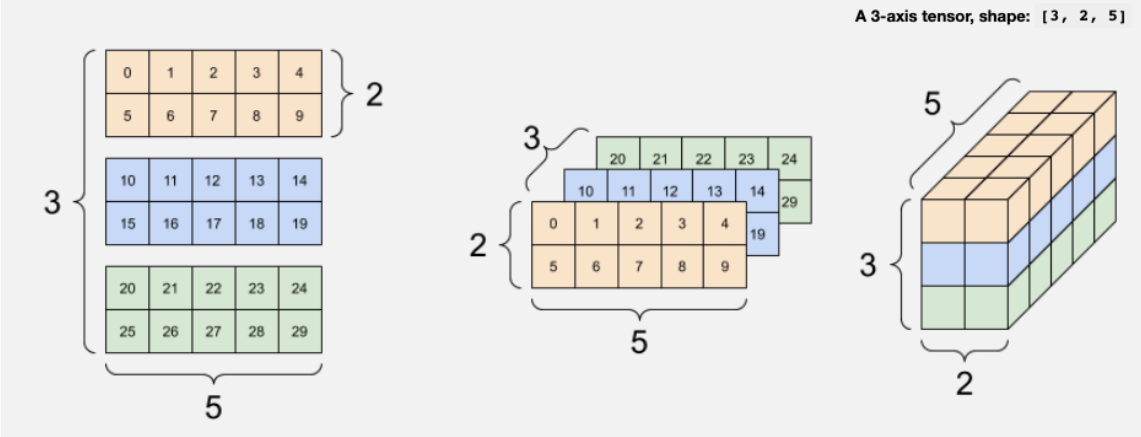
  - A 2D tensor can be formed by stacking 2 or more 1D tensors.
  - A 3D tensor can be formed by stacking 2 or more 2D tensors.
  - So on..

- A tensor can have an arbitrary number of axes (or dimensions)
    - There are many ways to visualize a tensor with more than 2 axes.
    - Example below:

```
rank_3_tensor = tf.constant([
  [[0, 1, 2, 3, 4],
   [5, 6, 7, 8, 9]],
  [[10, 11, 12, 13, 14],
   [15, 16, 17, 18, 19]],
  [[20, 21, 22, 23, 24],
   [25, 26, 27, 28, 29]],])

print(rank_3_tensor)
```

```
tf.Tensor(
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]], shape=(3, 2, 5), dtype=int32)
```

There are many ways you might visualize a tensor with more than 2-axes.



- A Tensor can be converted to a numpy Array in 2 ways:
    - Use np.array(tensor)

```
np.array(rank_2_tensor)
```

```
array([[1., 2.],
       [3., 4.],
       [5., 6.]], dtype=float16)
```

- Use tensor.numpy()

```
rank_2_tensor.numpy()
```

```
array([[1., 2.],
       [3., 4.],
       [5., 6.]], dtype=float16)
```

1.1 Basic Operations

```
# You can simply create a constant tensor using `tf.constant`
a = tf.constant([[1, 2],
                 [3, 4]])
b = tf.constant([[1, 1],
                 [1, 1]]) # Could have also said `tf.ones([2,2])`
```

- **Basic math**

```
# Let's do some basic math on tensors.
print(tf.add(a, b), "\n")
```

```
tf.Tensor(
[[2 3]
 [4 5]], shape=(2, 2), dtype=int32)
```

```
print(tf.multiply(a, b), "\n")
```

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

```
print(tf.matmul(a, b), "\n")
```

```
tf.Tensor(
[[3 3]
 [7 7]], shape=(2, 2), dtype=int32)
```

- **Element-wise Math**

```
print(a + b, "\n") # element-wise addition
```

```
tf.Tensor(
[[2 3]
 [4 5]], shape=(2, 2), dtype=int32)
```

```
print(a * b, "\n") # element-wise multiplication
```

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

```
print(a @ b, "\n") # matrix multiplication
```

```
tf.Tensor(
[[3 3]
 [7 7]], shape=(2, 2), dtype=int32)
```

- **Other operations**

```
c = tf.constant([[4.0, 5.0], [10.0, 1.0]])
```

```
# Find the largest value
print(tf.reduce_max(c))
```

```
tf.Tensor(10.0, shape=(), dtype=float32)
```

```
# Find the index of the largest value
print(tf.argmax(c))
```

```
tf.Tensor([1 0], shape=(2,), dtype=int64)
```

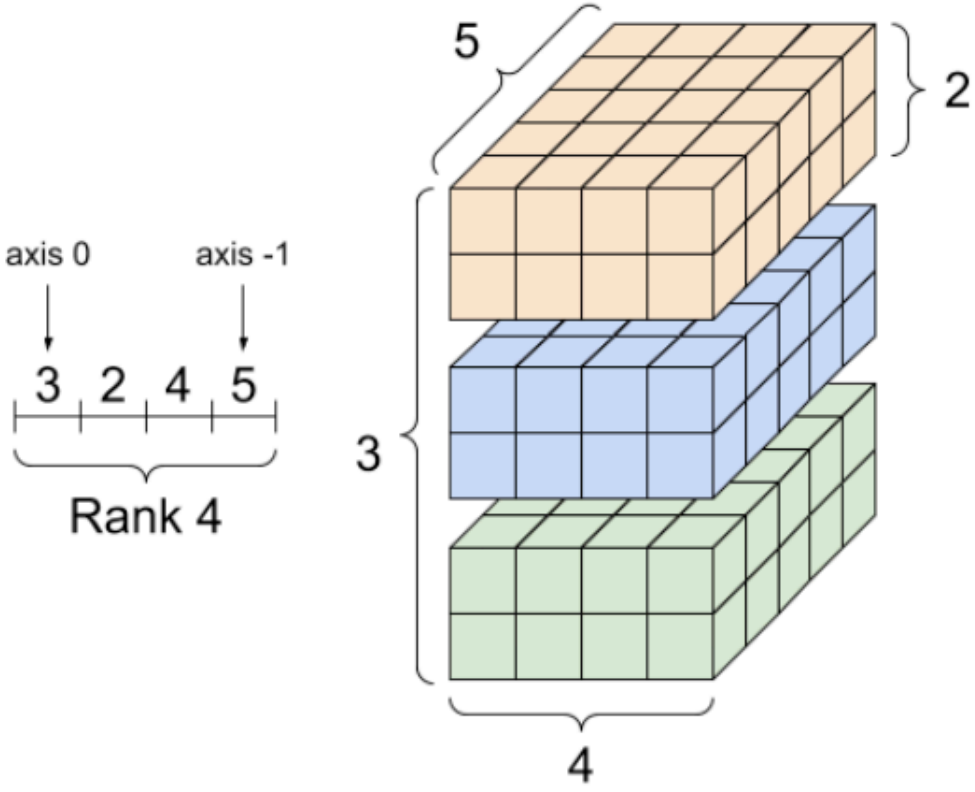
```
# Compute the softmax
print(tf.nn.softmax(c))
```

```
tf.Tensor(
[[2.6894143e-01 7.3105860e-01]
 [9.9987662e-01 1.2339458e-04]], shape=(2, 2), dtype=float32)
```

1.2 Tensor shapes:

- **Shape**
  - The length (number of elements) of each of the dimensions of a tensor.
- **Rank**
  - Number of tensor dimensions. A scalar has rank 0, a vector has rank 1, a matrix is rank 2.
- **Axis**
  - A particular dimension of a tensor
- **Size**
  - The total number of items in the tensor, the product shape vector

A rank-4 tensor, shape: `[3, 2, 4, 5]`



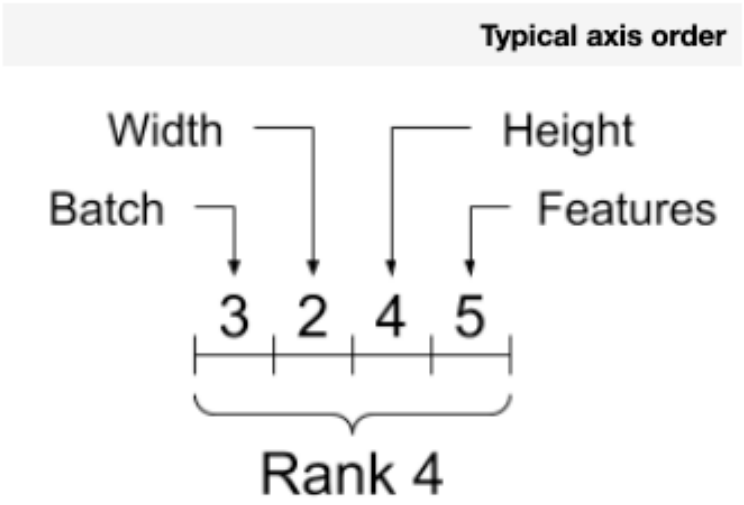
```
# `tf.zeros` creates a tensor with all elements set to zero.
rank_4_tensor = tf.zeros([3, 2, 4, 5])
```

```
print("Type of every element:", rank_4_tensor.dtype)
print("Number of dimensions:", rank_4_tensor.ndim)
print("Shape of tensor:", rank_4_tensor.shape)
print("Elements along axis 0 of tensor:", rank_4_tensor.shape[0])
print("Elements along the last axis of tensor:", rank_4_tensor.shape[-1])
print("Total number of elements (3*2*4*5): ", tf.size(rank_4_tensor).numpy())
```

```
Type of every element: <dtype: 'float32'>
Number of dimensions: 4
Shape of tensor: (3, 2, 4, 5)
Elements along axis 0 of tensor: 3
Elements along the last axis of tensor: 5
Total number of elements (3*2*4*5):  120
```

1.3 Tensorflow Axis and Indexing

- Axes are often referred to by their indices.
- Each axis also has a meaning.
- Often axes are ordered from global to local:
    - The batch axis first, followed by spatial dimensions, and features for each location last.
    - This way feature vectors are contiguous regions of memory



- **Single axis indexing**
  - Tensorflow follows standard python indexing and the basic rules of numpy indexing
  - Indexes start at 0
  - Negative indices count backward from the end

```
  0   1   2   3   4   5   6   7   8
  ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
  ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
 -9  -8  -7  -6  -5  -4  -3  -2  -1
```

  - Colon (:) is used for slicing.

```
arr[start:stop:step]
```

  - Indexing with a scalar removes the dimension

```
# When tensors are Indexed with a scalar, it removes the dimension
print("First:", rank_1_tensor[0].numpy())
print("Second:", rank_1_tensor[1].numpy())
print("Last:", rank_1_tensor[-1].numpy())
```

```
First: 0
Second: 1
Last: 34
```

- Indexing with a slice (:) keeps the dimension

```
# When tensors are Indexed with a `:` slice keeps the dimension
print("Everything:", rank_1_tensor[:].numpy())
print("Before 4:", rank_1_tensor[:4].numpy())
print("From 4 to the end:", rank_1_tensor[4:].numpy())
print("From 2, before 7:", rank_1_tensor[2:7].numpy())
print("Every other item:", rank_1_tensor[::2].numpy())
print("Reversed:", rank_1_tensor[::-1].numpy())
print("Single element:", rank_1_tensor[3:4:1].numpy())
```

```
Everything: [ 0  1  1  2  3  5  8 13 21 34]
Before 4: [0 1 1 2]
From 4 to the end: [ 3  5  8 13 21 34]
From 2, before 7: [1 2 3 5 8]
Every other item: [ 0  1  3  8 21]
Reversed: [34 21 13  8  5  3  2  1  1  0]
Single element: [2]
```

- **Multi-Axis Indexing**
    - Passing an integer for every index will result in a scalar

```
# Pull out a single value from a 2-rank tensor
print(rank_2_tensor.numpy(),'\n')
print(rank_2_tensor[1, 1].numpy())
```

```
[[1. 2.]
 [3. 4.]
 [5. 6.]]

4.0
```

    - Indexing rank 2 tensor

```
# Indexed tensors using any combination integers and slices `:`
# Get row and column tensors
print("Second row                 :", rank_2_tensor[1, :].numpy())
print("Second column              :", rank_2_tensor[:, 1].numpy())
print("Last row                   :", rank_2_tensor[-1, :].numpy())
print("First item in last column :", rank_2_tensor[0, -1].numpy())
print("Skip the first row         :")
print(rank_2_tensor[1:, :].numpy(), "\n")
```

