TensorFlow

By: Shuhao Lai

Date: 8/18/2019

Keras Basics

tensorflow.keras has all the functionality of keras but also allows TensorFlow specific functions as well. tf.keras can run any Keras-compatible code (assuming versions of Keras are the same).

Setup:

```
from __future__ import absolute_import, division, print_function, unicode_literals

import tensorflow as tf
from tensorflow.keras import layers
```

In Keras, you assemble layers to build models.

To build a simple, fully-connected network:

```
model = tf.keras.Sequential() # A model with stack of layers.
# Adds a densely-connected/fully-connected layer with 64 units to the model:
model.add(layers.Dense(64, activation='relu'))
# Add another:
model.add(layers.Dense(64, activation='relu'))
# Add a softmax layer with 10 output units:
model.add(layers.Dense(10, activation='softmax'))
```

Visit tf.keras.layers for more layer options. Common constructor parameters for the layers:

- activation: Specified by the name of a built-in function or as a callable object. By default, no activation is applied.
- kernel_initializer and bias_initializer: This parameter is a name or a callable object. This defaults to the "Glorot uniform" initializer.
- kernel_regularizer and bias_regularizer: options include L1 or L2 regularization. By default, no regularization is applied.

```
# Create a sigmoid layer:
layers.Dense(64, activation='sigmoid')
# Or:
layers.Dense(64, activation=tf.sigmoid)

# A linear layer with L1 regularization of factor 0.01 applied to the kernel matrix:
layers.Dense(64, kernel_regularizer=tf.keras.regularizers.l1(0.01))

# A linear layer with L2 regularization of factor 0.01 applied to the bias vector:
layers.Dense(64, bias_regularizer=tf.keras.regularizers.l2(0.01))

# A linear layer with a kernel initialized to a random orthogonal matrix:
layers.Dense(64, kernel_initializer='orthogonal')

# A linear layer with a bias vector initialized to 2.0s:
layers.Dense(64, bias_initializer=tf.keras.initializers.constant(2.0))
```

Configure learning process with compile method:

```
model.compile(optimizer=tf.train.AdamOptimizer(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

tf.keras.Model.compile takes three important arguments:

- optimizer: This object specifies the training procedure. Pass it optimizer instances from the tf.train module, such as tf.train.AdamOptimizer, tf.train.RMSPropOptimizer, or tf.train.GradientDescentOptimizer.
- loss: The function to minimize during optimization. Common choices include mean square error (mse), categorical_crossentropy, and binary_crossentropy. Loss functions are specified by name or by passing a callable object from the tf.keras.losses module.
- metrics: Used to monitor training. These are string names or callables from the tf.keras.metrics module.

More examples of optimizer parameters:

```
# Configure a model for categorical classification.
model.compile(optimizer=tf.train.RMSPropOptimizer(0.01),
              loss=tf.keras.losses.categorical_crossentropy,
              metrics=[tf.keras.metrics.categorical_accuracy])
```

Train model with fit method (Use in-memory NumPy arrays for small datasets):

```
import numpy as np

# To create data
def random_one_hot_labels(shape):
  n, n_class = shape
  classes = np.random.randint(0, n_class, n)
  labels = np.zeros((n, n_class))
  labels[np.arange(n), classes] = 1
  return labels
#Creating 1000 training data and their labels.
data = np.random.random((1000, 32))
labels = random_one_hot_labels((1000, 10))


model.fit(data, labels, epochs=10, batch_size=32) # Notice that numpy arrays are accepted as data and labels.
```

tf.keras.Model.fit takes three important arguments:

- epochs
- batch_size
- validation_data: Used to monitor performance; consists of a tuple of data and labels.

An example with validation data:

```
model.fit(data, labels, epochs=10, batch_size=32, validation_data=(val_data, val_labels))
```

Can use tf.data.Dataset:

```
# Instantiates a toy dataset instance:
dataset = tf.data.Dataset.from_tensor_slices((data, labels))
dataset = dataset.batch(32)
dataset = dataset.repeat()

# Don't forget to specify `steps_per_epoch` when calling `fit` on a dataset.
model.fit(dataset, epochs=10, steps_per_epoch=30)
```

Complete fit method using all referenced parameters.

```
val_dataset = tf.data.Dataset.from_tensor_slices((val_data, val_labels))
val_dataset = val_dataset.batch(32).repeat()

model.fit(dataset, epochs=10, steps_per_epoch=30,
        validation_data=val_dataset,
        validation_steps=3)
```

Evaluate and predict:

```
data = np.random.random((1000, 32))
labels = random_one_hot_labels((1000, 10))

model.evaluate(data, labels, batch_size=32) # Can use numpy array or dataset object.
model.evaluate(dataset, steps=30)

# To predict with a batch of data.
result = model.predict(data, batch_size=32)
```

Keras functional API allows complex models beyond the sequential model. The following example uses the functional API to build a simple, fully-connected network:

```
inputs = tf.keras.Input(shape=(32,))  # Returns a placeholder tensor

# A layer instance is callable on a tensor, and returns a tensor.
x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
predictions = layers.Dense(10, activation='softmax')(x)

#Instantiate the model given inputs and outputs. This creates a model that includes the Input layer and
# three dense layers. Note this replaces tf.keras.Sequential()
model = tf.keras.Model(inputs=inputs, outputs=predictions)

# The compile step specifies the training configuration.
model.compile(optimizer=tf.train.RMSPropOptimizer(0.001),
            loss='categorical_crossentropy',
            metrics=['accuracy'])
```

```
# Trains for 5 epochs
model.fit(data, labels, batch_size=32, epochs=5)
```

The value of the functional API can be shared with shared vectors:

```
# We can then concatenate the two vectors:
merged_vector = keras.layers.concatenate([encoded_a, encoded_b], axis=-1)

# And add a logistic regression on top
predictions = Dense(1, activation='sigmoid')(merged_vector)

# We define a trainable model linking the
# tweet inputs to the predictions
model = Model(inputs=[tweet_a, tweet_b], outputs=predictions)
```

Build a fully-customizable model by subclassing [tf.keras.Model]:

```
class MyModel(tf.keras.Model):

  def __init__(self, num_classes=10):
    super(MyModel, self).__init__(name='my_model')
    self.num_classes = num_classes
    # Define your layers here.
    self.dense_1 = layers.Dense(32, activation='relu')
    self.dense_2 = layers.Dense(num_classes, activation='sigmoid')

  def call(self, inputs):
    # Define your forward pass here,
    # using layers you previously defined (in `__init__`).
    x = self.dense_1(inputs)
    return self.dense_2(x)

  def compute_output_shape(self, input_shape):
    # You need to override this function if you want to use the subclassed model
    # as part of a functional-style model.
    # Otherwise, this method is optional.
    shape = tf.TensorShape(input_shape).as_list()
    shape[-1] = self.num_classes
    return tf.TensorShape(shape)
```

Instantiate the new model:

```
model = MyModel(num_classes=10)

# The compile step specifies the training configuration.
model.compile(optimizer=tf.train.RMSPropOptimizer(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Trains for 5 epochs.
model.fit(data, labels, batch_size=32, epochs=5)
```

Create a custom layer by subclassing tf.keras.layers.Layer and implementing the following methods:

- build: Create the weights of the layer. Add weights with the add_weight method.
- call: Define the forward pass.
- compute_output_shape: Specify how to compute the output shape of the layer given the input shape.
- Optionally, a layer can be serialized by implementing the get_config method and the from_config class method.

```
class MyLayer(layers.Layer):

‘’’
**kwargs break up a diction to pass arguments like a=2 while *args break up tuples and pass them as arguments.

*args extracts positional parameters and **kwargs extract keyword parameters.
‘’’
  def __init__(self, output_dim, **kwargs):
    self.output_dim = output_dim
    super(MyLayer, self).__init__(**kwargs)

  def build(self, input_shape):
    shape = tf.TensorShape((input_shape[1], self.output_dim))
    # Create a trainable weight variable for this layer.
    self.kernel = self.add_weight(name='kernel',
                                  shape=shape,
                                  initializer='uniform',
                                  trainable=True)
    # Make sure to call the `build` method at the end
    super(MyLayer, self).build(input_shape)

  def call(self, inputs):
    return tf.matmul(inputs, self.kernel)

  def compute_output_shape(self, input_shape):
    shape = tf.TensorShape(input_shape).as_list()
    shape[-1] = self.output_dim
    return tf.TensorShape(shape)

  def get_config(self):
    base_config = super(MyLayer, self).get_config()
    base_config['output_dim'] = self.output_dim
    return base_config

  @classmethod
  def from_config(cls, config):
    return cls(**config)
```

Create a model with custom layers:

```
model = tf.keras.Sequential([
   MyLayer(10),
   layers.Activation('softmax')])
```

Callback is an object passed to a model to execute during training: You can write custom callback, or use built-in tf.keras.callbacks that include:

- tf.keras.callbacks.ModelCheckpoint: Save checkpoints of your model at regular intervals.
- tf.keras.callbacks.LearningRateScheduler: Dynamically change the learning rate.
- tf.keras.callbacks.EarlyStopping: Interrupt training when validation performance has stopped improving.
- tf.keras.callbacks.TensorBoard: Monitor the model's behavior using TensorBoard.

Example in use:

```
callbacks = [
  # Interrupt training if `val_loss` stops improving for over 2 epochs
  tf.keras.callbacks.EarlyStopping(patience=2, monitor='val_loss'),
  # Write TensorBoard logs to `./logs` directory
  tf.keras.callbacks.TensorBoard(log_dir='./logs')
]
model.fit(data, labels, batch_size=32, epochs=5, callbacks=callbacks,
          validation_data=(val_data, val_labels))
```

save and load weights from TensorFlow Checkpoint file:

```
# Save weights to a TensorFlow Checkpoint file
model.save_weights('./weights/my_model')

# Restore the model's state,
# this requires a model with the same architecture.
model.load_weights('./weights/my_model')
```

Can also save weights to HDF5 format, which is the default method for keras:

```
# Save weights to a HDF5 file
model.save_weights('my_model.h5', save_format='h5')
# Restore the model's state
model.load_weights('my_model.h5')
```

Save and load a model’s configuration. Weights are not saved, however.

```
# Serialize a model to JSON format
json_string = model.to_json()
```

To read in model configurations:

```
import json
import pprint
pprint.pprint(json.loads(json_string)) # Prints out model info
fresh_model = tf.keras.models.model_from_json(json_string)
```

Serializing a model to YAML format requires pyyaml to be installed before importing to TensorFlow:

```
yaml_string = model.to_yaml()
fresh_model = tf.keras.models.model_from_yaml(yaml_string)
```

Subclassed models are not serializable because their architecture is defined by the Python code in the body of the **call** method.

The entire model can be saved to a file that contains the weight values, the model's configuration, and the optimizer's configuration. Allows you to resume training later at exact same state without need for original code:

```
# Save entire model to a HDF5 file
model.save('my_model.h5')

# Recreate the exact same model, including weights and optimizer.
model = tf.keras.models.load_model('my_model.h5') # Have to compile model again after this step.
```

Keras also support Eager Execution, Estimators, and multiple GPUs. More info is available on guide listed in references.

Basic Fashion Classification with Kera

Basics details about Kera. Some notable information is recorded.

```
keras.layers.Flatten(input_shape=(28, 28)) # Flattens ndarray into 1D.
```

tf.keras models are optimized to make predictions on a *batch*, or collection, of examples at once. So even though we're using a single image, we need to add it to a list to change 28x28 to 1x28x28.

Text Classification:

Use the following to pad sequences to maintain same lengths:

```
train_data = keras.preprocessing.sequence.pad_sequences(train_data,
                                                        value=word_index["<PAD>"],
                                                        padding='post',
                                                        maxlen=256)
```

The Embedding layer takes the integer-encoded vocabulary and looks up the embedding vector for each word-index. These vectors are learned as the model trains.

```
model.add(keras.layers.Embedding(vocab_size, 16))
```

Get a fixed-length output vector out to handle inputs of variable lengths:

```
model.add(keras.layers.GlobalAveragePooling1D()) # Not too clear.
```

model.fit() returns a History object that contains a dictionary with everything that happened during training:

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=40,
                    batch_size=512,
                    validation_data=(x_val, y_val),
                    verbose=1)

history_dict = history.history


acc = history_dict['acc']
val_acc = history_dict['val_acc']
loss = history_dict['loss']
val_loss = history_dict['val_loss']
```

Keras Regularization

Dropout layer:

```
    keras.layers.Dropout(0.5),
```

Further, with little data, one regularization technique is to reduce the capacity of the network to prevent overfitting.

Save uniquely named checkpoint once every 5-epochs:

```
# include the epoch in the file name. (uses `str.format`)
checkpoint_path = "training_2/cp-{epoch:04d}.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

cp_callback = tf.keras.callbacks.ModelCheckpoint(
    checkpoint_path, verbose=1, save_weights_only=True,
    # Save weights, every 5-epochs.
    period=5)

model = create_model()
model.fit(train_images, train_labels,
          epochs = 50, callbacks = [cp_callback],
          validation_data = (test_images,test_labels),
          verbose=0)
```

Chooses the latest checkpoint:

```
latest = tf.train.latest_checkpoint(checkpoint_dir)
```

```
model = create_model()
model.load_weights(latest)
```

Eager Execution

In order to interact with the model as it is being built, must use eager execution. Necessary imports and code for eager execution:

```
from __future__ import absolute_import, division, print_function

import tensorflow as tf

tf.enable_eager_execution()
```

Tensors

Tensors are multi-dimensional arrays with data type and shape. They come with various operations, including overloaded ones, that consume and produce tensors (these operations will automatically convert native Python types). The benefit of Tensors is that they can be used by GPU or TPU. They are also immutable.

some example methods:

```
print(tf.add(1, 2))
print(tf.add([1, 2], [3, 4]))
print(tf.square(5))
print(tf.reduce_sum([1, 2, 3]))
print(tf.encode_base64("hello world"))

# Operator overloading is also supported
print(tf.square(2) + tf.square(3))
```

TensorFlow operations automatically convert NumPy ndarrays to Tensors and NumPy operations automatically convert Tensors to NumPy ndarrays.

