Custom Models and Training with TensorFlow

Agenda

- Quick Tour of TensorFlow
- Using TensorFlow like NumPy
- Customizing Models and Training Algorithms
- TensorFlow Functions and Graphs

Quick Tour of TensorFlow

- Until now, we have used only TensorFlow's high-level API, tf.keras
- 95% cases we will not require anything other than tf.keras
- In the chapter, we will go deeper into TensorFlow lower-level Python API

Quick Tour of TensorFlow

- This will be useful when we need extra control
    - Custom Loss Function
    - Custom metrics
    - Models etc

Using TensorFlow like NumPy

- Quick Tour of TensorFlow
- Using TensorFlow like NumPy
- Customizing Models and Training Algorithms
- TensorFlow Functions and Graphs

Quick Tour of TensorFlow

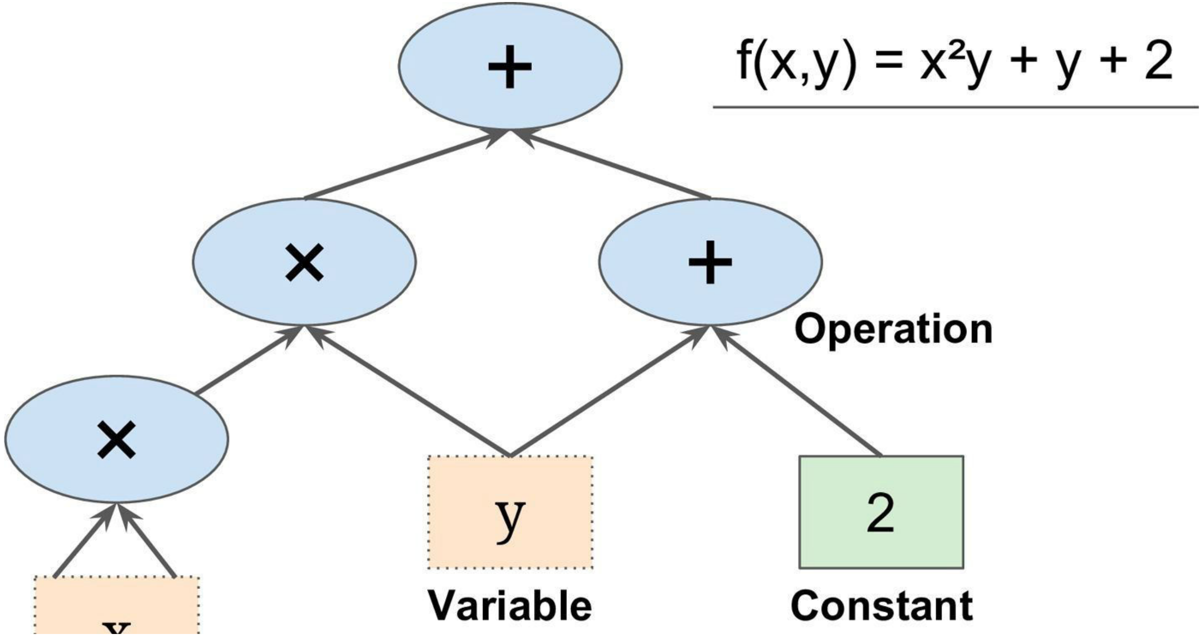
- Powerful library for numerical computation
- Well suited for large scale machine-learning
- Developed by Google Brain’s team and powers their large scale services like
  - Google Photos
  - Google Cloud Speech
  - Google Search

TensorFlow's Python API

Modules	Purpose
tf.keras tf.estimator	High-level Deep Learning APIs
tf.nn tf.losses tf.metrics tf.optimizers tf.train tf.initializers	Low-level Deep Learning APIs
tf.GradientTape tf.gradients()	Autodiff
tf.data tf.feature_column tf.audio tf.image tf.io tf.queue	I/O and Preprocessing
tf.summary	Visualization with Tensorboard
tf.distribute tf.saved_model tf.autograph tf.graph_util tf.lite tf.quantization tf.tpu tf.xla	Deployment and optimization
tf.lookup tf.nest tf.ragged tf.sets tf.sparse tf.strings	Special data structures
tf.math tf.linalg tf.signal tf.random tf.bitwise	Mathematics including linear algebra and signal processing
tf.compat	

