concepts, mental model and API

- tensorflow **core** → low level api for researchers. high level apis like tf.estimator use this.
- tensor is a high dimensional array. **rank** is number of dimensions.
- tf core programming consists of building the graph and running the graph. A computational graph is a series of TensorFlow operations arranged into a graph of nodes. each node is an operation with tensors flowing between nodes.
- To actually evaluate the nodes, we must run the computational graph within a session. A **session** encapsulates the control and state of the TensorFlow runtime.
- `sess.run(nodes)` takes a list of nodes and runs enough of the graph to evaluate those nodes and returns their values in a list.
- we could have tensor nodes that output a tensor or operation nodes that output the result of an operation.
- Tensorboard can display the computation graph.
- inputs are framed as `placeholder`s, which is a promise to provide a value later.
- **Variables** allow us to add trainable parameters to a graph. They are constructed with a type and initial value.
- Constants are initialized when you call `tf.constant`, and their value can never change. By contrast, variables are not initialized when you call `tf.Variable`.
- To initialize all the variables in a TensorFlow program, you must explicitly call a special operation as follows:

```
init = tf.global_variables_initializer()
sess.run(init)
```

  It is important to realize `init` is a handle to the TensorFlow sub-graph that initializes all the global variables. `tf.global_variables_initializer()` returns an op that initializes global variables.
- A variable is created when you first run the `tf.Variable.initializer` operation for that variable in a session. It is destroyed when that `tf.Session.close`.
- TensorFlow relies on a highly efficient C++ backend to do its computation. The connection to this backend is called a session.
- Variables are assigned by `tf.assign(ref, value)` which is an operation that outputs a tensor that is the new value of the ref → note that it really outputs a node which needs to be run for it to output the tensor. everything is an operation/node!
- `tf.train` provides classes that perform different kinds of training, most commonly gradient descent and its variants (Adam, RMSProp etc). Again these are operations! define them and `sess.run()`
- `tf.estimator` is a high level api that simplifies training and evaluation loops and helps manage datasets.
- `tf.estimator` can also help you define, train and evaluate custom models, not just its pre-defined ones.
- making a model work on MNIST is the hello world of machine learning.
- softmax regression is like a 0 hidden layer network that connects input layer to output layer.
- we often want to do expensive compute intensive operations outside python. tf lets us build the computation graph in python and start a session that sends off the computation to a non-python backend (like multi threaded C or GPU etc)

- The common usage for TensorFlow programs is to first create a graph and then launch it in a session. another option is to use `InteractiveSession`, which makes TensorFlow more flexible about how you structure your code. It allows you to interleave operations which build a computation graph with ones that run the graph.
- We make the input vector 2-d. like for 28 x 28 images, we make the input tensor shape `[None, 784]`. Because of matrix multiply libraries, we can feed forward multiple inputs in a batch. So None accounts for a variable number of inputs to be configured by batch size.

```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

- What TensorFlow actually did in that single line was to add new operations to the computation graph. These operations included ones to compute gradients, compute parameter update steps, and apply update steps to the parameters.
- If you have a `Tensor` t, calling `t.eval()` is equivalent to calling `tf.get_default_session().run(t)`
- it takes the loss tensor from the `loss()` function and hands it to a `tf.summary.scalar`, an op for generating summary values into the events file, which can be used by TensorBoard, for example.
- the value of the `loss` tensor may become NaN if the model diverges during training, so better capture this value for logging.
- `tf.estimator` provides higher level APIs with blackbox networks that are reasonably configurable but that let us work a lot more easily than the tf core API. If this is sufficient for your use case, you should probably use this.
- `tf.estimator.inputs.numpy_input_fn` returns input function that would feed dict of numpy arrays into the model. This returns a function outputting `features` and `target` based on the dict of numpy arrays.
- if you're looking to track the model while it trains, you'll likely want to instead use a TensorFlow `SessionRunHook` to perform logging operations.
- For sparse, categorical data (data where the majority of values are 0), you'll instead want to populate a `tf.SparseTensor` (more efficient memory representation)
- We can use neural nets for regression with `tf.estimator.DNNRegressor` . An example is to use a single neuron output layer which simply sums up the output from the previous layer, or maybe a weighted sum with learnt weights. other layers could ReLU, for example.
- To make it easier to understand, debug, and optimize TensorFlow programs, we've included a suite of visualization tools called TensorBoard.
- TensorBoard operates by reading TensorFlow events files, which contain summary data that you can generate when running TensorFlow. for each information you want to record, like loss, learning rate, gradient distribution, weight distribution etc, attach a `tf.summary.scalar`, `tf.summary.histogram` etc op to the node whose value is to be recorded. these are written to event files, which are then read by TensorBoard.
- summary nodes are usually peripheral to your graph: none of the ops you are currently running depend on them. So, to generate summaries, we need to run all of these summary nodes → they won't be run on their own if since no other nodes' output depends on them. Managing them by hand would be tedious, so use `tf.summary.merge_all` to combine them into a single op that generates all the summary data, then just run that single op.
- To run tensorboard, run `tensorboard --logdir=path/to/log-directory`
- Typical TensorFlow graphs can have many thousands of nodes--far too many to see easily all at once, or even to lay out using standard graph tools. To simplify, variable names can be scoped and the visualization uses this information to define a hierarchy on the nodes in the graph. The better your name scopes, the better your visualization.

- TensorBoard structure view: nodes with similar structure (like convolutional layers) have same color
- TensorBoard device view: Name scopes are colored proportionally to the fraction of devices of the operation nodes inside them.
- dotted arrow means control dependency (like initialization). lined arrow means data dependency (flow of tensors between layers)
- Tensor dimensions can also be visualized at each edge using TensorBoard.
- Often it is useful to collect runtime metadata for a run, such as total memory usage, total compute time, and tensor shapes for nodes using `tf.RunOptions()` and `tf.RunMetadata()`
- The TensorBoard Histogram Dashboard displays how the distribution of some `Tensor` in your TensorFlow graph has changed over time.
- A `tf.Tensor` has the following properties:
  - a data type (`float32`, `int32`, or `string`, for example) → every element of tensor has the same datatype.
  - a shape
- tensorflow allows for exact tensor shape specification at graph execution time.
- With the exception of `tf.Variable`, the value of a tensor is immutable, which means that in the context of a single execution tensors only have a single value. However, evaluating the same tensor twice can return different values; for example that tensor can be the result of reading data from disk, or generating a random number.
- it's often convenient to be able to change the shape of a `tf.Tensor`, keeping its elements fixed. This can be done with `tf.reshape`
- It is possible to cast `tf.Tensor`s from one datatype to another using `tf.cast`
- To correctly use `tf.Print` its return value must be used. See the example below