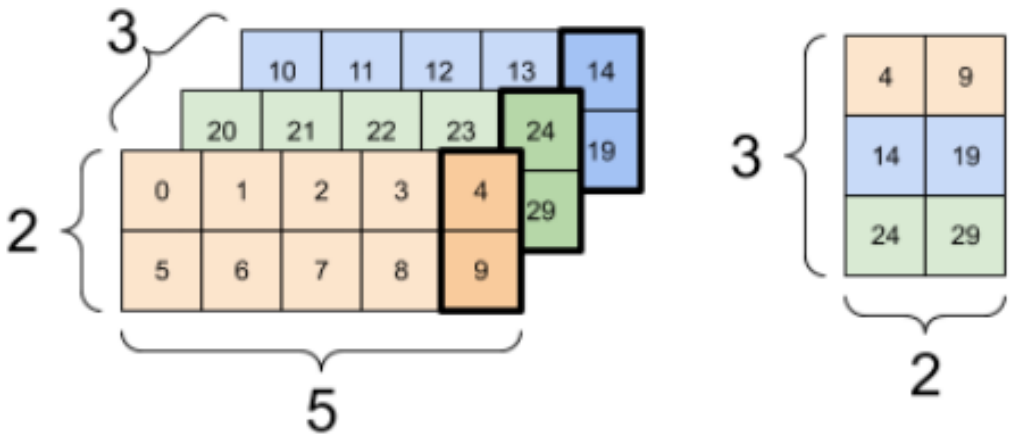
```
Second row                 : [3. 4.]
Second column              : [2. 4. 6.]
Last row                   : [5. 6.]
First item in last column : 2.0
Skip the first row         :
[[3. 4.]
 [5. 6.]]
```

- Example for Rank 3 tensor

```
print(rank_3_tensor)
```

```
tf.Tensor(
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]], shape=(3, 2, 5), dtype=int32)
```

Selecting the last feature across all locations in each example in the batch




```
# Get 3-axis tensor
print(rank_3_tensor[:, :, 4])
```

```
tf.Tensor(
[[ 4  9]
 [14 19]
 [24 29]], shape=(3, 2), dtype=int32)
```

- **To combine, use tf.stack()**
    - Example: Going from 1D to 2D

```
x1 = tf.constant([3,5,7])
x2 = tf.constant([4,6,8])

x = tf.stack([x1,x2])
x
```

```
<tf.Tensor: id=6, shape=(2, 3), dtype=int32, numpy=
array([[3, 5, 7],
       [4, 6, 8]], dtype=int32)>
```

```
tf.shape(x)
```

```
<tf.Tensor: id=8, shape=(2,), dtype=int32, numpy=array([2, 3], dtype=int32)>
```

1.4 Reshaping Tensors

- To view the shape of a tensor, we use <tensor>.shape

```
# Shape returns a `TensorShape` object that shows the size on each dimension
var_x = tf.Variable(tf.constant([[1], [2], [3]]))
print(var_x.shape)
```

```
(3, 1)
```

- The shape can be retrieves as a list

```
# You can convert this object into a Python list, too
print(var_x.shape.as_list())
```

```
[3, 1]
```

- TensorFlow uses C-style "row-major" memory ordering, where incrementing the right-most index corresponds to a single step in memory.

- If you **flatten** the tensor, you see the order it is laid out in memory.
    - Note: A -1 passed in the shape argument implies whatever fits.

```
print(rank_3_tensor)
```

```
tf.Tensor(
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]], shape=(3, 2, 5), dtype=int32)
```

```
# A `-1` passed in the `shape` argument says "Whatever fits".
print(tf.reshape(rank_3_tensor, [-1]))
```

```
tf.Tensor(
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29], shape=(30,), dtype=int32)
```

- **tf.reshape()** reads elements row by row and puts it into the output tensor.
    - For example,

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

    - Let

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

        - Here, we are reshaping x to a 3D tensor with 2 columns.
        - It starts reading x in the sequence: 3,5,7,4,6,8 and fills output in desired shape.
        - So, the output we get is:

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

    - Reshaping will "work" for any new shape with the same total number of elements, but it will not do anything useful if you do not respect the order of the axes.
    - To swap the axes, reshape will not work. We need to use tf.transpose() for that.

1.5 DTypes

- To inspect a tensors data type, use <tensor>.dtype
- When creating a tensor, the data type can be optionally specified.
    - If not specified, tf chooses a dtype that can represent the data.
    - tf converts python
        - integers to tf.int32
        - Floating point numbers to tf.float32
- Use **tf.cast**() to cast from type to type.

```
the_f64_tensor = tf.constant([2.2, 3.3, 4.4], dtype=tf.float64)
the_f16_tensor = tf.cast(the_f64_tensor, dtype=tf.float16)
```

```
# Now, let's cast to an uint8 and lose the decimal precision
the_u8_tensor = tf.cast(the_f16_tensor, dtype=tf.uint8)
print(the_u8_tensor)
```

```
tf.Tensor([2 3 4], shape=(3,), dtype=uint8)
```

1.6 Broadcasting

- Concept borrowed from numpy
  - Details: https://numpy.org/doc/stable/user/basics.broadcasting.html

- In short, smaller tensors are stretched automatically to fit larger tensors when running combined operations on both of them.

- Examples

```
x = tf.constant([1, 2, 3])
y = tf.constant(2)
z = tf.constant([2, 2, 2])

# All of these are the same computation
print(tf.multiply(x, 2))
print(x * y)
print(x * z)
```

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

```
# These are the same computations
x = tf.reshape(x,[3,1])
y = tf.range(1, 5)
print(x, "\n")
print(y, "\n")
print(tf.multiply(x, y))
```

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

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

tf.Tensor(
[[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]], shape=(3, 4), dtype=int32)
```

- **NOTE**: “y” in the example has shape (1,4). But the leading 1 is optional. Thus, the shape of y is displayed as [4]

- Most of the time, broadcasting is both time and space efficient, as the broadcast operation never materializes the expanded tensors in memory.

- You see what broadcasting looks like using **tf.broadcast_to**

```
print(tf.broadcast_to(tf.constant([1, 2, 3]), [3, 3]))
```

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

1.7 tf.convert_to_tensor

- Most ops, like tf.matmul and tf.reshape take arguments of class tf.Tensor.

- Most, but not all, ops call convert_to_tensor on non-tensor arguments.

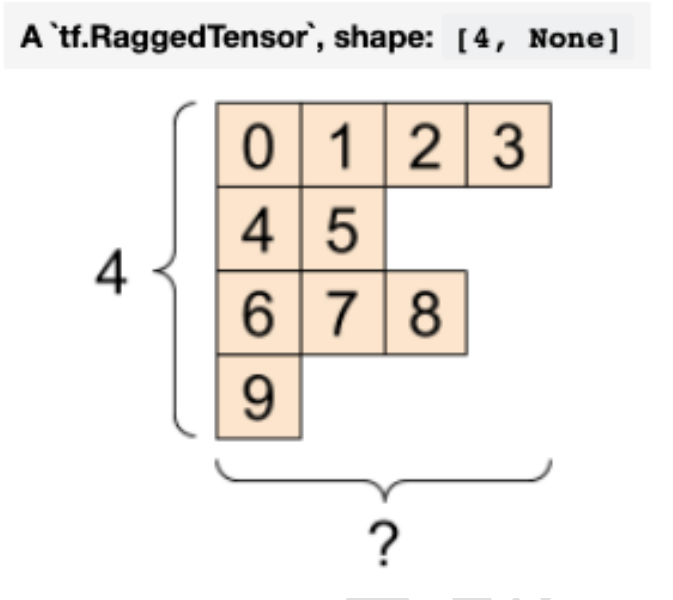
- There is a registry of conversions, and most object classes like NumPy's ndarray, TensorShape, Python lists, and tf.Variable will all convert automatically.

- See tf.register_tensor_conversion_function for more details, and if you have your own type you'd like to automatically convert to a tensor.

- The base tf.Tensor class requires tensors to be "rectangular". However, there are specialized types of Tensors that can handle different shapes:
    - ragged
    - sparse

1.8 Ragged Tensors

- A tensor with variable numbers of elements along some axis is called "ragged".
- Use tf.ragged.RaggedTensor for ragged data.
- For example, this cannot be represented as a regular tensor:




```
ragged_list = [
    [0, 1, 2, 3],
    [4, 5],
    [6, 7, 8],
    [9]]
```

```
try:
  tensor = tf.constant(ragged_list)
except Exception as e: print(e)
```

```
Can't convert non-rectangular Python sequence to Tensor.
```

```
# `tf.ragged.constant` constructs a constant RaggedTensor from a nested Python list.
ragged_tensor = tf.ragged.constant(ragged_list)
print(ragged_tensor)
```

```
<tf.RaggedTensor [[0, 1, 2, 3], [4, 5], [6, 7, 8], [9]]>
```

```
print(ragged_tensor.shape)
```

```
(4, None)
```

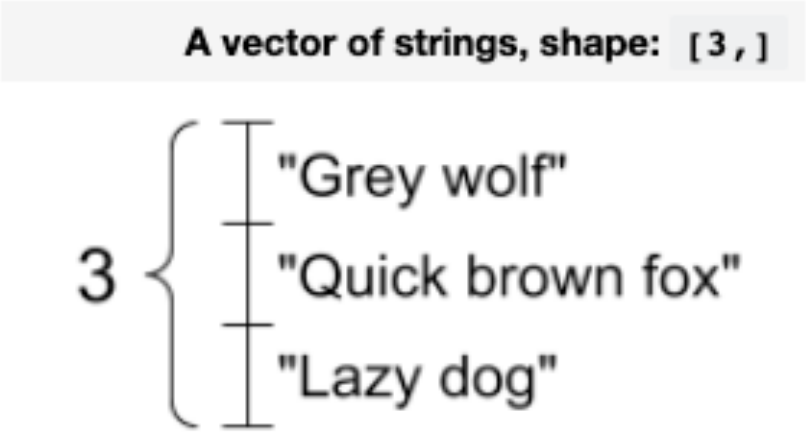
1.9 String Tensors

- tf.string is a dtype, which is to say we can represent data as strings (variable-length byte arrays) in tensors.
- The strings are atomic and cannot be indexed the way Python strings are.
- The length of the string is not one of the dimensions of the tensor.
- See tf.strings for functions to manipulate them.
- Below is a scalar

```
# Tensors can be strings, too here is a scalar string.
scalar_string_tensor = tf.constant("Gray wolf")
print(scalar_string_tensor)
```

```
tf.Tensor(b'Gray wolf', shape=(), dtype=string)
```

- Below is a vector




```
# If we have two string tensors of different lengths, this is OK.
tensor_of_strings = tf.constant(["Gray wolf",
                                 "Quick brown fox",
                                 "Lazy dog"])
# Note that the shape is (2,), indicating that it is 2 x unknown.
print(tensor_of_strings)
```

```
tf.Tensor([b'Gray wolf' b'Quick brown fox' b'Lazy dog'], shape=(3,), dtype=string)
```

- In the above printout the b prefix indicates that tf.string dtype is a byte-string (not a unicode string).

- If you pass unicode characters they are utf-8 encoded.

```
tf.constant("🥳👍")
```

```
<tf.Tensor: id=212, shape=(), dtype=string, numpy=b'\xf0\x9f\xa5\xb3\xf0\x9f\x91\x8d'>
```

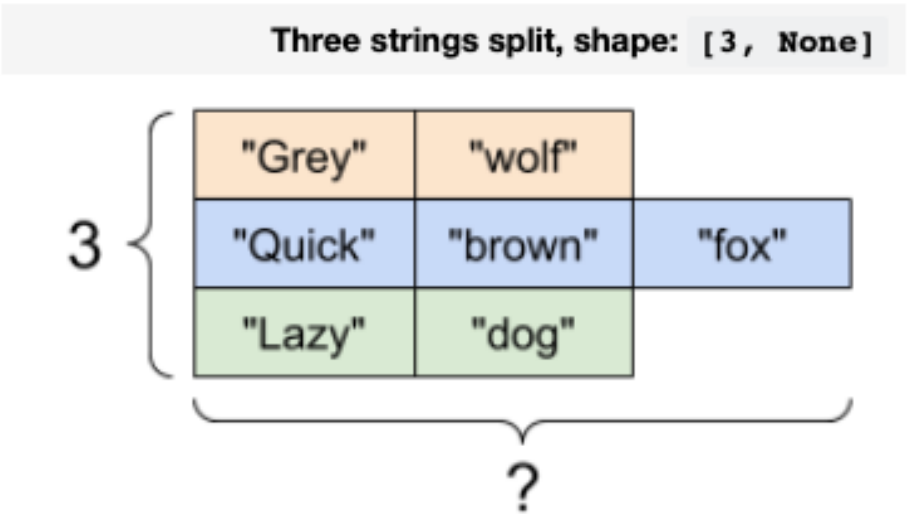
- **String Functions**
  - **Split**

```
# We can use split to split a string into a set of tensors
print(tf.strings.split(scalar_string_tensor, sep=" "))
```

```
tf.Tensor([b'Gray' b'wolf'], shape=(2,), dtype=string)
```

```
# ...but it turns into a `RaggedTensor` if we split up a tensor of strings,
# as each string might be split into a different number of parts.
print(tf.strings.split(tensor_of_strings))
```

```
<tf.RaggedTensor [[b'Gray', b'wolf'], [b'Quick', b'brown', b'fox'], [b'Lazy', b'dog']]>
```



- **To Number**

```
# `tf.strings.to_number` converts each string in the input Tensor to the specified numeric type.
text = tf.constant("1 10 100")
print(tf.strings.to_number(tf.strings.split(text, " ")))
```

```
tf.Tensor([  1.  10. 100.], shape=(3,), dtype=float32)
```

  - **Alternate approach 1**