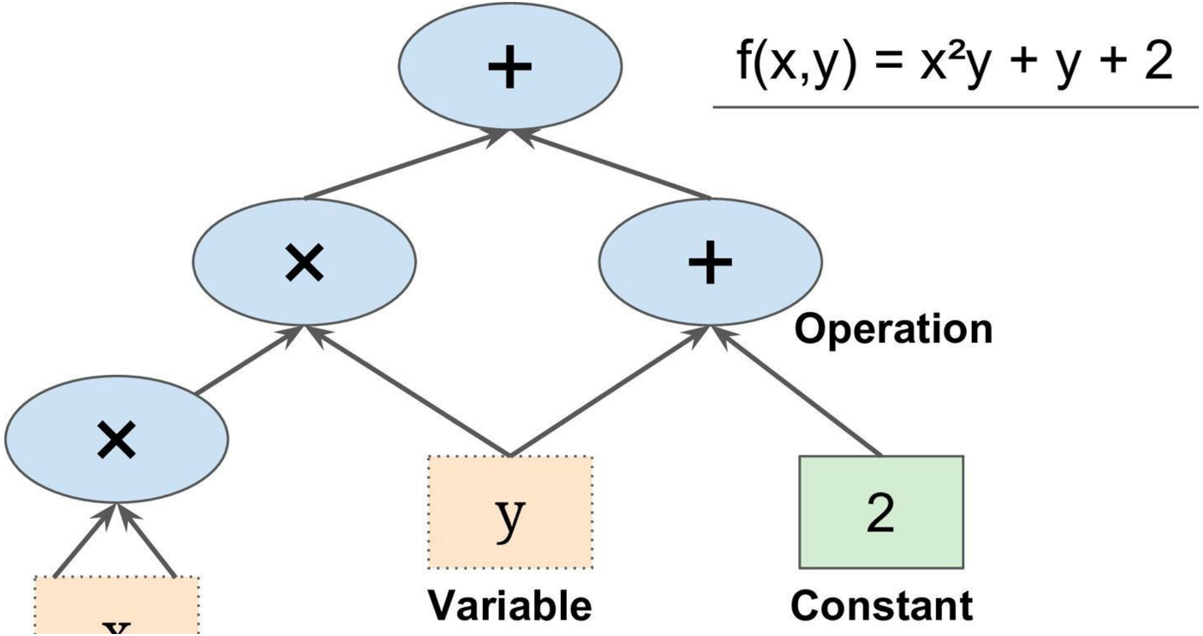
TensorFlow – Principal

- First define graph of computation



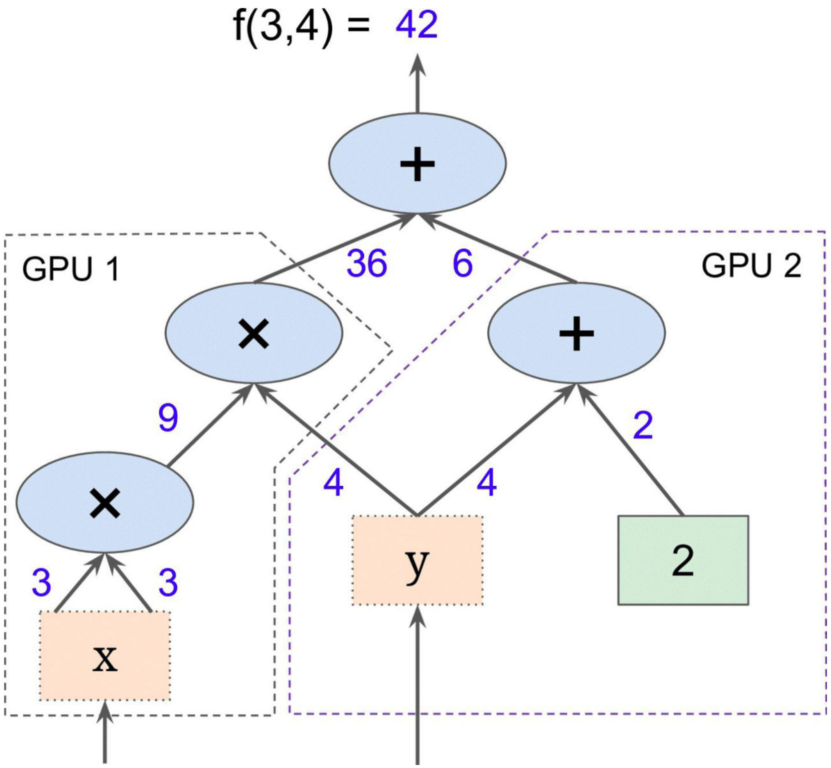
TensorFlow – Principal

- Then TensorFlow runs this graph efficiently using optimised C++ code



TensorFlow – Parallel Computation

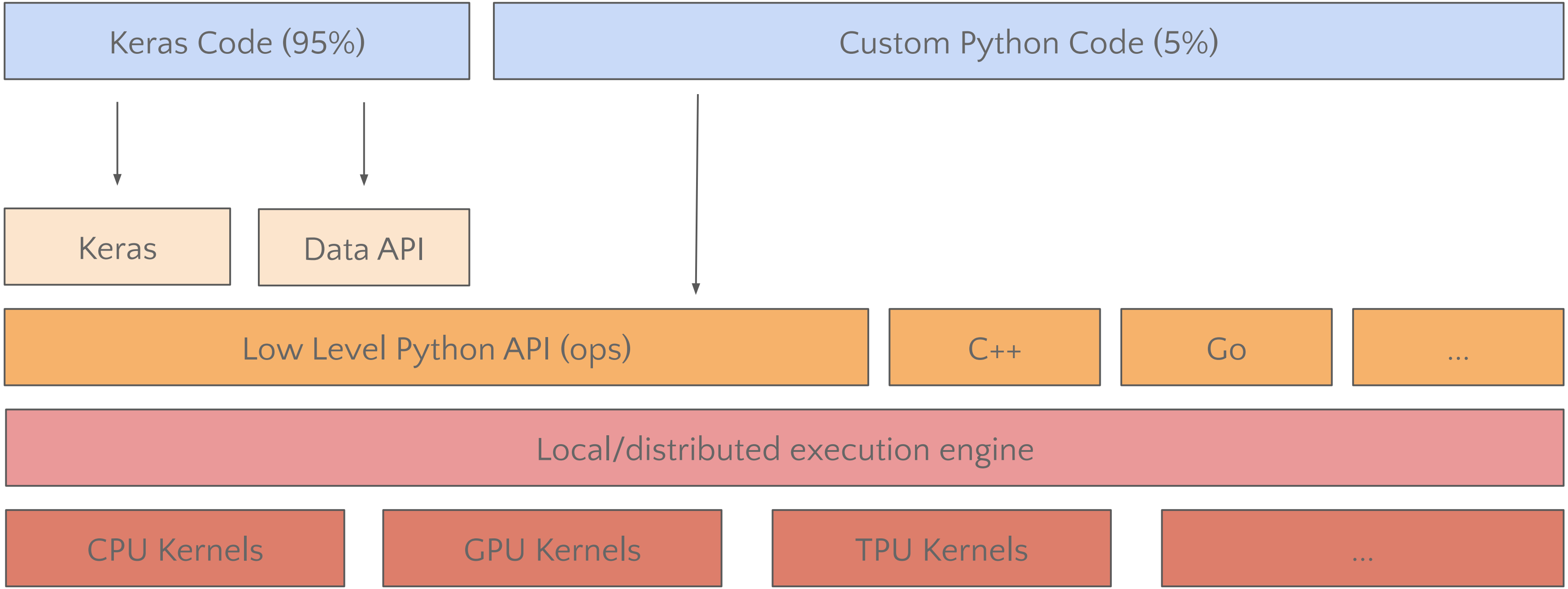
- Also the graph can be broken into multiple chunks
- Each chunk can run
  - Parallel across multiple
  - CPUs and
  - GPUs



Quick Tour of TensorFlow

- Similar to Numpy but with GPU support
- Supports Distributed Computing
  - Multiple servers

TensorFlow's architecture



Quick Tour of TensorFlow

- Includes JIT - Just-in-time compiler
  - Allows it to optimize computations for
    - Speed and
    - Memory

Quick Tour of TensorFlow

- Works by
  - Extracting computational graph from a Python function
  - Optimizing it (Pruning unused nodes)
  - Running the computational graph efficiently in parallel

Quick Tour of TensorFlow

- Computational Graphs can be exported to portable format
    - Train a TensorFlow model in one environment
        - Using Python on Linux
    - And run it in another environment
        - Using Java on Android

Quick Tour of TensorFlow

- At the lowest level, each TensorFlow operation (“op” for short)
  - Implemented using highly efficient C++ code
- Many ops have multiple implementations, called “Kernels”
- Each kernel is dedicated to specific device types
  - CPU – Central Processing Unit
  - GPU – Graphics Processing Unit
  - TPU – Tensor Processing Unit

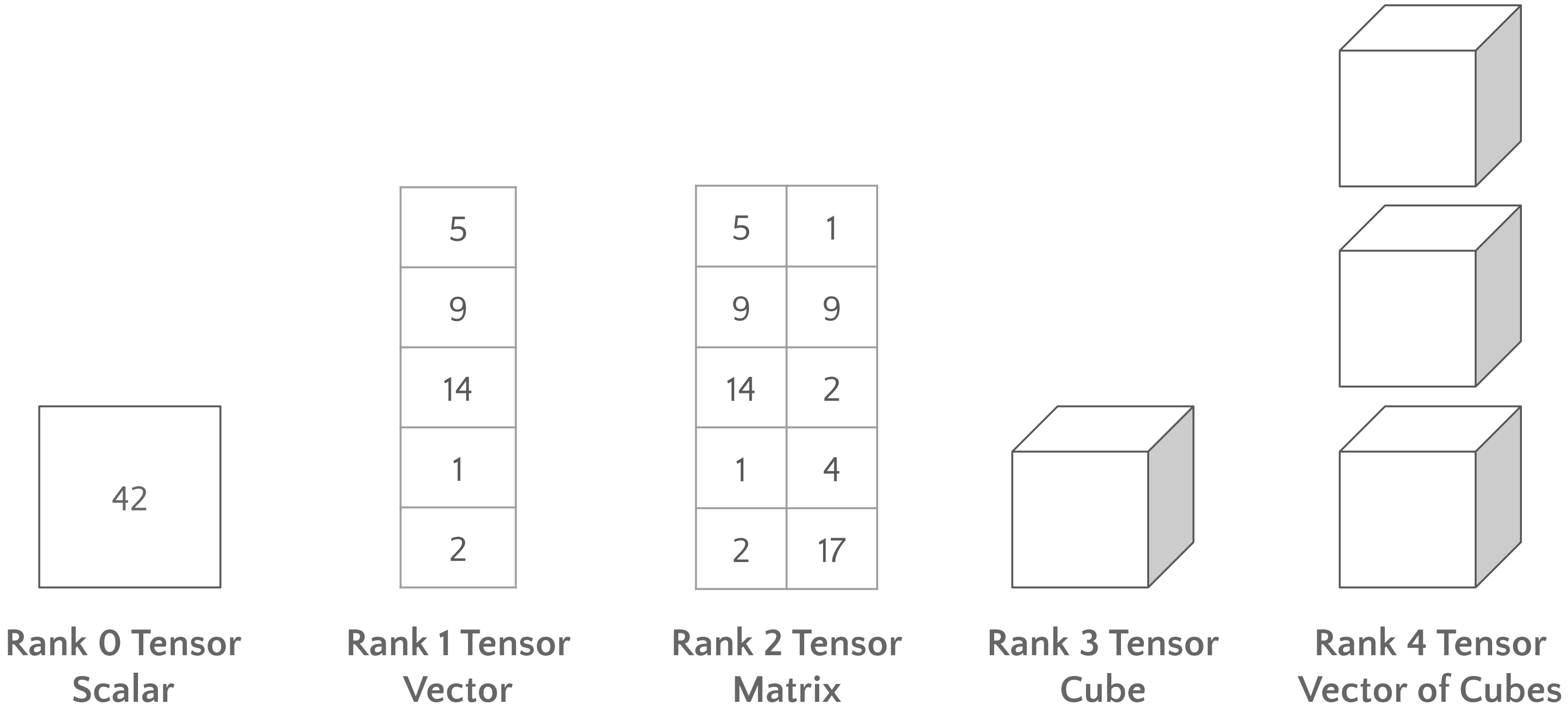
Using TensorFlow like NumPy

- Tensorflow API
    - Have “tensor” which flows from operation to operation
    - Hence the name “TensorFlow”