```
t = <<some tensorflow operation>>
tf.Print(t, [t])  # This does nothing
t = tf.Print(t, [t])  # Here we are using the value returned by tf.Print
result = t + 1  # Now when result is evaluated the value of `t` will be printed.
```

- When you evaluate `result` you will evaluate everything `result` depends upon. Since `result` depends upon `t`, and evaluating `t` has the side effect of printing its input (the old value of `t`), `t` gets printed.
- A TensorFlow **variable** is the best way to represent shared, persistent state manipulated by your program.
- Variables are manipulated via the `tf.Variable` class. A `tf.Variable` represents a tensor whose value can be changed by running ops on it. Unlike `tf.Tensor` objects, a `tf.Variable` exists outside the context of a single `session.run` call.
- TensorFlow provides **collections**, which are named lists of tensors or other objects, such as `tf.Variable` instances.
- By default every `tf.Variable` gets placed in the following two collections: * `tf.GraphKeys.GLOBAL_VARIABLES` --- variables that can be shared across multiple devices, * `tf.GraphKeys.TRAINABLE_VARIABLES`--- variables for which TensorFlow will calculate gradients.
- Control dependencies (RAW dependency): Because variables are mutable it's sometimes useful to know what version of a variable's value is being used at any point in time. To force a re-read of the value of a variable after something has happened, you can use `tf.Variable.read_value`. For example:

```
v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
assignment = v.assign_add(1)
```

```
with tf.control_dependencies([assignment]):
  w = v.read_value()  # w is guaranteed to reflect v's value after the
                      # assign_add operation.
```

(not sure why this dependency needs to be specified explicitly)

- **Distributed execution.** By using explicit edges to represent the values that flow between operations, it is possible for TensorFlow to partition your program across multiple devices (CPUs, GPUs, and TPUs) attached to different machines. TensorFlow inserts the necessary communication and coordination between devices.
- **Portability.** The dataflow graph is a language-independent representation of the code in your model. You can build a dataflow graph in Python, store it in a SavedModel, and restore it in a C++ program for low-latency inference.
- A `tf.Graph` object defines a **namespace** for the `tf.Operation` objects it contains. TensorFlow automatically chooses a unique name for each operation in your graph, but giving operations descriptive names can make your program easier to read and debug. You should also add name scopes to each operation. This is used by TensorBoard to group nodes together during visualization of the graph.
- If you want your TensorFlow program to use multiple different devices, the `tf.device` function provides a convenient way to request that all operations created in a particular context are placed on the same device (or type of device).
- A **device specification** has the following form:

```
/job:<JOB_NAME>/task:<TASK_INDEX>/device:<DEVICE_TYPE>:<DEVICE_INDEX>
```

- GPU vs CPU scheduling:

