ALTOROS

TensorFlow™

TensorFlow is an open-source software library for high-performance numerical computation. Its flexible architecture enables to easily deploy computation across a variety of platforms (CPUs, GPUs, and TPUs), as well as mobile and edge devices, desktops, and clusters of servers. TensorFlow comes with strong support for machine learning and deep learning.

High-Level APIs for Deep Learning

Keras is a handy high-level API standard for deep learning models widely adopted for fast prototyping and state-of-the-art research. It was originally designed to run on top of different low-level computational frameworks and therefore the TensorFlow platform fully implements it.

The Sequential API is the most common way to define your neural network model. It corresponds to the mental image we use when thinking about deep learning: a sequence of layers.

```python
import tensorflow as tf
from tensorflow.keras import datasets, layers, models

# Load data set
mnist = datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Construct a neural network model
model = models.Sequential()
model.add(layers.Flatten(input_shape=(28, 28)))
model.add(layers.Dense(512, activation=tf.nn.relu))
model.add(layers.Dropout(0.2))
model.add(layers.Dense(10, activation=tf.nn.softmax))
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train and evaluate the model
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

The Functional API enables engineers to define complex topologies, including multi-input and multi-output models, as well as advanced models with shared layers and models with residual connections.

```python
from tensorflow.keras.layers import Flatten, Dense, Dropout
from tensorflow.keras.models import Model

# Loading data set must be here <...>

inputs = tf.keras.Input(shape=(28, 28))
x = Flatten()(inputs)
x = Dense(512, activation='relu')(x)
x = Dropout(0.2)(x)
predictions = Dense(10, activation='softmax')(x)
model = Model(inputs=inputs, outputs=predictions)

# Compile, train and evaluate the model here <...>
```

A layer instance is called on a tensor and returns a tensor. An input tensor and output tensor can then be used to define a `Model`, which is compiled and trained just as a `Sequential` model. Models are callable by themselves and can be stacked the same way while reusing trained weights.

Transfer learning and fine-tuning of pretrained models saves your time if your data set does not differ significantly from the original one.

```python
import tensorflow as tf
import tensorflow_datasets as tfds

dataset = tfds.load(name='tf_flowers', as_supervised=True)
NUMBER_OF_CLASSES_IN_DATASET = 5
IMG_SIZE = 160

def preprocess_example(image, label):
    image = tf.cast(image, tf.float32)
    image = (image / 127.5) - 1
    image = tf.image.resize(image, (IMG_SIZE, IMG_SIZE))
    return image, label

DATASET_SIZE = 3670
BATCH_SIZE = 32
train = dataset['train'].map(preprocess_example)
train_batches = train.shuffle(DATASET_SIZE).batch(BATCH_SIZE)

# Load MobileNetV2 model pretrained on ImageNet data
model = tf.keras.applications.MobileNetV2(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False, weights='imagenet', pooling='avg')
model.trainable = False

# Add a new layer for multiclass classification
new_output = tf.keras.layers.Dense(
    NUMBER_OF_CLASSES_IN_DATASET, activation='softmax')
new_model = tf.keras.Sequential([model, new_output])
new_model.compile(
    loss=tf.keras.losses.categorical_crossentropy,
    optimizer=tf.keras.optimizers.RMSprop(lr=1e-3),
    metrics=['accuracy'])

