A Primer on Tensorflow

Sheroze Sheriffdeen

November 30, 2018

INSTITUTE FOR COMPUTATIONAL ENGINEERING & SCIENCES

The University of Texas at Austin

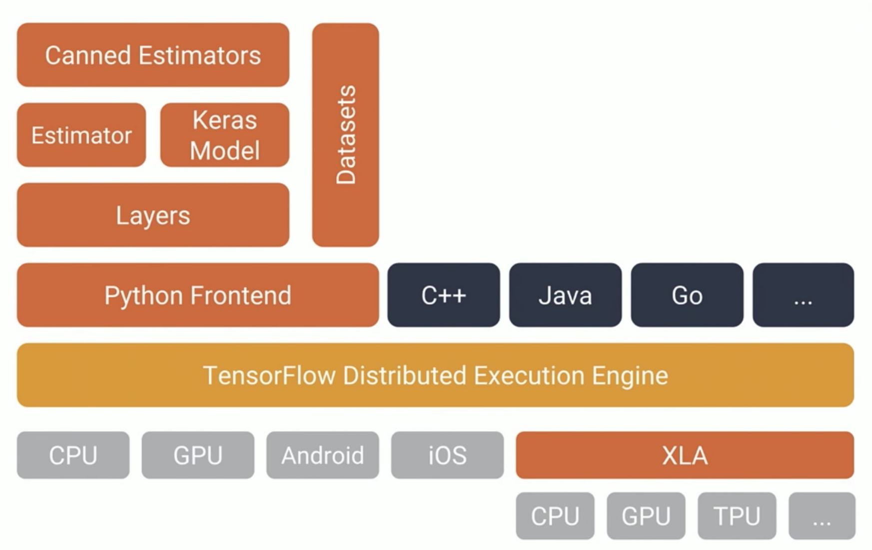
What is Tensorflow?

- Open-source software library for machine learning, deep learning, and more
- Easy deployment on multiple platforms (CPUs, GPUs, and TPUs), server clusters and even mobile devices.
- Backed by Google
- High level `tf.keras` interface for easy prototyping
- Estimators library to simplify training, evaluating, and serving machine learning models in a distributed setting
- `tf.data` is a library to process and feed data into models.
- `tf.layers` and `tf.keras.layers` to modularly build up deep learning models
- Low-level interface defines a computational graph equipped with automatic differentiation

Tensorflow Ecosystem

- Tensorflow Lite (mobile and embedded), Tensorflow.js, Tensorflow Probability
- TensorFlow Hub, a library for reusable machine learning modules
- Tensorflow Research Cloud provides cloud Tensor Processing Units (TPUs) which are Google's proprietary systems optimized for deep learning applications. (https://www.tensorflow.org/tfrc/)

Organization



A Tensorflow program is abstracted as a computational graph (`tf.Graph`).

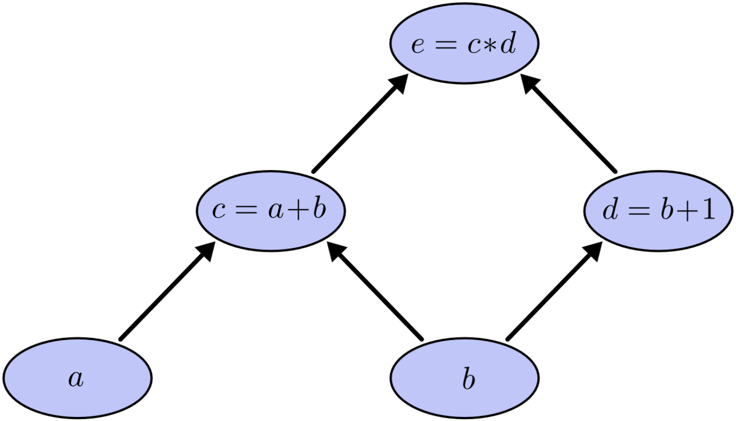


Figure: A simple computational graph with two variables

Low-level architecture

A running instance of a program is called a session (`tf.Session`). The `tf.Tensor` objects flow through the computational graph.

```
a = tf.constant(3.0, dtype=tf.float32)
b = tf.constant(4.0) # also tf.float32 implicitly
total = a + b
print(a)
print(b)
print(total)
```

The print statements will produce:

```
Tensor("Const:0", shape=(), dtype=float32)
Tensor("Const_1:0", shape=(), dtype=float32)
Tensor("add:0", shape=(), dtype=float32)
```

Instantiate a session to perform the computation:

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

Low-level architecture

- Once a computational graph is defined with the appropriate input function, it can be deployed in many different frameworks.
- This improves performance but is tricky to tinker with as the graph data is not exposed.
- There are ways to understand how your TensorFlow models train:
  - Add training metrics (e.g. training accuracy)
  - Use TensorBoard to visualize the evolution of data during training
  - Eager execution to instantly perform calculations. Trading performance benefits for easy debugging.