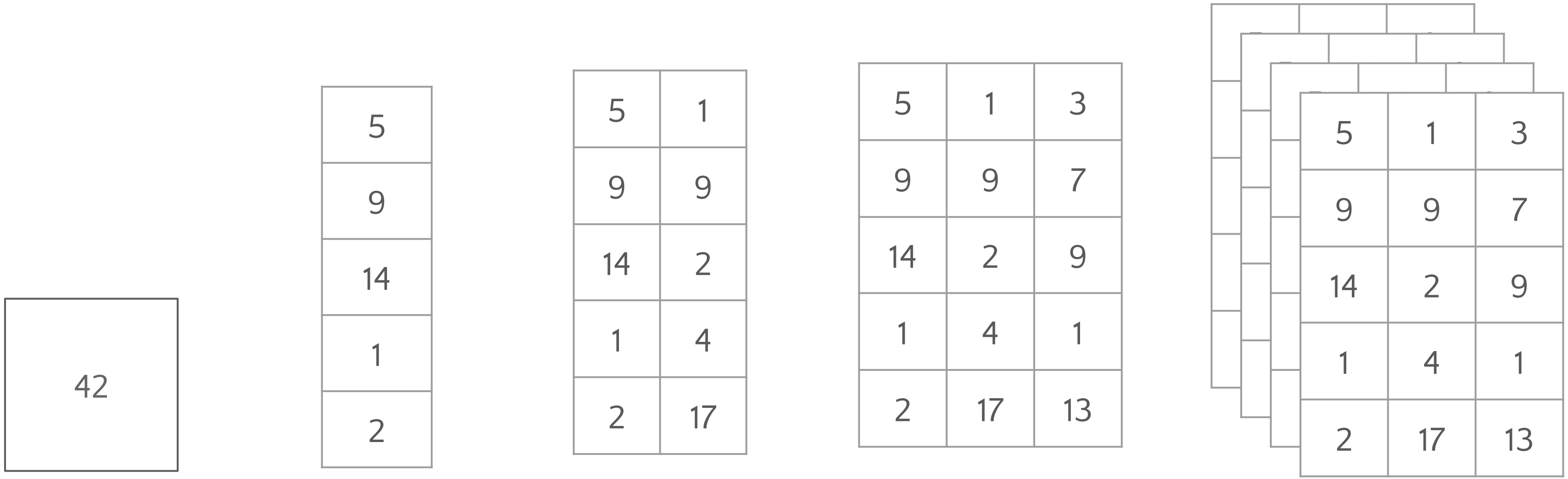
Using TensorFlow like NumPy

- What is Tensor?
  - Multidimensional array like a NumPy ndarray
  - But can hold scalar too like number 21

What is Tensor?



What is Tensor?



Rank 0 Tensor
Scalar

Rank 1 Tensor
Vector

Rank 2 Tensor
Matrix

Rank 2 Tensor
Matrix

Rank 3 Tensor
Cube

Tensor & Operations

- Create tensors with tf.constant()
- Scalar Tensor of integer

```
>>> import tensorflow as tf
>>> tf.constant(42)
```

Output-

```
<tf.Tensor: shape=(), dtype=int32, numpy=42>
```

Tensor & Operations

- Create tensors with tf.constant()
- Scalar Tensor of float

```
>>> tf.constant(42.1)
```

Output-

```
<tf.Tensor: shape=(), dtype=float32, numpy=42.1>
```

Tensor & Operations

- Create tensors with tf.constant()
- Tensor representing matrix with two rows and three columns of floats

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

Output-

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

1.0	2.0	3.0
4.0	5.0	6.0

2 x 3

Tensor & Operations

- Check shape of Tensor with shape

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.shape
```

Output-

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

1.0	2.0	3.0
4.0	5.0	6.0

2 x 3

Tensor & Operations

- Check data type of Tensor with dtype

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.dtype
```

Output-

```
tf.float32
```

1.0	2.0	3.0
4.0	5.0	6.0

2 x 3

Tensor & Operations

- Tensor Operation - Addition

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t + 5 # or tf.add(t,5)
```

1.0	2.0	3.0
4.0	5.0	6.0

2 x 3

Output-

```
<tf.Tensor: shape=(2, 3), dtype=float32,
numpy=array([[ 6.,  7.,  8.],[ 9., 10., 11.]],
dtype=float32)>
```

Tensor & Operations

- Tensor Operation - Square

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> tf.square(t)
```

1.0	2.0	3.0
4.0	5.0	6.0

2 x 3

Output-

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>
```

Tensor & Operations

- Tensor Operation – Multiply

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> tf.multiply(t, 5)
```

1.0	2.0	3.0
4.0	5.0	6.0

2 x 3

Output-

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 5., 10., 15.],
       [20., 25., 30.]], dtype=float32)>
```

Tensor & Operations

- Tensor Operation - Square root

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> tf.sqrt(t)
```

1.0	2.0	3.0
4.0	5.0	6.0

2 x 3

Output-

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1.        , 1.4142135, 1.7320508],
       [2.        , 2.236068 , 2.4494898]],
dtype=float32)>
```

Tensor & Operations

- Tensor Operation – Transpose

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> tf.transpose(t)
```

1.0	2.0	3.0
4.0	5.0	6.0

2 x 3

Output-

```
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[1., 4.],
       [2., 5.],
       [3., 6.]], dtype=float32)>
```

Tensor & Operations

- Tensor Operation – Matrix Multiplication

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> tf.matmul(t, tf.transpose(t))
# Or t @ tf.transpose(t)
```

1.0	2.0	3.0
4.0	5.0	6.0

2 x 3

Output-

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

Tensors & NumPy

Create Tensor from a NumPy array

- We can
  - Create a tensor from a NumPy array, and
  - Vice versa
- We can also apply
  - TensorFlow operations to NumPy arrays and
  - NumPy operations to tensors

Tensors & NumPy

Create Tensor from a NumPy array

```
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
```

Output –

```
<tf.Tensor: shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
```

Tensors & NumPy

Convert Tensor to NumPy

```
>>> t.numpy() # or np.array(t)
```

Output –

```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```

Tensors & NumPy

Convert Tensor to NumPy

```
>>> tf.square(a)
```

Output -

```
<tf.Tensor: shape=(3,), dtype=float64, numpy=array([ 4., 16., 25.])>
```

Tensors & NumPy

Convert Tensor to NumPy

```
>>> np.square(t)
```

Output –

```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```

Tensors & NumPy

- Important to note
  - NumPy uses 64-bit precision by default
  - Tensorflow uses 32-bit
- This is because
  - 32-bit precision is more than enough for neural networks
  - Runs faster and uses less RAM

Tensors & NumPy

Make sure to set dtype=tf.float32 while creating a Tensor from a Numpy array

Type Conversions

- We can not operate on incompatible types
- Type Conversions can significantly hurt performance
- Can easily get unnoticed when they are done automatically

42.0 + 21 --> Adding Float and Int

Type Conversions

- TensorFlow does not do type conversion automatically
- It just raises an exception when we execute an operation on tensors of incompatible types

`tf.constant(42.0) + tf.constant(21) -->` Adding Tensor of Float and Int

Type Conversions

- Handling exceptions – Wrap in try catch block

- Can't add float and integer

```
try:
    tf.constant(2.0) + tf.constant(40)
except tf.errors.InvalidArgumentError as ex:
    print(ex)
```

Type Conversions

- Handling exceptions - Wrap in try catch block
- Can't add 32-bit float and 64-bit float

```
>>> tf.constant(2.) + tf.constant(40., dtype=tf.float64)
```

Type Conversions

- Handling exceptions - Wrap in try catch block
- Can't add 32-bit float and 64-bit float

```
try:
    tf.constant(2.0) + tf.constant(40., dtype=tf.float64)
except tf.errors.InvalidArgumentError as ex:
    print(ex)
```

Using TensorFlow Like Numpy - Type Conversions

- Can't add 32-bit float and 64-bit float - Use tf.cast

```
>>> t2 = tf.constant(40., dtype=tf.float64)
>>> tf.constant(2.0) + tf.cast(t2, tf.float32)
```

Using TensorFlow Like Numpy – Variables

- Tensors we created so far are immutable – We can’t modify them
- So we can not use regular tensors to implement weight in neural network
- Since then they can not be tweaked by backpropagation
- Solution – tf.Variable

Using TensorFlow Like Numpy – Variables

- Create tf variable with two rows and three columns

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
>>> v

<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

Using TensorFlow Like Numpy – Variables

- Modify tf variable using assign()

```
>>> v.assign(2 * v)
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.]], dtype=float32)>
```

Using TensorFlow Like Numpy - Variables

- Update cells with index (0,1) to 42.0

```
>>> v[0, 1].assign(42.0)
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2., 42.,  6.],
       [ 8., 10., 12.]], dtype=float32)>
```

Using TensorFlow Like Numpy - Variables

- Update cell with scatter_nd_update()

```
>>> v.scatter_nd_update(indices=[[0, 0], [1, 2]], updates=[100., 200.])

