Google Cloud

ASL Tech Talk:

TensorFlow 2.0

In case you haven't heard...

TensorFlow

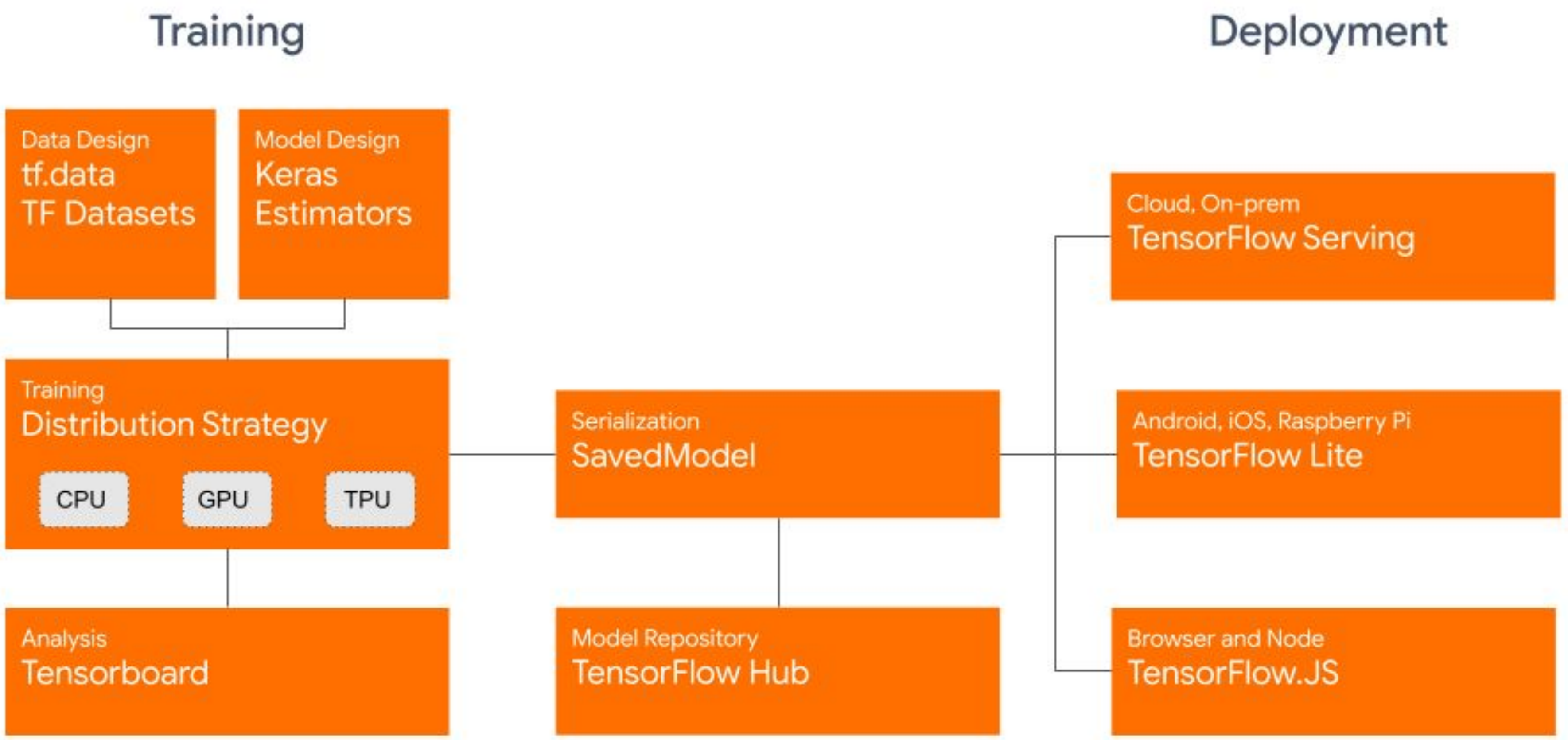
1.x

→

TensorFlow

2.0

Simplified conceptual diagram



API Cleanup

- TensorFlow had grown so much over the years with so many contributors that the API became bloated
- `tf.contrib` had a lot of abandoned packages, so some popular ones were moved to core before getting rid of contrib
- Often there were 3+ ways of doing the same thing, so much the API has been consolidated

Easy model building

- Load your data using `tf.data`.
- Build, train and validate your model with `tf.keras`, or use Premade Estimators.
- Run and debug with eager execution, then use `tf.function` for the benefits of graphs.
- Use Distribution Strategies for distributed training.
- Export to SavedModel.

