# TensorFlow

## Overview

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications. Essentially, TensorFlow provides:

* Easy model building
  + It is easy to build and train ML models using intuitive high-level APIs like Keras with eager execution, which makes for immediate model iteration and easy debugging.
* Robust ML production anywhere
  + Easily train and deploy models in the cloud, on-prem, in the browser, or on-device no matter what language you use.
* Powerful experimentation for research
  + A simple and flexible architecture to take new ideas from concept to code, to state-of-the-art models, and to publication faster.

All things considered, TensorFlow is one of the best ways to solve ML problems like Neural Networks, Recommender Systems and Generative adversarial networks. Whether an expert or a beginner, TensorFlow is an end-to-end platform that makes it easy for anyone to build and deploy ML models.1

## Case Studies

* Airbnb
  + Uses TensorFlow to classify images and detect objects at scale.
  + Uses TensorFlow to extract information from satellite images and deliver valuable insights.
* Arm
  + The Hardware Abstraction Layer leads to a more than 4x performance boost to TensorFlow Lite.
* Carousell
  + Uses TensorFlow to improve the buyer and seller experience.
* CEVA
  + Converts TensorFlow trained networks in their Deep Learning processors.
* China Mobile
  + Uses TensorFlow to improve their success rate of network element cutovers.
* Coca Cola
  + Uses TensorFlow to provide mobile proof-at-purchase.
* GE Healthcare
  + Trained a neural network using TensorFlow to identify anatomy MRIs of the brain.
* Google
  + Built Tensorflow to bring ML everywhere. It uses it to power ML implementations in products like Search Gmail and Translate, to aid researchers in new discoveries and even forge advances in humanitarian and environmental challenges.
* inSpace
  + Uses TensorFlow.js for real time toxicity filters in online chat.
* Intel
  + Uses TensorFlow inference performance on the Xeon Scalable processor.

are some of the case studies surrounding TensorFlow.

## Libraries

There are many additional resources that extend TensorFlow, such as:

* TensorBoard
  + A suite of visualization tools to understand, debug, and optimize TensorFlow programs.
* TensorFlow Hub
  + A library for the publication, discovery, and consumption of reusable parts of machine learning models.
* Model Optimization
  + The TensorFlow Model Optimization Toolkit is a suite of tools for optimizing ML models for deployment and execution.
* TensorFlow Federated
  + A framework for machine learning and other computations on decentralized data.
* Neural Structured Learning
  + A learning paradigm to train neural networks by leveraging structured signals in addition to feature inputs.
* TensorFlow Graphics
  + A library of computer graphics functionalities ranging from cameras, lights, and materials to renderers.
* Datasets
  + A collection of datasets ready to use with TensorFlow.
* Serving
  + A TFX serving system for ML models, designed for high-performance in production environments.
* Probability
  + TensorFlow Probability is a library for probabilistic reasoning and statistical analysis.
* MLIR
  + MLIR unifies the infrastructure for high-performance ML models in TensorFlow.
* XLA
  + A domain-specific compiler for linear algebra that accelerates TensorFlow models with potentially no source code changes.
* SIG Addons
  + Extra functionality for TensorFlow, maintained by SIG Addons.
* SIG IO
  + Dataset, streaming, and file system extensions, maintained by SIG IO.

## Breakdown of Basic Elements

### Eager execution

TensorFlow's eager execution is an imperative programming environment that evaluates operations immediately, without building graphs: operations return concrete values instead of constructing a computational graph to run later. This makes it easy to get started with TensorFlow and debug models, and it reduces boilerplate as well. To follow along with this guide, run the code samples below in an interactive python interpreter.

Eager execution is a flexible machine learning platform for research and experimentation, providing:

* An intuitive interface – Structuring code naturally and using Python data structures. Quickly iterate on small models and small data.
* Easier debugging — Call ops directly to inspect running models and test changes. Use standard Python debugging tools for immediate error reporting.
* Natural control flow — Use Python control flow instead of graph control flow, simplifying the specification of dynamic models.