```
# Split string elements of input into bytes using `tf.strings.bytes_split`.
byte_strings = tf.strings.bytes_split(tf.constant("Duck"))
# `tf.io.decode_raw` reinterpret the bytes of a string as a vector of numbers.
byte_ints = tf.io.decode_raw(tf.constant("Duck"), tf.uint8)
print("Byte strings:", byte_strings)
print("Bytes:", byte_ints)
```

```
Byte strings: tf.Tensor([b'D' b'u' b'c' b'k'], shape=(4,), dtype=string)
Bytes: tf.Tensor([ 68 117  99 107], shape=(4,), dtype=uint8)
```

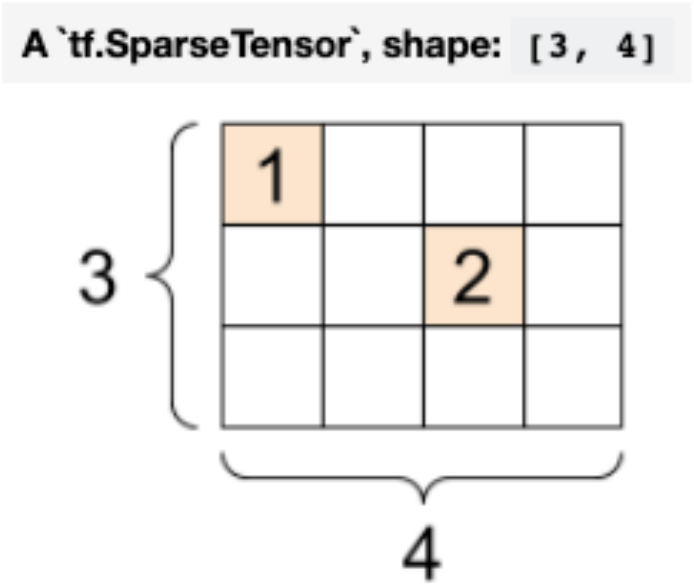
  - **Alternate approach 2**

```
# Or split it up as unicode and then decode it
unicode_bytes = tf.constant("アヒル 🦆")
# `tf.strings.unicode_split` splits each string in input into a sequence of Unicode code points.
unicode_char_bytes = tf.strings.unicode_split(unicode_bytes, "UTF-8")
# `tf.strings.unicode_decode` decodes each string in input into a sequence of Unicode code points.
unicode_values = tf.strings.unicode_decode(unicode_bytes, "UTF-8")

print("\nUnicode bytes:", unicode_bytes)
print("\nUnicode chars:", unicode_char_bytes)
print("\nUnicode values:", unicode_values)
```

```
Unicode bytes: tf.Tensor(b'\xe3\x82\xa2\xe3\x83\x92\xe3\x83\xab \xf0\x9f\xa6\x86', shape=(), dtype=string)

Unicode chars: tf.Tensor([b'\xe3\x82\xa2' b'\xe3\x83\x92' b'\xe3\x83\xab' b' ' b'\xf0\x9f\xa6\x86'], shape=(5,), dtyp
e=string)

Unicode values: tf.Tensor([ 12450  12498  12523     32 129414], shape=(5,), dtype=int32)
```

1.10 Sparse Tensors

- Sometimes, your data is sparse, like a very wide embedding space.
- TensorFlow supports tf.sparse.SparseTensor and related operations to store sparse data efficiently.




```
# Sparse tensors store values by index in a memory-efficient manner
sparse_tensor = tf.sparse.SparseTensor(indices=[[0, 0], [1, 2]],
                                       values=[1, 2],
                                       dense_shape=[3, 4])
print(sparse_tensor, "\n")
```

```
SparseTensor(indices=tf.Tensor(
[[0 0]
 [1 2]], shape=(2, 2), dtype=int64), values=tf.Tensor([1 2], shape=(2,), dtype=int32), dense_shape=tf.Tensor([3 4], s
hape=(2,), dtype=int64))
```

```
# We can convert sparse tensors to dense
print(tf.sparse.to_dense(sparse_tensor))
```

```
tf.Tensor(
[[1 0 0 0]
 [0 0 2 0]
 [0 0 0 0]], shape=(3, 4), dtype=int32)
```

1.11 Variable Tensors

- A TensorFlow variable is the recommended way to represent shared, persistent state your program manipulates.
- A variable looks and acts like a tensor, and, in fact, is a data structure backed by a tf.Tensor.
- Like tensors, they have a dtype and a shape, and can be exported to NumPy.
- A variable tensor is a tensor whose ***value*** can be changed.
- A variable requires an initial value that can be a tensor of any shape and type.
- Once initialized, the shape and type of the variable cannot be modified. Only the value can be changed.

```
import tensorflow as tf

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

- Value can be changed using one of the assign methods.

```
# x <- 48.5
x.assign(45.8)

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

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

- tf.Variable() is typically used to hold values that are modified during training like the model weights.

- Variables can be used as inputs to any tf operation.

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

- Most tensor operations work on variables as expected, although variables cannot be reshaped.

```
print("A variable:",my_variable)
```

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

```
# This creates a new tensor; it does not reshape the variable.
print("\nCopying and reshaping: ", tf.reshape(my_variable, ([1,4])))
```

```
Copying and reshaping:  tf.Tensor([[1. 2. 3. 4.]], shape=(1, 4), dtype=float32)
```

- Variables are backed by tensors. You can reassign the tensor using tf.Variable.assign. Calling assign does not (usually) allocate a new tensor; instead, the existing tensor's memory is reused.
- If you use a variable like a tensor in operations, you will usually operate on the backing tensor. Creating new variables from existing variables duplicates the backing tensors. Two variables will not share the same memory.

```
a = tf.Variable([2.0, 3.0])
# Create b based on the value of a
b = tf.Variable(a)
a.assign([5, 6])
```

```
<tf.Variable 'UnreadVariable' shape=(2,) dtype=float32, numpy=array([5., 6.], dtype=float32)>
```

```
# a and b are different
print(a.numpy())
print(b.numpy())
```

```
[5. 6.]
[2. 3.]
```

- In Python-based TensorFlow, tf.Variable instance have the same lifecycle as other Python objects.
    - When there are no references to a variable it is automatically deallocated
- Variables can also be named which can help you track and debug them.
    - You can give two variables the same name.

```
# Create a and b; they have the same value but are backed by different tensors.
a = tf.Variable(my_tensor, name="Mark")
# A new variable with the same name, but different value
# Note that the scalar add is broadcast
b = tf.Variable(my_tensor + 1, name="Mark")

# These are elementwise-unequal, despite having the same name
print(a == b)
```

```
tf.Tensor(
[[False False]
 [False False]], shape=(2, 2), dtype=bool)
```

    - Variable names are preserved when saving and loading models.
    - By default, variables in models will acquire unique variable names automatically, so you don't need to assign them yourself unless you want to.
- Although variables are important for differentiation, some variables will not need to be differentiated.
    - You can turn off gradients for a variable by setting trainable to false at creation.
    - An example of a variable that would not need gradients is a training step counter.

```
# You can turn off gradients for a variable by setting trainable to false at creation.
step_counter = tf.Variable(1, trainable=False)
```

1.12 Variable and Tensor Placement

- For better performance, TensorFlow will attempt to place tensors and variables on the fastest device compatible with its dtype.
- This means most variables are placed on a GPU if one is available.
- However, we can override this.
- By turning on device placement logging (at the start of the session), we can see where the variable is placed.

```
import tensorflow as tf

# Uncomment to see where your variables get placed (see below)
tf.debugging.set_log_device_placement(True)
```

- Below, we can place a float tensor and a variable on the CPU, even if a GPU is available.

```
# If you would like a particular operation to run on a
# device of your choice instead of what's automatically
# selected for you, you can use with `tf.device`
# to create a device context.
with tf.device('CPU:0'):

  # Create some tensors
  a = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
  b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
  c = tf.matmul(a, b)

print(c)
```

```
tf.Tensor(
[[22. 28.]
 [49. 64.]], shape=(2, 2), dtype=float32)
```

- **Note**: Because ***tf.config.set_soft_device_placement*** is turned on by default, even if you set context to use a GPU where one doesn't exist, the program will still run and execute the code on the available CPU.

1.13 GradientTape

- Tensorflow can compute the partial derivative of any ***function*** with respect to any ***variable***.
- In gradient descent, we know the weights are updated using the partial derivative of the loss w.r.t to the weights.
- To differentiate automatically, TensorFlow needs to remember what operations happened in what order during that forward pass. Then during the backward pass, TensorFlow traverses this list of operations in reverse order to compute those gradients.
- ***GradientTape*** is a context manager in which those partial differentiations are calculated. 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])

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

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

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

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

- **NOTE**:
    - For this to work, the functions have to be expressed within TensorFlow operations only, but since most basic operations like addition, multiplication, subtraction, are overloaded by TensorFlow Ops, this happens seamless.

2 Input Data Pipeline using Tensorflow

- ***tf.data*** API helps build efficient data pipelines.
- A data pipeline is a series of data processing steps.
- For example:
    - The pipeline for an ***image model*** might do the following
        - Aggregate data from files in a distributed file system.
        - Apply random perturbations to each image and
        - Merge randomly selected images into a batch for training.
    - The pipeline for a ***text model*** might involve:
        - Extracting symbols from raw text data,
        - Converting them to embedding identifiers with the lookup table and,
        - Batching together sequences of different lengths.
- The ***tf.data*** API makes it possible to handle large amounts of data, read from different data formats and perform complex transformations.

- **In-Memory datasets**
    - When using in memory datasets, we can use **from_tensors()** or **from_tensor_slices()**

```
t = tf.constant([[4, 2], [5, 3]])
ds = tf.data.Dataset.from_tensors(t)   # [[4, 2], [5, 3]]
```

```
t = tf.constant([[4, 2], [5, 3]])
ds = tf.data.Dataset.from_tensor_slices(t)   # [4, 2], [5, 3]
```

- from_tensors() combines the input and returns a single element as output for the dataset.
- from_tensor_slices() combines the input and returns a separate element for each row of the input tensor to create the dataset.

- **Reading dataset from a file(s)**
  - TextLineDataset()
    - Takes a file name (path) or list of file names as input.
  - <dataset>.map()
    - Can be used to parse each row of input data

  - **Example to read a csv file**

property type

sq_footage PRICE in K$

```
1001, house,  501
2001, house, 1001
3001, house, 1501
1001, apt,    701
2001, apt,   1301
3001, apt,   1901
1101, house,  526
2101, house, 1026
```

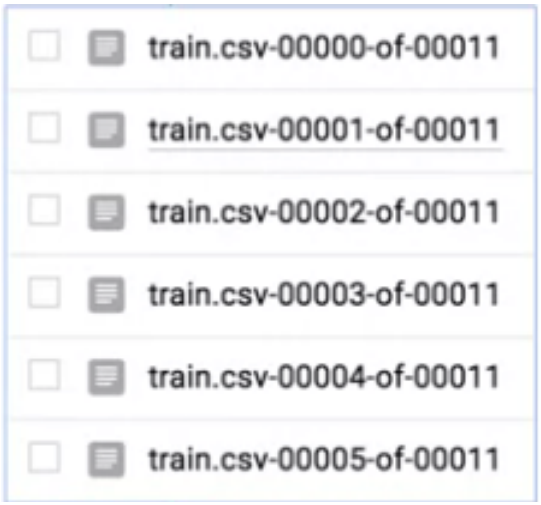
```
def parse_row(records):
    cols = tf.decode_csv(records, record_defaults=[[0], ['house'], [0]])
    features = {'sq_footage': cols[0], 'type': cols[1]}
    label = cols[2]
    return features, label


