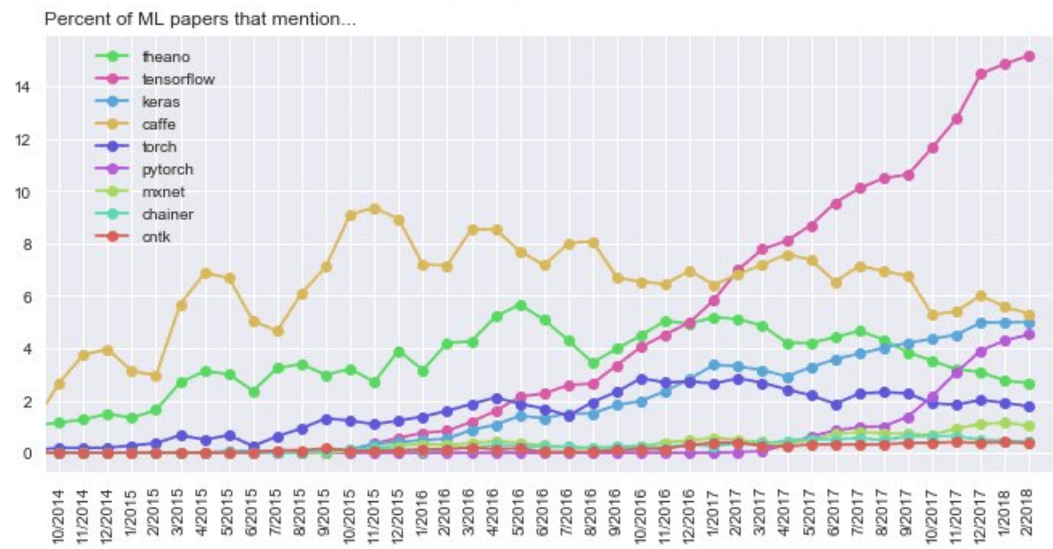
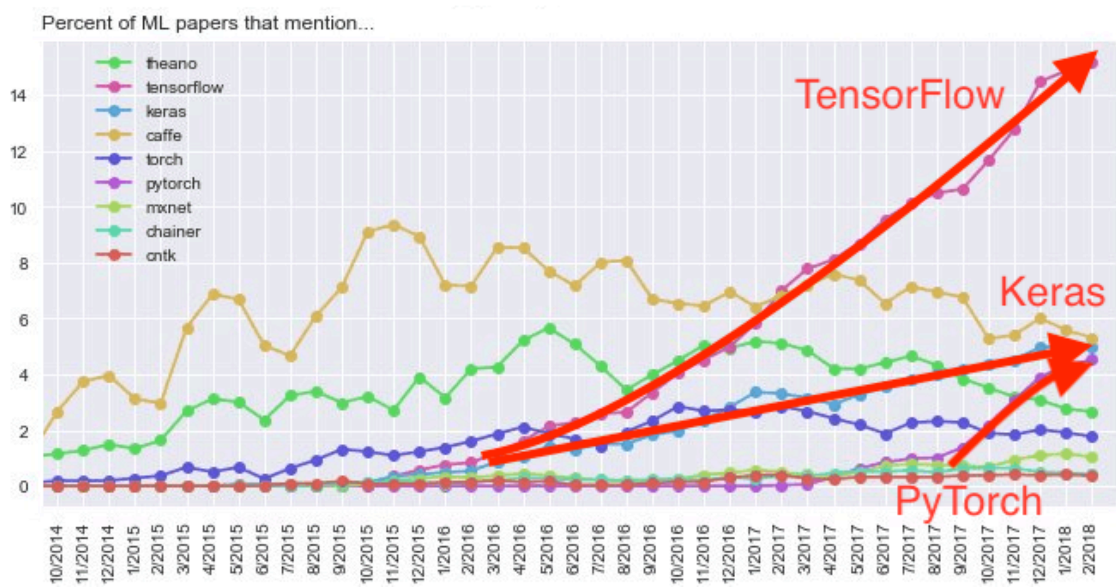
Why TensorFlow 2.0?