Use tensor.numpy() to convert to numpy ndarray. This operation is usually inexpensive.

GPU Acceleration

Without any annotations, TensorFlow automatically decides whether to use the GPU or CPU for an operation (and copies the tensor between CPU and GPU memory if necessary).

Detailed GPU management is possible if desired. Visit tutorial for more details.,

Datasets

Create dataset using one of the factory functions
like Dataset.from_tensors, Dataset.from_tensor_slices or using objects that read from files like TextLineDataset or TFRecordDataset:

```
ds_tensors = tf.data.Dataset.from_tensor_slices([1, 2, 3, 4, 5, 6])

# Create a CSV file
import tempfile
_, filename = tempfile.mkstemp()

with open(filename, 'w') as f:
  f.write("""Line 1
Line 2
Line 3
  """)

ds_file = tf.data.TextLineDataset(filename)
```

Use the transformations functions like map, batch, shuffle etc. to apply transformations to the records of the dataset:

```
# Map applies the function to the elements in dataset. Shuffle and batch have intuitive functions.
ds_tensors = ds_tensors.map(tf.square).shuffle(2).batch(2)
ds_file = ds_file.batch(2)
```

With eager execution, dataset objects support iteration:

```
print('Elements of ds_tensors:')
for x in ds_tensors:
  print(x)

print('\nElements in ds_file:')
for x in ds_file:
  print(x)
```

Automatic Differentiation

Enable Eager Execution to follow.

tf.GradientTape is an API for computing the gradient of a computation with respect to its input variables. TensorFlow records operations onto a GradientTape, which can be used to compute gradients of those operations.

Operations are recorded if at least one of their inputs is being "watched". Trainable variables are automatically watched.

```
x = tf.ones((2, 2))

with tf.GradientTape() as t:
  t.watch(x) # This line is necessary as we are finding gradient with respect to x and x is not a part of a model.
  y = tf.reduce_sum(x)
  z = tf.multiply(y, y)

# Derivative of z with respect to the original input tensor x
```

```
dz_dx = t.gradient(z, x) # Can also do dz_dy = t.gradient(x, y)
for i in [0, 1]:
  for j in [0, 1]:
    assert dz_dx[i][j].numpy() == 8.0
```

GradientTape are released after GradientTape.gradient() is called. Used persistent gradient tape to compute multiple gradients over the same record of computations.

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

The following is another more complicated example of GradientTape:

```
def f(x, y):
  output = 1.0
  for i in range(y):
    if i > 1 and i < 5:
      output = tf.multiply(output, x)
  return output

def grad(x, y):
  with tf.GradientTape() as t:
    t.watch(x)
    out = f(x, y)
  return t.gradient(out, x)

x = tf.convert_to_tensor(2.0)

assert grad(x, 6).numpy() == 12.0
assert grad(x, 5).numpy() == 12.0
assert grad(x, 4).numpy() == 4.0
```

Higher-order gradients are allowed as well:

```
x = tf.Variable(1.0)  # Create a Tensorflow variable initialized to 1.0

with tf.GradientTape() as t:
  with tf.GradientTape() as t2:
    y = x * x * x
  # Compute the gradient inside the 't' context manager
  # which means the gradient computation is differentiable as well.
  dy_dx = t2.gradient(y, x)
d2y_dx2 = t.gradient(dy_dx, x)
```

```
assert dy_dx.numpy() == 3.0
assert d2y_dx2.numpy() == 6.0
```

Custom Training Basics

Enable eager execution to follow.

Tensors are immutable objects; however, during training, there should be changes over time. Variable objects can be changed and are traced for computing gradients.

```
v = tf.Variable(1.0)
assert v.numpy() == 1.0

# Re-assign the value
v.assign(3.0)
assert v.numpy() == 3.0

# Use `v` in a TensorFlow operation like tf.square() and reassign
v.assign(tf.square(v))
assert v.numpy() == 9.0
```

Implementing a Simple Linear Model

Trying to fit f(x) = x*W + b where W = 3, b = 2

```
class Model(object):
  def __init__(self):
    # Initialize variable to (5.0, 0.0)
    # In practice, these should be initialized to random values.
    self.W = tf.Variable(5.0)
    self.b = tf.Variable(0.0)

  def __call__(self, x):
    return self.W * x + self.b

# Test setup
model = Model()
assert model(3.0).numpy() == 15.0


def loss(predicted_y, desired_y):
  return tf.reduce_mean(tf.square(predicted_y - desired_y))


TRUE_W = 3.0
TRUE_b = 2.0
NUM_EXAMPLES = 1000

inputs  = tf.random_normal(shape=[NUM_EXAMPLES])
```

```
noise   = tf.random_normal(shape=[NUM_EXAMPLES])
outputs = inputs * TRUE_W + TRUE_b + noise



def train(model, inputs, outputs, learning_rate):
   with tf.GradientTape() as t:
      current_loss = loss(model(inputs), outputs)
   dW, db = t.gradient(current_loss, [model.W, model.b]) # Can pass array as argument!
   model.W.assign_sub(learning_rate * dW)
   model.b.assign_sub(learning_rate * db)



model = Model()

# Collect the history of W-values and b-values to plot later
Ws, bs = [], []
epochs = range(10)
for epoch in epochs:
   Ws.append(model.W.numpy())
   bs.append(model.b.numpy())
   current_loss = loss(model(inputs), outputs)

   train(model, inputs, outputs, learning_rate=0.1)
   print('Epoch %2d: W=%1.2f b=%1.2f, loss=%2.5f' %
         (epoch, Ws[-1], bs[-1], current_loss))
```

Note that GradientTape is needed only for eager execution. In static graphics, the gradients are automatically computed with information from the created graphs. GradientTape is needed to access gradients as the model is being built.

More Complex Custom Model

The main class used when creating a layer-like thing which contains other layers is tf.keras.Model. Implementing one is done by inheriting from tf.keras.Model.

```
class ResnetIdentityBlock(tf.keras.Model):
  def __init__(self, kernel_size, filters):
    super(ResnetIdentityBlock, self).__init__(name='')
    filters1, filters2, filters3 = filters

    self.conv2a = tf.keras.layers.Conv2D(filters1, (1, 1))
    self.bn2a = tf.keras.layers.BatchNormalization()

    self.conv2b = tf.keras.layers.Conv2D(filters2, kernel_size, padding='same')
    self.bn2b = tf.keras.layers.BatchNormalization()

    self.conv2c = tf.keras.layers.Conv2D(filters3, (1, 1))
    self.bn2c = tf.keras.layers.BatchNormalization()
```

```
  def call(self, input_tensor, training=False):
    x = self.conv2a(input_tensor)
    x = self.bn2a(x, training=training)
    x = tf.nn.relu(x)

    x = self.conv2b(x)
    x = self.bn2b(x, training=training)
    x = tf.nn.relu(x)

    x = self.conv2c(x)
    x = self.bn2c(x, training=training)

    x += input_tensor
    return tf.nn.relu(x)


block = ResnetIdentityBlock(1, [1, 2, 3])
print(block(tf.zeros([1, 2, 3, 3])))
print([x.name for x in block.trainable_variables])
```

Much of the time, however, models which compose many layers simply call one layer after the other. This can be done in very little code using tf.keras.Sequential:

```
 my_seq = tf.keras.Sequential([tf.keras.layers.Conv2D(1, (1, 1)),
                               tf.keras.layers.BatchNormalization(), # New layer type!
                               tf.keras.layers.Conv2D(2, 1,
                                                      padding='same'),
                               tf.keras.layers.BatchNormalization(),
                               tf.keras.layers.Conv2D(3, (1, 1)),
                               tf.keras.layers.BatchNormalization()])
my_seq(tf.zeros([1, 2, 3, 3])) # Execute a forward pass/prediction.
```

Custom Training: Walkthrough

Let’s walkthrough building an iris classifier from measurements on the flower.

Setup:

```
from __future__ import absolute_import, division, print_function, unicode_literals

import os
import matplotlib.pyplot as plt

import tensorflow as tf

tf.enable_eager_execution()
```

Download and inspect the data to understand format. The data is in a csv.

Next, parse dataset:

```
# column order in CSV file
column_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species']

feature_names = column_names[:-1]
label_name = column_names[-1]

class_names = ['Iris setosa', 'Iris versicolor', 'Iris virginica']
```

Create a tf.data.Dataset:

```
batch_size = 32


# train_dataset_fp is a file path created when downloading and analyzing data.
train_dataset = tf.contrib.data.make_csv_dataset(
    train_dataset_fp,
    batch_size,
    column_names=column_names,
    label_name=label_name,
    num_epochs=1)
```

The default behavior is to shuffle the data (shuffle=True, shuffle_buffer_size=10000), and repeat the dataset forever (num_epochs=None).

The make_csv_dataset function returns a tf.data.Dataset of (features, label) pairs, where features is a dictionary: {'feature_name': value}. With eager execution enabled, these Dataset objects are iterable.

```
features, labels = next(iter(train_dataset)) # Gets batch from dataset.
```

Repackages array into shape (batch_size, num_features):

```
def pack_features_vector(features, labels):
  """Pack the features into a single array instead of using a dictionary."""
  features = tf.stack(list(features.values()), axis=1)
  return features, labels

train_dataset = train_dataset.map(pack_features_vector) # Applies function on all data in dataset.
```

Now select a model. Let's create a simple fully connected NN:

```
model = tf.keras.Sequential([
  tf.keras.layers.Dense(10, activation=tf.nn.relu, input_shape=(4,)),
  tf.keras.layers.Dense(10, activation=tf.nn.relu),
  tf.keras.layers.Dense(3)
])
```

Make a prediction:

```
predictions = model(features) # Note that features is a regular tensor array.
```

```
tf.argmax(predictions, axis=1) # Grabs index of max value actual classification prediction.
```

Let's try writing our own training loop. Necessary components for training loop:

```
def loss(model, x, y):
  y_ = model(x)
  return tf.losses.sparse_softmax_cross_entropy(labels=y, logits=y_)
```

Find gradients:

```
def grad(model, inputs, targets):
  with tf.GradientTape() as tape:
    loss_value = loss(model, inputs, targets)
  return loss_value, tape.gradient(loss_value, model.trainable_variables)
```

Let's setup an optimizer and gloab_step counter:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
global_step = tf.Variable(0) # Stores the number of learning steps taken.
loss_value, grads = grad(model, features, labels)

print("Step: {}, Initial Loss: {}".format(global_step.numpy(),
                                          loss_value.numpy()))

optimizer.apply_gradients(zip(grads, model.trainable_variables), global_step)

print("Step: {},         Loss: {}".format(global_step.numpy(),
                                          loss(model, features, labels).numpy()))

# Step: 0, Initial Loss: 1.7886989116668701
# Step: 1,         Loss: 1.3980517387390137
```

Now let's train!

```
from tensorflow import contrib
tfe = contrib.eager # contrib module is for experimental code.

# keep results for plotting
train_loss_results = []
train_accuracy_results = []

num_epochs = 201

for epoch in range(num_epochs):
  epoch_loss_avg = tfe.metrics.Mean()
  epoch_accuracy = tfe.metrics.Accuracy()
```

```
  # Training loop - using batches of 32
  for x, y in train_dataset:
    # Optimize the model
    loss_value, grads = grad(model, x, y)
    optimizer.apply_gradients(zip(grads, model.trainable_variables),
                              global_step)

    # Track progress
    epoch_loss_avg(loss_value)  # add current batch loss
    # compare predicted label to actual label
    epoch_accuracy(tf.argmax(model(x), axis=1, output_type=tf.int32), y)

  # end epoch
  train_loss_results.append(epoch_loss_avg.result())
  train_accuracy_results.append(epoch_accuracy.result())

  if epoch % 50 == 0:
    print("Epoch {:03d}: Loss: {:.3f}, Accuracy: {:.3%}".format(epoch,
                                                epoch_loss_avg.result(),
                                                epoch_accuracy.result()))
```

Testing:

```
test_dataset = tf.contrib.data.make_csv_dataset(
    test_fp,
    batch_size,
    column_names=column_names,
    label_name='species',
    num_epochs=1,
    shuffle=False)

test_dataset = test_dataset.map(pack_features_vector)

test_accuracy = tfe.metrics.Accuracy()

for (x, y) in test_dataset:
  logits = model(x)
  prediction = tf.argmax(logits, axis=1, output_type=tf.int32)
  test_accuracy(prediction, y)

print("Test set accuracy: {:.3%}".format(test_accuracy.result()))
```

Lets make prediction on three inputs:

```
# Creating three inputs.

predict_dataset = tf.convert_to_tensor([
    [5.1, 3.3, 1.7, 0.5,],
    [5.9, 3.0, 4.2, 1.5,],
```

```
    [6.9, 3.1, 5.4, 2.1]
])

predictions = model(predict_dataset)

for i, logits in enumerate(predictions):
  class_idx = tf.argmax(logits).numpy() # Get index of max value
  p = tf.nn.softmax(logits)[class_idx] # Get percentage
  name = class_names[class_idx] # Matches index to class name.
  print("Example {} prediction: {} ({:4.1f}%)".format(i, name, 100*p))
```

Linear Model with Estimators…Outdated

Estimators are essentially premade models, just need to specify some configurations then feed input data.

Setup:

```
from __future__ import absolute_import, division, print_function, unicode_literals

import tensorflow as tf
import tensorflow.feature_column as fc

import os
import sys

import matplotlib.pyplot as plt
from IPython.display import clear_output

tf.enable_eager_execution()
```

The tutorial tells you how to download a model, data, and run it from the command line. I skipped this part. I am only interested in downloading the data and building our own model, which I will detail below.

Download dataset:

```
from official.wide_deep import census_dataset
from official.wide_deep import census_main

census_dataset.download("/tmp/census_data/")
```

Analyze data. Then load data into dataset object. Note this shows another way to load CSV data:

```
import pandas

train_df = pandas.read_csv(train_file, header = None, names = census_dataset._CSV_COLUMNS)
test_df = pandas.read_csv(test_file, header = None, names = census_dataset._CSV_COLUMNS)

# This method is not scalable; try the method input_fn written below instead.
```

```
def easy_input_function(df, label_key, num_epochs, shuffle, batch_size):
  label = df[label_key]
  ds = tf.data.Dataset.from_tensor_slices((dict(df),label))

  if shuffle:
    ds = ds.shuffle(10000)

  ds = ds.batch(batch_size).repeat(num_epochs)

  return ds

ds = easy_input_function(train_df, label_key='income_bracket', num_epochs=5, shuffle=True, batch_size=10)
```

```
def input_fn(data_file, num_epochs, shuffle, batch_size):
  """Generate an input function for the Estimator."""
  assert tf.gfile.Exists(data_file), (
      '%s not found. Please make sure you have run census_dataset.py and '
      'set the --data_dir argument to the correct path.' % data_file)

  def parse_csv(value):
    tf.logging.info('Parsing {}'.format(data_file))
    columns = tf.decode_csv(value, record_defaults=_CSV_COLUMN_DEFAULTS)
    features = dict(zip(_CSV_COLUMNS, columns))
    labels = features.pop('income_bracket')
    classes = tf.equal(labels, '>50K')  # binary classification
    return features, classes

  # Extract lines from input files using the Dataset API.
  dataset = tf.data.TextLineDataset(data_file)

  if shuffle:
    # Where did _NUM_EXAMPLES dict come from? Not important though.
    dataset = dataset.shuffle(buffer_size=_NUM_EXAMPLES['train'])

  dataset = dataset.map(parse_csv, num_parallel_calls=5)

  # We call repeat after shuffling, rather than before, to prevent
separate
  # epochs from blending together.
  dataset = dataset.repeat(num_epochs)
  dataset = dataset.batch(batch_size)
  return dataset
```

```
ds = census_dataset.input_fn(train_file, num_epochs=5, shuffle=True,
batch_size=10)
```

Estimators expect an input_fn with no arguments, so we much wrap input function into an object with expected signature.