Eager execution supports most TensorFlow operations and GPU acceleration. Enabling eager execution changes how TensorFlow operations behave—now they immediately evaluate and return their values to Python. tf.Tensor objects reference concrete values instead of symbolic handles to nodes in a computational graph. Since there isn't a computational graph to build and run later in a session, it's easy to inspect results using print() or a debugger. Evaluating, printing, and checking tensor values does not break the flow for computing gradients. Eager execution works nicely with NumPy. NumPy operations accept tf.Tensor arguments. The TensorFlow tf.math operations convert Python objects and NumPy arrays to tf.Tensor objects. The tf.Tensor.numpy method returns the object's value as a NumPy ndarray.

### Tensors

Tensors are multi-dimensional arrays with a uniform type (called a dtype). Comparing to numpy, tensors are, kind of, like np.arrays. All tensors are immutable like Python numbers and strings: the contents of a tensor can never be updated, only a new one may be created.

### Variables

A TensorFlow variable is the recommended way to represent shared, persistent state the program manipulates. Variables are created and tracked via the tf.Variable class. A tf.Variable represents a tensor whose value can be changed by running ops on it. Specific ops allow you to read and modify the values of this tensor. Higher level libraries like tf.keras use tf.Variable to store model parameters.

### Ragged Tensors

Data comes in many shapes; tensors should too. Ragged tensors are the TensorFlow equivalent of nested variable-length lists. They make it easy to store and process data with non-uniform shapes, including:

* Variable-length features, such as the set of actors in a movie.
* Batches of variable-length sequential inputs, such as sentences or video clips.
* Hierarchical inputs, such as text documents that are subdivided into sections, paragraphs, sentences, and words.
* Individual fields in structured inputs, such as protocol buffers.

Ragged tensors are supported by more than a hundred TensorFlow operations, including math operations (such as tf.add and tf.reduce\_mean), array operations (such as tf.concat and tf.tile), string manipulation ops (such as tf.substr), control flow operations (such as tf.while\_loop and tf.map\_fn), and many others: There are also a number of methods and operations that are specific to ragged tensors, including factory methods, conversion methods, and value-mapping operations.

### Sparse Tensors

When working with tensors that contain a lot of zero values, it is important to store them in a space- and time-efficient manner. Sparse tensors enable efficient storage and processing of tensors that contain a lot of zero values. Sparse tensors are used extensively in encoding schemes like TF-IDF as part of data pre-processing in NLP applications and for pre-processing images with a lot of dark pixels in computer vision applications.

TensorFlow represents sparse tensors through the tf.SparseTensor object. Currently, sparse tensors in TensorFlow are encoded using the coordinate list (COO) format. This encoding format is optimized for hyper-sparse matrices such as embeddings. The COO encoding for sparse tensors is comprised of:

* values: A 1D tensor with shape [N] containing all nonzero values.
* indices: A 2D tensor with shape [N, rank], containing the indices of the nonzero values.
* dense\_shape: A 1D tensor with shape [rank], specifying the shape of the tensor.

A nonzero value in the context of a tf.SparseTensor is a value that's not explicitly encoded. It is possible to explicitly include zero values in the values of a COO sparse matrix, but these "explicit zeros" are generally not included when referring to nonzero values in a sparse tensor.

### Tensor Slicing

When working on ML applications such as object detection and NLP, it is sometimes necessary to work with sub-sections (slices) of tensors. For example, if the model architecture includes routing, where one layer might control which training example gets routed to the next layer. In this case, one could use tensor slicing ops to split the tensors up and put them back together in the right order.

In NLP applications, using tensor slicing to perform word masking while training is possible. For example, one can generate training data from a list of sentences by choosing a word index to mask in each sentence, taking the word out as a label, and then replacing the chosen word with a mask token.

## Keras

### Sequential Model

A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor. It is not appropriate when:

* The model has multiple inputs or multiple outputs
* Any of the layers has multiple inputs or multiple outputs
* Layer sharing is required
* Non-linear topology is desired (e.g. a residual connection, a multi-branch model)

Once the model architecture is ready, the following steps are:

* Train the model, evaluate it, and run inference.
* Save the model to disk and restore it.
* Speed up model training by leveraging multiple GPUs.

#### Feature extraction

Once a Sequential model has been built, it behaves like a Functional API model. This means that every layer has an input and output attribute. These attributes can be used to do neat things, like quickly creating a model that extracts the outputs of all intermediate layers in a Sequential model.