Automatic differentiation

Every differentiable TensorFlow operation has an associated gradient function. TensorFlow keeps track of all the operations applied to compute the output of the function.

```
from math import pi

def f(x):
  return tf.square(tf.sin(x))

assert f(pi/2).numpy() == 1.0
```

Figure: Function definition $f(x) = \sin^2(x)$

```
grad_f = tfe.gradients_function(f)
assert tf.abs(grad_f(pi/2)[0]).numpy() < 1e-7
```

Figure: Gradient evaluation $f'(x) = 2\sin(x)\cos(x)$

Automatic differentiation

```
def f(t, x):
    u = u(t, x)
    u_t = tf.gradients(u, t)[0]
    u_x = tf.gradients(u, x)[0]
    u_xx = tf.gradients(u_x, x)[0]
    f = u_t + u*u_x - (0.01/tf.pi)*u_xx
    return f
```

Figure: Burgers' equation: $u_t + uu_x - (0.01/\pi)u_{xx} = 0$

Keras

- High-level interface to build and train deep learning models
- Easy to use and provides good support for common use cases
- Modular: Keras models are made by connecting configurable building blocks together
- Easily extensible to add your own layers, loss functions, etc.

Basic Classification with tf.keras

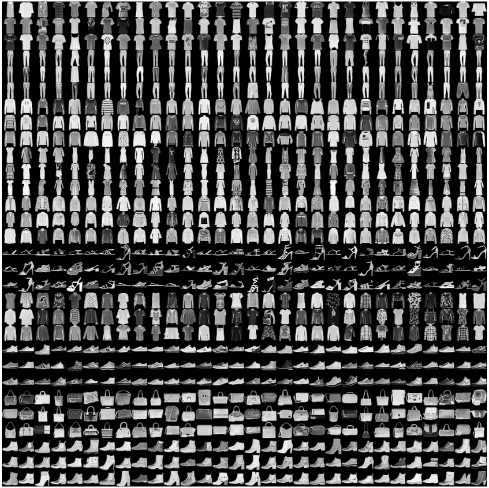


Figure: Fashion MNIST Dataset

Basic Classification

Problem Statement

Given 28x28 pixel grey-scale images of clothing articles, classify them by class (dress, coat, bag, etc.)

Simple Model

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

Keras Model

Simple Model

- Flattened input layer (not the best choice for images)
- One hidden layer with 128 neurons.
- Rectified Linear Unit (ReLU) as an activation function
- Output layer passed through softmax function for normalization

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

Keras Layers

There are many `tf.keras.layers` available (convolution, pooling, dropout, LSTM, etc.) with some common constructor parameters:

- **activation**: set the activation function for the layer.
- **kernel_initializer** and **bias_initializer**: initialization schemes that create the layer's weights (kernel and bias).
- **kernel_regularizer** and **bias_regularizer**: describe regularization if necessary

Basic Classification

Compiling the model

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

- Pick optimizer (Adadelta, Adagrad, Adam, Adamax, Nesterov Adam, RMSprop, Stochastic gradient descent, or implement your own)
- Pick objective (loss) function. (Mean squared, cross entropy, hinge loss, cosine loss, or implement your own)
- Pick metrics to be recorded (Fraction of correctly classified images in our example)

Basic Classification

Train the model

```
model.fit(train_images, train_labels, epochs=5)
```

- Feed the data to the model. May have to write a more complicated 'input function' to feed in different forms of data.
- As the model trains, the loss and other metrics are stored.
- The given simple model reaches 87% accuracy on the Fashion MNIST training set.

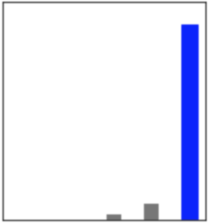
Basic Classification

Predicting with the model

```
predictions = model.predict(test_images)
```

- Given new test images, predict class probabilities in the online stage.

Ankle boot 90% (Ankle boot)



Regression Example

Problem Statement: Given housing information of houses in Boston, predict their market price. This is an example of a simple regression problem.

Sample of Inputs

- Per capita crime rate.
- The proportion of residential land zoned for lots over 25,000 square feet.
- The proportion of non-retail business acres per town.
- Charles River dummy variable ($= 1$ if tract bounds river; 0 otherwise).
- The average number of rooms per dwelling.
- The proportion of owner-occupied units built before 1940.
- Weighted distances to five Boston employment centers.
- Index of accessibility to radial highways.