```
import functools

train_inpf = functools.partial(census_dataset.input_fn, train_file, num_epochs=2, shuffle=True,
batch_size=64)
test_inpf = functools.partial(census_dataset.input_fn, test_file, num_epochs=1, shuffle=False, batch_size=64)
```

An Estimator expects a vector of numeric inputs, and feature columns describe how the model should convert each feature.

```
import tensorflow.feature_column as fc # I believe this is needed.

age = fc.numeric_column('age')

fc.input_layer(feature_batch, [age]).numpy() # Inspect resulting data that will be feed to model.

# Training model using only age feature.

classifier = tf.estimator.LinearClassifier(feature_columns=[age])
classifier.train(train_inpf)
result = classifier.evaluate(test_inpf)


clear_output()  # used for display in notebook
print(result)
```

For categorical data, use one of the tf.feature_column.categorical_column functions. Create sparse one hot vectors with categorical_column_with_vocabulary_list.

```
relationship = fc.categorical_column_with_vocabulary_list('relationship',
   ['Husband', 'Not-in-family', 'Wife', 'Own-child', 'Unmarried', 'Other-relative'])

fc.input_layer(feature_batch, [age, fc.indicator_column(relationship)]) # The indicator_column creates a dense one-hot output.
```

If the number of categorical data is unknow, use the following:

```
occupation = tf.feature_column.categorical_column_with_hash_bucket(
   'occupation', hash_bucket_size=1000)
```

More examples:

```
education = tf.feature_column.categorical_column_with_vocabulary_list(
   'education', [
      'Bachelors', 'HS-grad', '11th', 'Masters', '9th', 'Some-college',
      'Assoc-acdm', 'Assoc-voc', '7th-8th', 'Doctorate', 'Prof-school',
      '5th-6th', '10th', '1st-4th', 'Preschool', '12th'])
```

```
marital_status = tf.feature_column.categorical_column_with_vocabulary_list(
   'marital_status', [
      'Married-civ-spouse', 'Divorced', 'Married-spouse-absent',
      'Never-married', 'Separated', 'Married-AF-spouse', 'Widowed'])

workclass = tf.feature_column.categorical_column_with_vocabulary_list(
   'workclass', [
      'Self-emp-not-inc', 'Private', 'State-gov', 'Federal-gov',
      'Local-gov', '?', 'Self-emp-inc', 'Without-pay', 'Never-worked'])



my_categorical_columns = [relationship, occupation, education, marital_status, workclass]
```

Using them to train and evaluate:

```
classifier = tf.estimator.LinearClassifier(feature_columns=my_numeric_columns+my_categorical_columns)
classifier.train(train_inpf)
result = classifier.evaluate(test_inpf)
```

Make continuous features categorical with bucketization:

```
age_buckets = tf.feature_column.bucketized_column(
   age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])

fc.input_layer(feature_batch, [age, age_buckets]).numpy()

# Output

array([[35.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [48.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.],
       [43.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.],
       [64.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.],
       [41.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.],
       [55.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.],
       [31.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [34.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [46.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.],
       [44.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.]],
      dtype=float32)
```

This gives the model information about the value of the data and its location relative to other data. More powerful at times.

Cross columns to give connections between feature columns (Not too useful and confusing):

```
age_buckets_x_education_x_occupation = tf.feature_column.crossed_column(
   [age_buckets, 'education', 'occupation'], hash_bucket_size=1000)
```

Define the logistic regression model:

```
import tempfile

base_columns = [
    education, marital_status, relationship, workclass, occupation,
    age_buckets,
]

crossed_columns = [
    tf.feature_column.crossed_column(
        ['education', 'occupation'], hash_bucket_size=1000),
    tf.feature_column.crossed_column(
        [age_buckets, 'education', 'occupation'], hash_bucket_size=1000),
]

model = tf.estimator.LinearClassifier(
    model_dir=tempfile.mkdtemp(),
    feature_columns=base_columns + crossed_columns,
    optimizer=tf.train.FtrlOptimizer(learning_rate=0.1))
```

The learned model files are stored in model_dir.

Train and evaluate:

```
train_inpf = functools.partial(census_dataset.input_fn, train_file,
                               num_epochs=40, shuffle=True, batch_size=64)

model.train(train_inpf)



results = model.evaluate(test_inpf)
```

Adding regularization:

```
model_l1 = tf.estimator.LinearClassifier(
    feature_columns=base_columns + crossed_columns,
    optimizer=tf.train.FtrlOptimizer(
        learning_rate=0.1,
        l1_regularization_strength=10.0,
        l2_regularization_strength=0.0))
```

Boosted Trees

Skipped. Revisit at the end.

Boosted Trees Model Understanding

Skipped. Revisit at the end.

Build a CNN Using Estimators…Outdated

Setup:

```
from __future__ import absolute_import, division, print_function, unicode_literals

import tensorflow as tf
import numpy as np

tf.logging.set_verbosity(tf.logging.INFO)
```

This function takes MNIST feature data, labels, and mode (from[tf.estimator.ModeKeys](): TRAIN, EVAL, PREDICT) as arguments; configures the CNN; and returns predictions, loss, and a training operation:

```
def cnn_model_fn(features, labels, mode):
  """Model function for CNN."""
  # Input Layer, [batch_size, height, width, channel]. Can also channel data mode to have
  #channel before height and width.
  input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

  # Convolutional Layer #1. Can change input format as well to be channel before h x w.
  conv1 = tf.layers.conv2d(
      inputs=input_layer,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  # Pooling Layer #1
  pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)

  # Convolutional Layer #2 and Pooling Layer #2. Each filter is applied to one channel.
  conv2 = tf.layers.conv2d(
      inputs=pool1,
      filters=64,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)
  pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)

  # Dense Layer
  pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])
  dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)
  dropout = tf.layers.dropout(
      inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

  # Logits Layer
  logits = tf.layers.dense(inputs=dropout, units=10)
```

```
  predictions = {
      # Generate predictions (for PREDICT and EVAL mode)
      "classes": tf.argmax(input=logits, axis=1),
      # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
      # `logging_hook`.
      "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
  }

  if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

  # Calculate Loss (for both TRAIN and EVAL modes)
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

  # Configure the Training Op (for TRAIN mode)
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

  # Add evaluation metrics (for EVAL mode)
  eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])
  }
  return tf.estimator.EstimatorSpec(
      mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

Load and process data:

```
# Load training and eval data
((train_data, train_labels),
 (eval_data, eval_labels)) = tf.keras.datasets.mnist.load_data()

train_data = train_data/np.float32(255)
train_labels = train_labels.astype(np.int32)  # not required

eval_data = eval_data/np.float32(255)
eval_labels = eval_labels.astype(np.int32)  # not required
```

Creating the estimator:

```
# Create the Estimator
mnist_classifier = tf.estimator.Estimator(
    model_fn=cnn_model_fn, model_dir="/tmp/mnist_convnet_model")
```

The model_fn argument specifies the model function to use for training, evaluation, and prediction; we pass it the cnn_model_fn we created in "Building the CNN MNIST

Classifier." The model_dir argument specifies the directory where model data (checkpoints) will be saved.

Set up logging hook to track progress:

```
# Set up logging for predictions
tensors_to_log = {"probabilities": "softmax_tensor"} # softmax_tensor is the name of a tensor

#in the TensorFlow graph.

logging_hook = tf.train.LoggingTensorHook(
    tensors=tensors_to_log, every_n_iter=50)
```

Training the model:

```
# Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": train_data},
    y=train_labels,
    batch_size=100,
    num_epochs=None, # Will train until specified number of steps is reached.
    shuffle=True)

# train one step and display the probabilties
mnist_classifier.train(
    input_fn=train_input_fn,
    steps=1,
    hooks=[logging_hook])
```

Evaluate performance:

```
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": eval_data},
    y=eval_labels,
    num_epochs=1,
    shuffle=False)

eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
```

Generative Models

Skipped DCGAN and VAE.

Distributed Training

The tf.distribute.Strategy API provides an abstraction for distributing your training across multiple processing units.

This tutorial uses the tf.distribute.MirroredStrategy, which does in-graph replication with synchronous training on many GPUs on one machine. Essentially, it copies all of the model's variables to each processor. Then, it uses all-reduce to combine the gradients from all processors

and applies the combined value to all copies of the model. For more techniques, visit distribution strategy guide.

Setup:

```
from __future__ import absolute_import, division, print_function, unicode_literals


# Import TensorFlow
!pip install -q tf-nightly-gpu
import tensorflow as tf
import tensorflow_datasets as tfds

import os
```

Download and load data (with_info downloads metadata like number of training and testing examples)

```
datasets, ds_info = tfds.load(name='mnist', with_info=True, as_supervised=True)
mnist_train, mnist_test = datasets['train'], datasets['test']
```

Define strategy:

```
strategy = tf.distribute.MirroredStrategy()
```

Setup input pipeline:

```
# Gets num of training and testing examples.

num_train_examples = ds_info.splits['train'].num_examples
num_test_examples = ds_info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE_PER_REPLICA = 64

# Total batch size across all GPUs.
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync
```

Scale inputs (labels are accepted in order to pass it to map function):

```
def scale(image, label):
  image = tf.cast(image, tf.float32)
  image /= 255

  return image, label
```

Apply changes to data:

```
train_dataset = mnist_train.map(scale).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

Create and compile the Keras model in the context of strategy.scope:

```
with strategy.scope():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(10, activation='softmax')
  ])

  model.compile(loss='sparse_categorical_crossentropy',
                optimizer=tf.keras.optimizers.Adam(),
                metrics=['accuracy'])
```

Defining callbacks:

```
 # Define the checkpoint directory to store the checkpoints

checkpoint_dir = './training_checkpoints'
# Name of the checkpoint files
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")


# Function for decaying the learning rate.
# You can define any decay function you need.
def decay(epoch):
  if epoch < 3:
    return 1e-3
  elif epoch >= 3 and epoch < 7:
    return 1e-4
  else:
    return 1e-5


# Callback for printing the LR at the end of each epoch. Custom callback class!
class PrintLR(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs=None):
    print ('\nLearning rate for epoch {} is {}'.format(
        epoch + 1, tf.keras.backend.get_value(model.optimizer.lr)))


callbacks = [
    tf.keras.callbacks.TensorBoard(log_dir='./logs'),
    tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_prefix,
                                       save_weights_only=True),
    tf.keras.callbacks.LearningRateScheduler(decay),
    PrintLR()
]
```

Training:

```
model.fit(train_dataset, epochs=10, callbacks=callbacks)
```

Evaluating:

```
model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))

eval_loss, eval_acc = model.evaluate(eval_dataset)
print ('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

To see the output, you can download and view the TensorBoard logs at the terminal:

```
$ tensorboard --logdir=path/to/log-directory
```

Save your model:

```
tf.keras.experimental.export_saved_model(model, path)
```

Load the model without strategy.scope:

```
unreplicated_model = tf.keras.experimental.load_from_saved_model(path)

unreplicated_model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy'])

eval_loss, eval_acc = unreplicated_model.evaluate(eval_dataset)
print ('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

Training Loops

Setup:

```
from __future__ import absolute_import, division, print_function, unicode_literals

# Import TensorFlow
!pip install -q tf-nightly-gpu
import tensorflow as tf

# Helper libraries
import numpy as np
import os
```

Downloading fashion mnist dataset:

```
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

```
# Adding a dimension to the array -> new shape == (28, 28, 1)
# We are doing this because the first layer in our model is a convolutional
# layer and it requires a 4D input (batch_size, height, width, channels).
# batch_size dimension will be added later on.
train_images = train_images[..., None]
test_images = test_images[..., None]

# Getting the images in [0, 1] range.
train_images = train_images / np.float32(255)
test_images = test_images / np.float32(255)

train_labels = train_labels.astype('int64')
test_labels = test_labels.astype('int64')
```

Getting strategy for distributed training:

```
# If the list of devices is not specified in the
# `tf.distribute.MirroredStrategy` constructor, it will be auto-detected.
strategy = tf.distribute.MirroredStrategy()
```

Input pipeline:

```
BUFFER_SIZE = len(train_images)

BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync

EPOCHS = 10
```

tf.distribute.Strategy.experimental_distribute_dataset evenly distributes the dataset across all the replicas.

```
with strategy.scope():
  train_dataset = tf.data.Dataset.from_tensor_slices(
  (train_images, train_labels)).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
  train_ds = strategy.experimental_distribute_dataset(train_dataset)

  test_dataset = tf.data.Dataset.from_tensor_slices(
      (test_images, test_labels)).batch(BATCH_SIZE)
  test_ds = strategy.experimental_distribute_dataset(test_dataset)
```

Model creation:

```
with strategy.scope():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32, 3, activation='relu',
                             input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Conv2D(64, 3, activation='relu'),
```

```
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
  ])
optimizer = tf.train.GradientDescentOptimizer(0.001)
```

Note, loss is divided by the GLOBAL_BATCH_SIZE even though the global batch is divided evenly among the GPUs because when the GPUs sync up, the gradients are added.