def create_dataset(csv_file_path):
    dataset = tf.data.TextLineDataset(csv_file_path)
    dataset = dataset.map(parse_row)
    dataset = dataset.shuffle(1000).repeat(15).batch(128)
    return dataset
```

dataset = "[line1, line2, etc.]"

dataset = "[parse_row(line1), parse_row(line2), etc.]"

  - ***Note***: It is recommended to shuffle() only training data.

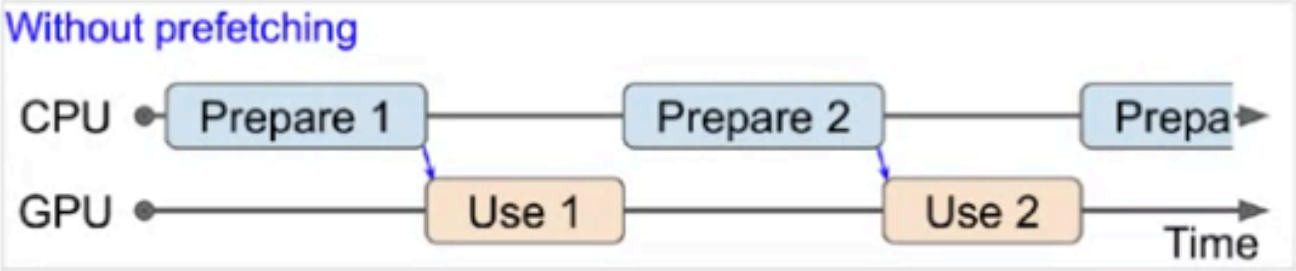
- **Example Reading a set of sharded CSV files**

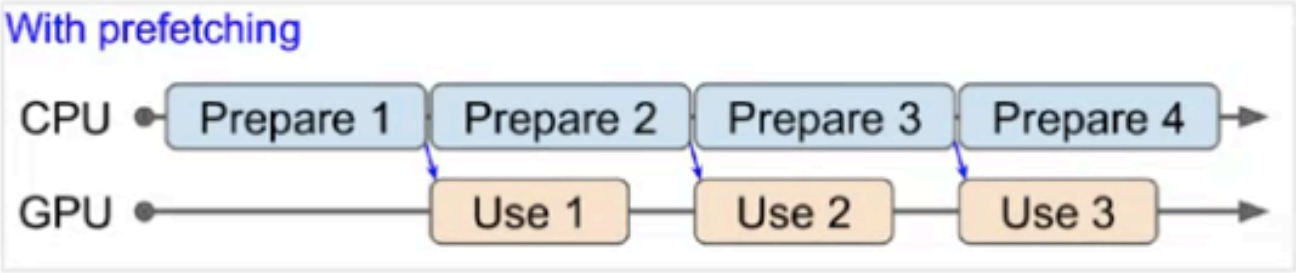


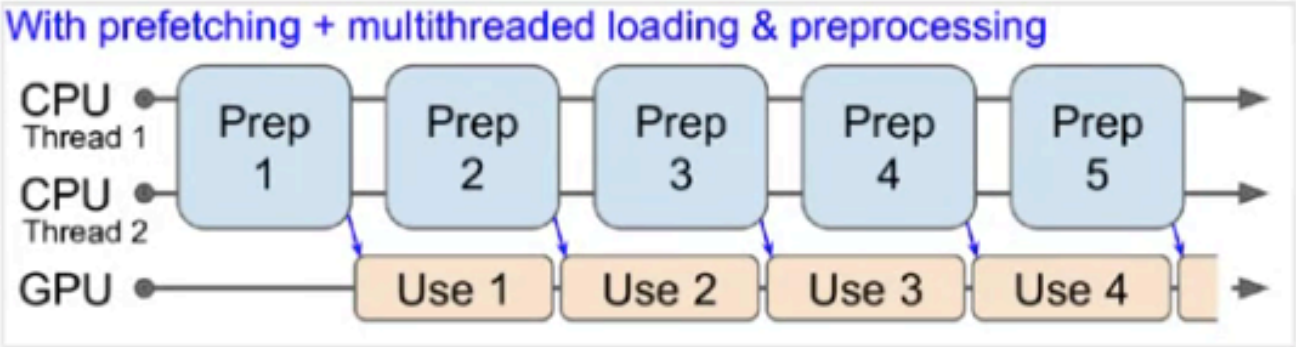

```
def parse_row(row):
   cols = tf.decode_csv(row, record_defaults=[[0],['house'],[0]])
   features = {'sq_footage': cols[0], 'type': cols[1]}
   label = cols[2] # price
   return features, label


def create_dataset(path):
    dataset = tf.data.Dataset.list_files(path)                    \
                             .flat_map(tf.data.TextLineDataset) \
                             .map(parse_row)

    dataset = dataset.shuffle(1000) \
                     .repeat(15)    \
                     .batch(128)
    return dataset
```

  - flat_map() does a one to many transformation
    - Converts one file name to a collection of text files.
  - map() does a one to one transformation
    - Applies parse_row to every line in the dataset

- **Datasets allow for prefetching and preprocessing**







- **Datasets** provide a rich hierarchy of functions to not only ingest data, but also transform and process the data

```
dataset = tf.data.TextLineDataset(filename)\
                 .skip(num_header_lines)\
                 .map(add_key)\
                 .map(decode_csv)\
                 .map(lambda feats, labels: preproc(feats), labels)
                 .filter(is_valid)\
                 .cache()
```

2.1 tf.feature_column

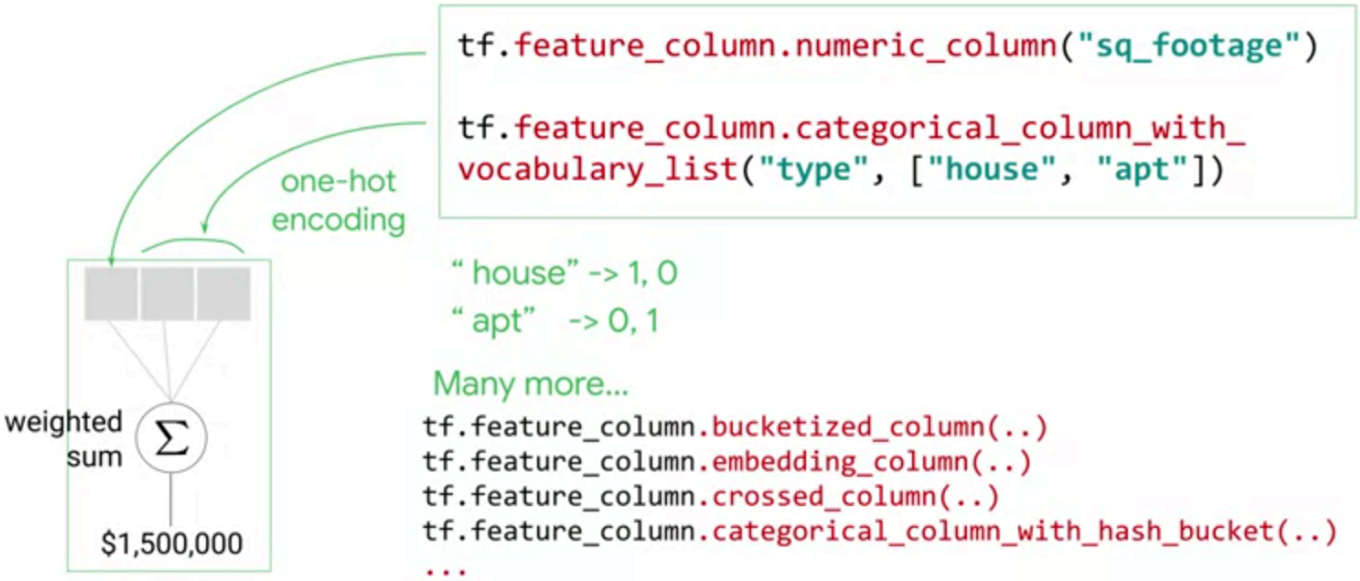
- A feature column describes how the model should use raw input data from your features dictionary.
- It provides methods for the input data to be properly transformed before sending it to a model for training. It tells the model what inputs to expect.
- The model just wants to work with numbers, that's the tensors part.

```
import tensorflow as tf

featcols = [
  tf.feature_column.numeric_column("sq_footage"),
  tf.feature_column.categorical_column_with_vocabulary_list("type",
                                                            ["house", "apt"])
]

...
```

"Features"

- **categorical column with vocabulary list**
  - Use this when your inputs are in a string or integer format and
  - you have an in-memory vocabulary mapping each value to an integer ID.
  - By default, out of vocabulary values are ignored.
- **categorical column with a vocabulary file**
  - when inputs are in a string or integer format,
  - and there's a vocabulary file that map's each value in your integer ID.
- **categorical column with identity**
  - used when inputs are integers in the range of zero to the number of buckets.
  - You want to use the input value itself as the categorical ID,
- **categorical column with a hash bucket**
  - used when features are sparse in the string or integer format.
  - And you want to distribute your inputs into a finite number of buckets by hashing them.

Under the hood: Feature columns take care of packing the inputs into the input vector of the model



- Besides categorical columns, there are many other types too
  - **Bucketized column**
    - Helps with discretizing continuous feature values

```
NBUCKETS = 16
latbuckets = np.linspace(start=38.0, stop=42.0, num=NBUCKETS).tolist()
lonbuckets = np.linspace(start=-76.0, stop=-72.0, num=NBUCKETS).tolist()
```

set up numeric ranges

```
fc_bucketized_plat = fc.bucketized_column(
    source_column=fc.numeric_column("pickup_longitude"),
    boundaries=lonbuckets)

fc_bucketized_plon = fc.bucketized_column(
    source_column=fc.numeric_column("pickup_latitude"),
    boundaries=latbuckets)

...
```

create bucketized columns for pickup latitude and pickup longitude

    - In this example, if we were to consider the latitude and longitude of the house or apartment that we're training or predicting on, we wouldn't want to feed in the raw latitude and longitude values. Instead, we would create buckets that could group the ranges of values for latitude and longitude. It's kind of like zooming out, if you're looking at just like a zip code.

- **Embedding Column**
  - Categorical columns are represented in TensorFlow as sparse tensors.
    - So categorical columns are an example of something that's sparse.
    - TensorFlow can do math operations on sparse tensors without having to convert them into dense values first, and this saves memory and optimizes compute time.
    - But as the number of categories of the feature grow large, it becomes infeasible to train a neural network using that 1 hot encodings.
  - Embeddings overcome this limitation.

```
fc_ploc = fc.embedding_column(categorical_column=fc_crossed_ploc,
                              dimension=3)
```

lower dimensional, dense vector in which each cell contains a number, not just 0 or 1

  - Instead of representing the data as a one hot vector of many dimensions, an embedding column represents that data, at a lower dimensional level or a dense vector in which each cell can contain any number not just a 0 or a 1.

2.2 TFRecord and tf.Example

- The TFRecord format is a simple format for storing a sequence of binary records.
- Protocol buffers are a cross-platform, cross-language library for efficient serialization of structured data.
- Protobuf messages are defined by .proto files, these are often the easiest way to understand a message type.
- The tf.Example message (or protobuf) is a flexible message type that represents a {"string": value} mapping.
- It is designed for use with TensorFlow and is used throughout the higher-level APIs such as TFX.

2.2.1 tf.Example

- A tf.Example is a {"string": tf.train.Feature} mapping.
- The tf.train.Feature message type can accept one of the following three types. Most other generic types can be coerced into one of these:
    - tf.train.BytesList (the following types can be coerced)
        - string
        - byte
    - tf.train.FloatList (the following types can be coerced)
        - float (float32)
        - double (float64)
    - tf.train.Int64List (the following types can be coerced)
        - bool
        - enum
        - int32
        - uint32
        - int64
        - uint64

- Converting a standard tensor scalar to a tf.Example compatible tf.train.Feature

```
# The following functions can be used to convert a value to a type compatible
# with tf.Example.

def _bytes_feature(value):
    """Returns a bytes_list from a string / byte."""
    if isinstance(value, type(tf.constant(0))):
        value = value.numpy() # BytesList won't unpack a string from an EagerTensor.
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

def _float_feature(value):
    """Returns a float_list from a float / double."""
    return tf.train.Feature(float_list=tf.train.FloatList(value=[value]))

def _int64_feature(value):
    """Returns an int64_list from a bool / enum / int / uint."""
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))
```

- The above converts scalars.
    - The simplest way to handle non-scalar features is to use tf.serialize_tensor to convert tensors to binary-strings.
    - Strings are scalars in tensorflow.
    - Use tf.parse_tensor to convert the binary-string back to a tensor.

- Converting proto message back to a string

```
# `SerializeToString()` serializes the message and returns it as a string
feature = _float_feature(np.exp(1))
feature.SerializeToString()
```

```
b'\x12\x06\n\x04T\xf8@'
```

- Creating a tf.Example from existing data
    - **STEP 1**: Within each observation, each value needs to be converted to a tf.train.Feature containing one of the 3 compatible types, using one of the functions above.
    - **STEP 2**: Create a map (dictionary) from the feature name string to the encoded feature value produced in #1.
    - **STEP 3**: The map produced in step 2 is converted to a Features message.

- Code example:
    - 10,000 observations
    - Data has 4 features
        - a boolean feature, False or True with equal probability
            - an integer feature uniformly randomly chosen from [0, 5]
            - a string feature generated from a string table by using the integer feature as an index
            - a float feature from a standard normal distribution

```
# The number of observations in the dataset.
n_observations = int(1e4)

# Boolean feature, encoded as False or True.
feature0 = np.random.choice([False, True], n_observations)

# Integer feature, random from 0 to 4.
feature1 = np.random.randint(0, 5, n_observations)

# String feature
strings = np.array([b'cat', b'dog', b'chicken', b'horse', b'goat'])
feature2 = strings[feature1]

# Float feature, from a standard normal distribution
feature3 = np.random.randn(n_observations)
```

    - Each of these features can be coerced into a tf.Example-compatible type using one of _bytes_feature, _float_feature, _int64_feature.

    - You can then create a tf.Example message from these encoded features:

```python
def serialize_example(feature0, feature1, feature2, feature3):
    """
    Creates a tf.Example message ready to be written to a file.
    """
    # Create a dictionary mapping the feature name to the tf.Example-compatible
    # data type.
    feature = {
      'feature0': _int64_feature(feature0),
      'feature1': _int64_feature(feature1),
      'feature2': _bytes_feature(feature2),
      'feature3': _float_feature(feature3),
    }