# Train the classification layer
new_model.fit(train_batches.repeat(), epochs=10,
              steps_per_epoch=DATASET_SIZE // BATCH_SIZE)
```

After the execution of the given transfer learning code, you can make MobileNetV2 layers trainable and perform fine-tuning of the resulting model to achieve better results.
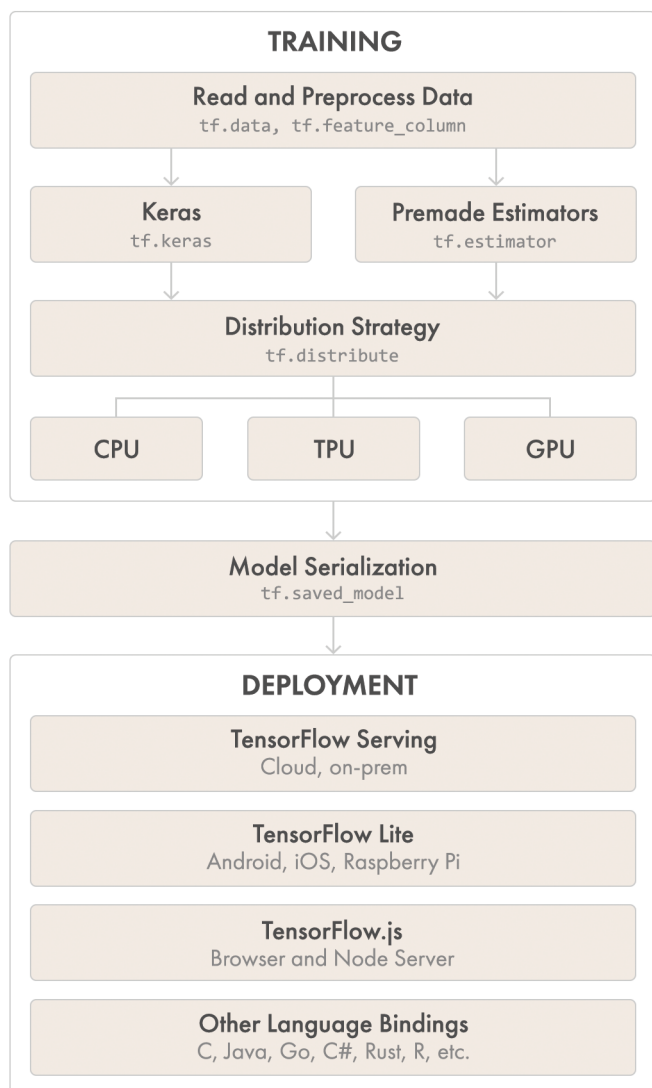
Jupyter Notebook

Jupyter Notebook is a web-based interactive computational environment for data science and scientific computing.

Google Colaboratory is a free notebook environment that requires no setup and runs entirely in the cloud. Use it for jump-starting a machine learning project.

A Reference Machine Learning Workflow

Here's a conceptual diagram and a workflow example:

TRAINING

Read and Preprocess Data
tf.data, tf.feature_column

Keras
tf.keras

Premade Estimators
tf.estimator

Distribution Strategy
tf.distribute

CPU | TPU | GPU

Model Serialization
tf.saved_model

DEPLOYMENT

TensorFlow Serving
Cloud, on-prem

TensorFlow Lite
Android, iOS, Raspberry Pi

TensorFlow.js
Browser and Node Server

Other Language Bindings
C, Java, Go, C#, Rust, R, etc.

01 Load the training data using pipelines created with `tf.data`. As an input, you can use either in-memory data (NumPy), or a local storage, or a remote persistent storage.

02 Build, train, and validate a model with `tf.keras`, or use premade estimators.

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

04 For large ML training tasks, use the Distribution Strategy API for deploying training on Kubernetes clusters within on-premises or cloud environments.

05 Export to SavedModel—an interchange format for TensorFlow Serving, TensorFlow Lite, TensorFlow.js, etc.

The `tf.data API` enables to build complex input pipelines from simple pieces. The pipeline aggregates data from a distributed file system, applies transformation to each object, and merges shuffled examples into training batches.

`tf.data.Dataset` represents a sequence of elements each containing one or more Tensor object(-s). This can be exemplified by a pair of tensors representing an image and a corresponding class label.