Load your data using tf.data

- Training data is read using input pipelines which are created using tf.data.
- Feature characteristics, for example bucketing and feature crosses are described using `tf.feature_column`.
- Convenient input from in-memory data (for example, NumPy) is also supported.

Build, train and validate your model with tf.keras, or use Premade Estimators

- Keras integrates tightly with the rest of TensorFlow so you can access TensorFlow's features whenever you want.
- A set of standard packaged models (for example, linear or logistic regression, gradient boosted trees, random forests) are also available to use directly (implemented using the `tf.estimator` API).
- If you're not looking to train a model from scratch, you'll soon be able to use transfer learning to train a Keras or Estimator model using modules from [TensorFlow Hub].

Run and debug with eager execution, then use tf.function for the benefits of graphs.

- TensorFlow 2.0 runs with eager execution by default for ease of use and smooth debugging.
- The `tf.function` decorator transparently translates your Python programs into TensorFlow graphs.
- This process retains all the advantages of 1.x TensorFlow graph-based execution: Performance optimizations, remote execution and the ability to serialize, export and deploy easily, while adding the flexibility and ease of use of expressing programs in simple Python.

Run and debug with eager execution, then use tf.function for the benefits of graphs.

```
@tf.function
def f(x):
    return tf.add(x, 1)

scalar = tf.constant(1)
vector = tf.constant([1, 1])
matrix = tf.constant([[3]])
```

Three separate graphs are made!

Use Distribution Strategies for distributed training.

- For large ML training tasks, the Distribution Strategy API makes it easy to distribute and train models on different hardware configurations without changing the model definition.
- Since TensorFlow provides support for a range of hardware accelerators like CPUs, GPUs, and TPUs, you can enable training workloads to be distributed to single-node/multi-accelerator as well as multi-node/multi-accelerator configurations, including TPU Pods.
- Although this API supports a variety of cluster configurations, templates to deploy training on Kubernetes clusters in on-prem or cloud environments are provided.

Export to SavedModel.

- TensorFlow will standardize on SavedModel as an interchange format for TensorFlow Serving, TensorFlow Lite, TensorFlow.js, TensorFlow Hub, and more.

TensorFlow Datasets

- TensorFlow Datasets is a collection of datasets ready to use with TensorFlow.
- All datasets are exposed as tf.data.Datasets, enabling easy-to-use and high-performance input pipelines.
- Guide to get started.
- List of datasets to try out.
  - Audio
  - Image
  - Structured
  - Summarization
  - Text
  - Translate
  - Video

TensorFlow Datasets

```
import tensorflow as tf
import tensorflow_datasets as tfds

# tfds works in both Eager and Graph modes
tf.compat.v1.enable_eager_execution()

# See available datasets
print(tfds.list_builders())

# Construct a tf.data.Dataset
dataset = tfds.load(name="mnist", split=tfds.Split.TRAIN)

# Build your input pipeline
dataset = dataset.shuffle(1024).batch(32).prefetch(tf.data.experimental.AUTOTUNE)
for features in dataset.take(1):
  image, label = features["image"], features["label"]
```

Converting TF 1.x code to 2.0

The overall process is:

1. Run the upgrade script.
2. Remove contrib symbols.
3. Switch your models to an object oriented style (Keras).
4. Use `tf.keras` or `tf.estimator` training and evaluation loops where you can.
5. Otherwise, use custom loops, but be sure to avoid sessions & collections.

It takes a little work to convert code to idiomatic TensorFlow 2.0, but every change results in:

- Fewer lines of code.
- Increased clarity and simplicity.
- Easier debugging.

Automatic conversion script

- The first step, before attempting to implement any changes yourself, is to try running the upgrade script.
- This will do an initial pass at upgrading your code to TensorFlow 2.0.
- But it can't make your code idiomatic to 2.0. Your code may still make use of `tf.compat.v1` endpoints to access placeholders, sessions, collections, and other 1.x-style functionality.
- Therefore, to get everything into the 2.0 style, you may want to then do a pass where you manually change whatever's left over

Top-level behavioral changes

- If your code works in TensorFlow 2.0 using [`tf.compat.v1.disable_v2_behavior()`](), there are still global behavioral changes you may need to address. The major changes are:
- Eager execution
- Resource variables
- Tensor shapes
- Control flow

Eager execution

- `v1.enable_eager_execution()` : Any code that implicitly uses a `tf.Graph` will fail. Be sure to wrap this code in a

  `with tf.Graph().as_default()` context.