<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[100.,  42.,   6.],
       [  8.,  10., 200.]], dtype=float32)>
```

Other Data Structures – Strings

- String Tensor – Byte String

```
>>> tf.constant(b"hello world")

<tf.Tensor: shape=(), dtype=string, numpy=b'hello world'>
```

Other Data Structures – Strings

- String Tensor – Unicode strings get encoded to utf-8 automatically

```
>>> tf.constant("café")

<tf.Tensor: shape=(), dtype=string, numpy=b'caf\xc3\xa9'>
```

Other Data Structures – Strings

- Represent Unicode strings using tensor of type tf.int32

```
>>> u = tf.constant([ord(c) for c in "café"])
>>> u

<tf.Tensor: shape=(4,), dtype=int32, numpy=array([ 99,  97, 102, 233],
dtype=int32)>
```

String Arrays

- Tensor of string arrays

```
>>> p = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
>>> p

<tf.Tensor: shape=(4,), dtype=string, numpy=
array([b'Caf\xc3\xa9', b'Coffee', b'caff\xc3\xa8',
       b'\xe5\x92\x96\xe5\x95\xa1'], dtype=object)>
```

Ragged Tensors

- List of lists, with each being of variable length

```
>>> speech = tf.ragged.constant(
  [['All', 'the', 'world', 'is', 'a', 'stage'],
  ['And', 'all', 'the', 'men', 'and', 'women', 'merely', 'players'],
  ['They', 'have', 'their', 'exits', 'and', 'their', 'entrances']])
>>> speech
```

Output –

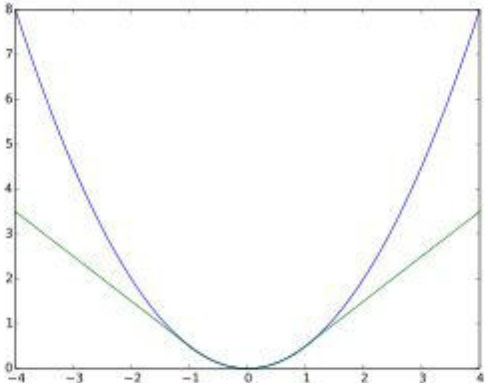
```
<tf.RaggedTensor [[b'All', b'the', b'world', b'is', b'a', b'stage'],
[b'And', b'all', b'the', b'men', b'and', b'women', b'merely',
b'players'], [b'They', b'have', b'their', b'exits', b'and', b'their',
b'entrances']]>
```

Custom Loss Function

Custom Loss Function

- What is Huber Loss function?
  Huber loss is less sensitive to outliers in data than mean squared error. The Huber loss is not currently part of the official Keras API, but it is available in tf.keras (just use an instance of the keras.losses.Huber class). Below is the formula for Huber Loss and a plot of Huber loss (green) vs squared error loss (blue)

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$



Custom Loss Function

- The Huber Loss function that takes the labels and predictions as arguments, and use TensorFlow operations to compute every instance's loss.

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)
```

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

tf.where return the elements where condition is True (multiplexing x and y). You can find example of the same towards the end of the notebook.

Custom Loss Function

- Now you can use this loss when you compile the Keras model, then train your model:

```
>>> model.compile(loss=huber_fn, optimizer="nadam")
>>> model.fit(X_train, y_train, [...])
```

Saving and Loading Models That Contain Custom Components

Saving and Loading Models That Contain Custom Components

- When you load a model containing custom objects, you need to map the names to the objects

```
>>> model = keras.models.load_model("my_model_with_a_custom_loss.h5",
    custom_objects={"huber_fn": huber_fn})
```

Saving and Loading Models That Contain Custom Components

- Although, you will have to specify the threshold value when loading the model

```
>>> model =
keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
custom_objects={"huber_fn": create_huber(2.0)})
```

- This can be solved this by creating a subclass of the keras.losses.Loss class, and then implementing its get_config() method

Saving and Loading Models That Contain Custom Components

```python
class HuberLoss(keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)
    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < self.threshold
        squared_loss = tf.square(error) / 2
        linear_loss = self.threshold * tf.abs(error) - self.threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

Saving and Loading Models That Contain Custom Components

Let's walk through this code:

- The constructor accepts **kwargs and passes them to the parent constructor, which handles standard hyperparameters
- The call() method takes the labels and predictions, computes all the instance losses, and returns them
- The get_config() method returns a dictionary mapping each hyperparameter name to its value. It first calls the parent class's get_config() method, then adds the new hyperparameters to this dictionary

Saving and Loading Models That Contain Custom Components

- You can then use any instance of this class when you compile the model:

```
>>> model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

- When you save the model, the threshold will be saved along with it; and when you load the model, you just need to map the class name to the class itself:

```
>>> model = keras.models.load_model
("my_model_with_a_custom_loss_class.h5", custom_objects={"HuberLoss":
HuberLoss})
```

Custom Activation Functions, Initializers, Regularizers, and Constraints

Customizability in Keras

Most Keras functionalities, such as

- Losses,
- Regularizers,
- Constraints,
- Initializers,
- Metrics,
- Activation functions,
- Layers,
- and even full models, can be customized in very much the same way.

Example of Custom Activation Function

Here is an example of a custom activation function:

```
def my_softplus(z):
    return tf.math.log(tf.exp(z) + 1.0)
```

This is equivalent to keras.activations.softplus() or tf.nn.softplus(),

f(x) = ln(1+exp x)

Example of Custom Glorot initializer

Here is an example of a custom Glorot initializer:

```
def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)
```

This is equivalent to keras.initializers.glorot_normal(). Equation for Glorot initializer:

$$\sigma = \sqrt{\frac{2}{a+b}}$$

where a is the number of input units in the weight tensor, and b b is the number of output units in the weight tensor.

Example of Custom Regularizer

Here is an example of a custom Regularizer:

```
def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))
```

This is equivalent to keras.regularizers.l1(0.01).

Example of Custom Constraint

Here is an example of a custom Constraint:

```
def my_positive_weights(weights):
    return tf.where(weights < 0., tf.zeros_like(weights), weights)
```

This is equivalent to keras.constraints.nonneg() or tf.nn.relu(), and this ensures that weights are all positive.

Using a Custom Function

These custom functions can then be used normally; for example:

```
layer = keras.layers.Dense(30, activation=my_softplus,
    kernel_initializer=my_glorot_initializer,
    kernel_regularizer=my_l1_regularizer,
    kernel_constraint=my_positive_weights)
```

In this example, the activation function, initializer, regularizer, and the constraint are all custom functions. Here, `kernel_constrain` is a custom constraint that ensures weights are all positive.

Hyperparameters in Custom Function

If a function has hyperparameters that need to be saved along with the model, then

- You will want to subclass the appropriate class, such as:
  - `keras.regularizers.Regularizer`,
  - `keras.constraints.Constraint`,
  - `keras.initializers.Initializer`, or
  - `keras.layers.Layer` (for any layer, including activation functions).

Example of Custom Function with Hyperparameters

Here is a simple class for ℓ1 regularization that saves its factor hyperparameter:

```
class MyL1Regularizer(keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor
    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))
    def get_config(self):
        return {"factor": self.factor}
```

Custom Metrics

Difference between Loss and Metrics

Losses

- Must be differentiable (at least where they are evaluated),
- Their gradients should not be 0 everywhere,
- It's OK if they are not easily interpretable by humans,
- E.g: cross entropy.

Metrics

- Must be more easily interpretable,
- They can be non-differentiable,
- They can have 0 gradients everywhere,
- E.g: accuracy.

Creating Custom Loss/Metrics Function

In most cases, defining a custom metric function is exactly the same as defining a custom loss function.

```
>>> model.compile(loss="mse", optimizer="nadam",
metrics=[create_huber(2.0)])
```

For each batch during training, Keras will compute this metric and keep track of its mean since the beginning of the epoch.

Streaming Metric

A streaming metric or a stateful metric is gradually updated, batch after batch. E.g:

```
precision = keras.metrics.Precision()
precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: id=581729, shape=(), dtype=float32, numpy=0.8>
precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])
<tf.Tensor: id=581780, shape=(), dtype=float32, numpy=0.5>
```

This code returns a precision of 80% after the first batch; then after the second batch, it returns 50% (which is the overall precision so far, not the second batch's precision).

Creating a Streaming Metric

- If you need to create a streaming metric, create a subclass of the `keras.metrics.Metric` class.

Example of a Streaming Metric

```
class HuberMetric(keras.metrics.Metric):
   def __init__(self, threshold=1.0, **kwargs):
         super().__init__(**kwargs)
         self.threshold = threshold
         self.huber_fn = create_huber(threshold)
         self.total = self.add_weight("total", initializer="zeros")
         self.count = self.add_weight("count", initializer="zeros")
```

Example of a Streaming Metric (contd.)

```
def update_state(self, y_true, y_pred, sample_weight=None):
      metric = self.huber_fn(y_true, y_pred)
      self.total.assign_add(tf.reduce_sum(metric))
      self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))
def result(self):
      return self.total / self.count
def get_config(self):
      base_config = super().get_config()
      return {**base_config, "threshold": self.threshold}
```

Example of a Streaming Metric (contd.)

Now let's walk through the code:

- The add_weight() method to create the variables needed to keep track of the metric's state over multiple batches—in this case, the sum of all Huber losses (total) and the number of instances seen so far (count).
- The update_state() method updates the variables, given the labels and predictions for one batch (and sample weights).
- The result() method computes and returns the final result, in this case the mean Huber metric over all instances.

Example of a Streaming Metric (contd.)

Now let's walk through the code:

- The get_config() method to ensure the threshold gets saved along with the model.
- The default implementation of the reset_states() method resets all variables to 0.0 (but you can override it if needed).