This information is so confusing.

When writing own training loop, don't forget to divide loss by GLOBAL_BATCH_SIZE. But, if you are using regularization losses in your model then you need to scale the loss value by number of replicas. You can do this by using the tf.nn.scale_regularization_loss function.

This reduction and scaling to the loss is done automatically in keras model.compile and model.fit

If using tf.keras.losses classes, the loss reduction needs to be explicitly specified to be one of NONE or SUM.

Training Loop (replaces fit in keras), VERY CONFUSING:

```
with strategy.scope():
  def train_step(dist_inputs):
    def step_fn(inputs):
      images, labels = inputs
      logits = model(images)
      cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
          logits=logits, labels=labels)
      loss = tf.nn.compute_average_loss(cross_entropy, global_batch_size=BATCH_SIZE)
      train_op = optimizer.minimize(loss)
      with tf.control_dependencies([train_op]):
        return tf.identity(loss)

    per_replica_losses = strategy.experimental_run_v2(
        step_fn, args=(dist_inputs,))
    mean_loss = strategy.reduce(
        tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)
    return mean_loss


with strategy.scope():
  train_iterator = train_ds.make_initializable_iterator()
  iterator_init = train_iterator.initialize()
  var_init = tf.global_variables_initializer()
  loss = train_step(next(train_iterator))
  with tf.Session() as sess:
    sess.run([var_init])
    for epoch in range(EPOCHS):
```

```
    sess.run([iterator_init])
    for step in range(10000):
      if step % 1000 == 0:
        print('Epoch {} Step {} Loss {:.4f}'.format(epoch+1,
                                                    step,
                                                    sess.run(loss)))
```

TPU Custom Training

Setup:

```
from __future__ import absolute_import, division, print_function, unicode_literals

# Import TensorFlow
import tensorflow as tf

# Helper libraries
import numpy as np
import os
```

Create model:

```
def create_model(input_shape):
  """Creates a simple convolutional neural network model using the Keras API"""
  return tf.keras.Sequential([
      tf.keras.layers.Conv2D(28, kernel_size=(3, 3), activation='relu', input_shape=input_shape),
      tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(128, activation=tf.nn.relu),
      tf.keras.layers.Dropout(0.2),
      tf.keras.layers.Dense(10, activation=tf.nn.softmax),
  ])
```

Loss and gradient methods:

```
def loss(model, x, y):
  """Calculates the loss given an example (x, y)"""
  logits = model(x)
  return logits, tf.losses.sparse_softmax_cross_entropy(labels=y, logits=logits)

def grad(model, x, y):
  """Calculates the loss and the gradients given an example (x, y)"""
  logits, loss_value = loss(model, x, y)
  return logits, loss_value, tf.gradients(loss_value, model.trainable_variables)
```

Main function:

```
tf.keras.backend.clear_session()

resolver = tf.contrib.cluster_resolver.TPUClusterResolver(tpu=TPU_WORKER)
```

```
tf.contrib.distribute.initialize_tpu_system(resolver)
strategy = tf.contrib.distribute.TPUStrategy(resolver)

# Load MNIST training and test data
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# All MNIST examples are 28x28 pixel greyscale images (hence the 1
# for the number of channels).
input_shape = (28, 28, 1)

# Only specific data types are supported on the TPU, so it is important to
# pay attention to these.
# More information:
# https://cloud.google.com/tpu/docs/troubleshooting#unsupported_data_type
x_train = x_train.reshape(x_train.shape[0], *input_shape).astype(np.float32) # * splits tuple.
x_test = x_test.reshape(x_test.shape[0], *input_shape).astype(np.float32)
y_train, y_test = y_train.astype(np.int64), y_test.astype(np.int64)

# The batch size must be divisible by the number of workers (8 workers),
# so batch sizes of 8, 16, 24, 32, ... are supported.
BATCH_SIZE = 32

NUM_EPOCHS = 5

train_steps_per_epoch = len(x_train) // BATCH_SIZE
test_steps_per_epoch = len(x_test) // BATCH_SIZE
```

Create objects within strategy's scope:

```
with strategy.scope():
  model = create_model(input_shape)

  optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)

  training_loss = tf.keras.metrics.Mean('training_loss', dtype=tf.float32)
  training_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
      'training_accuracy', dtype=tf.float32)
  test_loss = tf.keras.metrics.Mean('test_loss', dtype=tf.float32)
  test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
      'test_accuracy', dtype=tf.float32)
```

Define custom train and test steps:

```
with strategy.scope():
  def train_step(inputs):
    """Each training step runs this custom function which calculates
    gradients and updates weights.
    """
```

```
    x, y = inputs

    logits, loss_value, grads = grad(model, x, y)

    # Show that this is truly a custom training loop
    # Multiply all gradients by 2.
    grads = grads * 2

    update_vars = optimizer.apply_gradients(
        zip(grads, model.trainable_variables))

    update_loss = training_loss.update_state(loss_value)
    update_accuracy = training_accuracy.update_state(y, logits)

    with tf.control_dependencies([update_vars, update_loss, update_accuracy]):
      return tf.identity(loss_value)

  def test_step(inputs):
    """Each training step runs this custom function"""
    x, y = inputs

    logits, loss_value = loss(model, x, y)

    update_loss = test_loss.update_state(loss_value)
    update_accuracy = test_accuracy.update_state(y, logits)

    with tf.control_dependencies([update_loss, update_accuracy]):
      return tf.identity(loss_value)
```

Do the training:

```
def run_train():
  # Train
  session.run(train_iterator_init)
  while True:
    try:
      session.run(dist_train)
    except tf.errors.OutOfRangeError:
      break
  print('Train loss: {:0.4f}\t Train accuracy: {:0.4f}%'.format(
      session.run(training_loss_result),
      session.run(training_accuracy_result) * 100))
  training_loss.reset_states()
  training_accuracy.reset_states()

def run_test():
  # Test
  session.run(test_iterator_init)
```

```
  while True:
    try:
      session.run(dist_test)
    except tf.errors.OutOfRangeError:
      break
  print('Test loss: {:0.4f}\t Test accuracy: {:0.4f}%'.format(
      session.run(test_loss_result),
      session.run(test_accuracy_result) * 100))
  test_loss.reset_states()
  test_accuracy.reset_states()
```

```
with strategy.scope():
  training_loss_result = training_loss.result()
  training_accuracy_result = training_accuracy.result()
  test_loss_result = test_loss.result()
  test_accuracy_result = test_accuracy.result()

  config = tf.ConfigProto()
  config.allow_soft_placement = True
  cluster_spec = resolver.cluster_spec()
  if cluster_spec:
    config.cluster_def.CopyFrom(cluster_spec.as_cluster_def())

  print('Starting training...')

  # Do all the computations inside a Session (as opposed to doing eager mode)
  with tf.Session(target=resolver.master(), config=config) as session:
    all_variables = (
        tf.global_variables() + training_loss.variables +
        training_accuracy.variables + test_loss.variables +
        test_accuracy.variables)

    train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(BATCH_SIZE,
drop_remainder=True)
    train_iterator = strategy.make_dataset_iterator(train_dataset)

    test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(BATCH_SIZE,
drop_remainder=True)
    test_iterator = strategy.make_dataset_iterator(train_dataset)

    train_iterator_init = train_iterator.initialize()
    test_iterator_init = test_iterator.initialize()

    session.run([v.initializer for v in all_variables])

    dist_train = strategy.experimental_run(train_step, train_iterator).values
    dist_test = strategy.experimental_run(test_step, test_iterator).values
```

```
  # Custom training loop
  for epoch in range(0, NUM_EPOCHS):
    print('Starting epoch {}'.format(epoch))

    run_train()

    run_test()
```

Transfer Learning with TFHub

```
 !pip install -q -U tensorflow_hub
!pip install -q tf-nightly-gpu


from __future__ import absolute_import, division, print_function, unicode_literals

import matplotlib.pylab as plt

import tensorflow as tf
tf.enable_eager_execution()


import tensorflow_hub as hub

from tensorflow.keras import layers

classifier_url ="https://tfhub.dev/google/tf2-preview/mobilenet_v2/classification/2" #@param {type:"string"}


IMAGE_SHAPE = (224, 224)

classifier = tf.keras.Sequential([
   hub.KerasLayer(classifier_url, input_shape=IMAGE_SHAPE+(3,))
])
```

Testing on one image:

```
import numpy as np
import PIL.Image as Image

grace_hopper =
tf.keras.utils.get_file('image.jpg','https://storage.googleapis.com/download.tensorflow.org/example_images/grace_hopper.jpg')
grace_hopper = Image.open(grace_hopper).resize(IMAGE_SHAPE)

grace_hopper = np.array(grace_hopper)/255.0
result = classifier.predict(grace_hopper[np.newaxis, ...])

predicted_class = np.argmax(result[0], axis=-1)
```

```
labels_path =
tf.keras.utils.get_file('ImageNetLabels.txt','https://storage.googleapis.com/download.tensorflow.org/data/ImageNetLabels.txt')
imagenet_labels = np.array(open(labels_path).read().splitlines())

# Now can link predicted_class with label.
```

Transfer learning now.

Get new dataset:

```
data_root = tf.keras.utils.get_file(
  'flower_photos','https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz'
, untar=True) # untar necessary because the file is a tgz, or TAR Archive file.

# Provides tons of augmentations
image_generator = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1/255)
# Applies augmentations to images in data_root. Resizing all images to target_size.
image_data = image_generator.flow_from_directory(str(data_root), target_size=IMAGE_SHAPE)
```

The resulting object is an iterator that returns image_batch, label_batch pairs.

In the guide, they only passed the new data directly to the model without retraining top layer. Not useful.

Can download headless model to train classification layer:

```
feature_extractor_url = "https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/2" #@param {type:"string"}. URL for model with classifiers/softmax.

feature_extractor_layer = hub.KerasLayer(feature_extractor_url,
                                         input_shape=(224,224,3))
feature_batch = feature_extractor_layer(image_batch) # Gets a batch of features.
```

Now freeze the variables in the model so only the new classifier layer is trained:

```
feature_extractor_layer.trainable = False

model = tf.keras.Sequential([
  feature_extractor_layer,
  layers.Dense(image_data.num_classes, activation='softmax')
])

 # Use model.summary() to get summary of components in model.
```

Compile model:

```
model.compile(
  optimizer=tf.keras.optimizers.Adam(),
  loss='categorical_crossentropy',
  metrics=['acc'])
```

Now train with callbacks:

```
class CollectBatchStats(tf.keras.callbacks.Callback):
  def __init__(self):
    self.batch_losses = []
    self.batch_acc = []

  def on_train_batch_end(self, batch, logs=None):
    self.batch_losses.append(logs['loss'])
    self.batch_acc.append(logs['acc'])
    self.model.reset_metrics()


steps_per_epoch = np.ceil(image_data.samples/image_data.batch_size)

batch_stats_callback = CollectBatchStats()

history = model.fit(image_data, epochs=2,
                    steps_per_epoch=steps_per_epoch,
                    callbacks = [batch_stats_callback])
```

Export your model:

```
import time
t = time.time()

export_path = "/tmp/saved_models/{}".format(int(t))
tf.keras.experimental.export_saved_model(model, export_path)
```

Load model:

```
reloaded = tf.keras.experimental.load_from_saved_model(export_path,
custom_objects={'KerasLayer':hub.KerasLayer})
```

Transfer Learning Using Pretrained ConvNets

Setup:

```
from __future__ import absolute_import, division, print_function, unicode_literals

import os

import tensorflow as tf
from tensorflow import keras

import numpy as np

import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

Data collection and sorting:

```
zip_file = tf.keras.utils.get_file(origin="https://storage.googleapis.com/mledu-
datasets/cats_and_dogs_filtered.zip",
                                   fname="cats_and_dogs_filtered.zip", extract=True)
base_dir, _ = os.path.splitext(zip_file) # Splits at extension, which is .zip

train_dir = os.path.join(base_dir, 'train') # Gets training path.
validation_dir = os.path.join(base_dir, 'validation') # Gets validation path.

# Directory with our training cat pictures
train_cats_dir = os.path.join(train_dir, 'cats') # Gets training cat path.
print ('Total training cat images:', len(os.listdir(train_cats_dir)))

# Directory with our training dog pictures
train_dogs_dir = os.path.join(train_dir, 'dogs') # Gets training dog path.
print ('Total training dog images:', len(os.listdir(train_dogs_dir)))

# Directory with our validation cat pictures
validation_cats_dir = os.path.join(validation_dir, 'cats')
print ('Total validation cat images:', len(os.listdir(validation_cats_dir)))

# Directory with our validation dog pictures
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
print ('Total validation dog images:', len(os.listdir(validation_dogs_dir)))
```

Preprocessing:

```
image_size = 160 # All images will be resized to 160x160
batch_size = 32

# Rescale all images by 1./255 and apply image augmentation
train_datagen = keras.preprocessing.image.ImageDataGenerator(
        rescale=1./255)

validation_datagen = keras.preprocessing.image.ImageDataGenerator(rescale=1./255)

# Flow training images in batches of 20 using train_datagen generator. Tensor (feature, label) pairs are return.
train_generator = train_datagen.flow_from_directory(
        train_dir,  # Source directory for the training images
        target_size=(image_size, image_size),
        batch_size=batch_size,
        # Autogenerated labels. Since we use binary_crossentropy loss, we need binary labels.
        class_mode='binary')

# Flow validation images in batches of 20 using test_datagen generator
validation_generator = validation_datagen.flow_from_directory(
        validation_dir, # Source directory for the validation images
        target_size=(image_size, image_size),
        batch_size=batch_size,
        class_mode='binary')
```

Create pretrained model with dense layers left out:

```
IMG_SHAPE = (image_size, image_size, 3)

# Create the base model from the pre-trained model MobileNet V2
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                               include_top=False,
                                               weights='imagenet')

base_model.trainable = False # Freeze conv layers from training

model = tf.keras.Sequential([
  base_model,
  keras.layers.GlobalAveragePooling2D(),
  keras.layers.Dense(1, activation='sigmoid')
])



model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=0.0001),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Train the model:

```
epochs = 10
steps_per_epoch = train_generator.n // batch_size
validation_steps = validation_generator.n // batch_size

history = model.fit_generator(train_generator,
                              steps_per_epoch = steps_per_epoch,
                              epochs=epochs,
                              workers=4,
                              validation_data=validation_generator,
                              validation_steps=validation_steps)
```

Lets train some of the last convolution layers, where abstraction and specialization takes place:

```
base_model.trainable = True


# Let's take a look to see how many layers are in the base model
print("Number of layers in the base model: ", len(base_model.layers))

# Fine tune from this layer onwards
fine_tune_at = 100

# Freeze all the layers before the `fine_tune_at` layer
```

```
for layer in base_model.layers[:fine_tune_at]:
  layer.trainable =  False
```

Compile the model using a much-lower training rate.

```
model.compile(optimizer = tf.keras.optimizers.RMSprop(lr=2e-5),
              loss='binary_crossentropy',
              metrics=['accuracy'])

history_fine = model.fit_generator(train_generator,
                                   steps_per_epoch = steps_per_epoch,
                                   epochs=epochs,
                                   workers=4,
                                   validation_data=validation_generator,
                                   validation_steps=validation_steps)
```

Advanced CNN

All the code is neatly provided for this tutorial.