#### Transfer learning

Transfer learning consists of freezing the bottom layers in a model and only training the top layers.

### Functional API

The Keras functional API is a way to create models that are more flexible than the tf.keras.Sequential API. The functional API can handle models with non-linear topology, shared layers, and even multiple inputs or outputs. The main idea is that a deep learning model is usually a directed acyclic graph (DAG) of layers. So the functional API is a way to build graphs of layers.

#### Training, evaluation, and inference

Training, evaluation, and inference work exactly in the same way for models built using the functional API as for Sequential models. The Model class offers a built-in training loop (the fit() method) and a built-in evaluation loop (the evaluate() method).

#### Save and serialize

Saving the model and serialization work the same way for models built using the functional API as they do for Sequential models. The standard way to save a functional model is to call model.save() to save the entire model as a single file. The model can later be recreated from this file, even if the code that built the model is no longer available. This saved file includes the:

* model architecture
* model weight values (that were learned during training)
* model training config, if any (as passed to compile)
* optimizer and its state, if any (to restart training where you left off)

### An End=to-End example

When passing data to the built-in training loops of a model, either NumPy arrays (if the data is small and fits in memory) or tf.data Dataset objects should be used. A typical, end-to-end workflow is consisting of the following:

* Training
* Validation on a holdout set generated from the original training data
* Evaluation on the test data

Fit() can be called to train the model, by slicing the data into "batches" of size batch\_size, and repeatedly iterating over the entire dataset for a given number of epochs. To evaluate the model, evaluate() is used. To train a model with fit() however, a loss function and an optimizer need to be specified. Additionally but also optionally, some metrics to monitor would be nice to be specified as well. The metrics argument should be a list and the model can have any number of metrics. If it has multiple outputs as well, different losses and metrics for each output can be specified and the contribution of each output to the total loss of the model can also be modulated. Generally speaking however, many built-in optimizers, losses, and metrics are available so it probably won’t be necessary for someone to start creating their own. Some optimizers, metrics and losses available from TensorFlow are:

* Optimizers:
  + SGD() (with or without momentum)
  + RMSprop()
  + Adam()
* Losses:
  + MeanSquaredError()
  + KLDivergence()
  + CosineSimilarity()
* Metrics:
  + AUC()
  + Precision()
  + Recall()

### Saving and Loading

A Keras model consists of multiple components:

* The architecture, or configuration, which specifies what layers the model contain, and how they're connected.
  + A set of weights values (the "state of the model").
  + An optimizer (defined by compiling the model).
  + A set of losses and metrics (defined by compiling the model or calling add\_loss() or add\_metric()).
* The Keras API makes it possible to save all of these pieces to disk at once, or to only selectively save some of them:
  + Saving everything into a single archive in the TensorFlow SavedModel format (or in the older Keras H5 format). This is the standard practice.
  + Saving the architecture / configuration only, typically as a JSON file.
  + Saving the weights values only. This is generally used when training the model.

#### Whole-model saving & loading

The entire model can be saved to a single artifact. It will include:

* The model's architecture/config
* The model's weight values (which were learned during training)
* The model's compilation information (if compile() was called)
* The optimizer and its state, if any (this enables you to restart training where you left)

APIs

* model.save() or tf.keras.models.save\_model()
* tf.keras.models.load\_model()

There are two formats one can use to save an entire model to disk: the TensorFlow SavedModel format, and the older Keras H5 format. The recommended format is SavedModel. It is the default when model.save() is used.

#### SavedModel format

SavedModel is the more comprehensive save format that saves the model architecture, weights, and the traced Tensorflow subgraphs of the call functions. This enables Keras to restore both built-in layers as well as custom objects. The model architecture, and training configuration (including the optimizer, losses, and metrics) are stored in saved\_model.pb. The weights are saved in the variables/ directory. When saving the model and its layers, the SavedModel format stores the class name, call function, losses, and weights (and the config, if implemented). The call function defines the computation graph of the model/layer. In the absence of the model/layer config, the call function is used to create a model that exists like the original model which can be trained, evaluated, and used for inference. Nevertheless, it is always a good practice to define the get\_config and from\_config methods when writing a custom model or layer class. This allows you to easily update the computation later if needed. See the section about Custom objects for more information.