Custom Layers

Why we need a Custom Layer

A Custom Layer is needed:

- To build an architecture that contains an exotic layer,
- To build a very repetitive architecture,
  - Containing identical blocks of layers repeated many times.

Layers with No Weights

Some layers have no weights. If you want to create a custom layer without any weights, the simplest option is to write a function and wrap it in a keras.layers.Lambda layer.

```
>>> exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

Layers with No Weights (contd.)

- This custom layer can then be used like any other layer, using the Sequential API, the Functional API, or the Subclassing API.
- You can also use it as an activation function (or you could use activation=tf.exp, activation=keras.activations.exponential, or simply activation="exponential").

Layers with Weights

To build a custom stateful layer (i.e., a layer with weights), you need to create a subclass of the keras.layers.Layer class. The following class implements a simplified version of the Dense layer:

```
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)
```

Layers with Weights (contd.)

```
    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
        name="kernel", shape=[batch_input_shape[-1], self.units],
initializer="glorot_normal")
        self.bias = self.add_weight(name="bias", shape=[self.units],
    initializer="zeros")
        super().build(batch_input_shape)
    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)
    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] +
    [self.units])
```

Layers with Weights (contd.)

```
def get_config(self):
    base_config = super().get_config()
    return {**base_config, "units": self.units,
"activation":keras.activations.serialize(self.activation)}
```

Layers with Weights (contd.)

Let's walk through this code:

- The constructor takes all the hyperparameters as argument
- The build() method's role is to create the layer's variables by calling the
- add_weight() method for each weight
- The call() method performs the desired operations
- The compute_output_shape() method simply returns the shape of this layer's outputs
- The get_config() method is just like in the previous custom classes

Custom Layers with Varied Behaviors in Training/Testing

If your layer needs to have a different behavior during training and during testing (e.g., if it uses Dropout or BatchNormalization layers), then

- You must add a training argument to the call() method,
- And use this argument to decide what to do.

Custom Models

How to create a Custom Model

Creating a Custom Model is simple:

- Subclass the keras.Model class,
- Create layers and variables in the constructor,
- And implement the call() method to do whatever you want the model to do.

How to create a Custom Model

The Model class is a subclass of the Layer class, so models can be defined and used exactly like layers. But a model has some extra functionalities, including

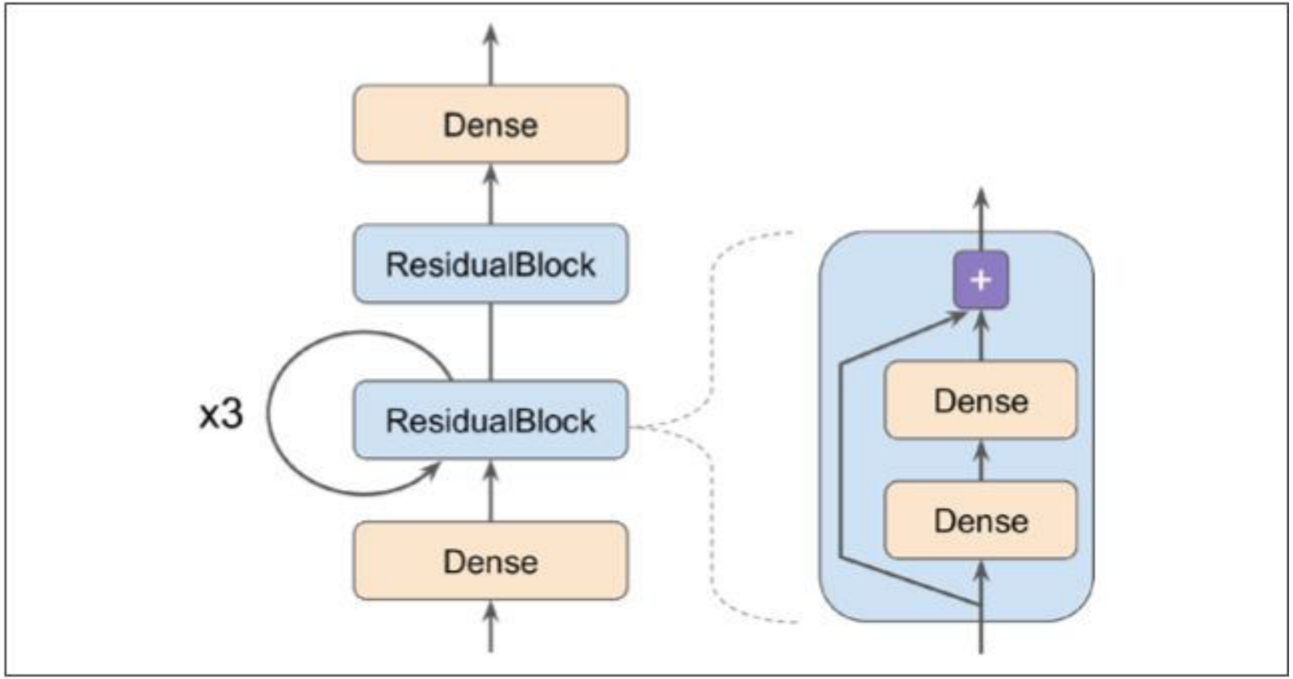
- compile(), fit(), evaluate(), and predict() methods (and a few variants), plus the get_layers() method
- and the save() method (and support for keras.models.load_model() and keras.models.clone_model()).

Why use Custom Layers instead of Custom Models

If models provide more functionality than layers, why not just define every layer as a model?

- Technically you could, but it is usually cleaner to distinguish the internal components of your model (i.e., layers or reusable blocks of layers) from the model itself (i.e., the object you will train).
- The former should subclass the Layer class, while the latter should subclass the Model class.

Custom Model Creation (Notebook)



Losses and Metrics Based on Model Internal

Losses and Metrics Based on Model Internals

- The custom losses and metrics we defined earlier were all based on the labels and the predictions (and optionally sample weights).
- There will be times when you want to define losses based on other parts of your model,
    - such as the weights or activations of its hidden layers.
- This may be useful for regularization purposes or to monitor some internal aspect of your model.

Custom Loss Based on Model Internals

To define a custom loss based on model internals,

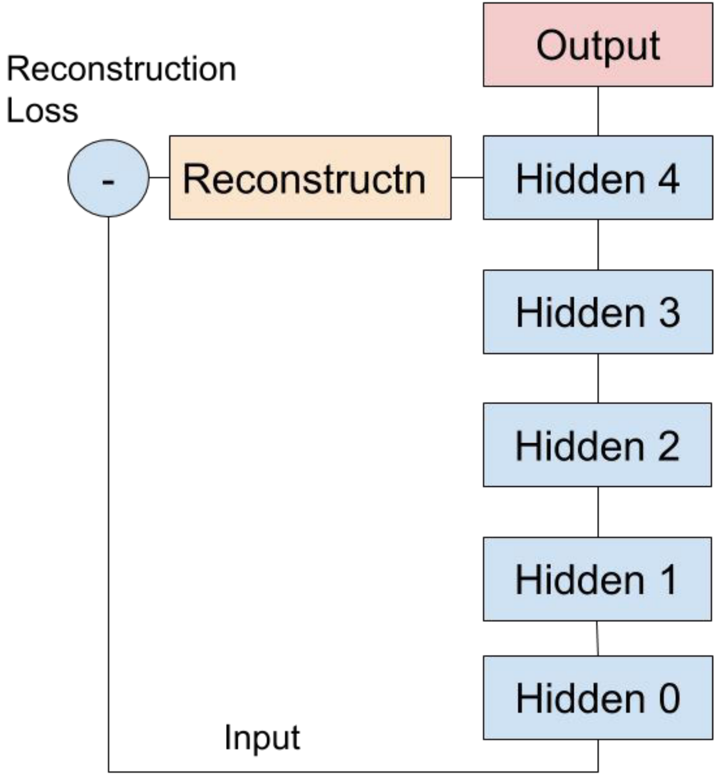
- Compute it based on any part of the model you want,
- Then pass the result to the add_loss() method.

Custom Metric Based on Model Internals

You can add a custom metric based on model internals by

- Computing it in any way you want, as long as the result is the output of a metric object,
- For example, you can create a keras.metrics.Mean object in the constructor,
- Then call it in the call() method,
- Passing it the recon_loss,
- And finally add it to the model by calling the model's add_metric() method.

Reconstruction Loss (Custom Loss on Custom Model in Notebook)



Computing Gradients Using Autodiff

Calculating Gradient using Autodiff in Tensorflow

Using autodiff in Tensorflow is very simple. Let's see an example:

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2

>>> w1, w2 = tf.Variable(5.), tf.Variable(3.)
>>> with tf.GradientTape() as tape:
...     z = f(w1, w2)
...     gradients = tape.gradient(z, [w1, w2])
```

Output:

```
>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
<tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]
```

Calculating Gradient using Autodiff in Tensorflow

In the previous example:

- We first define two variables w1 and w2,
- We create a tf.GradientTape context that will automatically record every operation that involves a variable,
- Finally we ask this tape to compute the gradients of the result z with regard to both variables [w1, w2].