Image distortions:

- tf.random_crop for random cropping
- tf.image.per_image_standardization to make the model insensitive to dynamic range.
- tf.image.random_flip_left_right flips the image from left to right.
- Randomly distort the image with tf.image.random_brightness.
- Randomly distort the image with tf.image.random_contrast.

Please see the Images page for the list of available distortions.

Reading images from disk and distorting them can use a non-trivial amount of processing time. To prevent these operations from slowing down training, we apply the transformation in parallel (num_parallel_calls argument of dataset.map()), and prefetch the data.

The rest is a standard CNN model.

This tutorial briefly mentions Tensorboard.

This tutorial also talks about training on multiple GPUs.

Text Generation with an RNN

Setup:

```
from __future__ import absolute_import, division, print_function, unicode_literals

import tensorflow as tf
tf.enable_eager_execution()
```

```
import numpy as np
import os
import time
```

Download and read text:

```
path_to_file = tf.keras.utils.get_file('shakespeare.txt',
'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt')

# Read, then decode for py2 compat.
text = open(path_to_file, 'rb').read().decode(encoding='utf-8')
```

Map text to indices, indices to text, and vectorize text:

```
# Creating a mapping from unique characters to indices
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)

text_as_int = np.array([char2idx[c] for c in text])
```

Creating training examples and targets with the following format: the input sequence would be "Hell", and the target sequence "ello".

```
# The maximum length sentence we want for a single input in characters
seq_length = 100
examples_per_epoch = len(text)//seq_length

# Create training examples / targets
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)

sequences = char_dataset.batch(seq_length+1, drop_remainder=True)
```

Create example and target:

```
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text

dataset = sequences.map(split_input_target)
```

Shuffle and batch data:

```
# Batch size
BATCH_SIZE = 64
steps_per_epoch = examples_per_epoch//BATCH_SIZE

# Buffer size to shuffle the dataset
```

```
# (TF data is designed to work with possibly infinite sequences,
# so it doesn't attempt to shuffle the entire sequence in memory. Instead,
# it maintains a buffer in which it shuffles elements).
BUFFER_SIZE = 10000

dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
```

Building model:

```
# Length of the vocabulary in chars
vocab_size = len(vocab)

# The embedding dimension
embedding_dim = 256

# Number of RNN units
rnn_units = 1024
```

Note, CuDNNGRU expects running on GPU:

```
if tf.test.is_gpu_available():
  rnn = tf.keras.layers.CuDNNGRU
else:
  import functools
  rnn = functools.partial(
    tf.keras.layers.GRU, recurrent_activation='sigmoid')


def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
  model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              batch_input_shape=[batch_size, None]),
    rnn(rnn_units,
        return_sequences=True,
        recurrent_initializer='glorot_uniform',
        stateful=True),
    tf.keras.layers.Dense(vocab_size)
  ])
  return model


model = build_model(
  vocab_size = len(vocab),
  embedding_dim=embedding_dim,
  rnn_units=rnn_units,
  batch_size=BATCH_SIZE)
```

Get prediction:

```
for input_example_batch, target_example_batch in dataset.take(1):
  example_batch_predictions = model(input_example_batch)
```

Get indices from distribution:

```
sampled_indices = tf.random.categorical(example_batch_predictions[0], num_samples=1)
sampled_indices = tf.squeeze(sampled_indices,axis=-1).numpy()
```

Lost function:

```
def loss(labels, logits):
  return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)
```

Compile:

```
model.compile(
    optimizer = tf.train.AdamOptimizer(),
    loss = loss)
```

Callbacks:

```
# Directory where the checkpoints will be saved
checkpoint_dir = './training_checkpoints'
# Name of the checkpoint files
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")

checkpoint_callback=tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_prefix,
    save_weights_only=True)
```

Train:

```
EPOCHS=3


history = model.fit(dataset.repeat(), epochs=EPOCHS, steps_per_epoch=steps_per_epoch,
callbacks=[checkpoint_callback])
```

Get latest checkpoint:

```
model = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)

model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))

model.build(tf.TensorShape([1, None]))
```

Generate sample text:

```
def generate_text(model, start_string):
  # Evaluation step (generating text using the learned model)

  # Number of characters to generate
```

```
  num_generate = 1000

  # Converting our start string to numbers (vectorizing)
  input_eval = [char2idx[s] for s in start_string]
  input_eval = tf.expand_dims(input_eval, 0)

  # Empty string to store our results
  text_generated = []

  # Low temperatures results in more predictable text.
  # Higher temperatures results in more surprising text.
  # Experiment to find the best setting.
  temperature = 1.0

  # Here batch size == 1
  model.reset_states()
  for i in range(num_generate):
      predictions = model(input_eval)
      # remove the batch dimension
      predictions = tf.squeeze(predictions, 0)

      # using a multinomial distribution to predict the word returned by the model
      predictions = predictions / temperature
      predicted_id = tf.multinomial(predictions, num_samples=1)[-1,0].numpy()

      # We pass the predicted word as the next input to the model
      # along with the previous hidden state
      input_eval = tf.expand_dims([predicted_id], 0)

      text_generated.append(idx2char[predicted_id])

  return (start_string + ''.join(text_generated))


print(generate_text(model, start_string=u"ROMEO: "))
```

Custom training: first, initialize the RNN state. We do this by calling the tf.keras.Model.reset_states method. Reset_states clear hidden states, which is important for GRU and LSTM which has hidden_states that capture information from surrounding inputs.

```
model = build_model(
  vocab_size = len(vocab),
  embedding_dim=embedding_dim,
  rnn_units=rnn_units,
  batch_size=BATCH_SIZE)


optimizer = tf.train.AdamOptimizer()
```

```
# Training step
EPOCHS = 1

for epoch in range(EPOCHS):
    start = time.time()

    # initializing the hidden state at the start of every epoch
    # initially hidden is None
    hidden = model.reset_states()

    for (batch_n, (inp, target)) in enumerate(dataset):
        with tf.GradientTape() as tape:
            # feeding the hidden state back into the model
            # This is the interesting step
            predictions = model(inp)
            loss = tf.losses.sparse_softmax_cross_entropy(target, predictions)

        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        if batch_n % 100 == 0:
            template = 'Epoch {} Batch {} Loss {:.4f}'
            print(template.format(epoch+1, batch_n, loss))

    # saving (checkpoint) the model every 5 epochs
    if (epoch + 1) % 5 == 0:
      model.save_weights(checkpoint_prefix.format(epoch=epoch))

    print ('Epoch {} Loss {:.4f}'.format(epoch+1, loss))
    print ('Time taken for 1 epoch {} sec\n'.format(time.time() - start))

model.save_weights(checkpoint_prefix.format(epoch=epoch))
```

Neural Machine Translation with Attention

Skipped because I should learn this when I want to actually built it. For now, focus on getting basics of Tensorflow down.

Image Captioning

Skipped for same reason as above.

Recurrent Neural Network with LSTM

Hard to follow. Look else where for tutorial.

Drawing Classification

Skipped for same reason as NN translation with Attention.

Simple Audio Recognition

Literally tells you have to use the provided files. What I want is to build a model. This section does not provide that.

Load Images

Setup:

```
from __future__ import absolute_import, division, print_function, unicode_literals

import tensorflow as tf
tf.enable_eager_execution()
AUTOTUNE = tf.data.experimental.AUTOTUNE # Sets number of elements to prefetch
import pathlib
data_root_orig = tf.keras.utils.get_file('flower_photos',
                                         'https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz', untar=True) # Untar as tgz files are a TAR Archive.

‘’’
The folder contained in the path data_root_orig has more folders, one for each label. The names of those folders are the labels. Inside label folders are the actual images.
‘’’

data_root = pathlib.Path(data_root_orig)
```

Flower photos now in data_root.

```
import random
all_image_paths = list(data_root.glob('*/*')) # Gets all the images. * is a greedy any regex.
all_image_paths = [str(path) for path in all_image_paths] # Converts to string path.
random.shuffle(all_image_paths)
```

Determine labels for each image:

```
# Gets labels, which are folder names.
label_names = sorted(item.name for item in data_root.glob('*/') if item.is_dir())
```

Assign index to label. For example, “daisy” ➔ 1:

```
label_to_index = dict((name, index) for index,name in enumerate(label_names))
```

Create list of every file and its label index:

```
all_image_labels = [label_to_index[pathlib.Path(path).parent.name]
                    for path in all_image_paths] # Gets labels for all images. Order of images and labels match up.
```

Print raw data:

```
img_path = all_image_paths[0]
img_raw = tf.io.read_file(img_path)
print(repr(img_raw)[:100]+"...")
```

Get image tensor:

```
img_tensor = tf.image.decode_image(img_raw) # Before, images where paths
```

Resize:

```
img_final = tf.image.resize(img_tensor, [192, 192])
img_final = img_final/255.0
```

Put neatly in function:

```
def preprocess_image(image):
  image = tf.image.decode_jpeg(image, channels=3)
  image = tf.image.resize(image, [192, 192])
  image /= 255.0  # normalize to [0,1] range

  return image

def load_and_preprocess_image(path):
  image = tf.read_file(path)
  return preprocess_image(image)
```

Build tf.data.Dataset:

```
path_ds = tf.data.Dataset.from_tensor_slices(all_image_paths)
```

Load and format all dataset image paths:

```
image_ds = path_ds.map(load_and_preprocess_image, num_parallel_calls=AUTOTUNE)
```

Dataset of labels:

```
label_ds = tf.data.Dataset.from_tensor_slices(tf.cast(all_image_labels, tf.int64))
```

Since labels and images datasets are in the same order, use zip to get image and labels:

```
image_label_ds = tf.data.Dataset.zip((image_ds, label_ds))
```

Another method to create the dataset:

```
ds = tf.data.Dataset.from_tensor_slices((all_image_paths, all_image_labels))

# The tuples are unpacked into the positional arguments of the mapped function
def load_and_preprocess_from_path_label(path, label):
  return load_and_preprocess_image(path), label

image_label_ds = ds.map(load_and_preprocess_from_path_label)
```

Shuffling, batching, repeating, and prefetching data:

```
BATCH_SIZE = 32
```

```
# buffer_size shuffles only that many data.
# Setting a shuffle buffer size as large as the dataset ensures that the entire data is shuffled.
ds = image_label_ds.shuffle(buffer_size=image_count)
ds = ds.repeat() # Items can be seen multiple times.
ds = ds.batch(BATCH_SIZE)
# `prefetch` lets the dataset fetch batches, in the background while the model is training.
ds = ds.prefetch(buffer_size=AUTOTUNE)
```

- The repeat() method allows the dataset to be reinitialize after it has been completely read. Further, the order of shuffle, repeat, and batch matter.

Merge shuffle and repeat function:

```
ds = image_label_ds.apply(
  tf.data.experimental.shuffle_and_repeat(buffer_size=image_count))
ds = ds.batch(BATCH_SIZE)
ds = ds.prefetch(buffer_size=AUTOTUNE)
```

Pipe dataset to a model. But first, create the model:

```
mobile_net = tf.keras.applications.MobileNetV2(input_shape=(192, 192, 3), include_top=False)
mobile_net.trainable=False
```

Using the following method, you know that the input should be between [-1, 1]:

```
help(keras_applications.mobilenet_v2.preprocess_input)
```

Adjust data:

```
def change_range(image,label):
  return 2*image-1, label

keras_ds = ds.map(change_range)
```

Next build model, compile it, and train:

```
model = tf.keras.Sequential([
  mobile_net,
  tf.keras.layers.GlobalAveragePooling2D(),
  tf.keras.layers.Dense(len(label_names))])

model.compile(optimizer=tf.train.AdamOptimizer(),
              loss=tf.keras.losses.sparse_categorical_crossentropy,
              metrics=["accuracy"])

model.fit(ds, epochs=1, steps_per_epoch=tf.ceil(len(all_image_paths)/BATCH_SIZE))

# I believe ds should be keras_ds.
```

This simple input pipeline above is slow and only sufficient on CPU. However, before continuing on enhancing performance, examine if loading input data is the bottleneck.

Speed up shuffling, repeating, and prefetching with cache for other epochs to use:

```
ds = image_label_ds.cache()
ds = ds.apply(
  tf.data.experimental.shuffle_and_repeat(buffer_size=image_count))
ds = ds.batch(BATCH_SIZE).prefetch(buffer_size=AUTOTUNE)
```

Cache must be built on first run though.

If data doesn't fit in memory, use cache file:

```
ds = image_label_ds.cache(filename='./cache.tf-data')
ds = ds.apply(
  tf.data.experimental.shuffle_and_repeat(buffer_size=image_count))
ds = ds.batch(BATCH_SIZE).prefetch(1)
```

No need to rebuild with cache file but it is slower than cache in memory.