```python
import tensorflow as tf

DATASET_URL = "https://archive.ics.uci.edu/ml/machine-" \
              "learning-databases/covtype/covtype.data.gz"
DATASET_SIZE = 387698
dataset_path = tf.keras.utils.get_file(
    fname=DATASET_URL.split('/')[-1], origin=DATASET_URL)

COLUMN_NAMES = [
    'Elevation', 'Aspect', 'Slope',
    'Horizontal_Distance_To_Hydrology',
    'Vertical_Distance_To_Hydrology',
    'Horizontal_Distance_To_Roadways',
    'Hillshade_9am', 'Hillshade_Noon', 'Hillshade_3pm',
    'Horizontal_Distance_To_Fire_Points', 'Soil_Type',
    'Cover_Type']

def _parse_line(line):
    # Decode the line into values
    fields = tf.io.decode_csv(
        records=line, record_defaults=[0.0] * 54 + [0])

    # Pack the result into a dictionary
    features = dict(zip(COLUMN_NAMES,
        fields[:10] + [tf.stack(fields[14:54])] + [fields[-1]]))

    # Extract one-hot encoded class label from the features
    class_label = tf.argmax(fields[10:14], axis=0)
    return features, class_label

def csv_input_fn(csv_path, test=False,
                 batch_size=DATASET_SIZE // 1000):

    # Create a dataset containing the csv lines
    dataset = tf.data.TextLineDataset(filenames=csv_path,
                                      compression_type='GZIP')
    # Parse each line
    dataset = dataset.map(_parse_line)

    # Shuffle, repeat, batch the examples for train and test
    dataset = dataset.shuffle(buffer_size=DATASET_SIZE,
                              seed=42)

    TEST_SIZE = DATASET_SIZE // 10
    return dataset.take(TEST_SIZE).batch(TEST_SIZE) if test \
        else dataset.skip(TEST_SIZE).repeat().batch(batch_size)
```
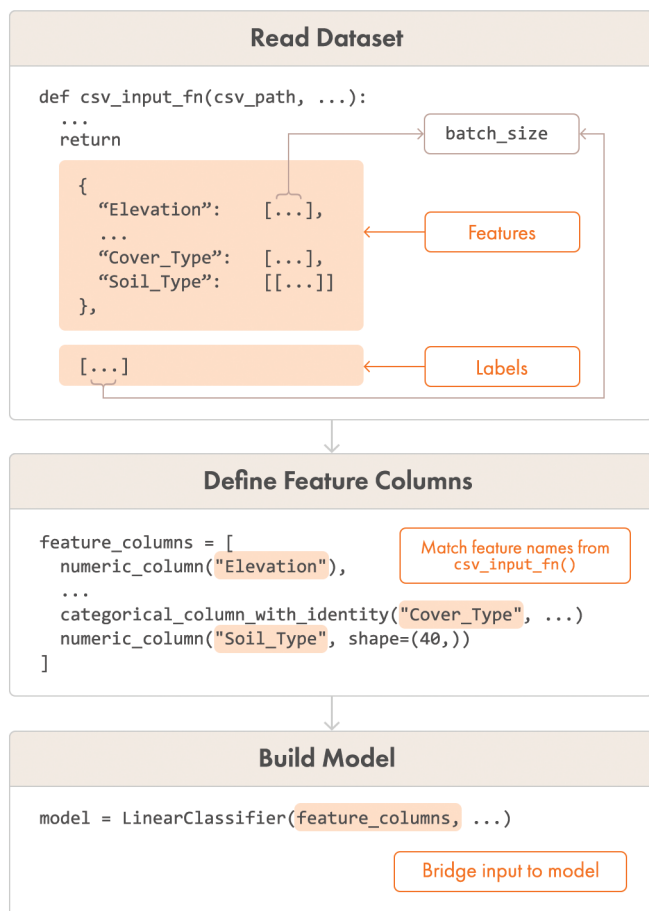
Functions from the `tf.feature_column` namespace are used to put raw data into a TensorFlow data set. A feature column is a high-level configuration abstraction for ingesting and representing features. It does not contain any data but tells the model how to transform the raw data so that it matches the expectation. The exact feature column to choose depends on the feature type and the model type. The continuous feature type is handled by `numeric_column` and can be directly fed into a neural network or a linear model.

Read Dataset