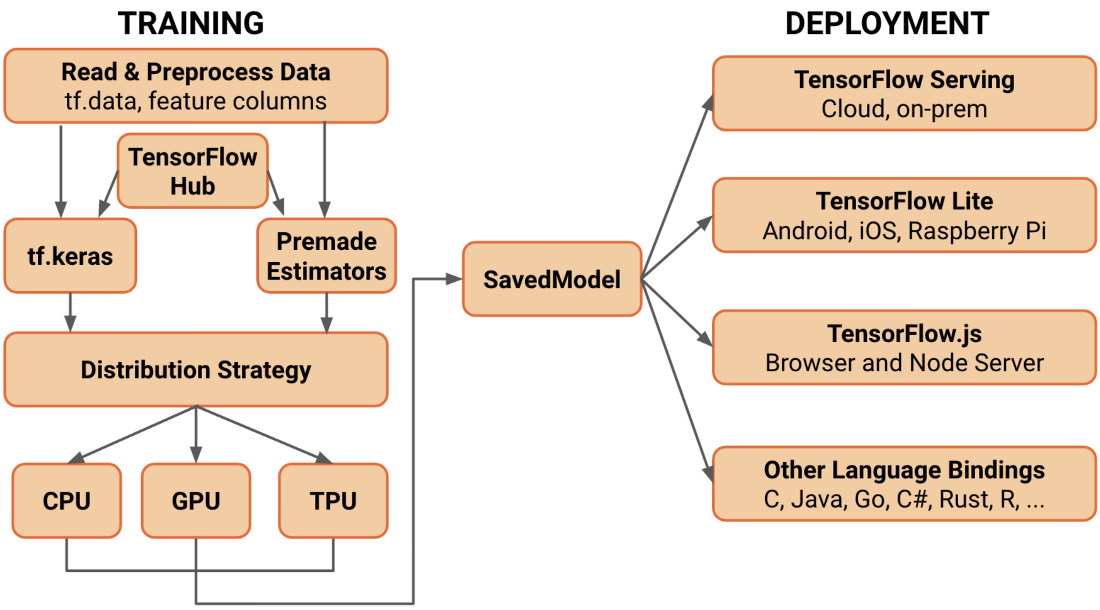
> Unique mentions of deep learning frameworks in arxiv papers (full text) over time, based on 43K ML papers over last 6 years. So far TF mentioned in 14.3% of all papers, PyTorch 4.7%, Keras 4.0%, Caffe 3.8%, Theano 2.3%, Torch 1.5%, mxnet/chainer/cntk <1%.

(Source: [Andrej Karpathy (@karpathy) 10th March 2018 (pic.twitter.com/YOYAvc33iN)](pic.twitter.com/YOYAvc33iN))



Major Changes

- API Clean-Up
- Getting rid of globals
- Eager Execution
- Functions and Autograph
- Migrating from TF1 to TF2
- Deployment



(Source: [Whats coming in TensorFlow 2.0 (https://medium.com/tensorflow/whats-coming-in-tensorflow-2-0-d3663832e9b8)](https://medium.com/tensorflow/whats-coming-in-tensorflow-2-0-d3663832e9b8))

API Clean-Up

High-Level goals:

- Add a few additional namespaces.
- Add additional endpoints for TensorFlow symbols in relevant namespaces.
- Remove some of the existing endpoints.

Examples

- `tf.math` namespace added recently, but e.g. `tf.round` is in root
- `tf.zeta` is in root but it is rarely used
- prefixes should be replaced by subnamespaces (e.g. `tf.string_strip` vs `tf.string.strip`)
- omit redundant hierarchies (e.g. flatten `tf.saved_model.signature_constants.CLASSIFY_INPUTS` to `tf.saved_model.CLASSIFY_INPUTS`)
- all layers, losses and metrics will be collected under the `tf.keras` namespace.

Additional namespaces

- **tf.random** will contain random sampling ops
- **tf.keras.layers** will contain all symbols that are currently under `tf.layers`
- **tf.keras.losses** will contain all symbols that are currently under `tf.losses`
- **tf.keras.metrics** will contain all symbols that are currently under `tf.metrics`

Already added namespaces:

- **tf.debugging** ops helpful for debugging, also TensorFlow Debugger will be moved there
- **tf.dtypes** - data types
- **tf.io** - ops for reading and writing
- **tf.quantization** - ops related to quantization

Deprecated namespaces

- **tf.logging**, **tf.app**, **tf.flags** (open-sourced as absl-py (https://abseil.io/docs/python/quickstart))
- **tf.manip** Endpoints will be kept in root
- **tf.contrib** gets removed