This is new information! Can also save preprocessed dataset as TFRecord. But let's start with something simple:

Save unpreprocessed images in TFRecord:

```
image_ds = tf.data.Dataset.from_tensor_slices(all_image_paths).map(tf.read_file)
tfrec = tf.data.experimental.TFRecordWriter('images.tfrec')
tfrec.write(image_ds) # Might need to close stream.
```

Read in TFRecord:

```
image_ds = tf.data.TFRecordDataset('images.tfrec').map(preprocess_image)

# To get image, label pair

ds = tf.data.Dataset.zip((image_ds, label_ds))
ds = ds.apply(
  tf.data.experimental.shuffle_and_repeat(buffer_size=image_count))
ds=ds.batch(BATCH_SIZE).prefetch(AUTOTUNE)
```

In this case, the preprocessed Tensors are not saved in TFRecord, so read in is slowed down. To save preprocessed Tensors:

```
paths_ds = tf.data.Dataset.from_tensor_slices(all_image_paths)
image_ds = paths_ds.map(load_and_preprocess_image)

ds = image_ds.map(tf.serialize_tensor) # Converts tensors to strings


tfrec = tf.data.experimental.TFRecordWriter('images.tfrec')
tfrec.write(ds)
```

Read in and deserialize:

```
ds = tf.data.TFRecordDataset('images.tfrec')


def parse(x):
```

```
  result = tf.parse_tensor(x, out_type=tf.float32)
  result = tf.reshape(result, [192, 192, 3])
  return result

ds = ds.map(parse, num_parallel_calls=AUTOTUNE)
```

Add labels:

```
ds = tf.data.Dataset.zip((ds, label_ds))
ds = ds.apply(
  tf.data.experimental.shuffle_and_repeat(buffer_size=image_count))
ds=ds.batch(BATCH_SIZE).prefetch(AUTOTUNE)
```

TFRecords and tf.example

Important for storing and reading in data using TFRecords. Will be important. Can only serialize and store models right now. Serializing processed data will be important.

Vector Representation of Words

Word2Vec article. Builds a simple Word2Vec model. Not useful.

Kernal Methods

This is for people with SVM background

Large-scale Linear Model

A one hidden layer model.

Unicode

Important for storing characters from foreign languages of emojis.

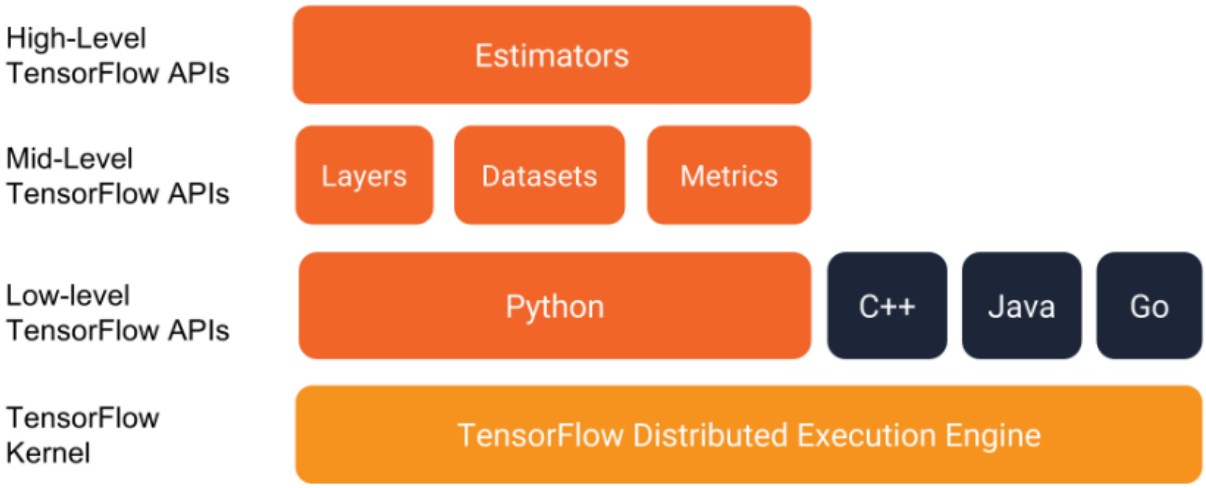
Loading From Model Checkpoint (Double Check Section)

```
Model_name.ckpt.data-00000-of-00001
Model_name.ckpt.index
Model_name.ckpt.meta
```

When saving a model checkpoint, three files will get saved. To load a model, all three files must be in the same directory and then you can simply load the "*name*.ckpt" file.

Estimators from GUIDE

Keras layers module is considered lower level than estimators. TensorFlow made creating computational graphics easy with Estimators; a pre-made Estimator has a pre-made model while a custom Estimator requires the user to build his/her own. It handles the details of initialization, logging, saving and restoring, and many other features so you can concentrate on your model.



Premade Estimators

To write a TensorFlow program based on pre-made Estimators, you must perform the following tasks:

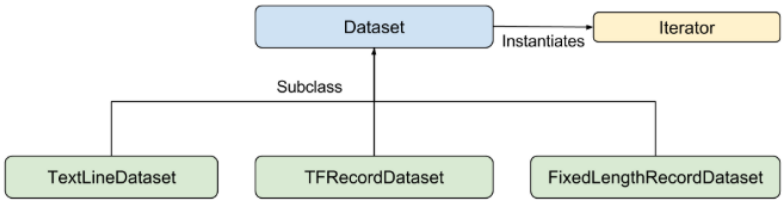
- Create one or more input functions.
- Define the model's feature columns.
- Instantiate an Estimator, specifying the feature columns and various hyperparameters.
- Call one or more methods on the Estimator object, passing the appropriate input function as the source of the data.

Let's complete those following steps now, starting with the input function. An **input function** returns a tf.data.Dataset object which outputs the following two-element tuple:

- features - A Python dictionary in which:
  - Each key is the name of a feature.
  - Each value is an array containing all of that feature's values.
- label

An example USELESS input function:

```
def input_evaluation_set():
    features = {'SepalLength': np.array([6.4, 5.0]),
                'SepalWidth':  np.array([2.8, 2.3]),
                'PetalLength': np.array([5.6, 3.3]),
                'PetalWidth':  np.array([2.2, 1.0])}
    labels = np.array([2, 1]) # Two labels b/c only 2 examples.
    return features, labels
```



Pre-made functions to parse data.

Input function for training:

```
def train_input_fn(features, labels, batch_size):
    """An input function for training"""
    # Convert the inputs to a Dataset.
    dataset = tf.data.Dataset.from_tensor_slices((dict(features), labels))

    # Shuffle, repeat, and batch the examples.
    return dataset.shuffle(1000).repeat().batch(batch_size)
```

When you build an Estimator model, you pass it a list of feature columns that describes each of the features you want the model to use. The numeric_column tells the Estimator to use 32float for the values in each column:

```
# Feature columns describe how to use the input.
my_feature_columns = []
for key in train_x.keys():
    my_feature_columns.append(tf.feature_column.numeric_column(key=key))
```

Instantiate an Estimator. The link, https://www.tensorflow.org/api_docs/python/tf/estimator, provides list of pre-made Estimators.

```
# Build a DNN with 2 hidden layers and 10 nodes in each hidden layer.
classifier = tf.estimator.DNNClassifier(
    feature_columns=my_feature_columns,
    # Two hidden layers of 10 nodes each.
    hidden_units=[10, 10],
    # The model must choose between 3 classes.
    n_classes=3)
```

We can now train, evaluate, or predict. Let's train first:

```
# Train the Model.
classifier.train(
    input_fn=lambda:iris_data.train_input_fn(train_x, train_y, args.batch_size),
    steps=args.train_steps)
```

Must use lambda as input_fn must have no arguments.

Evaluate the trained model:

```
# Evaluate the model.
eval_result = classifier.evaluate(
    input_fn=lambda:iris_data.eval_input_fn(test_x, test_y, args.batch_size))

print('\nTest set accuracy: {accuracy:0.3f}\n'.format(**eval_result))
```

The eval_result dictionary also contains the average_loss (mean loss per sample), the loss (mean loss per mini-batch) and the value of the estimator's global_step (the number of training iterations it underwent).

Making predictions:

```
predict_x = {
    'SepalLength': [5.1, 5.9, 6.9],
    'SepalWidth': [3.3, 3.0, 3.1],
    'PetalLength': [1.7, 4.2, 5.4],
    'PetalWidth': [0.5, 1.5, 2.1],
}


# Returns dictionary for each input. “class_ids” is one key and “probability” is second key. Both
# keys return a list.
predictions = classifier.predict(
    input_fn=lambda:iris_data.eval_input_fn(predict_x, batch_size=args.batch_size))
```

Checkpoints from Estimators

This will focus on saving checkpoints (model dependent; will save all variables but not actual model), not saving models (model independent; will save everything to replicate saved model).

Estimators automatically create checkpoints and event files (used for TensorBoard). Specific top-level directory to stored these files as follows:

```
classifier = tf.estimator.DNNClassifier(
    feature_columns=my_feature_columns,
    hidden_units=[10, 10],
    n_classes=3,
    model_dir='models/iris') # ← specify directory here.
```

If model_dir is not specified, a directory created by Python’s tempfile.mkdtemp function.

By default, the Estimator saves checkpoints in the directory according to the following schedule:

- Writes a checkpoint every 10 minutes.
- Writes a checkpoint when the train method starts (first iteration) and completes (final iteration).
- Retains only the 5 most recent checkpoints in the directory.

Alter schedule by:

1. Creating a tf.estimator.RunConfig object that defines the desired schedule.
2. When instantiating the Estimator, pass that RunConfig object to the Estimator's config argument.

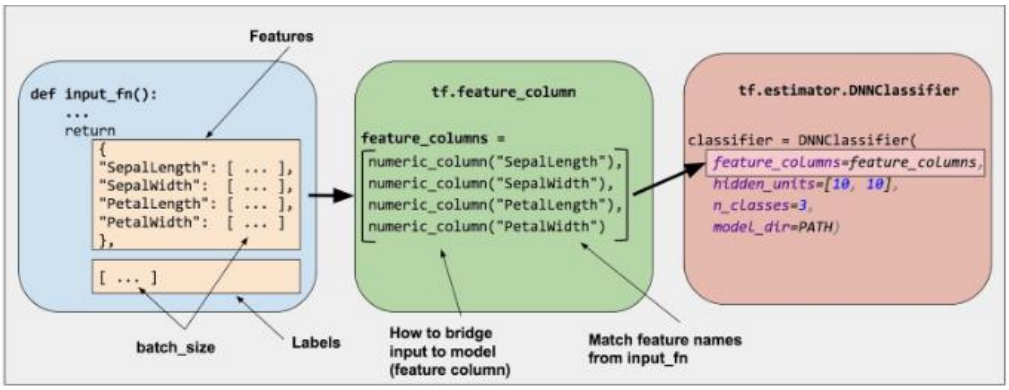
```
my_checkpointing_config = tf.estimator.RunConfig(
    save_checkpoints_secs = 20*60,  # Save checkpoints every 20 minutes.
    keep_checkpoint_max = 10,       # Retain the 10 most recent checkpoints.
)

classifier = tf.estimator.DNNClassifier(
    feature_columns=my_feature_columns,
    hidden_units=[10, 10],
    n_classes=3,
    model_dir='models/iris',
    config=my_checkpointing_config)
```

Each call to the Estimator's train, evaluate, or predict method builds model's graph from model_fn(), then initializes weights of the new model either from scratch (using default an algorithm) or from most recent checkpoint.

To compare slightly different versions of a model (say add more hidden neurons), save code to create the model and its checkpoints. One method is to save them in different branches in GIT.

Feature Columns



Note that feature_columns must store feature_column objects. This image spans creating data, creating feature_column, initializing model.

Categorical data is usually represented as a one hot vector. Let's visit 6 functions from tf.feature_column:

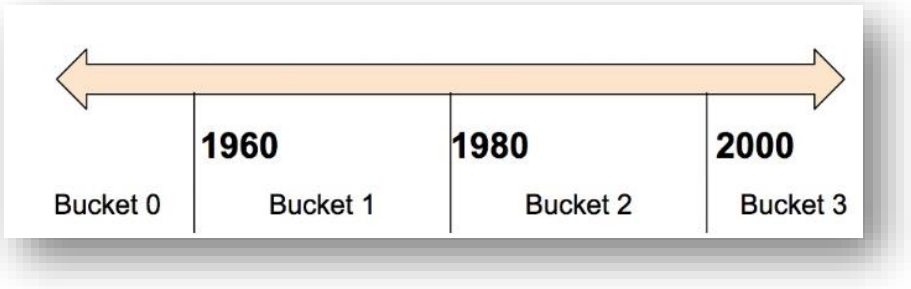
Numeric_column:

```
# Each entry for SepalLength is of shape [10, 5] (default shape is [1,]) where each cell is a tf.float64 scalar
#(default is folat32).
```

```
numeric_feature_column = tf.feature_column.numeric_column(key="SepalLength", dtype=tf.float64, shape = [10 ,5])
```

Bucketized Column:

Split the year a house was built into buckets as follows:



Each input will then be represented as one hot vector like [1, 0, 0, 0]. Why do this? This method uses 4 weights as oppose to 1. This creates a more expressive model. Create bucketized feature as follows:

```
# First, convert the raw input to a numeric column.
numeric_feature_column = tf.feature_column.numeric_column("Year")

# Then, bucketize the numeric column on the years 1960, 1980, and 2000.
bucketized_feature_column = tf.feature_column.bucketized_column(
    source_column = numeric_feature_column,
    boundaries = [1960, 1980, 2000])
```

Categorical Identity Column:

Each bucket represents one value as oppose to a range in bucketized column. Data is still converted to one hot-vector for model. Note, in categorical identity column, data must by integers.

```
# Create categorical output for an integer feature named "my_feature_b",
# The values of my_feature_b must be >= 0 and < num_buckets
identity_feature_column = tf.feature_column.categorical_column_with_identity(
    key='my_feature_b',
    num_buckets=4) # Values [0, 4)

# In order for the preceding call to work, the input_fn() must return
# a dictionary containing 'my_feature_b' as a key. Furthermore, the values
# assigned to 'my_feature_b' must belong to the set [0, 4).
def input_fn():
    ...
    return ({ 'my_feature_a':[7, 9, 5, 2], 'my_feature_b':[3, 1, 2, 2] },
            [Label_values])
```

Categorical Vocabulary Column:

This function creates a one-hot vector very similar to categorical identity column. The only difference is that the actual data can be strings. The feature_column will generate one hot vector from the strings.