### Keras preprocessing layers

The Keras preprocessing layers API allows developers to build Keras-native input processing pipelines. These input processing pipelines can be used as independent preprocessing code in non-Keras workflows, combined directly with Keras models, and exported as part of a Keras SavedModel.

With Keras preprocessing layers, they can build and export models that are truly end-to-end: models that accept raw images or raw structured data as input; models that handle feature normalization or feature value indexing on their own. The available preprocessing layers are:

* Core preprocessing layers
  + TextVectorization layer: turns raw strings into an encoded representation that can be read by an Embedding layer or Dense layer.
  + Normalization layer: performs feature-wise normalize of input features.
* Structured data preprocessing layers (These layers are for structured data encoding and feature engineering)
  + CategoryEncoding layer: turns integer categorical features into one-hot, multi-hot, or TF-IDF dense representations.
  + Hashing layer: performs categorical feature hashing, also known as the "hashing trick".
  + Discretization layer: turns continuous numerical features into integer categorical features.
  + StringLookup layer: turns string categorical values into integers indices.
  + IntegerLookup layer: turns integer categorical values into integers indices.
  + CategoryCrossing layer: combines categorical features into co-occurrence features. E.g. if you have feature values "a" and "b", it can provide with the combination feature "a and b are present at the same time".
* Image preprocessing layers (These layers are for standardizing the inputs of an image model.)
  + Resizing layer: resizes a batch of images to a target size.
  + Rescaling layer: rescales and offsets the values of a batch of image (e.g. go from inputs in the [0, 255] range to inputs in the [0, 1] range.
  + CenterCrop layer: returns a center crop of a batch of images.
* Image data augmentation layers (These layers apply random augmentation transforms to a batch of images. They are only active during training.)
  + RandomCrop layer
  + RandomFlip layer
  + RandomTranslation layer
  + RandomRotation layer
  + RandomZoom layer
  + RandomHeight layer
  + RandomWidth layer

#### The adapt() method

Some preprocessing layers have an internal state that must be computed based on a sample of the training data. The list of stateful preprocessing layers is:

* TextVectorization: holds a mapping between string tokens and integer indices
* Normalization: holds the mean and standard deviation of the features
* StringLookup and IntegerLookup: hold a mapping between input values and output indices.
* CategoryEncoding: holds an index of input values.
* Discretization: holds information about value bucket boundaries.

Crucially, these layers are non-trainable. Their state is not set during training; it must be set before training, a step called "adaptation". The developer can set the state of a preprocessing layer by exposing it to training data, via the adapt() method.

In addition, adaptable layers always expose an option to directly set state via constructor arguments or weight assignment. If the intended state values are known at layer construction time, or are calculated outside of the adapt() call, they can be set without relying on the layer's internal computation. For instance, if external vocabulary files for the TextVectorization, StringLookup, or IntegerLookup layers already exist, those can be loaded directly into the lookup tables by passing a path to the vocabulary file in the layer's constructor arguments.

## Transfer Learning

Transfer learning consists of taking features learned on one problem, and leveraging them on a new, similar problem. For instance, features from a model that has learned to identify racoons may be useful to kick-start a model meant to identify tanukis.

Transfer learning is usually done for tasks where the dataset has too little data to train a full-scale model from scratch. The most common incarnation of transfer learning in the context of deep learning is the following workflow:

* Take layers from a previously trained model.
* Freeze them, so as to avoid destroying any of the information they contain during future training rounds.
* Add some new, trainable layers on top of the frozen layers. They will learn to turn the old features into predictions on a new dataset.
* Train the new layers on your dataset.

A last, optional step, is fine-tuning, which consists of unfreezing the entire model obtained above (or part of it), and re-training it on the new data with a very low learning rate. This can potentially achieve meaningful improvements, by incrementally adapting the pretrained features to the new data.

# Bibliography

[1] <https://www.tensorflow.org/>

[2] <https://www.tensorflow.org/about/case-studies>

[3] <https://www.tensorflow.org/guide/>