Namespaces and endpoints overview (https://docs.google.com/spreadsheets/d/1FLFJLzg7WNP6JHODX5q8BDgptKafq_slHpnHVbJlteQ/edit#gi

Getting rid of globals

- TensorFlow 1.X relied heavily on global namespaces
- tf.Variable() will be put into default graph and will stay there
- When loosing track, it could only be recovered when the name is known
- This is bad software-engineering practice

```
>>> v1 = tf.constant(1., name="v")
>>> v2 = tf.constant(2., name="v")
>>> del v1, v2
>>> tf.get_default_graph().get_operations()
[<tf.Operation 'v' type=Const>, <tf.Operation 'v_1' type=Const>]
>>> tf.get_default_graph().get_operation_by_name('v_2')
<tf.Operation 'v_2' type=Const>
```

([Great explanation of A. Geron (https://github.com/ageron/tf2_course/issues/8)](https://github.com/ageron/tf2_course/issues/8))

The beauty of TensorFlow 2

In [1]:
```
import tensorflow as tf
print(tf.__version__)
v1 = tf.constant(1.)
v2 = tf.constant(2.)
del v1
v1
```

```
2.0.0-alpha0

------------------------------------------------------------------
---------
NameError                                 Traceback (most recent c
all last)
<ipython-input-1-6a8893a0bc3c> in <module>
      4 v2 = tf.constant(2.)
      5 del v1
----> 6 v1

NameError: name 'v1' is not defined
```

In [2]:
```
tf.get_default_graph()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-2-7697bfee6fcc> in <module>
----> 1 tf.get_default_graph()

AttributeError: module 'tensorflow' has no attribute 'get_default_graph'
```

In [4]:
```
tf.reset_default_graph()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-c193d79f7d5a> in <module>
----> 1 tf.reset_default_graph()

AttributeError: module 'tensorflow' has no attribute 'reset_default_graph'
```

That means ...

no more global namespaces and mechanisms to find variables

- Use Keras layers or keep track of the variables yourself
- Garbage Collector takes care of lost variables
- variable_scope and get_variable will be removed
- naming will be controlled via tf.name_scope + tf.Variable

Eager Execution

- Eager execution evaluates operations immediately instead building graphs.
- That means operations return concrete values instead of constructing a computational graph to run later
- Introduced with TensorFlow 1.8
- Default mode in Version 2

https://www.tensorflow.org/guide/eager (https://www.tensorflow.org/guide/eager)

https://www.tensorflow.org/guide/keras#eager_execution (https://www.tensorflow.org/guide/keras#eager_execution)

Let's get started with eager execution

In [5]:
```
import tensorflow as tf
x = [[2.]]
m = tf.matmul(x, x)
print("{}".format(m))
```

```
[[4.]]
```

- tf.Tensor object references concrete value
- easy to inspect with print() or even a debugger
- Evaluating, printing, and checking tensor values does not break the flow for computing gradients

More fun with eager execution

Broadcasting

In [6]:
```
a = tf.constant([[1, 2],
                 [3, 4]])
print(a)

b = a + tf.constant([1])
print(b)
```

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

Operator overloading

In [8]:
```
print(a * b)
print(a + b)
```

```
tf.Tensor(
[[ 2  6]
 [12 20]], shape=(2, 2), dtype=int32)
tf.Tensor(
[[3 5]
 [7 9]], shape=(2, 2), dtype=int32)
```

Working with Numpy

In [9]:
```
import numpy as np
c = np.multiply(a, b)
print(c)
```

```
[[ 2  6]
 [12 20]]
```

Building a model the OOP way

In [10]:
```
class MNISTModel(tf.keras.Model):
  def __init__(self):
    super(MNISTModel, self).__init__()
    self.conv1 = tf.keras.layers.Conv2D(16, (3,3), padding="same",
activation='relu')
    self.conv2 = tf.keras.layers.Conv2D(16, (3,3), padding="same",
activation='relu')
    self.flatten = tf.keras.layers.Flatten()
    self.dense = tf.keras.layers.Dense(units=(10), activation='soft
max')

  def call(self, input):
    result = self.conv1(input)
    result = self.conv2(result)
    result = self.flatten(result)
    result = self.dense(result)
    return result

model = MNISTModel()
model._set_inputs(tf.keras.Input(shape=(28,28,1,)))
```