There are two functions available for this: tf.feature_column.categorical_column_with_vocabulary_list and tf.feature_column.categorical_column_with_vocabulary_file. The list version allows you to directly type the unique vocabs for bucketing while the file version reads from a file:

```
vocabulary_feature_column =
    tf.feature_column.categorical_column_with_vocabulary_list(
        key=feature_name_from_input_fn,
        vocabulary_list=["kitchenware", "electronics", "sports"])

vocabulary_feature_column =
    tf.feature_column.categorical_column_with_vocabulary_file(
        key=feature_name_from_input_fn,
        vocabulary_file="product_class.txt", # Should contain one vocab per line.
        vocabulary_size=3) # Three vocab in file.
```

Hashed Column:

Rather than assign buckets manually for each vocab, the hash column will automatically hash the string into a bucket. All you have to do is determine how many buckets you are willing to have:

```
hashed_feature_column =
    tf.feature_column.categorical_column_with_hash_bucket(
        key = "some_feature",
        hash_bucket_size = 100) # The number of categories
```

Collisions are not that big of a problem.

Crossed Column:

This method combines multiple features into a single feature. For example, suppose we want to estimate pricing in Atlanta, GA. Suppose we represent Atlanta, GA as a 100x100 grid, if we have feature columns for latitude and longitude to pinpoint a cell, we can cross them to form 10,000 sections to pass into the model.

```
def make_dataset(latitude, longitude, labels):
    assert latitude.shape == longitude.shape == labels.shape

    features = {'latitude': latitude.flatten(),
                'longitude': longitude.flatten()}
    labels=labels.flatten()

    return tf.data.Dataset.from_tensor_slices((features, labels))


# Bucketize the latitude and longitude using the `edges`
```

```
latitude_bucket_fc = tf.feature_column.bucketized_column(
    tf.feature_column.numeric_column('latitude'),
    list(atlanta.latitude.edges)) # atlanta.latitude.edges represents the edges to make the 100x100 grid.

longitude_bucket_fc = tf.feature_column.bucketized_column(
    tf.feature_column.numeric_column('longitude'),
    list(atlanta.longitude.edges))

# Cross the bucketized columns, using 5000 hash bins. A pair of 5000 possibilities are created.
crossed_lat_lon_fc = tf.feature_column.crossed_column(
    [latitude_bucket_fc, longitude_bucket_fc], 5000)

fc = [
    latitude_bucket_fc,
    longitude_bucket_fc,
    crossed_lat_lon_fc]

# Build and train the Estimator.
est = tf.estimator.LinearRegressor(fc, ...)
```

You may create a feature cross from either of the following:

- Feature names; that is, names from the dict returned from input_fn.
- Any categorical column,
  except categorical_column_with_hash_bucket (since crossed_column hashes the input).

When creating feature crosses, you typically still should include the original (uncrossed) features in your model.

I believe crossing numerical column is not helpful as it's the same as providing the uncross columns into the model.

IMPORTANT: as of now, all categorical feature columns are stored as a sparse one-hot vector, meaning they can be stored efficiently but the entire vector is not stored. Not all models accept this representation of vectors, thus, indicator columns can be used to create a dense feature column from a sparse one. The dense version now stores all values in the vector.

Create indicator column:

```
# Represent the categorical column as an indicator column.
indicator_column = tf.feature_column.indicator_column(categorical_column)
```

Embedding Column:

Dataset for Estimators

In TensorFlow, arguments expecting an "array" can accept nearly anything that can be converted to an array with numpy.array except for tuples, which have a special meaning.

The tf.data.Dataset.from_tensor_slices function creates a tf.data.Dataset representing slices of the array. The array is sliced across the first dimension. Thus, 60,000x28x28 passed to from_tensor_slices would produce 60,000 tensors each representing a 28x28 array. Note that a Dataset can slice the first dimension in an array even if it is nested in any combination of dictionaries or tuples. For example, a dictionary where each key holds a 100 element array would get sliced to 100 dictionaries with each key holding 1 element (the dictionaries would be represented as a tensor however).

In eager mode, you can iterate through dictionary with iterator.

Know shuffle, repeat, and batch functions.

To read data from a text file:

```
ds = tf.data.TextLineDataset(train_path).skip(1)
```

The TextLineDataset is an object to read the file one line at a time. The skip method skips line 1.

The following takes a line and parses it as a CSV:

```
# Metadata describing the text columns
COLUMNS = ['SepalLength', 'SepalWidth',
         'PetalLength', 'PetalWidth',
         'label']
FIELD_DEFAULTS = [[0.0], [0.0], [0.0], [0.0], [0]]
def _parse_line(line):
    # Decode the line into its fields
    fields = tf.decode_csv(line, FIELD_DEFAULTS)

    # Pack the result into a dictionary
    features = dict(zip(COLUMNS,fields))

    # Separate the label from the features
    label = features.pop('label')

    return features, label
```

Apply the _parse_line function to all lines with map functions.

```
ds = ds.map(_parse_line)
```

Creating Custom Estimators

Premade Estimators subclass tf.estimator.Estimator while custom Estimators are an instance of it. In custom Estimator, must write the model function.

First, get dataset with batches of (feature_dictionary, labels) pairs. Next, create feature columns. Then, we can write the model function. The functions signature is as follows:

```
def my_model_fn(
    features, # This is batch_features from input_fn
```

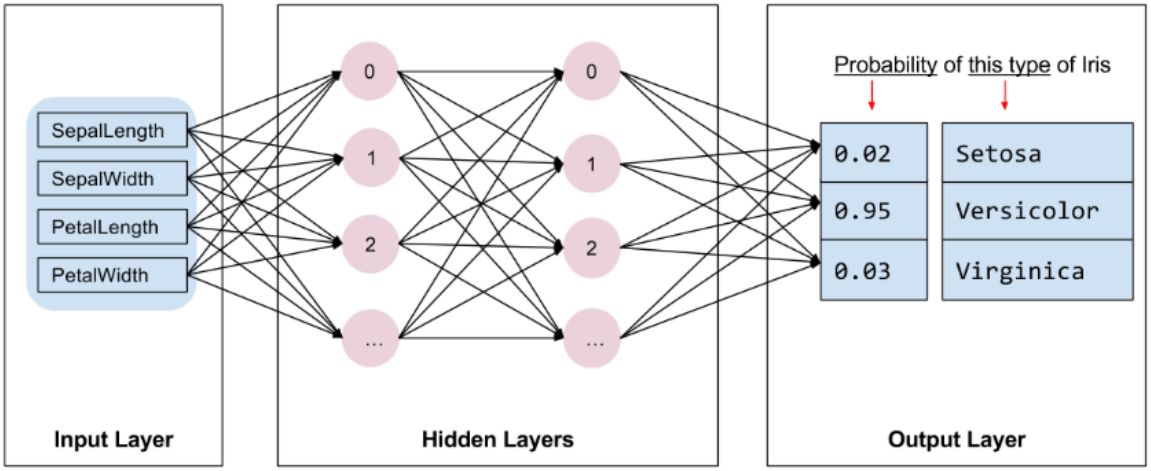
```
  labels,   # This is batch_labels from input_fn
  mode,     # An instance of tf.estimator.ModeKeys for training, evaluating, or predicting
  params):  # Additional configuration
```

The caller may pass params to an **Estimator's** constructor. Any params passed to the constructor are in turn passed on to the model_fn method. Next, create the Estimator and configure it:

```
classifier = tf.estimator.Estimator(
   model_fn=my_model_fn,
   params={
      'feature_columns': my_feature_columns,
      # Two hidden layers of 10 nodes each.
      'hidden_units': [10, 10],
      # The model must choose between 3 classes.
      'n_classes': 3,
   })
```

To implement a typical model function, you must define the model and specify calculations for predict, evaluate, and train mode.

For the model we are trying to build (



) we need to define input layer, hidden layers, and output layer.

Convert the feature dictionary and feature_columns into input for your model,:

```
# Use `input_layer` to apply the feature columns.
   net = tf.feature_column.input_layer(features, params['feature_columns'])
```

Create hidden layers:

```
 # Build the hidden layers, sized according to the 'hidden_units' param.
   for units in params['hidden_units']:
      net = tf.layers.dense(net, units=units, activation=tf.nn.relu)
```

More tf.layers objects available at https://www.tensorflow.org/api_docs/python/tf/layers.

Output layer:

```
# Compute logits (1 per class).
   logits = tf.layers.dense(net, params['n_classes'], activation=None)
```

How to call various modes (train, evaluate, predict):

```
classifier = tf.estimator.Estimator(...)
classifier.train(input_fn=lambda: my_input_fn(FILE_TRAIN, True, 500))
```

The preceding then calls model function with ModeKeys.TRAIN. Obviously, the method function must handle train, evaluate, and predict. Each mode must return a tf.estimator.EstimatorSpec.

Note, the model function is called (model is rebuilt) whenever someone calls train, eval, or predict.

Lets handle all three modes now. First, predict:

```
# Compute predictions.
predicted_classes = tf.argmax(logits, 1) # Logits is likely an array of shape (1, m)
if mode == tf.estimator.ModeKeys.PREDICT:
   predictions = {
      'class_ids': predicted_classes[:, tf.newaxis], # This is now of shape (1, 1).
      'probabilities': tf.nn.softmax(logits),
      'logits': logits,
   }
   return tf.estimator.EstimatorSpec(mode, predictions=predictions)
```

Calculate loss for evaluation and training:

```
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)
```

The tf.estimator.EstimatorSpec returned for evaluation contains the following information:

- loss, which is the model's loss
- And optionally, eval_metric_ops, which is an optional dictionary of metrics.

Calculate accuracy metric:

```
# Compute evaluation metrics.
accuracy = tf.metrics.accuracy(labels=labels,
                               predictions=predicted_classes,
                               name='acc_op')

metrics = {'accuracy': accuracy}
tf.summary.scalar('accuracy', accuracy[1]) # Makes accuracy available to TensorBoard in train and eval mode.


if mode == tf.estimator.ModeKeys.EVAL:
   return tf.estimator.EstimatorSpec(
      mode, loss=loss, eval_metric_ops=metrics)
```

When train is called, model function returns EstimatorSpec that contains loss and training operation.

The optimizer for the training operation:

```
optimizer = tf.train.AdagradOptimizer(learning_rate=0.1)
```

Now code to train the model:

```
train_op = optimizer.minimize(loss, global_step=tf.train.get_global_step())
```

global_step, which counts training steps, is important for TensorBoard and to know when to stop training.

The returned EstimatorSpec must have lost and train_op, which is the training operation.

```
return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)
```

Instantiate the custom estimator as follows:

```
# Build 2 hidden layer DNN with 10, 10 units respectively.
   classifier = tf.estimator.Estimator(
      model_fn=my_model_fn,
      params={
         'feature_columns': my_feature_columns,
         # Two hidden layers of 10 nodes each.
         'hidden_units': [10, 10],
         # The model must choose between 3 classes.
         'n_classes': 3,
      })
```

Training, evaluating, and predicting is the same as premade Estimators:

```
# Train the Model.
classifier.train(
   input_fn=lambda:iris_data.train_input_fn(train_x, train_y, args.batch_size),
   steps=args.train_steps)
```

Execute the following to use TensorBoard:

```
# Replace PATH with the actual path passed as model_dir
tensorboard --logdir=PATH
```

Then browse to http://localhost:6006.

Low Level APIs From GUIDE

Introduction

Tensors store primitives in arrays. They do not have values; they are just handlers.

Can think of TensorFlow Core in two sections: the computational graph and the session that runs it.

Computational Graph:

- Nodes are tf.Operations (or ops) that describe calculations that consume and produce tensor(s).
- Edges are tf.Tensor that represent values that flow through the graph.

A very simple computational graph:

```
a = tf.constant(3.0, dtype=tf.float32)
b = tf.constant(4.0) # also tf.float32 implicitly
total = a + b
print(a) #==> Tensor("Const:0", shape=(), dtype=float32)
print(b) #==> Tensor("Const_1:0", shape=(), dtype=float32)
print(total) #==> Tensor("add:0", shape=(), dtype=float32)
```

The outputs are not 3.0, 4.0, and 7.0 as expected because the preceding code builds the computational graph; it does not run it. The tensor objects just handle the results of the operations that will be run.

Visualize the graph with TensorBoard. First write the event file to a directory:

```
writer = tf.summary.FileWriter('.') # Specify directory to save event file
writer.add_graph(tf.get_default_graph())
writer.flush() # File will be in the format “events.out.tfevents.{timestamp}.{hostname}”
```

Command to launch TensorBoard:

```
tensorboard --logdir .
```

Then visit http://localhost:6006/#graphs.

Create a tf.Session object to run TensorFlow operations. The following creates a tf.Session object and then invokes its run method to evaluate the total tensor we created:

```
sess = tf.Session()
print(sess.run(total))
```

The Session.run backtracks through the graph and runs all the nodes that provide input to the requested output node.

Run can transparently handle any combination of tuples and dictionaries.

```
print(sess.run({'ab':(a, b), 'total':total})) # Returns {'total': 7.0, 'ab': (3.0, 4.0)}
```

Session.run recomputes graph each time.

```
vec = tf.random_uniform(shape=(3,))
out1 = vec + 1
out2 = vec + 2
```

```
print(sess.run(vec)) # Different from next line
print(sess.run(vec))
print(sess.run((out1, out2))) # out2 – out1 will always be 1
```

Session.run on a tf.Operation, which is returned by some function, will result in None.