Calculating Gradient more than once

The tape is automatically erased immediately after you call its gradient() method, so you will get an exception if you try to call gradient() twice.

```
>>> with tf.GradientTape() as tape:
...      z = f(w1, w2)

>>> dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
>>> dz_dw2 = tape.gradient(z, w2) # RuntimeError!
```

Calculating Gradient more than once

If you need to call gradient() more than once, you must make the tape persistent and delete it each time you are done with it to free resources.

```
>>> with tf.GradientTape(persistent=True) as tape:
...       z = f(w1, w2)

>>> dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
>>> dz_dw2 = tape.gradient(z, w2) # => tensor 10.0
>>> del tape
```

Calculating Gradient for constants

By default, the tape will only track operations involving variables, so if you try to compute the gradient of z with regard to anything other than a variable, the result will be None.

```
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)
gradients = tape.gradient(z, [c1, c2]) # returns [None, None]
```

Force the Tape to Watch any Tensor

You can force the tape to watch any tensors you like, to record every operation that involves them:

```
>>> with tf.GradientTape() as tape:
...     tape.watch(c1)
...     tape.watch(c2)
>>> z = f(c1, c2)
>>> gradients = tape.gradient(z, [c1, c2])
```

Force the Tape to Watch any Tensor (contd.)

This can be useful in some cases,

- If you want to implement a regularization loss that penalizes activations that vary a lot when the inputs vary little the loss will be based on the gradient of the activations with regard to the inputs.
- Since the inputs are not variables, you would need to tell the tape to watch them.

Stop Gradients from Backpropagating

- To stop gradients from backpropagating, you must use the tf.stop_gradient() function.
- The function returns its inputs during the forward pass (like tf.identity()), but it does not let gradients through during backpropagation (it acts like a constant).

Stop Gradients from Backpropagating (contd.)

```
>>> def f(w1, w2):
...       return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)

>>> with tf.GradientTape() as tape:
...       z = f(w1, w2) # same result as without stop_gradient()

>>> gradients = tape.gradient(z, [w1, w2]) # => returns [tensor 30., None]
```

Numerical Issues with Computing Gradients

- Due to floating-point precision errors, autodiff ends up computing infinity divided by infinity (which returns NaN),
- We can analytically find that the derivative of the function and see if it is numerically stable,
- Next, we can tell TensorFlow to use this stable function when computing the gradients of the function by decorating it with @tf.custom_gradient,
- And making it return both its normal output and the function that computes the derivatives.

Custom Training Loops

Custom Training Loops

- In some rare cases, the fit() method may not be flexible enough since the fit() method only uses one optimizer implementing this paper requires writing your own custom loop.
- However, writing a custom training loop will make your code longer, more error-prone, and harder to maintain

Example of Custom Training Loops

- First, let's build a simple model

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu",
kernel_initializer="he_normal", kernel_regularizer=l2_reg),
    keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

Example of Custom Training Loops (contd.)

- Next, let's create a tiny function that will randomly sample a batch of instances from the training set

```
def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]
```

Example of Custom Training Loops (contd.)

- Let’s also define a function that will display the training status, including the number of steps, the total number of steps, the mean loss since the start of the epoch, and other metrics:

```
def print_status_bar(iteration, total, loss, metrics=None):
    metrics = " - ".join(["{}: {:.4f}".format(m.name, m.result())
                          for m in [loss] + (metrics or [])])
    end = "" if iteration < total else "\n"
    print("\r{}/{} - ".format(iteration, total) + metrics, end=end)
```

Example of Custom Training Loops (contd.)

- Now let's define some hyperparameters and choose the optimizer, the loss function, and the metrics:

```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]
```

Example of Custom Training Loops (contd.)

- Finally, we build the custom loops:

```
for epoch in range(1, n_epochs + 1):
    print("Epoch {}/{}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
```

Example of Custom Training Loops (contd.)

```
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        mean_loss(loss)
        for metric in metrics:
            metric(y_batch, y_pred)
        print_status_bar(step * batch_size, len(y_train), mean_loss, metrics)
    print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
    for metric in [mean_loss] + metrics:
        metric.reset_states()
```

Example of Custom Training Loops (contd.)

Now, let’s walk through this code:

- We create two nested loops for the epochs, and for the batches within an epoch.
- Then we sample a random batch from the training set.
- Inside the tf.GradientTape() block, we make a prediction for one batch, and we compute the loss, compute the mean over the batch using tf.reduce_mean(). The regularization losses are already reduced to a single scalar each, so we just need to sum them using tf.add_n().

Example of Custom Training Loops (contd.)

Now, let's walk through this code:

- Next, we ask the tape to compute the gradient of the loss with regard to each trainable variable.
- Then we update the mean loss and the metrics (over the current epoch), and we display the status bar.
- At the end of each epoch, we display the status bar again to make it look complete.

Tensorflow Execution Modes

TensorFlow Execution Modes – Eager Mode

- Default mode of execution – eager mode
  - Execution happens immediately like Python and Numpy

```
def square(x):
    return x ** 2
```

Define the Logic

```
>> square(tf.constant(4))

<tf.Tensor: shape=(),
dtype=int32, numpy=16>
```

Execute it Immediately

TensorFlow Execution Modes - Eager Mode

Question - Limitation of Eager Mode Execution??

TensorFlow Execution Modes - Eager Mode

Question - Limitation of Eager Mode Execution??

Answer - It may be slow

TensorFlow Execution Modes – Eager Mode

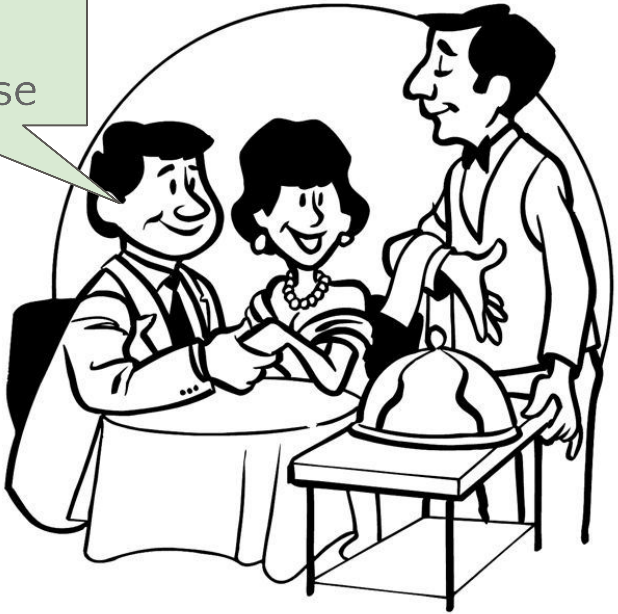
Question – Limitation of Eager Mode Execution??

Answer – It may be slow

Why? Lets understand it with an example

TensorFlow Execution Modes – Eager Mode

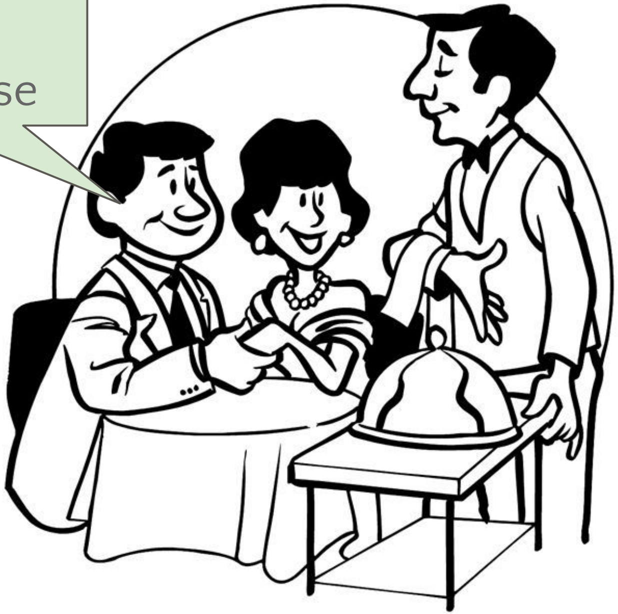
Cheese burger,
soup and
a Plate of
Noodles please