Eager training

In [11]:
```
(mnist_images, mnist_labels), _ = tf.keras.datasets.mnist.load_data
()
dataset = tf.data.Dataset.from_tensor_slices((tf.cast(mnist_images[
...,tf.newaxis]/255, tf.float32),
                                              tf.cast(mnist_labels,
tf.int64)))
dataset = dataset.shuffle(100).batch(32)
```

In [12]:
```
for images,labels in dataset.take(1):
    print("Logits: ", model(images[0:1]).numpy())
```

```
Logits:  [[0.10688005 0.10701542 0.09227316 0.11876098 0.09978637
0.09091892
  0.08603656 0.10980077 0.08442199 0.10410569]]
```

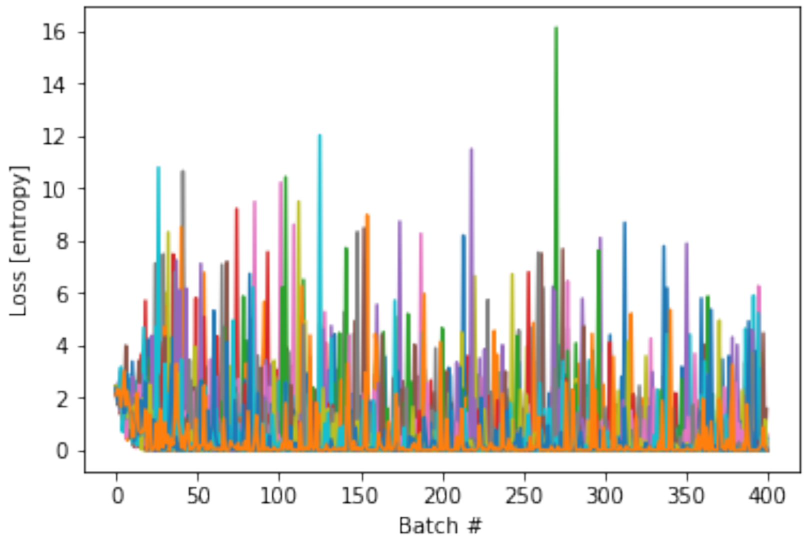
In [13]:
```
optimizer = tf.keras.optimizers.Adam()
loss_history = []
for (batch, (images, labels)) in enumerate(dataset.take(400)):
    if batch % 10 == 0:
        print('.', end='')
    with tf.GradientTape() as tape:
        logits = model(images)
        loss_value = tf.keras.losses.sparse_categorical_crossentrop
y(labels, logits)
    loss_history.append(loss_value.numpy())
    grads = tape.gradient(loss_value, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables)
)
```

```
........................................
```

- In eager mode tf.GradientTape traces operations to compute gradients later

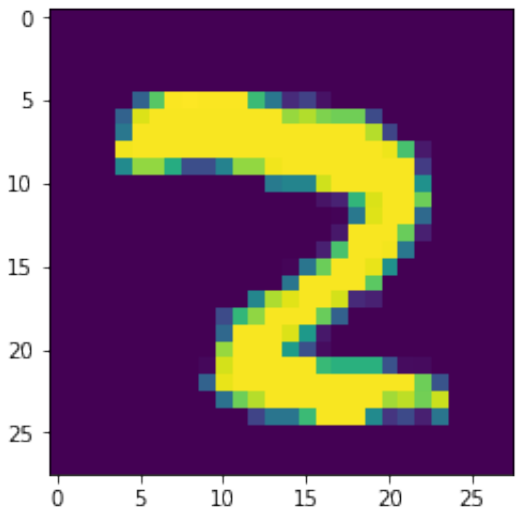
In [14]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(loss_history)
plt.xlabel('Batch #')
plt.ylabel('Loss [entropy]')
```

Out[14]:
```
Text(0, 0.5, 'Loss [entropy]')
```



In [15]:
```
for images,labels in dataset.take(1):
    print("Logits: ", model(images[0:1]).numpy().argmax())
    print(plt.imshow(images[0:1].numpy().reshape(28,28)))
tf.saved_model.save(model, "mnist_example")
```