Regression Example

```
def build_model():
  model = keras.Sequential([
    keras.layers.Dense(64, activation=tf.nn.relu,
                       input_shape=(train_data.shape[1],)),
    keras.layers.Dense(64, activation=tf.nn.relu),
    keras.layers.Dense(1)
  ])

  optimizer = tf.train.RMSPropOptimizer(0.001)

  model.compile(loss='mse',
                optimizer=optimizer,
                metrics=['mae'])
  return model

model = build_model()
model.summary()
```

Regression Example

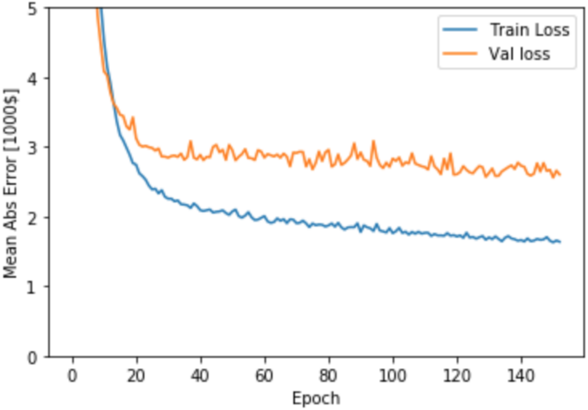
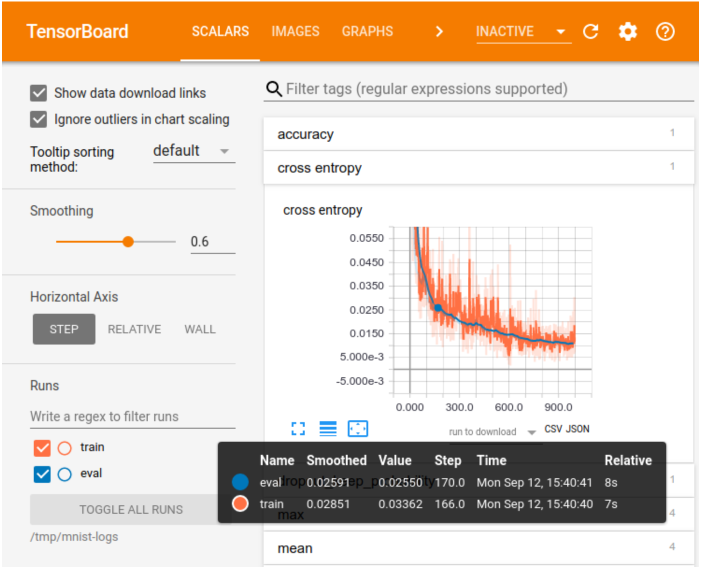


Figure: Training and Validation error with a 20% split obtained by training `history` variable output by the `fit` function

Extending Keras

- Easy to extend Keras to represent complex model topologies using the Keras functional API. (multi-input, multi-output models)
- Extend `tf.keras.Model` and `tf.keras.layers.Layer` to write your own models and layers having full control over the forward pass.
- Use `tf.keras.callbacks` to customize the model behavior during training:
  - `tf.keras.callbacks.LearningRateScheduler` to dynamically change the learning rate
  - `tf.keras.callbacks.TensorBoard` to monitor the model during training

TensorBoard



TensorBoard

Use TensorBoard to monitor training of a Keras model by adding a callback.

```
tensorboard = TensorBoard(log_dir="logs/{}".format(time()))

model.fit(x_train, y_train, verbose=1, callbacks=[tensorboard])
```

Launch TensorBoard using the following command:

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

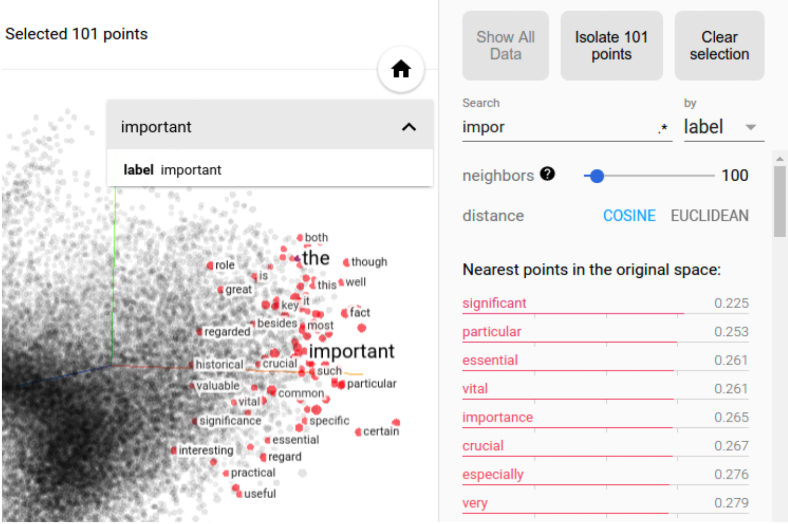
Once it is running, open `localhost:6006`.