Resource variables

- `v1.enable_resource_variables()`: Some code may depends on non-deterministic behaviors enabled by TF reference variables. Resource variables are locked while being written to, and so provide more intuitive consistency guarantees.
  - This may change behavior in edge cases.
  - This may create extra copies and can have higher memory usage.
  - This can be disabled by passing `use_resource=False` to the `tf.Variable` constructor.

Tensor shapes

- `v1.enable_v2_tensorshape()`: TF 2.0 simplifies the behavior of tensor shapes. Instead of `t.shape[0].value` you can say `t.shape[0]`. These changes should be small, and it makes sense to fix them right away. See `TensorShape` for examples.

Control flow

- `v1.enable_control_flow_v2()`: The TF 2.0 control flow implementation has been simplified, and so produces different graph representations.

Make your converted code TF 2.0 native

- Replace `v1.Session.run` calls
- Use Python objects to track variables and losses
- Upgrade your training loops
- Upgrade your data input pipelines
- Migrate off `compat.v1` symbols

Replace `v1.Session.run` calls

Every `v1.Session.run` call should be replaced by a Python function.

- The `feed_dict` and `v1.placeholder`s become function arguments.
- The `fetches` become the function's return value.
- During conversion eager execution allows easy debugging with standard Python tools like `pdb`.

After that add a `tf.function` decorator to make it run efficiently in graph. See the Autograph Guide for more on how this works.

Replace `v1.Session.run` calls

Note that:

- Unlike `v1.Session.run` a `tf.function` has a fixed return signature, and always returns all outputs. If this causes performance problems, create two separate functions.
- There is no need for a `tf.control_dependencies` or similar operations: A `tf.function` behaves as if it were run in the order written. `tf.Variable` assignments and `tf.asserts`, for example, are executed automatically.

Use Python objects to track variables and losses

All name-based variable tracking is strongly discouraged in TF 2.0. Use Python objects to to track variables.

Use `tf.Variable` instead of `v1.get_variable`.

Every `v1.variable_scope` should be converted to a Python object. Typically this will be one of:

- `tf.keras.layers.Layer`
- `tf.keras.Model`
- `tf.Module`

Use Python objects to track variables and losses

If you need to aggregate lists of variables (like [`tf.Graph.get_collection(tf.GraphKeys.VARIABLES)`]), use the `.variables` and `.trainable_variables` attributes of the `Layer` and `Model` objects.

These `Layer` and `Model` classes implement several other properties that remove the need for global collections. Their `.losses` property can be a replacement for using the `tf.GraphKeys.LOSSES` collection.

See the keras guides for details.

Upgrade your training loops

- Use the highest level API that works for your use case. Prefer `tf.keras.Model.fit` over building your own training loops.
- These high level functions manage a lot of the low-level details that might be easy to miss if you write your own training loop. For example, they automatically collect the regularization losses, and set the `training=True` argument when calling the model.

Upgrade your data input pipelines

- Use `tf.data` datasets for data input. These objects are efficient, expressive, and integrate well with tensorflow.
- They can be passed directly to the `tf.keras.Model.fit` method.
- `model.fit(dataset, epochs=5)`
- They can be iterated over directly using standard Python:
-
```
for example_batch, label_batch in dataset:
    break
```

Migrate off compat.v1 symbols

- The `tf.compat.v1` module contains the complete TensorFlow 1.x API, with its original semantics.
- The TF2 upgrade script will convert symbols to their 2.0 equivalents if such a conversion is safe, i.e., if it can determine that the behavior of the 2.0 version is exactly equivalent (for instance, it will rename `v1.arg_max` to `tf.argmax`, since those are the same function).
- After the upgrade script is done with a piece of code, it is likely there are many mentions of `compat.v1`. It is worth going through the code and converting these manually to the 2.0 equivalent (it should be mentioned in the log if there is one).