    # Create a Features message using tf.train.Example.
    example_proto = tf.train.Example(features=tf.train.Features(feature=feature))
    return example_proto.SerializeToString()
```

- Example of how data is written to file

```python
# This is an example observation from the dataset.

example_observation = []

serialized_example = serialize_example(False, 4, b'goat', 0.9876)
serialized_example
```

```
b'\nR\n\x11\n\x08feature1\x12\x05\x1a\x03\n\x01\x04\n\x14\n\x08feature2\x12\x08\n\x06\n\x04goat\n\x14\n\x08feature3\x12\x08\x12\x06\n\x04
[\xd3|?\n\x11\n\x08feature0\x12\x05\x1a\x03\n\x01\x00'
```

- Decoding serialized data

```
example_proto = tf.train.Example.FromString(serialized_example)
example_proto
```

```
features {
  feature {
    key: "feature0"
    value {
      int64_list {
        value: 0
      }
    }
  }
  feature {
    key: "feature1"
    value {
      int64_list {
        value: 4
      }
    }
  }
  feature {
    key: "feature2"
    value {
      bytes_list {
        value: "goat"
      }
    }
  }
  feature {
    key: "feature3"
    value {
      float_list {
        value: 0.9876000285148621
      }
    }
  }
}
```

2.2.2 TFRecord format details

- A TFRecord file contains a sequence of records. The file can only be read sequentially.
- Each record contains a byte-string, for the data-payload, plus the data-length, and CRC32C (32-bit CRC using the Castagnoli polynomial) hashes for integrity checking.
- Each record is stored in the following formats:
    - uint64 length
    - uint32 masked_crc32_of_length
    - byte data[length]
    - uint32 masked_crc32_of_data
- The records are concatenated together to produce the file.
- **Note**:
    - There is no requirement to use tf.Example in TFRecord files.
    - tf.Example is just a method of serializing dictionaries to byte-strings, lines of text, encoded image data, or serialized tensors (using tf.io.serialize_tensor, and tf.io.parse_tensor when loading). See the tf.io module for more options.

2.2.3 TFRecord using tf.data

2.2.3.1 Writing a TFRecord file

- Using the previous data, create a dataset

```
features_dataset = tf.data.Dataset.from_tensor_slices((feature0, feature1, feature2, feature3))
features_dataset
```

```
<TensorSliceDataset shapes: ((), (), (), ()), types: (tf.bool, tf.int64, tf.string, tf.float64)>
```

```
# Use `take(1)` to only pull one example from the dataset.
for f0,f1,f2,f3 in features_dataset.take(1):
    print(f0)
    print(f1)
    print(f2)
    print(f3)
```

```
tf.Tensor(False, shape=(), dtype=bool)
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(b'cat', shape=(), dtype=string)
tf.Tensor(0.6531766113234501, shape=(), dtype=float64)
```

- The serialize_example function defined earlier has a non-tensor return type. This can be wrapped in a tf.py_function to make it return a compatible type.

```
def tf_serialize_example(f0,f1,f2,f3):
    tf_string = tf.py_function(
                    serialize_example,
                    (f0,f1,f2,f3),  # pass these args to the above function.
                    tf.string)      # the return type is `tf.string`.
    return tf.reshape(tf_string, ()) # The result is a scalar
```

- Apply the function to each record in the dataset

```
serialized_features_dataset = features_dataset.map(tf_serialize_example)
serialized_features_dataset
```

```
<MapDataset shapes: (), types: tf.string>
```

- Create a generator to write the records

```
def generator():
    for features in features_dataset:
        yield serialize_example(*features)
```

```
serialized_features_dataset = tf.data.Dataset.from_generator(
                                    generator,
                                    output_types=tf.string,
                                    output_shapes=())
```

- Write to a TFRecord file

```
filename = 'test.tfrecord'
writer = tf.data.experimental.TFRecordWriter(filename)
writer.write(serialized_features_dataset)
```

2.2.3.2 Reading a TFRecord File

- Using tf.data.TFRecordDataset

```
filenames = [filename]
raw_dataset = tf.data.TFRecordDataset(filenames)
```

  - the dataset contains serialized tf.train.Example messages.
  - When iterated over it returns these as scalar string tensors.

- These tensors can be parsed using the function below.

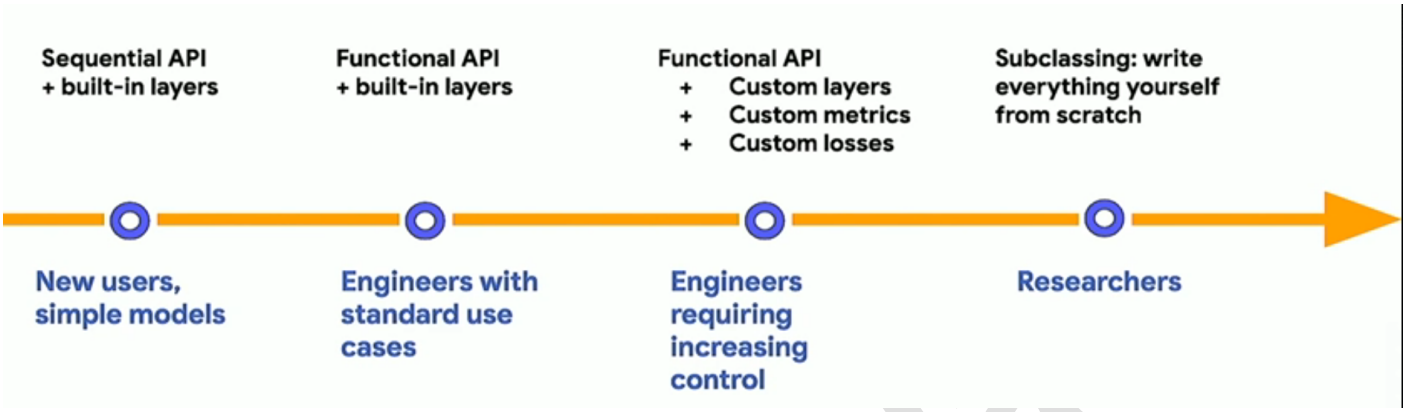
```
feature_description = {
    'feature0': tf.io.FixedLenFeature([], tf.int64, default_value=0),
    'feature1': tf.io.FixedLenFeature([], tf.int64, default_value=0),
    'feature2': tf.io.FixedLenFeature([], tf.string, default_value=''),
    'feature3': tf.io.FixedLenFeature([], tf.float32, default_value=0.0),
}

def _parse_function(example_proto):
  # Parse the input `tf.Example` proto using the dictionary above.
  return tf.io.parse_single_example(example_proto, feature_description)
```

  - Note that the feature_description is necessary here because datasets use graph-execution and need this description to build their shape and type signature.

- Apply this function to the dataset.

```
parsed_dataset = raw_dataset.map(_parse_function)
parsed_dataset
```

3 Keras



3.1 Keras Sequential API

- Tf.keras is TensorFlow's high-level API for building and training your deep learning models.
- A sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor.
- **Defining a Model**

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Input(shape=(64,))
    Dense(units=32, activation="relu", name="hidden1"),
    Dense(units=8, activation="relu", name="hidden2"),
    Dense(units=1, activation="linear", name="output")
])
```

The batch size is omitted. Here the model expects batches of vectors with 64 components.

The Keras sequential model stacks layers on the top of each other.

- Sequential models are not really advisable if
  - The model has multiple inputs or multiple outputs.
  - Any of the layers in the model have multiple inputs and multiple outputs.
  - The model needs to do layer sharing
  - The model has a non-linear topology, such as a residual connection or if it is multi-branches.

- Examples

  - Imports

```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

  - A simple linear model (multiclass logistic regression) with 10 classes

```
# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

A linear model (a single Dense layer) aka multiclass logistic regression

  - A simple neural network with 1 hidden layer

```
# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

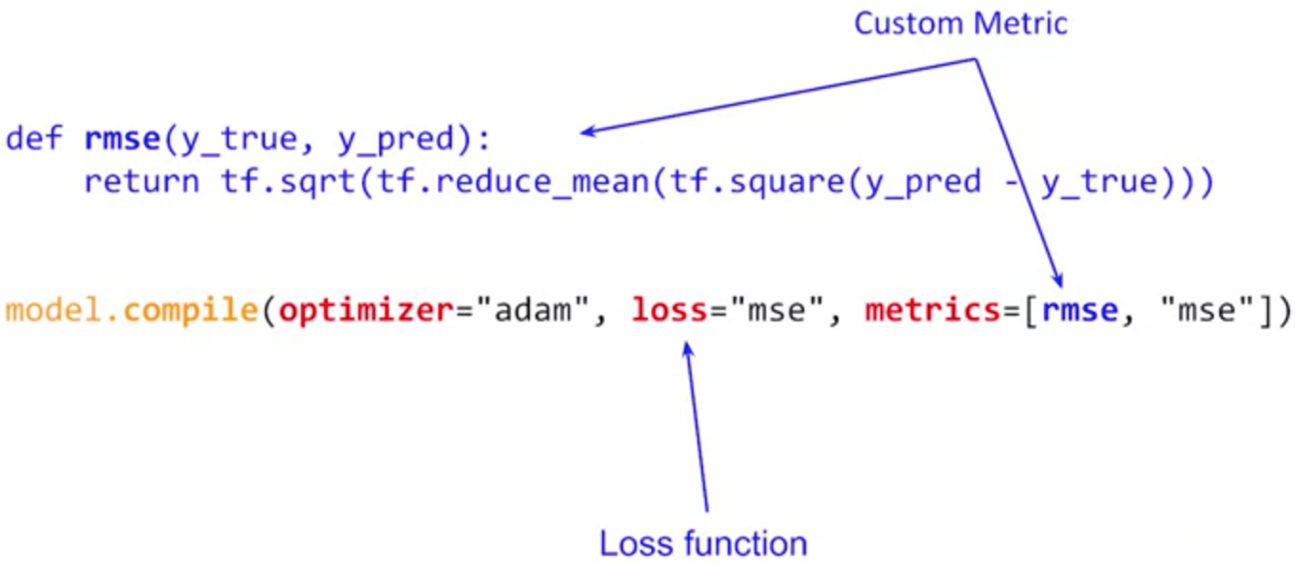
A neural network with one hidden layer

  - A deep neural network

```
# Define your model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

A deeper neural network

- **Compiling a model**



  - **Metrics**
    - These are evaluation metrics.
    - You can specify custom functions or use in-built metrics.
  - **Loss**
    - This is the loss function used by gradient descent.
  - **Optimizer**
    - Optimizers tie together the loss function and the model parameters by actually doing the updating of the model in response to the output of the loss function.
    - Some common optimizers are as follows:
      - ***SGD (Stochastic Gradient Descent)***
        - Gradient descent algorithm that descends the slope to reach the minimum or lowest point on the loss surface.
      - ***Adam***
        - An optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on the training data.
        - Some advantages
          - computationally efficient with low memory requirements.
          - Invariable to diagonal re-scaling of the gradients
        - When to use

        - well-suited for models that have large datasets, or a lot of parameters that you're adjusting.
        - very appropriate for problems with very noisy or sparse gradients and nonstationary objectives.
    - ***Momentum***
        - Reduces learning rate when the gradient values are small.
    - ***Adagrad***
        - Gives frequently occurring features low learning rates.
    - ***Adadelta***
        - Improves Adagrad by avoiding and reducing learning rate to 0.
    - ***FTRL (follow the regularized leader)***
        - works well on wide models.