Can have placeholders, which are promised to have a value later:

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = x + y

print(sess.run(z, feed_dict={x: 3, y: 4.5})) #==> 7.5
print(sess.run(z, feed_dict={x: [1, 3], y: [2, 4]})) #==> [ 3.  7.]
```

feed_dict can also overwrite any value a Tensor handles.

Run graph with Dataset:

```
my_data = [
   [0, 1,],
   [2, 3,],
   [4, 5,],
   [6, 7,],
]
slices = tf.data.Dataset.from_tensor_slices(my_data)
next_item = slices.make_one_shot_iterator().get_next() # Must get iterator.

while True:
  try:
    print(sess.run(next_item))
  except tf.errors.OutOfRangeError:
    break
```

If the Dataset depends on stateful operations (dataset changes with each run) you may need to initialize the iterator before using it, as shown below:

```
r = tf.random_normal([10,3])
dataset = tf.data.Dataset.from_tensor_slices(r)
iterator = dataset.make_initializable_iterator()
next_row = iterator.get_next()


sess.run(iterator.initializer)
while True:
  try:
    print(sess.run(next_row))
  except tf.errors.OutOfRangeError:
    break
```

Create layers:

```
x = tf.placeholder(tf.float32, shape=[None, 3]) # Why is it None?
linear_model = tf.layers.Dense(units=1)
y = linear_model(x)
```

The layer object inspects input to determine size of internal variables. Thus, must set shape of placeholder.

Must initialize variables now. Can initialize them individually, but the following initializes all variables in tf.GraphKeys.GLOBAL_VARIABLES **up to this point**:

```
init = tf.global_variables_initializer() # Return tf.Operations
sess.run(init)
```

Execute layer:

```
print(sess.run(y, {x: [[1, 2, 3],[4, 5, 6]]}))
```

Here's a shortcut!

```
linear_model = tf.layers.Dense(units=1)
y = linear_model(x)

# Same as

y = tf.layers.dense(x, units=1)
```

Easiest way to use feature columns is to use tf.feature_column.input_layer(), which only accept dense columns as inputs. Thus, categorical column must be wrapped in indicator_column.

```
features = {
   'sales' : [[5], [10], [8], [9]], # Comma is used to separate dictionary entries. Mx1 shape is necessary.
   'department': ['sports', 'sports', 'gardening', 'gardening']}

department_column = tf.feature_column.categorical_column_with_vocabulary_list(
       'department', ['sports', 'gardening'])
department_column = tf.feature_column.indicator_column(department_column)

columns = [
   tf.feature_column.numeric_column('sales'),
   department_column
]

inputs = tf.feature_column.input_layer(features, columns)
```

Feature columns have internal variables that also need to be initialize. Categorical feature columns need to be initialized separately from global_variable_initializer() since they use tf.contrib.lookup:

```
var_init = tf.global_variables_initializer()
table_init = tf.tables_initializer()
```

```
sess = tf.Session()
sess.run((var_init, table_init))
```

Let's use our knowledge to train a small regression model manually. The following is the complete program:

```
x = tf.constant([[1], [2], [3], [4]], dtype=tf.float32) # numbers in brackets to create 2D array instead of 1D.
y_true = tf.constant([[0], [-1], [-2], [-3]], dtype=tf.float32)

linear_model = tf.layers.Dense(units=1)

y_pred = linear_model(x)
loss = tf.losses.mean_squared_error(labels=y_true, predictions=y_pred)

optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)

init = tf.global_variables_initializer()

sess = tf.Session()
sess.run(init)
for i in range(100):
  _, loss_value = sess.run((train, loss))
  print(loss_value)

print(sess.run(y_pred))
```

Tensors

Has two properties: data type and shape. The datatype is always known though the shape might only be known at execution.

The main Tensors are: tf.Variable, tf.constant, tf.placeholder, tf.SparseTensor. Ony tf.Variable is mutable.

Example of creating a Tensor:

```
cool_numbers  = tf.Variable([3.14159, 2.71828], tf.float32)
```

Get rank:

```
r = tf.rank(a_tensor)
# After the graph runs, r will hold the value 4.
```

Dynamically indexing a multidimensional tensor can accept scalar tensors in index brackets, [].

Indexing and slicing a tensor works as normal. To get shape of Tensor while building model, read Tensor property by printing out Tensor. Can also use a_tensor.shape to get tuple of shape.

Reshaping Tensor:

```
rank_three_tensor = tf.ones([3, 4, 5])
matrixB = tf.reshape(matrix, [3, -1])  #  Reshape existing content into a 3x20
```

Cast tensors with:

```
float_tensor = tf.cast(tf.constant([1, 2, 3]), dtype=tf.float32)
```

When converting Python object to tensors, can specify type, otherwise automatically given.

.eval() and .Print() was discussed. However, much easier to run a session to print it.

Variables

Note, tf.Variable is mutable. tf.Variable exists outside the context of a single session.run call.

Modifications on tf.Variables are visible across multiple tf.Sessions, so multiple workers can see the same values for a tf.Variable.

To create a tf.Variable:

```
my_variable = tf.get_variable("my_variable", [1, 2, 3], dtype=tf.int32,
  initializer=tf.zeros_initializer)

 )
```

default dtype is float32 and default initializer is tf.glorot_uniform_initializer.

Can also initialize from a tf.Tensor:

```
other_variable = tf.get_variable("other_variable", dtype=tf.int32,
  initializer=tf.constant([23, 42]))
```

every tf.Variable gets placed in the following two collections:

- tf.GraphKeys.GLOBAL_VARIABLES --- variables that can be shared across multiple devices,
- tf.GraphKeys.TRAINABLE_VARIABLES --- variables for which TensorFlow will calculate gradients.

If you don't want a variable to be trainable, add it another collection as follows:

```
my_local = tf.get_variable("my_local", shape=(),
collections=[tf.GraphKeys.LOCAL_VARIABLES])
```

Or specify trainable = false:

```
my_non_trainable = tf.get_variable("my_non_trainable",
                           shape=(),
                           trainable=False)
```

Create own collection:

```
tf.add_to_collection("my_collection_name", my_local)
```

Retrieve all variables from a collection:

```
tf.get_collection("my_collection_name")
```

Device placement is discussed here; placing variables in certain GPUs. This is out of my scope for now.

Call tf.global_variables_initializer() to initialize all variables in the tf.GraphKeys.GLOBAL_VARIABLES collection.

```
session.run(tf.global_variables_initializer())
```

Initialize one variable:

```
session.run(my_variable.initializer)
```

Prints uninitialized variables:

```
print(session.run(tf.report_uninitialized_variables()))
```

Since global initializer does not initialize in a particular order, use the following if one variable depends another:

```
v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
w = tf.get_variable("w", initializer=v.initialized_value() + 1) # Note the v.initialized_value()
```

Can use tf.Variable like any other tf.Tensor. tf.Variable will simply get converted to tf.Tensor.

Assign or assign_add to tf.Variables as follows:

```
v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
assignment = v.assign_add(1)
tf.global_variables_initializer().run()
sess.run(assignment)  # or assignment.op.run(), or assignment.eval()
```

Force re-read of variable to ensure correct value (pretty confusing):

```
v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
assignment = v.assign_add(1)
with tf.control_dependencies([assignment]):
  w = v.read_value()  # w is guaranteed to reflect v's value after the
                      # assign_add operation.
```

If a function uses variables and the function is called multiple times, it is important to put each function call in its own scope as TensorFlow would not know to reuse or create new tf.Variables as the function is called repeatedly:

```
def my_image_filter(input_images):
  with tf.variable_scope("conv1"):
    # Variables created here will be named "conv1/weights", "conv1/biases".
    relu1 = conv_relu(input_images, [5, 5, 32, 32], [32])
  with tf.variable_scope("conv2"):
    # Variables created here will be named "conv2/weights", "conv2/biases".
    return conv_relu(relu1, [5, 5, 32, 32], [32])
```

Or reuse variables:

```
with tf.variable_scope("model"):
 output1 = my_image_filter(input1)
with tf.variable_scope("model", reuse=True):
 output2 = my_image_filter(input2)
```

same as:

```
with tf.variable_scope("model") as scope:
 output1 = my_image_filter(input1)
 scope.reuse_variables()
 output2 = my_image_filter(input2)
```

Graphs and Sessions

Name scope; all tf.Tensors and tf.Operations accept optional name argument:

```
c_0 = tf.constant(0, name="c")  # => operation named "c"

# Already-used names will be "uniquified".
c_1 = tf.constant(2, name="c")  # => operation named "c_1"

# Name scopes add a prefix to all operations created in the same context.
with tf.name_scope("outer"):
 c_2 = tf.constant(2, name="c")  # => operation named "outer/c"

 # Name scopes nest like paths in a hierarchical file system.
 with tf.name_scope("inner"):
  c_3 = tf.constant(3, name="c")  # => operation named "outer/inner/c"

 # Exiting a name scope context will return to the previous prefix.
 c_4 = tf.constant(4, name="c")  # => operation named "outer/c_1"

 # Already-used name scopes will be "uniquified".
 with tf.name_scope("inner"):
  c_5 = tf.constant(5, name="c")  # => operation named "outer/inner_1/c"
```

Name scope reduces visual complexity of the graph.

NOTE, all functions have a default graph they belong to.

Can place operation in different devices. This is out of my scope for now. For now, now that I can use “tf.device(tf.train.replica_device_setter(ps_tasks=3))” to apply a heuristic for device placement:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=3)):
  # tf.Variable objects are, by default, placed on tasks in "/job:ps" in a
  # round-robin fashion.
  w_0 = tf.Variable(...)  # placed on "/job:ps/task:0"
  b_0 = tf.Variable(...)  # placed on "/job:ps/task:1"
  w_1 = tf.Variable(...)  # placed on "/job:ps/task:2"
  b_1 = tf.Variable(...)  # placed on "/job:ps/task:0"

  input_data = tf.placeholder(tf.float32)     # placed on "/job:worker"
  layer_0 = tf.matmul(input_data, w_0) + b_0  # placed on "/job:worker"
  layer_1 = tf.matmul(layer_0, w_1) + b_1     # placed on "/job:worker"
```

TensorFlow automatically convert Tensors from numpy array or other tensor-like objects. However, the conversion occurs each time the numpy array is used, which could eat up memory. Do one conversion total with **tf.convert_to_tensor** to save memory.

Use session in with block or call tf.Session.close once you are done with the session.

Visualizing graph:

```
# Build your graph.
…
with tf.Session() as sess:
  # `sess.graph` provides access to the graph used in a `tf.Session`.
  writer = tf.summary.FileWriter("/tmp/log/...", sess.graph)

  # Perform your computation...
  for i in range(1000):
    sess.run(train_op)
    # ...

  writer.close()
```

Working with multiple graphs:

```
g_1 = tf.Graph()
with g_1.as_default():
  # Operations created in this scope will be added to `g_1`.
```

```
  c = tf.constant("Node in g_1")

  # Sessions created in this scope will run operations from `g_1`.
  sess_1 = tf.Session()

g_2 = tf.Graph()
with g_2.as_default():
  # Operations created in this scope will be added to `g_2`.
  d = tf.constant("Node in g_2")

# Alternatively, you can pass a graph when constructing a `tf.Session`:
# `sess_2` will run operations from `g_2`.
sess_2 = tf.Session(graph=g_2)

# Print all of the operations in the default graph.
g = tf.get_default_graph()
print(g.get_operations())
```

Save and Restore

Save variables to checkpoint:

```
tf.reset_default_graph()
# Create some variables.
v1 = tf.get_variable("v1", [3], initializer = tf.zeros_initializer)
v2 = tf.get_variable("v2", [5], initializer = tf.zeros_initializer)

# Add ops to save and restore only `v2` using the name "v2"
saver = tf.train.Saver({"v2": v2}) # Leave arg empty to save/restore all variables.

# Use the saver object normally after that.
with tf.Session() as sess:
  # Initialize v1 since the saver will not.
  v1.initializer.run()
  saver.restore(sess, "/tmp/model.ckpt") # use saver.save(sess, “path”) to save instead.

  print("v1 : %s" % v1.eval())
  print("v2 : %s" % v2.eval())
```

Inspect saved variables:

```
# import the inspect_checkpoint library
from tensorflow.python.tools import inspect_checkpoint as chkp

# Print all tensors in checkpoint file. Specify name of tensor and set all_tensor = false to search a
# specific tensor.
```

```
chkp.print_tensors_in_checkpoint_file("/tmp/model.ckpt", tensor_name='', all_tensors=True)

# tensor_name:  v1
# [ 1.  1.  1.]
# tensor_name:  v2
# [-1. -1. -1. -1. -1.]
```

Can save and restore models- variables, the graph, and the graph's metadata.

But first, some definitions: A **MetaGraph** is a dataflow graph, plus its associated variables, assets, and signatures. A **MetaGraphDef** is the protocol buffer representation of a MetaGraph. A **signature** is the set of inputs to and outputs from a graph. Further, Assets mean any external files that are needed for you model. Common examples are vocabularies and embedding matrices.

Example of saving model:

```
export_dir = ...
...
builder = tf.saved_model.builder.SavedModelBuilder(export_dir)
with tf.Session(graph=tf.Graph()) as sess:
  ...
# https://stackoverflow.com/questions/46513923/tensorflow-how-and-why-to-use-savedmodel
# Clears things up for me.
  builder.add_meta_graph_and_variables(sess,
                                       [tag_constants.TRAINING],
                                       signature_def_map=foo_signatures, # describes inputs and outputs.
                                       assets_collection=foo_assets, # Not mandatory to set.
                                       strip_default_attrs=True) # For forward capabilities.
...
# Add a second MetaGraphDef for inference.
with tf.Session(graph=tf.Graph()) as sess:
  ...
  builder.add_meta_graph([tag_constants.SERVING], strip_default_attrs=True)
...
builder.save()
```

Load and restore:

```
export_dir = ...
...
with tf.Session(graph=tf.Graph()) as sess:
  tf.saved_model.loader.load(sess, [tag_constants.TRAINING], export_dir)
  ...
```

There more complicated information outside my scope for a long time.

Control Flow

Normal if else statements don’t work as the condition is evaluated at graph construction time. If the condition depends on tensor1, which also depends on another tensor2, then tensor1 will not have a value at construction time.

Ragged Tensors

Skipped.

References

- https://www.tensorflow.org/tutorials