Low-level variables & operator execution

Examples of low-level API use include:

- using variable scopes to control reuse
- creating variables with `v1.get_variable`.
- accessing collections explicitly
- accessing collections implicitly with methods like :
  - `v1.global_variables`
  - `v1.losses.get_regularization_loss`
- using `v1.placeholder` to set up graph inputs
- executing graphs with `Session.run`
- initializing variables manually

Low-level variables & operator execution

```
in_a = tf.placeholder(dtype=tf.float32, shape=(2))
in_b = tf.placeholder(dtype=tf.float32, shape=(2))

def forward(x):
  with tf.variable_scope("matmul", reuse=tf.AUTO_REUSE):
    W = tf.get_variable("W", initializer=tf.ones(shape=(2,2)),
                        regularizer=tf.contrib.layers.l2_regularizer(0.04))
    b = tf.get_variable("b", initializer=tf.zeros(shape=(2)))
    return W * x + b

out_a = forward(in_a)
out_b = forward(in_b)

reg_loss = tf.losses.get_regularization_loss(scope="matmul")

with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  outs = sess.run([out_a, out_b, reg_loss],
                feed_dict={in_a: [1, 0], in_b: [0, 1]})
```

```
W = tf.Variable(tf.ones(shape=(2,2)), name="W")
b = tf.Variable(tf.zeros(shape=(2)), name="b")

@tf.function
def forward(x):
  return W * x + b
regularizer = tf.keras.regularizers.l2(0.04)
reg_loss = regularizer(W)
```

Low-level variables & operator execution

In the converted code:

- The variables are local Python objects.
- The `forward` function still defines the calculation.
- The `Session.run` call is replaced with a call to `forward`
- The optional `tf.function` decorator can be added for performance.
- The regularizations are calculated manually, without referring to any global collection.
- **No sessions or placeholders.**

Models based on tf.layers

- The `v1.layers` module is used to contain layer-functions that relied on `v1.variable_scope` to define and reuse variables.

Models based on tf.layers

```python
def model(x, training, scope='model'):
  with tf.variable_scope(scope, reuse=tf.AUTO_REUSE):
    x = tf.layers.conv2d(x, 32, 3, activation=tf.nn.relu,
          kernel_regularizer=tf.contrib.layers.l2_regularizer(0.04))
    x = tf.layers.max_pooling2d(x, (2, 2), 1)
    x = tf.layers.flatten(x)
    x = tf.layers.dropout(x, 0.1, training=training)
    x = tf.layers.dense(x, 64, activation=tf.nn.relu)
    x = tf.layers.batch_normalization(x, training=training)
    x = tf.layers.dense(x, 10, activation=tf.nn.softmax)
    return x

train_data = tf.ones(shape=(1, 28, 28, 1))
test_data = tf.ones(shape=(1, 28, 28, 1))

train_out = model(train_data, training=True)
test_out = model(test_data, training=False)
```

```python
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, activation='relu',
                           kernel_regularizer=tf.keras.regularizers.l2(0.04),
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10, activation='softmax')
])

train_data = tf.ones(shape=(1, 28, 28, 1))
test_data = tf.ones(shape=(1, 28, 28, 1))

train_out = model(train_data, training=True)
test_out = model(test_data, training=False)
```

Models based on tf.layers

- The simple stack of layers fits neatly into `tf.keras.Sequential`. (For more complex models see custom layers and models, and the functional API.)
- The model tracks the variables, and regularization losses.
- The conversion was one-to-one because there is a direct mapping from `v1.layers` to `tf.keras.layers`.

Most arguments stayed the same. But notice the differences:

- The `training` argument is passed to each layer by the model when it runs.
- The first argument to the original `model` function (the input `x`) is gone. This is because object layers separate building the model from calling the model.

Models based on tf.layers

Also note that:

- If you were using regularizers of initializers from `tf.contrib`, these have more argument changes than others.
- The code no longer writes to collections, so functions like [`v1.losses.get_regularization_loss`]() will no longer return these values, potentially breaking your training loops.

Mixed variables & `v1.layers`

- Existing code often mixes lower-level TF 1.x variables and operations with higher-level `v1.layers`.