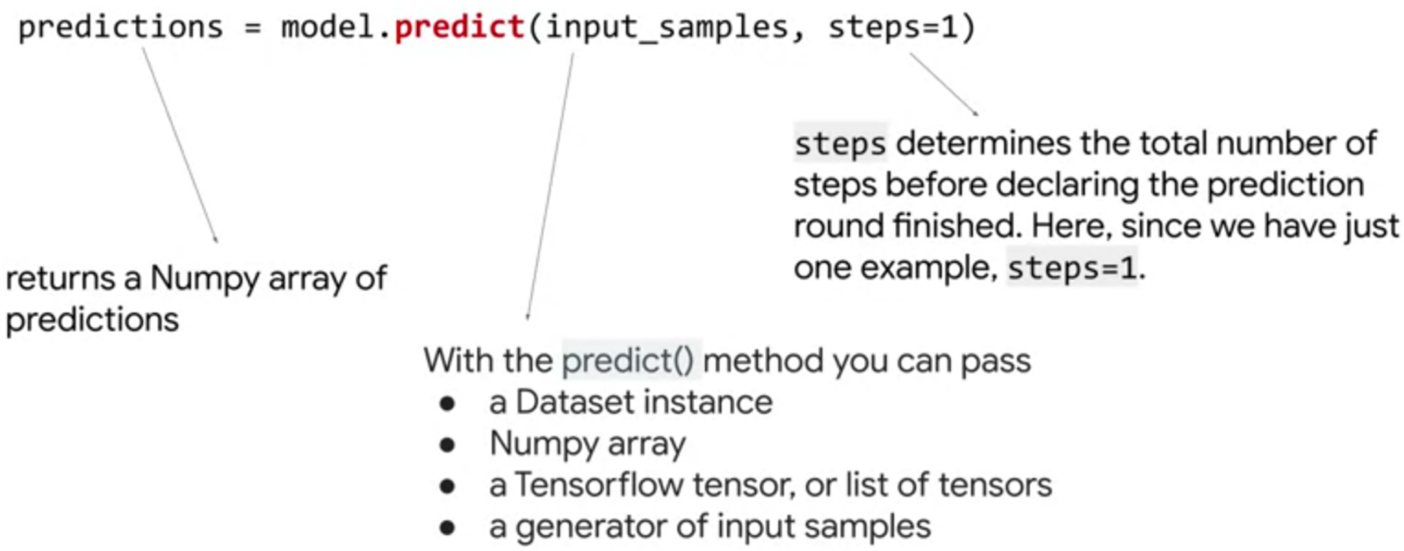
- **Training a model**

```
from tensorflow.keras.callbacks import TensorBoard


steps_per_epoch = NUM_TRAIN_EXAMPLES // (TRAIN_BATCH_SIZE * NUM_EVALS)


history = model.fit(
    x=trainds,
    steps_per_epoch=steps_per_epoch,
    epochs=NUM_EVALS,
    validation_data=evalds,
    callbacks=[TensorBoard(LOGDIR)]
)
```

This is a trick so that we have control on the total number of examples the model trains on (NUM_TRAIN_EXAMPLES) and the total number of evaluation we want to have during training (NUM_EVALS).

    - Use the fit() method.
    - This defines the number of epochs to use.
    - We can also specify the steps per epoch (which is the number of batch iterations before a training epoch is considered finished), the validation data, validation steps

- To train your model, Keras provides three functions that can be used:
    - **.fit()**
        - For training a model for a fixed number of epochs (iterations on a dataset).
        - works well for small datasets which can fit entirely in memory
    - **.fit_generator()**
        - for training a model on data yielded batch-by-batch by a generator
        - For large datasets (or if you need to manipulate the training data on the fly via data augmentation, etc)
    - **.train_on_batch()**
        - runs a single gradient update on a single batch of data.
        - for more fine-grained control over training and accepts only a single batch of data
- **Prediction**



    - If steps is set to None and input is a dataset iterator, predict() will run till the dataset is exhausted.

3.2 Keras Functional API

- The Sequential API is used to build stacks while the functional API is used to build DAGs
- Used when we have multiple inputs\outputs or more complex models.

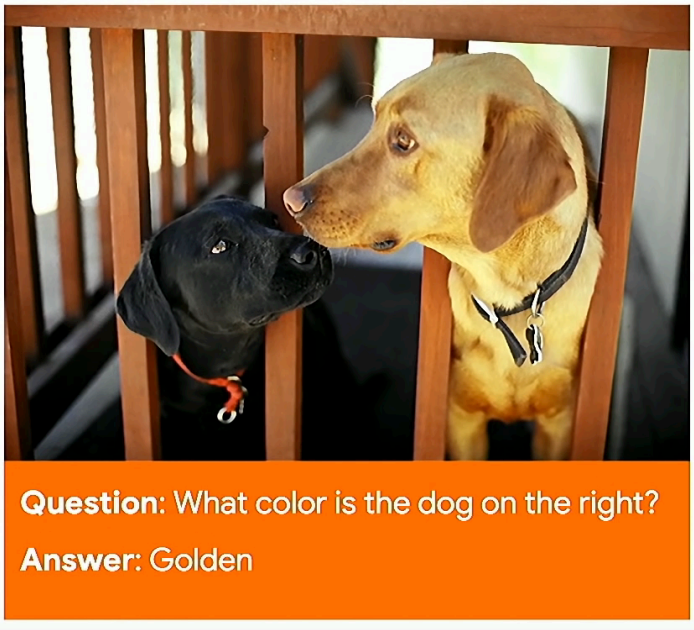
Strengths

- less verbose than using keras.Model subclasses
- validates your model while you're defining it
- your model is plottable and inspectable
- your model can be serialized or cloned

Weaknesses

- doesn't support dynamic architectures
- sometimes you have to write from scratch and you need to build subclasses, e.g. custom training or inference layers

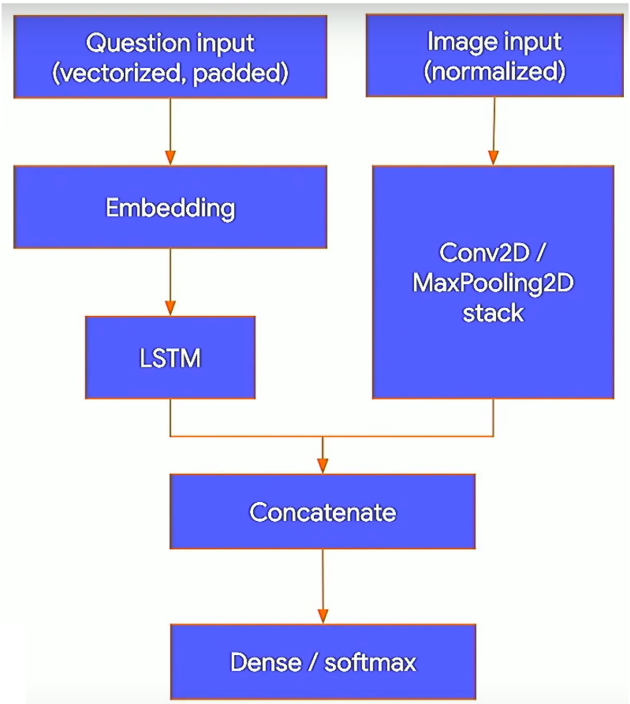
- **Example**: Visual Question Answering (VQA)
  - We are given an image and a question in natural language.
  - The answer to the question can be modelled as a classification problem or a natural language problem. In this example, we go with classification.
  - As an example, below is a VQA



- One possible workflow could be as follows:

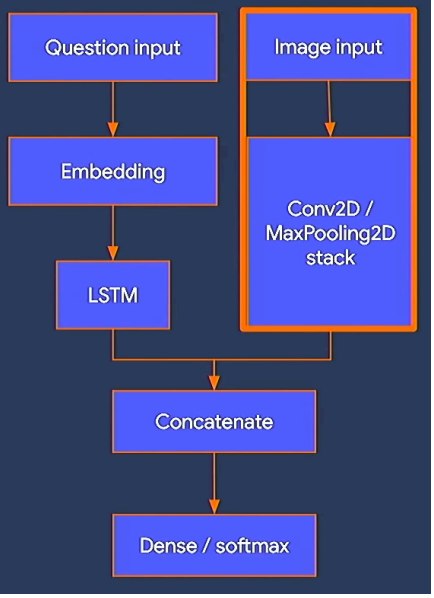
A multi-input model

1. Use a CNN to embed the image
2. Use a LSTM to embed the question
3. **Concatenate**
4. Classify with Dense layers, per usual



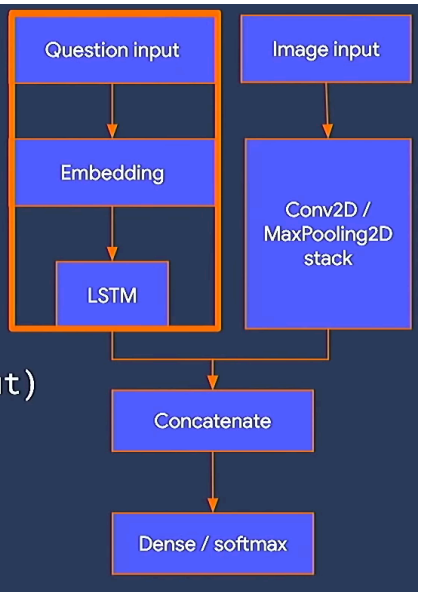
- The sample code is as follows
  - The vision model

```
# A vision model.
# Encode an image into a vector.
vision_model = Sequential()
vision_model.add(Conv2D(64, (3, 3),
                        activation='relu',
                        input_shape=(224, 224, 3)))
vision_model.add(MaxPooling2D())
vision_model.add(Flatten())

# Get a tensor with the output of your vision model
image_input = Input(shape=(224, 224, 3))
encoded_image = vision_model(image_input)
```



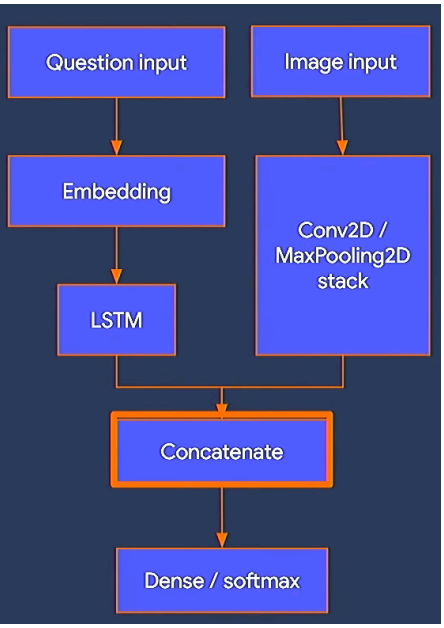
- The language model

```
# A language model.
# Encode the question into a vector.
question_input = Input(shape=(100,),
                       dtype='int32',
                       name="Question")

embedded = Embedding(input_dim=10000,
                     output_dim=256,
                     input_length=100)(question_input)

encoded_question = LSTM(256)(embedded_question)
```



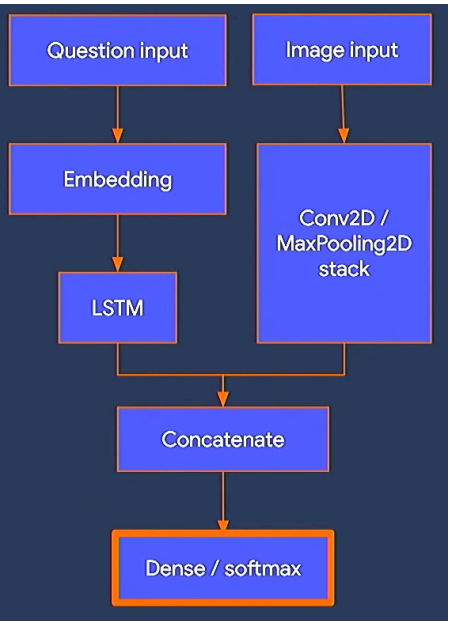
- Combining the outputs from the two models

```
# Concatenate the encoded image and question
merged = layers.concatenate([encoded_image,
                             encoded_question])
```



- The final functional model

```
# Train a classifier on top.
output = Dense(1000,
               activation='softmax')(merged)


# You can train w/ .fit, .train_on_batch,
# or with a GradientTape.
vqa_model = Model(inputs=[image_input,
                          question_input],
                          outputs=output)
```



3.3 Keras Subclassing

- Similar to how pytorch works
- Inside of Keras, the "Model" class is the root class used to define a model architecture.
- Since Keras utilizes object-oriented programming, we can actually subclass the Model class and then insert our architecture definition.
- Model subclassing is fully customizable and enables you to implement your own custom forward-pass of the model.

```
class MyModel(tf.keras.Model):
  def __init__(self, num_classes=10):
    super(MyModel, self).__init__(name='my_model')
    self.dense_1 = layers.Dense(32, activation='relu')
    self.dense_2 = layers.Dense(num_classes,activation='softmax')

  def call(self, inputs):
    # Define your forward pass here
    x = self.dense_1(inputs)
    return self.dense_2(x)
```

- The constructor (init)
  - Define the layers here
- The call method
  - Describe how the layers are chained together
- In the above code, for example, if we don't want to use the built-in ReLU, we could call activation as below too

```
class MyModel(tf.keras.Model):
  def __init__(self, num_classes=10):
    super(MyModel, self).__init__(name='my_model')
    self.dense_1 = layers.Dense(32)
    self.dense_2 = layers.Dense(num_classes,activation='softmax')

  def call(self, inputs):
    # Define your forward pass here
    x = self.dense_1(inputs)
    x = tf.nn.relu(x)
    return self.dense_2(x)
```

3.4 Saving a Model

- We'll export the model to a TensorFlow SavedModel format.
- Once we have a model in this format, we have lots of ways to serve the model:
    - web application,
    - code like JavaScript from a mobile application,
    - etc.