Waiter takes the order

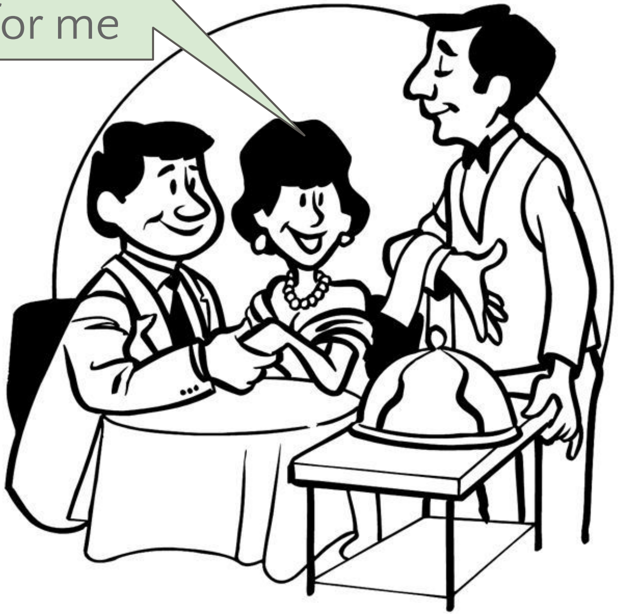
TensorFlow Execution Modes – Eager Mode

Cheese burger,
soup and
a Plate of
Noodles please



Waiter goes and tells to Chef

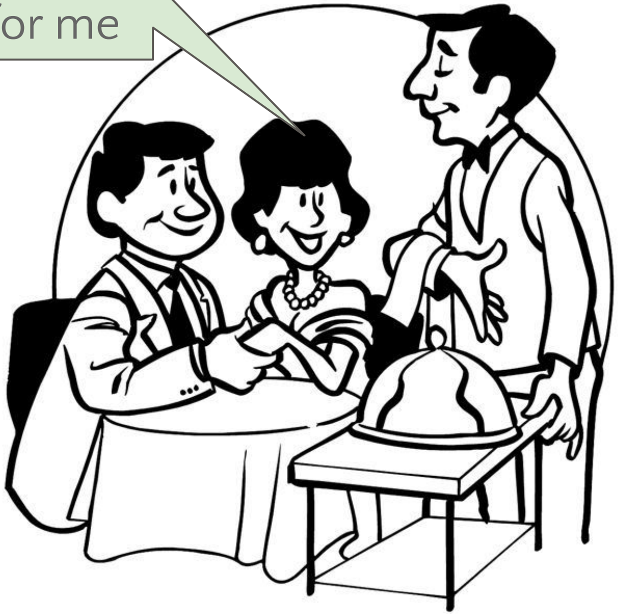
TensorFlow Execution Modes – Eager Mode



Waiter takes another order

TensorFlow Execution Modes – Eager Mode

Soup and
A Plate of
Noodles for me



Waiter goes and tells
to Chef

TensorFlow Execution Modes – Eager Mode

One cheese burger
Two soups
Two plates of Noodles

Waiter takes another order

TensorFlow Execution Modes – Eager Mode

Question – Do you see any problem in this approach?

TensorFlow Execution Modes – Eager Mode

Question – Do you see any problem in this approach?

Answer – Too much to and fro

TensorFlow Execution Modes – Eager Mode

Question – Do you see any problem in this approach?

Answer – Too much to and fro

This is how eager execution works

TensorFlow Execution Modes – Eager Mode

Question – Do you see any problem in this approach?

Answer – Too much to and fro

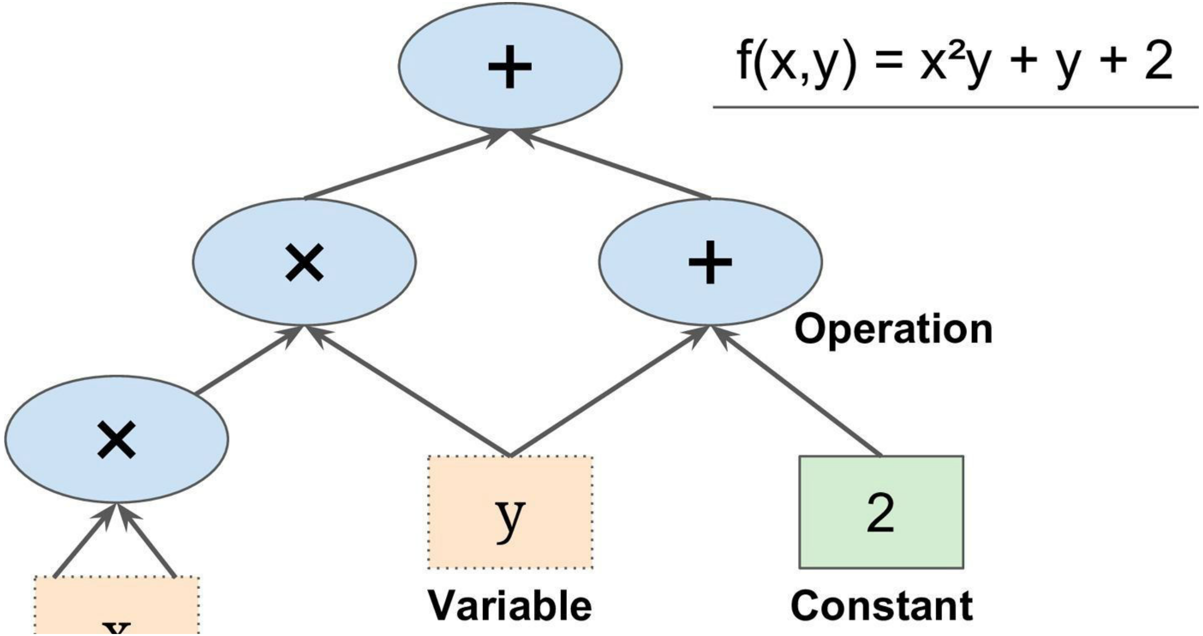
What would be better way?

TensorFlow Execution Modes – Graph Mode

The chef is able to optimize because of clubbing multiple order together

TensorFlow Execution Modes – Graph Mode

- Let's see how to enable graph mode execution in TensorFlow



TensorFlow Execution Modes – Graph Mode

- Convert Python function to TensorFlow functions

```
def square(x):
    return x ** 2
```

Python Function

TensorFlow Execution Modes – Graph Mode

- Convert Python function to TensorFlow function with @tf decorator

```
def square(x):
    return x ** 2
```

Python Function

Convert

```
@tf.function
def square(x):
    return x ** 2
```

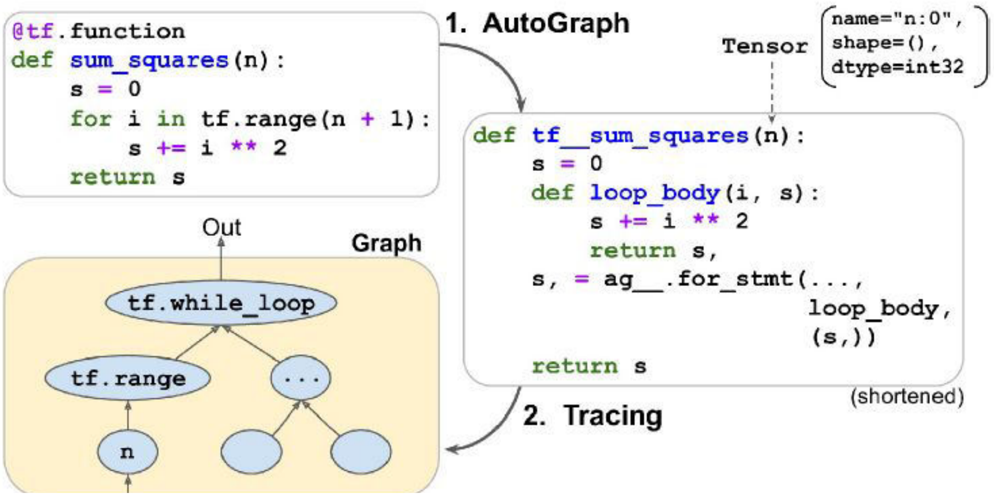
TF Function

TensorFlow Functions – AutoGraph & Tracing

- So how does TensorFlow generate graphs from functions?