Mixed variables & [v1.layers]()

```python
def model(x, training, scope='model'):
  with tf.variable_scope(scope, reuse=tf.AUTO_REUSE):
    W = tf.get_variable(
      "W", dtype=tf.float32,
      initializer=tf.ones(shape=x.shape),
      regularizer=tf.contrib.layers.l2_regularizer(0.04),
      trainable=True)
    if training:
      x = x + W
    else:
      x = x + W * 0.5
    x = tf.layers.conv2d(x, 32, 3, activation=tf.nn.relu)
    x = tf.layers.max_pooling2d(x, (2, 2), 1)
    x = tf.layers.flatten(x)
    return x

train_out = model(train_data, training=True)
test_out = model(test_data, training=False)
```

→

```python
# Create a custom layer for part of the model
class CustomLayer(tf.keras.layers.Layer):
  def __init__(self, *args, **kwargs):
    super(CustomLayer, self).__init__(*args, **kwargs)

  def build(self, input_shape):
    self.w = self.add_weight(
        shape=input_shape[1:],
        dtype=tf.float32,
        initializer=tf.keras.initializers.ones(),
        regularizer=tf.keras.regularizers.l2(0.02),
        trainable=True)

  # Call method will sometimes get used in graph mode,
  # training will get turned into a tensor
  @tf.function
  def call(self, inputs, training=None):
    if training:
      return inputs + self.w
    else:
      return inputs + self.w * 0.5

train_data = tf.ones(shape=(1, 28, 28, 1))
test_data = tf.ones(shape=(1, 28, 28, 1))

# Build the model including the custom layer
model = tf.keras.Sequential([
    CustomLayer(input_shape=(28, 28, 1)),
    tf.keras.layers.Conv2D(32, 3, activation='relu'),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
])

train_out = model(train_data, training=True)
test_out = model(test_data, training=False)
```

Mixed variables & `v1.layers`

- Subclassed Keras models & layers need to run in both v1 graphs (no automatic control dependencies) and in eager mode
    a. Wrap the `call()` in a `tf.function()` to get autograph and automatic control dependencies
- Don't forget to accept a `training` argument to `call`.
    a. Sometimes it is a `tf.Tensor`
    b. Sometimes it is a Python boolean.

Mixed variables & `v1.layers`

- Create model variables in constructor or `Model.build` using `self.add_weight()`.
    a. In `Model.build` you have access to the input shape, so can create weights with matching shape.
    b. Using `tf.keras.layers.Layer.add_weight` allows Keras to track variables and regularization losses.
- Don't keep tf.Tensors in your objects.
    a. They might get created either in a `tf.function` or in the eager context, and these tensors behave differently.
    b. Use `tf.Variable`s for state, they are always usable from both contexts
    c. `tf.Tensors` are only for intermediate values.

Still can't give up 1.x?

- It is still possible to run 1.X code, unmodified (except for contrib), in TensorFlow 2.0:
- ```
  import tensorflow.compat.v1 as tf
  tf.disable_v2_behavior()
  ```
- However, this does not let you take advantage of many of the improvements made in TensorFlow 2.0.
  ```

Simplified input function with `tf.data`

```python
# Create an input function reading a file using the Dataset API
# Then provide the results to the Estimator API
def read_dataset(filename, mode, batch_size = 512):
    def _input_fn():
        def decode_csv(value_column):
            columns = tf.decode_csv(value_column, record_defaults=DEFAULTS)
            features = dict(zip(CSV_COLUMNS, columns))
            label = features.pop(LABEL_COLUMN)
            return features, label

        # Create list of files that match pattern
        file_list = tf.gfile.Glob(filename)

        # Create dataset from file list
        dataset = (tf.data.TextLineDataset(file_list)  # Read text file
        .map(decode_csv))  # Transform each elem by applying decode_csv fn

        if mode == tf.estimator.ModeKeys.TRAIN:
            num_epochs = None # indefinitely
            dataset = dataset.shuffle(buffer_size=10*batch_size)
        else:
            num_epochs = 1 # end-of-input after this

        dataset = dataset.repeat(num_epochs).batch(batch_size)
        return dataset
    return _input_fn
```