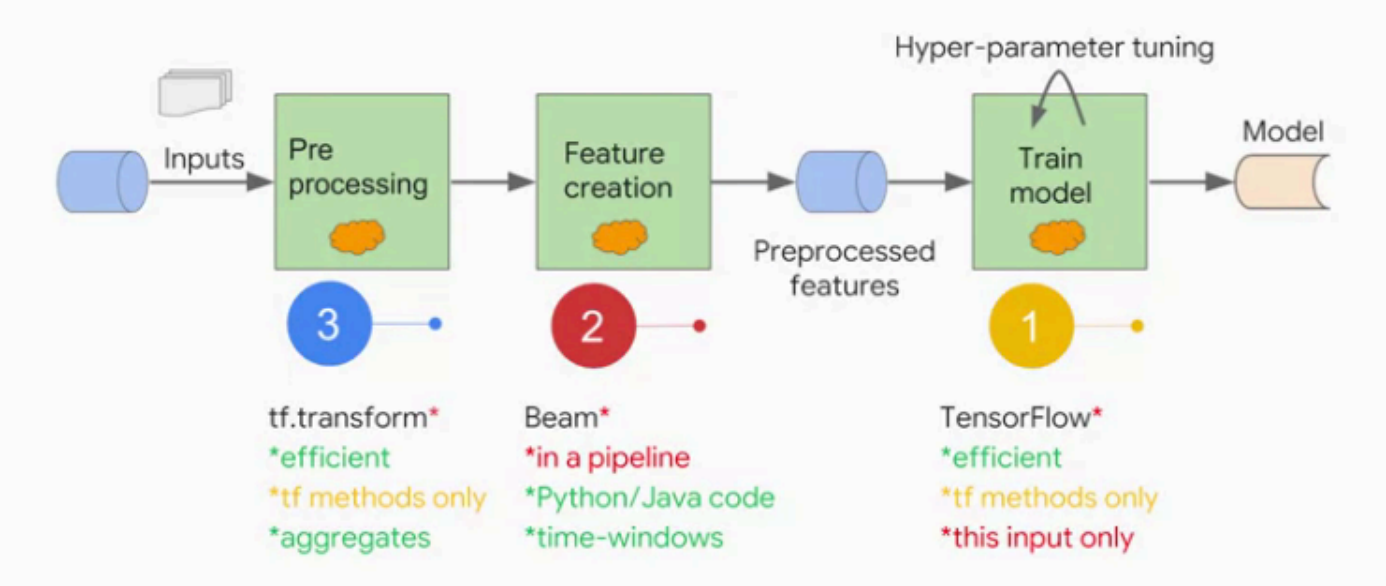
- Example code

```
OUTPUT_DIR = "./export/savedmodel"
shutil.rmtree(OUTPUT_DIR, ignore_errors=True)

EXPORT_PATH = os.path.join(OUTPUT_DIR,
datetime.datetime.now().strftime("%Y%m%d%H%M%S"))

tf.saved_model.save(model, EXPORT_PATH)
```

- SavedModel is a universal serialization format for TensorFlow models.
- SavedModel provides a language-neutral format to save your Machine Learning models that is both recoverable and hermetic.
- It enables higher-level systems and tools to produce, consume, and transform your TensorFlow models.
- The resulting SavedModel is then servable.
- Models saved in this format can be restored using the tf.keras.models.load_model and they're compatible with TensorFlow Serving.

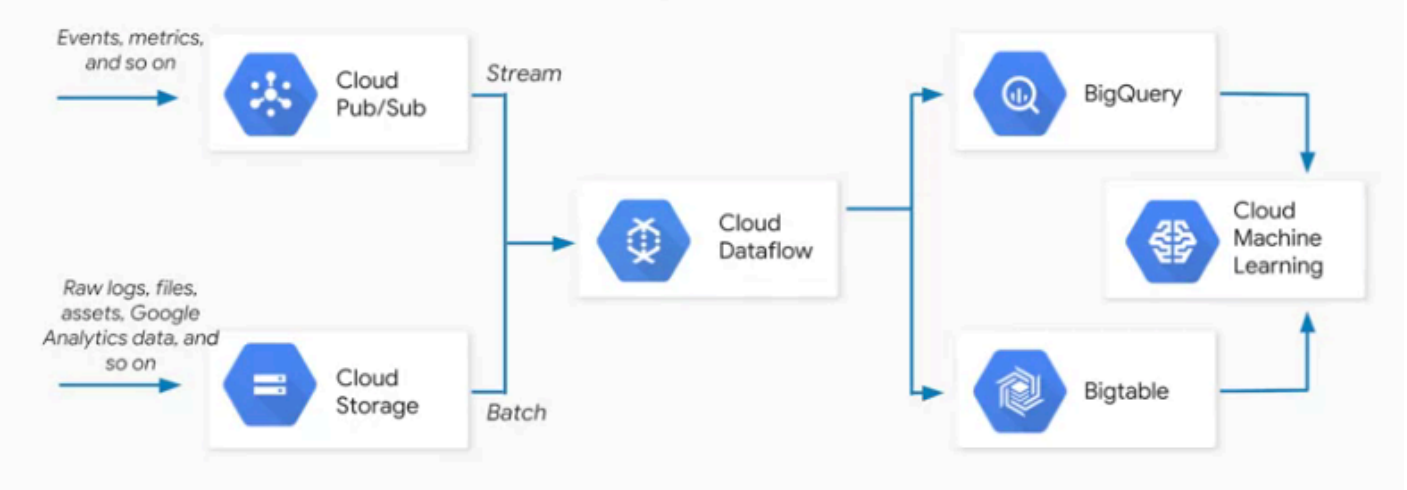
4 Tensorflow Transform

- Allows us to carry out feature processing efficiently, at scale and on streaming data.
- The limit with doing TensorFlow processing is that we can do preprocessing on a single input only. Sequence model is an exception, but TensorFlow models tend to be stateless.
- So for example, if we need min\max to scale the dataset, we would need to run a data pipeline even during prediction. Tensorflow transform provides a way to do so more efficiently.



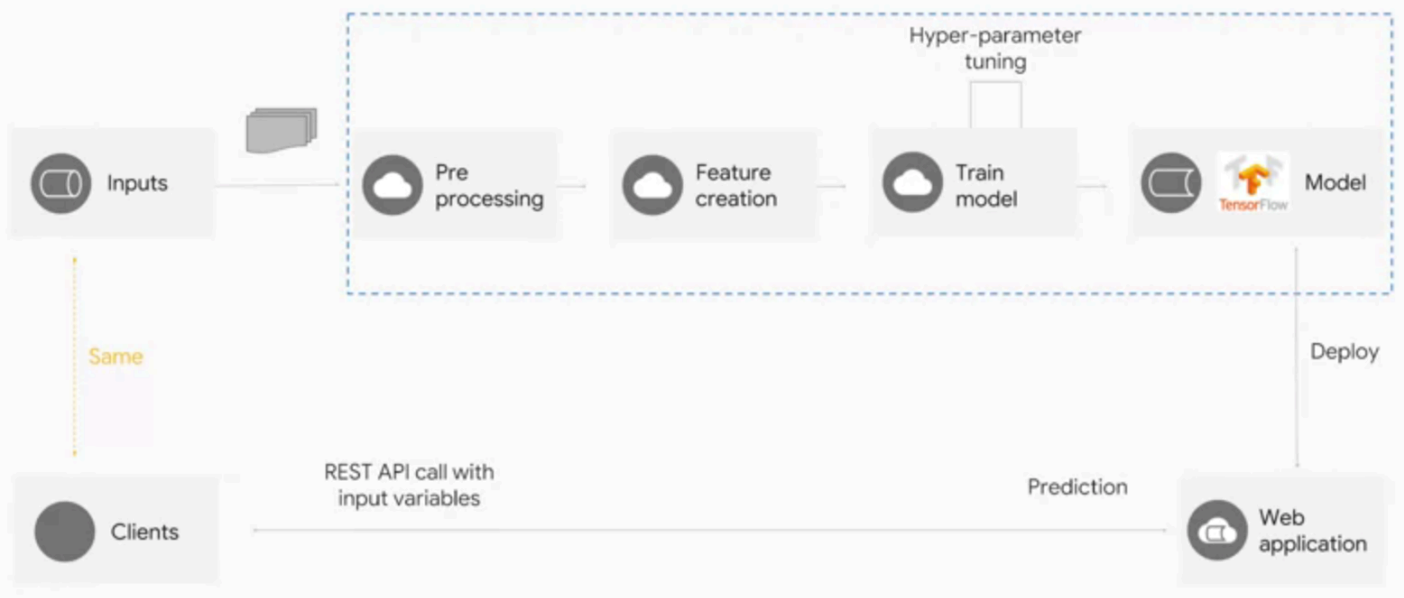
  - With TensorFlow transform, you're limited to TensorFlow methods but then you also get the efficiency of TensorFlow.
  - You can also use the aggregate of your entire training dataset because tf.transform uses Dataflow during training but only TensorFlow during prediction.

- Tensorflow transform is a hybrid of Apache Beam and tensorflow.
    - Dataflow preprocessing only works in the context of a pipeline.



- Think in terms of incoming streaming data such as IoT data, Internet of Things data, or flights data.
- The Dataflow pipeline might involve the predictions, and it might invoke those predictions and save those predictions to big table. These predictions are then served to anyone who visits the webpage in the next 60 seconds at which point a new prediction is available in big table.
- When you hear Dataflow think backend preprocessing for machine learning models.
- You can use Dataflow for preprocessing that needs to maintain state such as time Windows.

- For on-the-fly preprocessing for machine learning models, think TensorFlow.

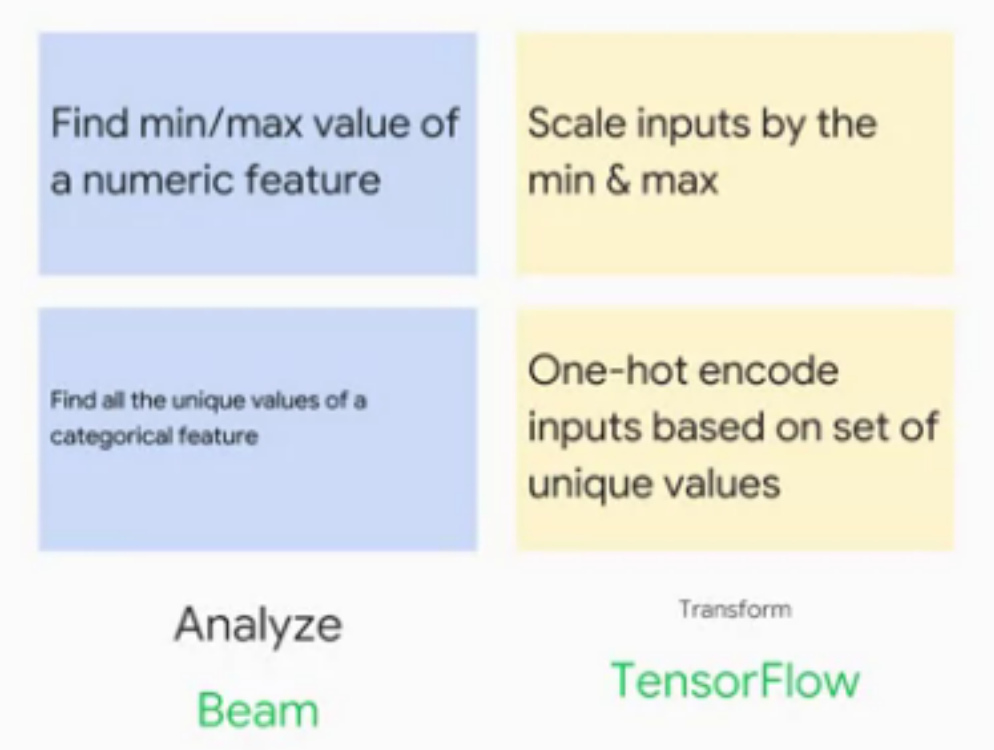


  - You use TensorFlow for preprocessing that is based on the provided input only.
  - If you put all the stuff in the dotted box into the TensorFlow graph, then it's quite easy for clients to just invoke a web application and get all the processing handled for them.

- To understand how tensorflow transform works, consider that general preprocessing has 2 stages:

  - **Analyze stage**:
    - This is the stage where we traverse the entire dataset to obtain insights. (Training Phase)
    - Apache Beam is best suited for this kind of work loads.