```python
def csv_input_fn(csv_path, ...):
    ...
    return
    {
        "Elevation":    [...],
        ...
        "Cover_Type":   [...],
        "Soil_Type":    [[...]]
    },

    [...]
```

batch_size

Features

Labels

Define Feature Columns

```python
feature_columns = [
    numeric_column("Elevation"),
    ...
    categorical_column_with_identity("Cover_Type", ...)
    numeric_column("Soil_Type", shape=(40,))
]
```

Match feature names from csv_input_fn()

Build Model

```python
model = LinearClassifier(feature_columns, ...)
```

Bridge input to model

Categorical features can be ingested by functions with the `categorical_column_` prefix, but they need to be wrapped by `embedding_column` or `indicator_column` before being fed into Neural Network models. For linear models, `indicator_column` is an internal representation when categorical columns are passed in directly.

```python
feature_columns = [tf.feature_column.numeric_column(name)
                   for name in COLUMN_NAMES[:10]]

feature_columns.append(
    tf.feature_column.categorical_column_with_identity(
        'Cover_Type', num_buckets=8)
)
# Soil_type[1-40] is a tensor of length 40
feature_columns.append(
    tf.feature_column.numeric_column('Soil_Type', shape=(40,))
)
```

The Estimator API provides high-level encapsulation for best practices: model training, evaluation, prediction, and export for serving. The `tf.estimator.Estimator` subclass represents a complete model. Its object creates and manages `tf.Graph` and `tf.Session` for you. Premade estimators include Linear Classifier, DNN Classifier, and Gradient Boosted Trees. `BaselineClassifier` and `BaselineRegressor` will help to establish a simple model for sanity check during further model development.

```python
# Build, train, and evaluate the estimator
model = tf.estimator.LinearClassifier(feature_columns,
                                      n_classes=4)

model.train(input_fn=lambda: csv_input_fn(dataset_path),
            steps=10000)
model.ev aluate(
    input_fn=lambda: csv_input_fn(dataset_path, test=True))
```

SavedModel contains a complete TF program and does not require the original model-building code to run, which makes it useful for deploying and sharing models.

```python
# Export model to SavedModel
_builder = tf.estimator.export. \
    build_parsing_serving_input_receiver_fn
_spec_maker = tf.feature_column.make_parse_example_spec

serving_input_fn = _builder(_spec_maker(feature_columns))

export_path = model.export_saved_model(
    "/tmp/from_estimator/", serving_input_fn)
```

The following code sample shows how to load and use the saved model with Python.

```python
# Import model from SavedModel
imported = tf.saved_model.load(export_path)

# Use imported model for prediction
def predict(new_object):
    example = tf.train.Example()

    # All regular continuous features
    for column in COLUMN_NAMES[:-2]:
        val = new_object[column]
        example.features.feature[column]. \
            float_list.value.extend([val])

    # One-hot encoded feature of 40 columns
    for val in new_object['Soil_Type']:
        example.features.feature['Soil_Type']. \
            float_list.value.extend([val])

    # Categorical column with ID
    example.features.feature['Cover_Type']. \
        int64_list.value.extend([new_object['Cover_Type']])

    return imported.signatures['predict'](
        examples=tf.constant([example.SerializeToString()]))

predict({
    'Elevation': 2296, 'Aspect': 312, 'Slope': 27,
    'Horizontal_Distance_To_Hydrology': 256,
    'Horizontal_Distance_To_Fire_Points': 836,
    'Horizontal_Distance_To_Roadways': 1273,
    'Vertical_Distance_To_Hydrology': 145,
    'Hillshade_9am': 136, 'Hillshade_Noon': 208,
    'Hillshade_3pm': 206,
    'Soil_Type': [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                  0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    'Cover_Type': 6})
```