```python
# Create estimator to train and evaluate
def train_and_evaluate(output_dir):
    EVAL_INTERVAL = 300
    run_config = tf.estimator.RunConfig(
        save_checkpoints_secs = EVAL_INTERVAL,
        keep_checkpoint_max = 3)

    estimator = tf.estimator.DNNRegressor(
        model_dir = output_dir,
        feature_columns = get_cols(),
        hidden_units = [64, 32],
        config = run_config)

    train_spec = tf.estimator.TrainSpec(
        input_fn = read_dataset('train.csv', mode = tf.estimator.ModeKeys.TRAIN),
        max_steps = TRAIN_STEPS)

    exporter = tf.estimator.LatestExporter('exporter', serving_input_fn)

    eval_spec = tf.estimator.EvalSpec(
        input_fn = read_dataset('eval.csv', mode = tf.estimator.ModeKeys.EVAL),
        steps = None,
        start_delay_secs = 60, # start evaluating after N seconds
        throttle_secs = EVAL_INTERVAL,  # evaluate every N seconds
        exporters = exporter)

    tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

```python
def features_and_labels(row_data):
    label = row_data.pop(LABEL_COLUMN)
    return row_data, label  # features, label

# Load the training data
def load_dataset(pattern, batch_size=1, mode=tf.estimator.ModeKeys.EVAL):
    # Make a CSV dataset
    dataset = tf.data.experimental.make_csv_dataset(
        file_pattern=pattern,
        batch_size=batch_size,
        column_names=CSV_COLUMNS,
        column_defaults=DEFAULTS)

    # Map dataset to features and label
    dataset = dataset.map(map_func=features_and_labels)  # features, label

    # Shuffle and repeat for training
    if mode == tf.estimator.ModeKeys.TRAIN:
        dataset = dataset.shuffle(buffer_size=1000).repeat()

    # Take advantage of multi-threading; 1=AUTOTUNE
    dataset = dataset.prefetch(buffer_size=1)

    return dataset
```

```python
TRAIN_BATCH_SIZE = 32
NUM_TRAIN_EXAMPLES = 10000 * 5  # training dataset repeats, it'll wrap around
NUM_EVALS = 5  # how many times to evaluate
# Enough to get a reasonable sample, but not so much that it slows down
NUM_EVAL_EXAMPLES = 10000

trainds = load_dataset(
    pattern="train*.csv",
    batch_size=TRAIN_BATCH_SIZE,
    mode=tf.estimator.ModeKeys.TRAIN)
evalds = load_dataset(
    pattern="eval*.csv",
    batch_size=1000,
    mode=tf.estimator.ModeKeys.EVAL).take(count=NUM_EVAL_EXAMPLES//1000)
```

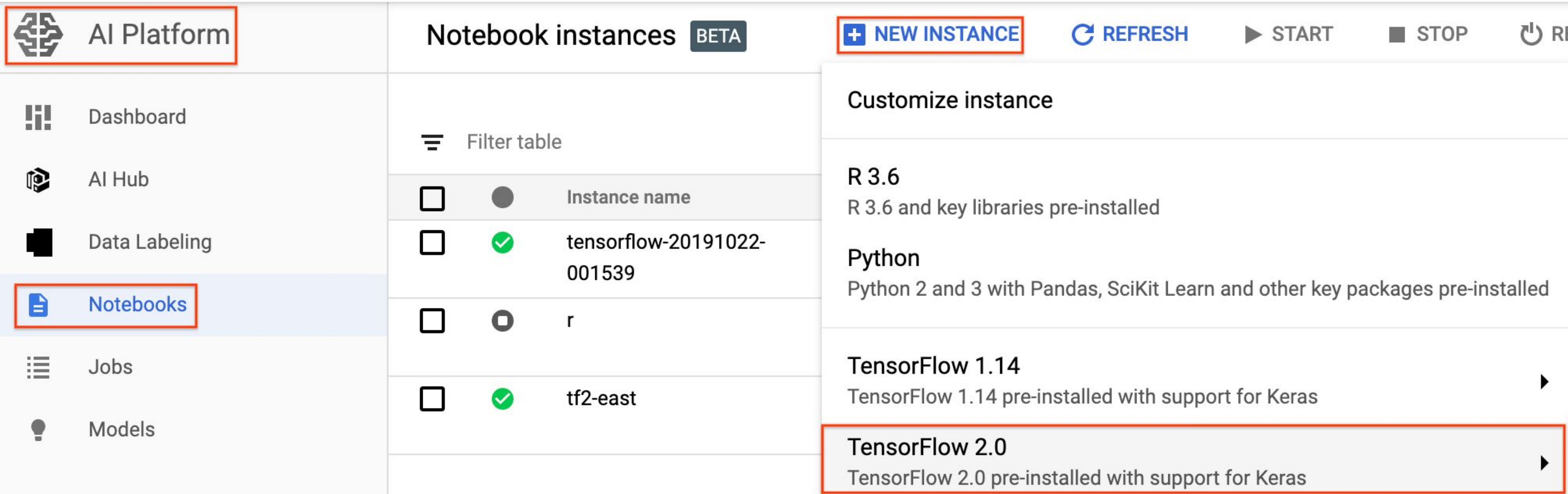
Keras training loops

```python
# Create estimator to train and evaluate
def train_and_evaluate(output_dir):
    EVAL_INTERVAL = 300
    run_config = tf.estimator.RunConfig(
        save_checkpoints_secs = EVAL_INTERVAL,
        keep_checkpoint_max = 3)

    estimator = tf.estimator.DNNRegressor(
        model_dir = output_dir,
        feature_columns = get_cols(),
        hidden_units = [64, 32],
        config = run_config)

    train_spec = tf.estimator.TrainSpec(
        input_fn = read_dataset('train.csv', mode = tf.estimator.ModeKeys.TRAIN),
        max_steps = TRAIN_STEPS)

    exporter = tf.estimator.LatestExporter('exporter', serving_input_fn)

    eval_spec = tf.estimator.EvalSpec(
        input_fn = read_dataset('eval.csv', mode = tf.estimator.ModeKeys.EVAL),
        steps = None,
        start_delay_secs = 60, # start evaluating after N seconds
        throttle_secs = EVAL_INTERVAL,  # evaluate every N seconds
        exporters = exporter)

    tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