TensorBoard

You can track various training metrics by annotating them within the model. TensorBoard can do far more than tracking scalars:

- Visualize your TensorFlow graph
- Show images that pass through the graph
- Histogram of tensor values over training period

Visualizing Embeddings



Data Pipeline

Data is the most important part of a deep learning model.

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

val_data = np.random.random((100, 32))
val_labels = np.random.random((100, 10))

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

What about more complicated data requiring more optimization, batching, shuffling, preprocessing, etc.?

`tf.data`

- A simple interface to build complex input pipelines for reusable pieces.
- For example, the pipeline for a PDE surrogate model might aggregate data from multiple forward solves in a distributed fashion, apply necessary preprocessing, and merge randomly selected solves into a batch for training.
- Two main abstractions:
  - `tf.data.Dataset` representing a sequence of elements defined from a source (pandas, numpy, Python generators) and can be transformed to perform batching, shuffling, and preprocessing.
  - `tf.data.Iterator` acts as the interface between input pipeline code and your model extracting elements from your dataset.

tf.data Interface

Basic example of a dataset with named components created from slicing tensors along the first dimension.

```
dataset = tf.data.Dataset.from_tensor_slices(
   {"a": tf.random_uniform([4]),
    "b": tf.random_uniform([4, 100], maxval=100, dtype=tf.int32)})
print(dataset.output_types)  # ==> "{'a': tf.float32, 'b': tf.int32}"
print(dataset.output_shapes)  # ==> "{'a': (), 'b': (100,)}"
```

Reading from multiple CSV files

```
# Creates a dataset that reads all of the records from two CSV files, each with
# eight float columns
filenames = ["/var/data/file1.csv", "/var/data/file2.csv"]
record_defaults = [tf.float32] * 8   # Eight required float columns
dataset = tf.contrib.data.CsvDataset(filenames, record_defaults)
```

Preprocessing Data

`Dataset.map(f)` transformation produces a new dataset by applying a given function f to each element of the input dataset.

```
# Reads an image from a file, decodes it into a dense tensor, and resizes it
# to a fixed shape.
def _parse_function(filename, label):
  image_string = tf.read_file(filename)
  image_decoded = tf.image.decode_jpeg(image_string)
  image_resized = tf.image.resize_images(image_decoded, [28, 28])
  return image_resized, label

# A vector of filenames.
filenames = tf.constant(["/var/data/image1.jpg", "/var/data/image2.jpg", ...])

# `labels[i]` is the label for the image in `filenames[i].
labels = tf.constant([0, 37, ...])

dataset = tf.data.Dataset.from_tensor_slices((filenames, labels))
dataset = dataset.map(_parse_function)
```

Input Data Pipeline

Shuffle, batch and repeat epochs with simple calls. Each operation is lazily evaluated.

```
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
dataset = tf.data.TFRecordDataset(filenames)
dataset = dataset.map(...)
dataset = dataset.shuffle(buffer_size=10000)
dataset = dataset.batch(32)
dataset = dataset.repeat(num_epochs)
iterator = dataset.make_one_shot_iterator()
```

The iterator object yields elements of the dataset (batched or otherwise) with the `get_next` method. `tf.Tensor` objects that correspond to the symbolic next element of an iterator. Each time these tensors are evaluated, they take the value of the next element in the underlying dataset.

```
next_example, next_label = iterator.get_next()
```

A few more tf.data notes

- Iterator state (therefore the whole input pipeline) can be saved and restored
- Instead of creating a one-shot iterator with the full dataset, an initializable iterator can be used to parametrize the dataset.
- `tf.data` interface is written with leveraging performance gains in a distributed setting in mind.