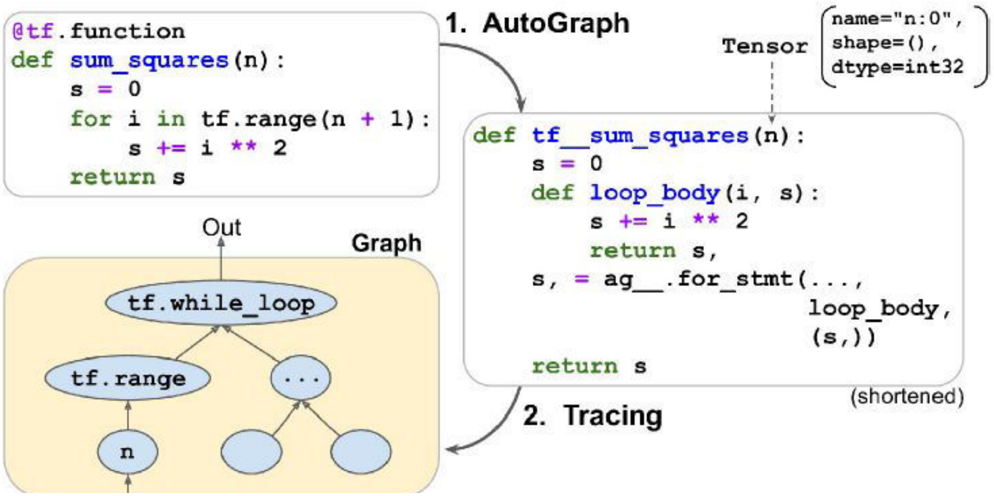
TensorFlow Functions – AutoGraph & Tracing

- So how does TensorFlow generate graphs from functions?
    - Using Autograph & Tracing



TensorFlow Functions – AutoGraph & Tracing

- Autograph
  - Analyzes the Python function and convert it to TensorFlow function



TensorFlow Functions – AutoGraph & Tracing

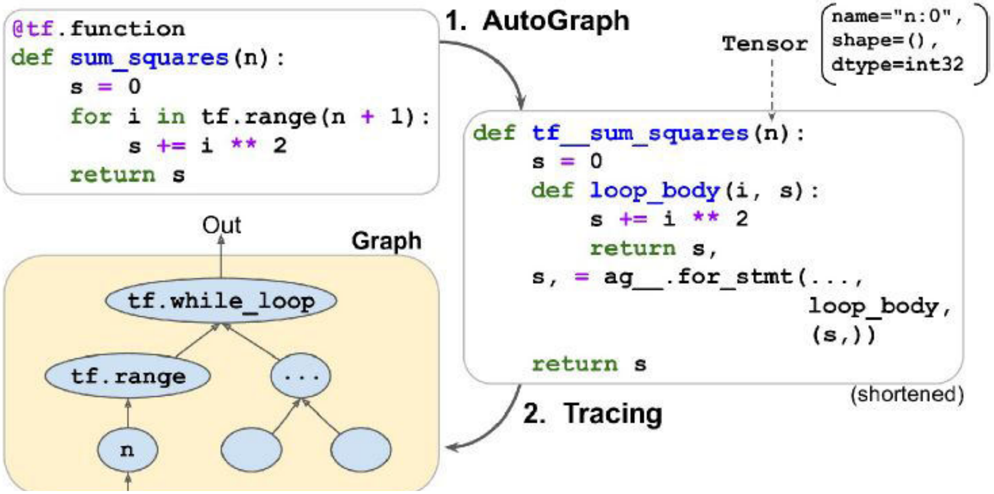
Get the code generated by TensorFlow

```
>>> print(tf.autograph.to_code(square.python_function))
```

```
def tf__square(x):
  do_return = False
  retval_ = ag__.UndefinedReturnValue()
  with ag__.FunctionScope('square', 'fscope',
ag__.ConversionOptions(recursive=True, user_requested=True,
optional_features=(), internal_convert_user_code=True)) as fscope:
    do_return = True
    retval_ = fscope.mark_return_value(x ** 2)
  do_return,
  return ag__.retval(retval_)
```

TensorFlow Functions – AutoGraph & Tracing

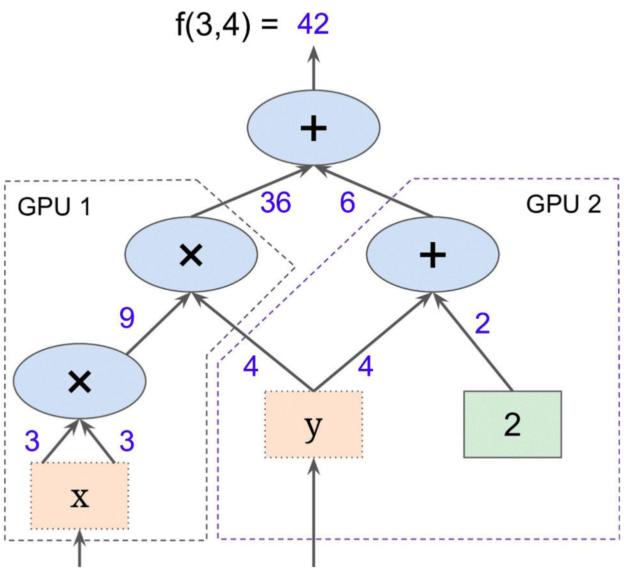
- Tracing
  - Generates Computational Graph
  - Nodes represent operations and arrow represent tensors



So what are the Advantages of Graph Execution

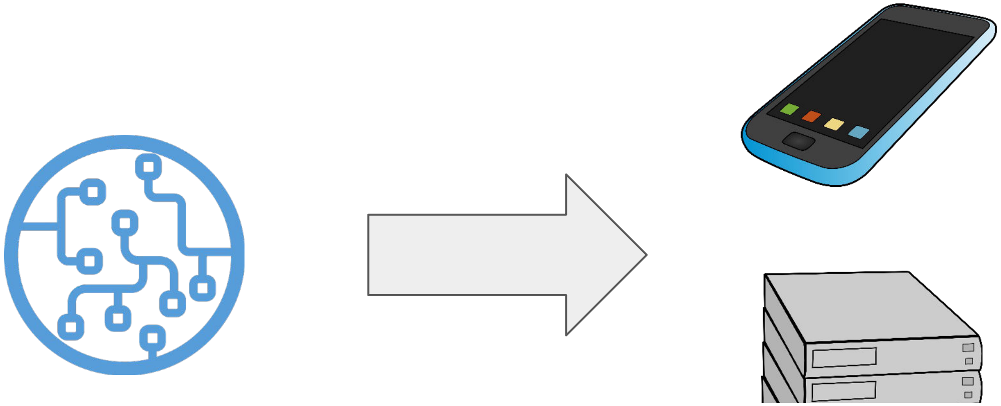
Advantages of Graph Mode

- Faster execution – Tensorflow takes advantage of underlying hardware
  - By placing set of operations in GPU
  - Grouping operations together



Advantages of Graph Mode

- Portability, For example
  - Train your model in Python
  - Export the Graph
  - And run in your mobile device using Java



Advantages of Graph Mode

- With Graph we can use feature of Tensorflow, called
  - XLA – Accelerated Linear Algebra
- XLA analyzes the graph and improves performance in terms of
  - Execution speed and
  - Memory

https://www.tensorflow.org/xla

TensorFlow Functions - AutoGraph & Tracing

- If we pass numerical values to a TF function
  - Then new graph will be generated every time

```
@tf.function
def square(x):
    return x ** 2
```

```
>> square(10) # New graph will be generated
>> square(20) # New graph will be generated
```

TensorFlow Functions - AutoGraph & Tracing

- But if we pass tensors to TF function
  - Then the new graph will be generated only if input tensors shape and datatype changes

```
@tf.function
def square(x):
    return x ** 2
```

```
>> square(tf.constant(10)) # New graph will be
generated
>> square(tf.constant(20)) # Same graph will be
reused
>> square(tf.constant([10, 20])) # New graph will be
generated
```

TensorFlow Functions – AutoGraph & Tracing

- Tracing – Get the graph operation

```
# TensorFlow computational graph is wrapped in concrete function
get_concrete_function
>>> concrete_function = square.get_concrete_function(tf.constant(2.0))
>>> ops = concrete_function.graph.get_operations()
>>> ops


[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'pow/y' type=Const>,
 <tf.Operation 'pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

TensorFlow Functions - Rules

- Rule - 1
  - Use TensorFlow Constructs as much as possible
  - Like tf.reduce_sum() instead of np.sum
  - tf.sort() instead of built in sorted()

TensorFlow Functions – Rules

- Rule – 2
    - The source code of Python function should be available to TensorFlow
    - Just having compiled *.pyc Python files does not help

```
def square(x):
    return x ** 2
```

Python Code

TensorFlow should have access to this Code

TensorFlow Functions - Rules

- Rule - 3
  - TensorFlow will only capture for loops that iterate over a tensor or a dataset
  - Use for i in tf.range(x)
    - Do not use for i in range(x)

TensorFlow Functions – Rules

- Rule – 4
  - Prefer vectorized implementation over for loop
  - Example – Vectorized sum

Check notebook for vectorized implementation

Tensorflow Control Flow

TensorFlow Control Flow

Example 1 – Direct modification works

```
if tf.greater(a,b):
    x = a # Direct modification works
else:
    x = b
```

TensorFlow Control Flow

Example 2 – Undefined values will cause error

```
if tf.greater(a,b):
    x = tf.constant(....) # First time initialization here
else:
    ......
    # No initialization of x
```

This will throw error

TensorFlow Control Flow

Example 2 – Undefined values will cause error

```
x = tf.constant(....) # Define default value of x
if tf.greater(a,b):
    x = tf.constant(....) # First time initialization here
else:
    ......
    # No initialization of x
```

Solution?? – Define default value of x

TensorFlow Control Flow

Example 3 – Return/break may also lead to undef

```
def (a,b):
    if tf.greater(a,b):
        return tf.constant(....)

f(a,b) # if a <= b would return None
```

Questions?