```
Logits:  2
AxesImage(54,36;334.8x217.44)
```



tf.functions and Autograph

- Function decorator for JIT compilation
- tf.function transforms Python syntax into TensorFlow graphs
- Eager execution with all of the benefits of graph mode for performance and portability

Working with graphs

- TensorFlow graph defines all computations
- actual computation is defined by the arguments to tf.Session.run

In [17]:
```
import tensorflow as tf

with tf.compat.v1.Session() as sess:
    x = tf.compat.v1.placeholder(tf.float32)
    y = tf.square(x)
    z = tf.add(x, y)

    z0 = sess.run([z], feed_dict={x: 2.})
    z1 = sess.run([z], feed_dict={x: 2., y: 2.})

print("z0:{},\nz1:{}".format(z0, z1))
```

```
z0:[6.0],
z1:[4.0]
```

There is one tf.Graph object (tf.get_default_graph()) but different subgraphs get executed

Equivalent:

```
def compute_z0(x):
  return tf.add(x, tf.square(x))

def compute_z1(x, y):
  return tf.add(x,  y)
```

tf.function to the rescue

- A "TensorFlow function" defines a computation as a graph of TensorFlow operations, with named arguments and explicit return values
- In fact, annotated functions are like tensorflow ops
- not necessary to decorate each function, just decorate the higher level functions

In [18]:
```
import tensorflow as tf

def add(x, y):
    return x + y

@tf.function
def compute_z1(x, y):
  return add(x, y)

@tf.function
def compute_z0(x):
  return tf.add(x, tf.square(x))

z0 = compute_z0(2.)
z1 = compute_z1(2., 2.)

print("z0:{},\nz1:{}".format(z0, z1))
```

```
z0:6.0,
z1:4.0
```

Autograph

- AutoGraph converts subset of Python constructs into TensorFlow equivalent
- for/while -> tf.while_loop (break and continue supported)
- if -> tf.cond
- for _ in dataset -> dataset.reduce

Using Graph

In [19]:
```
import tensorflow as tf

def true_fn(): return True
def false_fn(): return False
def loop_condition_fn(i, x, y): return tf.less(i, 10)
def loop_body_fn(i, x, y): return tf.add(i, 1), tf.add(x, 1), tf.ad
d(y, 1)

with tf.compat.v1.Session() as sess:
    # conditionals
    x = tf.compat.v1.placeholder(tf.float32)
    y = tf.compat.v1.placeholder(tf.float32)
    z = tf.cond(tf.less(x, y), true_fn, false_fn)
    # loops
    i = tf.constant(0)
    inc = tf.while_loop(loop_condition_fn,loop_body_fn, [i, x, y])
    res = sess.run([z, inc], feed_dict={x: 2., y: 1.})

print("result:{}".format(res))
```

```
result:[False, (10, 12.0, 11.0)]
```

Using tf.function and autograph

In [20]:
```
import tensorflow as tf

@tf.function
def lesser_than(x, y):
    return x < y

@tf.function
def increment(x, y):
    for i in range(10):
        x = x + 1
        y = y + 1
    return x, y

print(lesser_than(2., 1.))
print(increment(2., 1.))
print(tf.autograph.to_code(increment.python_function))
```

```
tf.Tensor(False, shape=(), dtype=bool)
(<tf.Tensor: id=67397, shape=(), dtype=float32, numpy=12.0>, <tf.T
ensor: id=67398, shape=(), dtype=float32, numpy=11.0>)
from __future__ import print_function

def tf__increment(x, y):
  try:
    with ag__.function_scope('increment'):
      do_return = False
      retval_ = None

      def loop_body(loop_vars, x_1, y_1):
        with ag__.function_scope('loop_body'):
          i = loop_vars
          x_1 = x_1 + 1
          y_1 = y_1 + 1
          return x_1, y_1
      x, y = ag__.for_stmt(ag__.range_(10), None, loop_body, (x, y
))
      do_return = True
      retval_ = x, y
      return retval_
  except:
    ag__.rewrite_graph_construction_error(ag_source_map__)



tf__increment.autograph_info__ = {}
```

Still a bit to think about

In [21]:
```
def f_eager():
    x = tf.Variable(1)
    return x