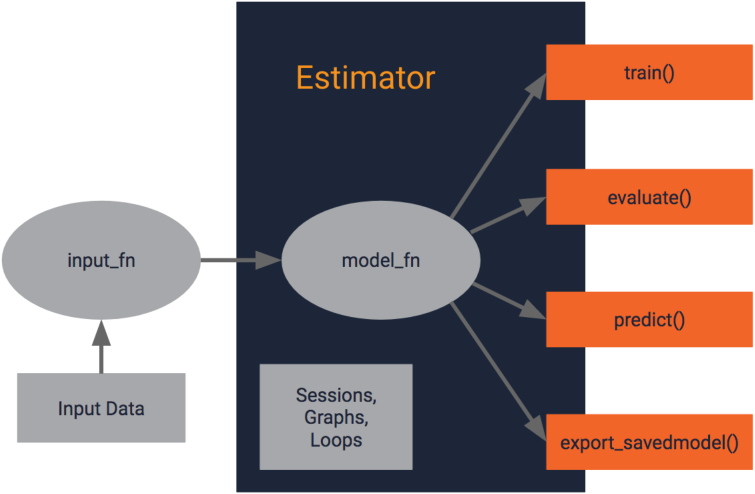
Estimators

Estimators is a high-level TensorFlow API that greatly simplifies machine learning programming. Estimators encapsulate the following actions:

- training
- evaluation
- prediction
- export for serving

Estimators are built on top of `tf.keras.layers`, and provide a safe distributed training loop to handle exceptions, create checkpoint files and recover from failures, and save metrics to be visualized in TensorBoard.

Overview of Estimators



Model Function

```python
def simple_dnn(features, labels, mode, params):
    batch_size = params["batch_size"]
    activation_fn = tf.nn.sigmoid
    layer_width = 80

    dense1 = tf.layers.dense(features, units=layer_width, activation=activation_fn)
    logits = tf.layers.dense(inputs=dense1, units=1)

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

    loss = tf.losses.mean_squared_error(labels, logits)

    if mode == tf.estimator.ModeKeys.TRAIN:
        optimizer = params["optimizer"](learning_rate = params["learning_rate"])
        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)

    rmse = tf.metrics.root_mean_squared_error(labels, logits)
    tf.summary.scalar('rmse', rmse)
    eval_metric_ops = {"rmse":rmse}

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

Layers

Different layers can be composed like in Keras models to create complex moels.

```
# Convolutional Layer #1
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)
```

Figure: Example of a convolutional layer with pooling

Estimator interface vs Keras interface

Estimator API Advantages

- Can conduct distributed training with Estimators
- Estimators provide premade estimators unlike Keras where you have to write code to build your model. (`tf.estimator.DNNRegressor`, `tf.estimator.DNNClassifier`, etc.)
- Easier to incorporate custom low-level TensorFlow operations into the deep learning model.

Keras API Advantages

- Simple and intuitive interface
- Can use `tf.keras.estimator.model_to_estimator` to convert a Keras model to an Estimator to run on distributed settings.

Saving, restoring, and serving models

- TensorFlow provides the ability to save model progress during or after training.
- Helpful when sharing models so that other people can recreate your work or to break up long training times.
- Saved TensorFlow models can be used to perform predictions in a production environment using `TensorFlow Serving`.

Tutorials

- TensorFlow website (https://www.tensorflow.org/tutorials/)
- Tensorflow's model garden (https://github.com/tensorflow/models) is a great learning resource for understanding TF models via pre-trained models with real-world training sets.
- Google Colaboratory is a free and hosted Jupyter notebook environment to play around with the TF examples. (https://colab.research.google.com/notebooks/welcome.ipynb)

Questions?

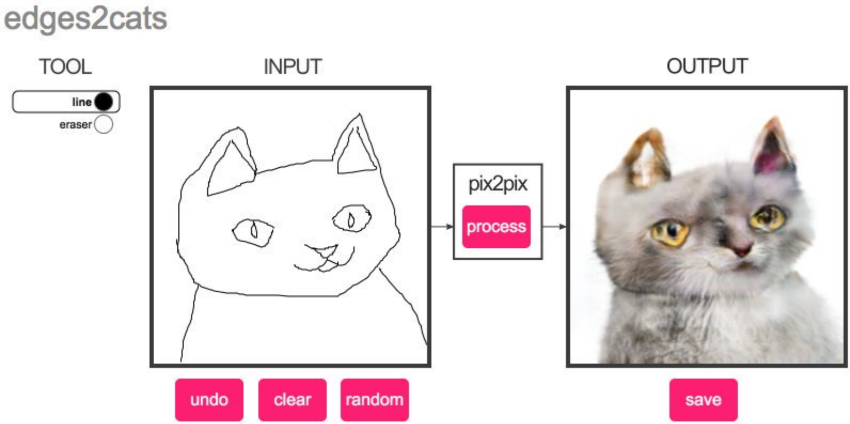


Figure: pix2pix-tensorflow interactive image translation