```python
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, activation='relu',
                           kernel_regularizer=tf.keras.regularizers.l2(0.02),
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Model is the full model w/o custom layers
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_data, epochs=NUM_EPOCHS)
loss, acc = model.evaluate(test_data)

print("Loss {}, Accuracy {}".format(loss, acc))
```

Training with gcloud on GCP

- TF 2.0 was just officially released, so all of the other product teams are racing to catch up
- Cloud AI Platform Notebooks can come with TF 2.0 prebuilt



Training with gcloud on GCP

- TF 2.0 is almost supported natively on CAIP for training and inference, but in the meantime a workaround is needed.
- We'll create a Docker image of our TF 2.0 environment and send that with our gcloud training call
- First create Dockerfile

```
%%writefile babyweight/Dockerfile
FROM gcr.io/deeplearning-platform-release/tf2-cpu
COPY trainer /babyweight/trainer
RUN apt update && \
    apt install --yes python3-pip && \
    pip3 install --upgrade --quiet tf-nightly-2.0-preview

ENV PYTHONPATH ${PYTHONPATH}:/babyweight
CMD ["python3", "-m", "trainer.task"]
```

Training with gcloud on GCP

- Then create shell script to export Docker image

```
%%writefile babyweight/push_docker.sh
export PROJECT_ID=$(gcloud config list project --format "value(core.project)")
export IMAGE_REPO_NAME=babyweight_training_container
export IMAGE_URI=gcr.io/$PROJECT_ID/$IMAGE_REPO_NAME

echo "Building  $IMAGE_URI"
docker build -f Dockerfile -t $IMAGE_URI ./
echo "Pushing $IMAGE_URI"
docker push $IMAGE_URI
```

Training with gcloud on GCP

- Then call your script in bash

```
%%bash
cd babyweight
bash push_docker.sh
```

- It will do a bunch of installs and loading of packages. It will take a few minutes.

Training with gcloud on GCP

- Lastly, send your Docker container to the gcloud AI Platform job

```
%%bash
OUTDIR=gs://${BUCKET}/babyweight/trained_model
JOBID=babyweight_$(date -u +%y%m%d_%H%M%S)
echo $OUTDIR $REGION $JOBNAME
gsutil -m rm -rf $OUTDIR

IMAGE=gcr.io/$PROJECT/babyweight_training_container

gcloud beta ai-platform jobs submit training $JOBID \
    --staging-bucket=gs://$BUCKET \
    --region=$REGION \
    --master-image-uri=$IMAGE \
    --master-machine-type=n1-standard-4 \
    --scale-tier=CUSTOM
```

Thank you!

Questions?

cloud.google.com