@tf.function
def f():
    x = tf.Variable(1)
    return x
```

- no guarantee about the number of times tf.function evaluates the Python function while converting
- tf.Variable in eager mode is just a plain Python object
- tf.Variable in a decorated function creates symbol in persistent graph (eager mode is disabled)

Migrating from TF1 to TF2

- a migration script is provided
- code practices for easy conversion

Using the migration script

In [ ]:
```
# %load tf1_script.py
import tensorflow as tf

def true_fn(): return True
def false_fn(): return False
def loop_condition_fn(i, x, y): return tf.less(i, 10)
def loop_body_fn(i, x, y): return tf.add(i, 1), tf.add(x, 1), tf.ad
d(y, 1)

with tf.Session() as sess:
    # conditionals
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.cond(tf.less(x, y), true_fn, false_fn)
    # loops
    i = tf.constant(0)
    inc = tf.while_loop(loop_condition_fn,loop_body_fn, [i, x, y])
    res = sess.run([z, inc], feed_dict={x: 2., y: 1.})

print("result:{}".format(res))
```

In [ ]:
```
!tf_upgrade_v2 --infile tf1_script.py --outfile tf2_script.py
```

In [ ]:
```
# %load tf2_script.py
import tensorflow as tf

def true_fn(): return True
def false_fn(): return False
def loop_condition_fn(i, x, y): return tf.less(i, 10)
def loop_body_fn(i, x, y): return tf.add(i, 1), tf.add(x, 1), tf.ad
d(y, 1)

with tf.compat.v1.Session() as sess:
    # conditionals
    x = tf.compat.v1.placeholder(tf.float32)
    y = tf.compat.v1.placeholder(tf.float32)
    z = tf.cond(pred=tf.less(x, y), true_fn=true_fn, false_fn=false
_fn)
    # loops
    i = tf.constant(0)
    inc = tf.while_loop(cond=loop_condition_fn,body=loop_body_fn, l
oop_vars=[i, x, y])
    res = sess.run([z, inc], feed_dict={x: 2., y: 1.})

print("result:{}".format(res))
```

Good migration practices

- tf.Session.run -> Python function where feed_dict and tf.placeholder become function arguments
- tf.get_variable -> tf.Variable
- variable_scope -> Python object (Keras Layer, Keras Model ...)
- own training loops -> tf.keras.Model.fit
- use tf.data datasets

see also [TensorFlow Migration Guid (https://www.tensorflow.org/alpha/guide/migration_guide)](https://www.tensorflow.org/alpha/guide/migration_guide)

Recommendations for coding in TensorFlow 2

- Refactor your code into smaller functions
- Use Keras layers and models to manage variables
- Combine tf.data.Datasets and @tf.function
- Take advantage of AutoGraph with Python control flow

see also [TensorFlow Best-Practices (https://github.com/tensorflow/docs/blob/master/site/en/r2/guide/effective_tf2.md)](https://github.com/tensorflow/docs/blob/master/site/en/r2/guide/effective_tf2.md)

Deployment

https://www.tensorflow.org/guide/distribute_strategy (https://www.tensorflow.org/guide/distribute_strategy)

> A SavedModel contains a complete TensorFlow program, including weights and computation. It does not require the original model building code to run, which makes it useful for sharing or deploying (with TFLite, TensorFlow.js, TensorFlow Serving, or TFHub).

see also SavedModel Guid (https://www.tensorflow.org/alpha/guide/saved_model)

An example

How to export a model for serving today:

```
model = tf.keras.applications.MobileNet()
tensor_info_input = tf.saved_model.utils.build_tensor_info(model.input)
tensor_info_output = tf.saved_model.utils.build_tensor_info(model.outpu
t)
prediction_signature = (
    tf.saved_model.signature_def_utils.build_signature_def(
        inputs={'input': tensor_info_input},
        outputs={'prediction': tensor_info_output},
        method_name=signature_constants.PREDICT_METHOD_NAME))

export_path = out_path
tf_builder = builder.SavedModelBuilder(export_path)
with keras.backend.get_session() as sess:
    tf_builder.add_meta_graph_and_variables(
        sess=sess,
        tags=[tag_constants.SERVING],
        signature_def_map={
            signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY: pred
iction_signature
        }
    )
tf_builder.save()
```

How to export a model with TensorFlow 2.0:

In [ ]:
```
model = tf.keras.applications.MobileNet()
tf.saved_model.save(model, "/tmp/mobilenet/1/")
```

In [ ]:
```
!saved_model_cli show --dir /tmp/mobilenet/1 --tag_set serve --sign
ature_def serving_default
```

Summary

- A big focus of TensorFlow 2 is on Keras
- tf.function and autograph leverage eager mode
- no more sessions, everything eager now
- Keras gets Serving-Ready with the new SavedModel formt
- A lot of ressource, best practices and helpers to get your project migrated