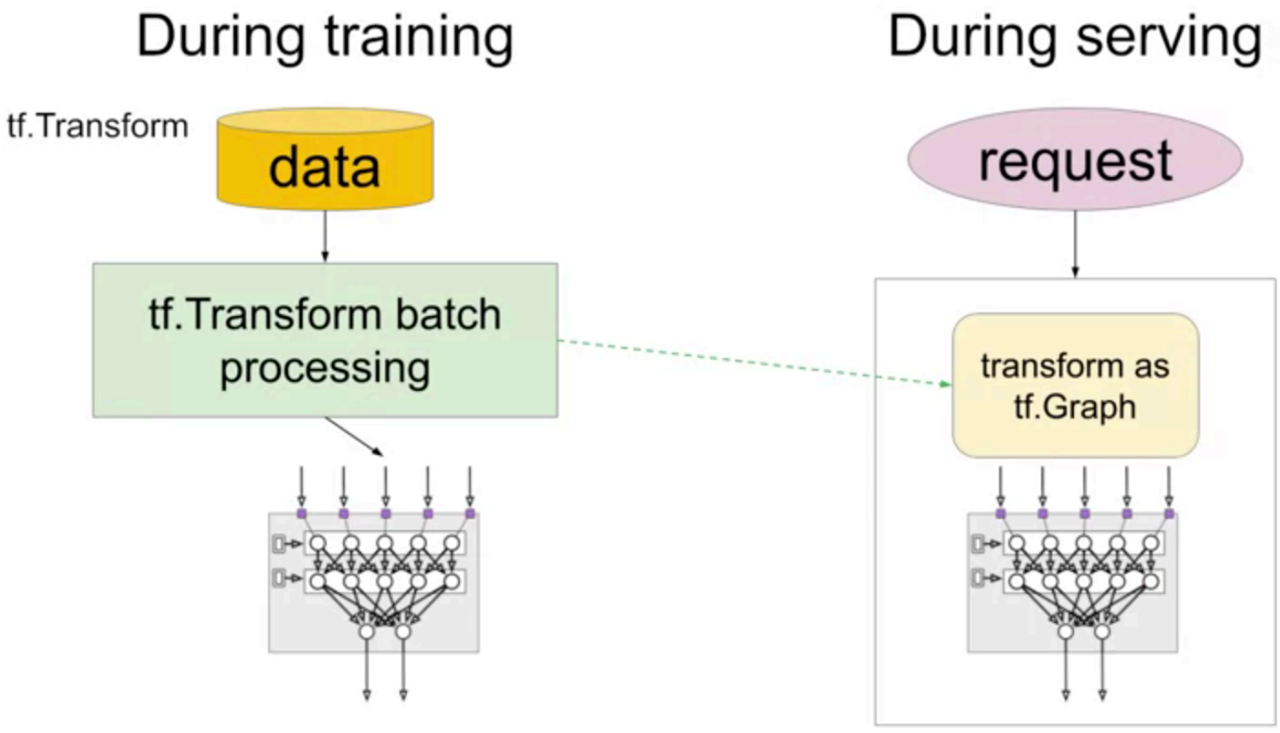
  - **Transform stage**:
    - This is the stage where we use the data obtained above and transform it or use it in some way to modify data at hand.
    - Since this is on the fly preprocessing, tensorflow is best suited for this.
    - Evaluation\Prediction phase.



4.1 PTransform (Parallel Transform) methods

- Remember that computing the min and max, et cetera during the analysis is done only on the training dataset. We cannot use the evaluation dataset for that.
  - So, the evaluation dataset is scaled using the min and max found in the training data.
  - But what if the max in the evaluation is bigger? This simulates the situation when you deploy your model and then you find that a bigger value comes in at prediction time. It's no different.
  - You cannot use an evaluation dataset to compute min and max, vocabulary et cetera.

- One way to think about it is that there are two phases.
  - The analysis phase is executed in Beam while creating the training dataset.
  - The transform phase is executed in TensorFlow during prediction.



4.1.1 Analyze Phase (training phase)

- **NOTE**:
  - We analyze only the training dataset
  - We use the AnalyzeAndTransform method in this phase.
  - This returns the transform function that can be used during evaluation and serving phases.

- **STEP 1:**
  - Setup a schema.
  - This schema will be used to create a metadata template used by beam in Dataflow.

```
raw_data_schema = {                                  TensorFlow type for input column
    colname : dataset_schema.ColumnSchema(tf.string, ...)
        for colname in 'dayofweek,key'.split(',')
}
raw_data_schema.update({                                                    float32
      colname : dataset_schema.ColumnSchema(tf.float32, ...)
        for colname in 'fare_amount,pickuplon, ... ,dropofflat'.split(',')
})
raw_data_metadata =
    dataset_metadata.DatasetMetadata(dataset_schema.Schema(raw_data_schema))
```

- **STEP 2:**
  - Run the analyzeAndTransform PTransform on the training dataset to get back the preprocessed training data and transform function

```
raw_data = (p                                   1. Read in data as usual for Beam
    | beam.io.Read(beam.io.BigQuerySource(query=myquery, use_standard_sql=True))
    | beam.Filter(is_valid))                    2. Filter out data that you don't want to train with

                                                3. Pass raw data + metadata template to AnalyzeAndTransformDataset
transformed_dataset, transform_fn = ((raw_data, raw_data_metadata)
    | beam_impl.AnalyzeAndTransformDataset(preprocess))
```

4. Get back transformed dataset and a reusable transform function

    - The returned transformed dataset is a PCollection
    - The transformations carried out as part of the preprocess function are returned as a transform function as shown above.

- **STEP 3:**
  - Write out the transformed data into a TFRecord (this is the most efficient format for tensorflow)

```
transformed_data |
   tfrecordio.WriteToTFRecord(
     os.path.join(OUTPUT_DIR,'train'),

     coder=ExampleProtoCoder(
           transformed_metadata.schema)
   )
```

    - **NOTE**: The schema used above is a transformed metadata schema since we are writing the transformed data and not the raw data.

4.1.2 Transform Phase (evaluation\serving phase)

- The preprocess function referenced earlier is a function that transforms the input data.

```
transformed_dataset, transform_fn = ((raw_data, raw_data_metadata)
     | beam_impl.AnalyzeAndTransformDataset(
                    preprocess))
```

The things you do in preprocess() will get added to the TensorFlow graph, and be executed in TensorFlow during serving

  - In **Beam**, it is called as part of the AnalyzeAndTransformDataset
  - In **TensorFlow**, the things you do in preprocess() will get called essentially as part of the serving_input_fn in TensorFlow.

- The preprocessing function is restricted to functions that can be called from Tensorflow Graph
  - You cannot call regular python functions, since the preprocess is part of the tensorflow graph during serving.

```
def preprocess(inputs):

    result = {}    Create features from the input tensors and put into "results" dict

  result['fare_amount'] = inputs['fare_amount'] Pass through

  result['dayofweek'] = tft.string_to_int(inputs['dayofweek']) vocabulary
  ...
  result['dropofflat'] = (tft.scale_to_0_1(inputs['dropofflat']))     scaling

  result['passengers'] = tf.cast(inputs['passengers'], tf.float32) Other TF fns

  return result
```

    - The 'inputs' is a dict whose values are tensors.
      - Input functions return features, labels, and this is the features. And features is a dict.
      - Tf.transform will take care of converting the data that comes in via PTransform into tensors during the analysis phase.
      - We take the tensors and use them to create new features and put these features into a new dictionary.

- **tft.string_to_int()**
  - We want 'dayofweek' to be an integer. However, in the input, it is a string like "Thu".
  - So, what we are doing it is that we are asking tf.transform to convert the string that is read (such as "Thu") into an integer (such as 3 or 5).
  - What tf transform will do is to compute the vocabulary of all the possible days of the week in the training dataset during the Analyze phase and use that information to do the string-to-int mapping.

- **tft.scale_to_0_1()**
  - Tf.transform will compute the min and max of the column and use those values to scale the inputs between 0 and 1.

- We can also invoke other TensorFlow functions like casting an integer in JSON to a real-valued number.

4.1.2.1 Evaluation Dataset

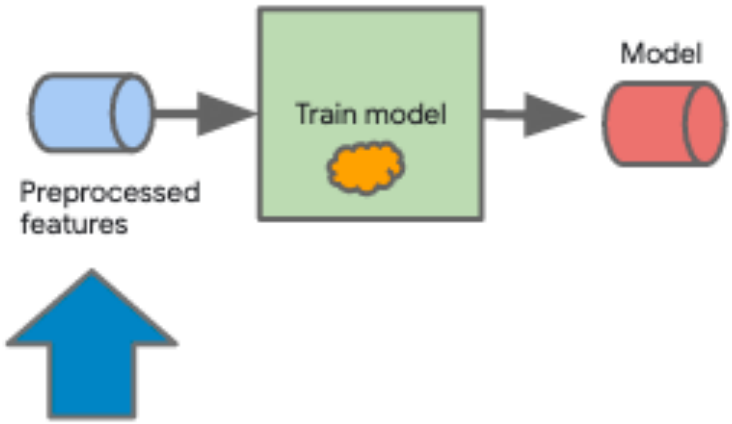
- AnalyzeAndTransform is done on training data
- For the evaluation dataset, we carry out pretty much the same Beam pipeline that we did on the training dataset.

```
raw_test_data = (p
        | beam.io.Read(beam.io.BigQuerySource(...)
        | 'eval_filter' >> beam.Filter(is_valid))
transformed_test_dataset = (((raw_test_data, raw_data_metadata), transform_fn)
        | beam_impl.TransformDataset())

transformed_test_data, _ = transformed_test_dataset
_ = transformed_test_data | tfrecordio.WriteToTFRecord(
          os.path.join(OUTPUT_DIR, 'eval'),
          coder=example_proto_coder.ExampleProtoCoder(
              transformed_metadata.schema))
```

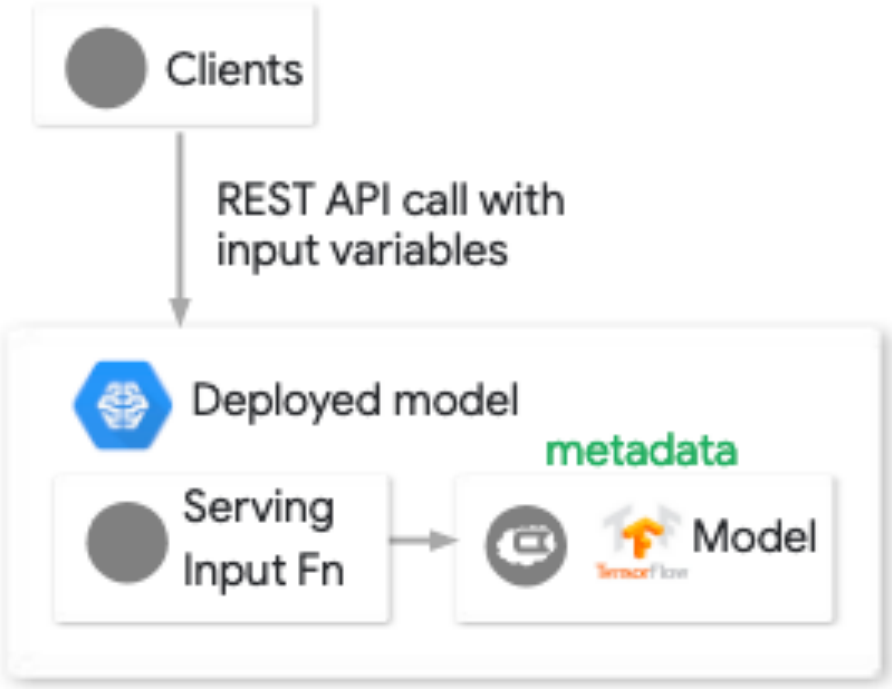
- There is one big exception, though.
  - We don't analyze the evaluation dataset.
- If we are scaling the values, the values in the evaluation dataset will be scaled based on the min & max found in the training dataset.
- So, on the evaluation dataset, we just call TransformDataset.
- This will take care of calling all the things that we did in preprocess!
- TransformDataset needs as input the transform_fn that was computed on the training data.
- Once we have the transformed data, it can be written out just like the way it was done for the training data.

4.1.2.2 Model Serving

- We use Beam for training and evaluation



- For the prediction phase, we will use Tensorflow.



- For the model to know what functions to call, we need to save the transform function. Below, all are added to a metadata folder alongside the trained model.

```
_ = transform_fn | transform_fn_io.WriteTransformFn(
     os.path.join(OUTPUT_DIR, 'metadata')))
```

- The reason these functions needed to be in TensorFlow was that they are part of the prediction graph, so that the end-user can give the model raw data and the model can do the necessary preprocessing.

- How does model know what function to call?
  - Change training and evaluation input functions to read preprocessed features.

```
def read_dataset(args, mode):
   if mode == tf.estimator.ModeKeys.TRAIN:
       input_paths = args['train_data_paths']
   else:
       input_paths = args['eval_data_paths']        Reading transformed features
   transformed_metadata = metadata_io.read_metadata(
             os.path.join(args['metadata_path'], 'transformed_metadata'))
   return input_fn_maker.build_training_input_fn(
         metadata = transformed_metadata,
         file_pattern = (
             input_paths[0] if len(input_paths) == 1 else input_paths),
        ...)
```

    - The helper function **build_training_input_fn()** is used for both training and evaluation data.

  - Change the serving input function as follows:
    - It accepts raw data as input

```
def make_serving_input_fn(args):
   raw_metadata = metadata_io.read_metadata(
       os.path.join(args['metadata_path'], 'rawdata_metadata'))
   transform_savedmodel_dir = (
       os.path.join(args['metadata_path'], 'transform_fn'))
   return input_fn_maker.build_parsing_transforming_serving_input_receiver_fn(
     raw_metadata,
     transform_savedmodel_dir,
     exclude_raw_keys = [LABEL_COLUMN])
```

    - The **build_parsing_transforming_...()** function
        - This parse the Json according to raw data schema, transforms the raw data based on the TensorFlow operations in saved_model.pb and then sends it along to the model for prediction.

- Thus, we just need to send raw data and tensorflow will do the necessary transformations required for prediction.