```
with tf.device("/device:CPU:0"):
  # Operations created in this context will be pinned to the CPU.
  img = tf.decode_jpeg(tf.read_file("img.jpg"))

with tf.device("/device:GPU:0"):
  # Operations created in this context will be pinned to the GPU.
  result = tf.matmul(weights, img)
```

- TensorFlow uses the `tf.Session` class to represent a connection between the client program---typically a Python program, although a similar interface is available in other languages---and the C++ runtime. A `tf.Session` object provides access to devices in the local machine, and remote devices using the distributed TensorFlow runtime.
- Since a `tf.Session` owns physical resources (such as GPUs and network connections), it is typically used as a context manager (in a `with` block) that automatically closes the session when you exit the block. It is also possible to create a session without using a `with` block, but you should explicitly call `tf.Session.close` when you are finished with it to free the resources.
- session init takes in a `target` argument. If this argument is left empty (the default), the session will only use devices in the local machine. However, you may also specify a `grpc://` URL to specify the address of a TensorFlow server, which gives the session access to all devices on machines that that server controls. See `tf.train.Server` for details of how to create a TensorFlow server.
- The `tf.train.Saver` class provides methods for saving and restoring models. The `tf.train.Saver` constructor adds `save` and `restore` ops to the graph for all, or a specified list, of the variables in the

graph. The `Saver` object provides methods to run these ops, specifying paths for the checkpoint files to write to or read from.

- TensorFlow saves variables in binary **checkpoint files** that, roughly speaking, map variable names to tensor values.
- When you want to save and load variables, the graph, and the graph's metadata--basically, when you want to save or restore your model--we recommend using SavedModel. **SavedModel** is a language-neutral, recoverable, hermetic serialization format. TensorFlow provides several mechanisms for interacting with SavedModel, including `tf.saved_model` APIs, Estimator APIs and a CLI (`saved_model_cli`).
- The `Dataset` API enables you to build complex input pipelines from simple, reusable pieces. For example, the pipeline for an image model might aggregate data from files in a distributed file system, apply random perturbations to each image, and merge randomly selected images into a batch for training. The pipeline for a text model might involve extracting symbols from raw text data, converting them to embedding identifiers with a lookup table, and batching together sequences of different lengths.
- The `Dataset` API makes it easy to deal with large amounts of data, different data formats, and complicated transformations. A `tf.data.Dataset` represents a sequence of elements, in which each element contains one or more `Tensor`objects.
- A `tf.data.Iterator` provides the main way to extract elements from a dataset. The operation returned by `Iterator.get_next()` yields the next element of a `Dataset` when executed.
- For pre-processing a dataset, you can apply per-element transformations such as `Dataset.map()` (to apply a function to each element), and multi-element transformations such as `Dataset.batch()`
- if your input data are on disk in the recommend TFRecord format, you can construct a `tf.data.TFRecordDataset`.
- A **one-shot** iterator is the simplest form of iterator, which only supports iterating once through a dataset, with no need for explicit initialization. A **reinitializable** iterator can be initialized from multiple different `Dataset` objects. For example, you might have a training input pipeline that uses random perturbations to the input images to improve generalization, and a validation input pipeline that evaluates predictions on unmodified data. These pipelines will typically use different `Dataset` objects that have the same structure (i.e. the same types and compatible shapes for each component).
- The `Dataset` API supports a variety of file formats so that you can process large datasets that do not fit in memory. For example, the TFRecord file format is a simple record-oriented binary format that many TensorFlow applications use for training data. The `tf.data.TFRecordDataset` class enables you to stream over the contents of one or more TFRecord files as part of an input pipeline.
- Many datasets are distributed as one or more text files. The `tf.data.TextLineDataset` provides an easy way to extract lines from one or more text files. Given one or more filenames, a `TextLineDataset` will produce one string-valued element per line of those files.
- For performance reasons, we encourage you to use TensorFlow operations for preprocessing your data whenever possible. However, it is sometimes useful to call upon external Python libraries when parsing your input data. To do so, invoke, the `tf.py_func()` operation in a `Dataset.map()` transformation.
- The simplest form of batching stacks `n` consecutive elements of a dataset into a single element. The `Dataset.batch()` transformation does exactly this.
- The simplest way to iterate over a dataset in multiple epochs is to use the `Dataset.repeat()` transformation. For example, to create a dataset that repeats its input for 10 epochs:

```
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
dataset = tf.data.TFRecordDataset(filenames)
```

```
dataset = dataset.map(...)
dataset = dataset.repeat(10)
dataset = dataset.batch(32)
```

- The `tf.train.MonitoredTrainingSession` API simplifies many aspects of running TensorFlow in a distributed setting. `MonitoredTrainingSession` uses the `tf.errors.OutOfRangeError` to signal that training has completed.
- TensorFlow debugger (**tfdbg**) is a specialized debugger for TensorFlow, available as a CLI. It lets you view the internal structure and states of running TensorFlow graphs during training and inference, which is difficult to debug with general-purpose debuggers such as Python's `pdb` due to TensorFlow's computation-graph paradigm.
- To add support for tfdbg in our example, all that is needed is to add the following lines of code and wrap the Session object with a debugger wrapper.

```
from tensorflow.python import debug as tf_debug
sess = tf_debug.LocalCLIDebugWrapperSession(sess)
```

- Enter the `run` command (or just `r`) at the command prompt:

```
tfdbg> run
```

- The `run` command causes tfdbg to execute until the end of the next `Session.run()` call, which calculates the model's accuracy using a test data set. tfdbg augments the runtime graph to dump all intermediate tensors. After the run ends, tfdbg displays all the dumped tensors values in the *run-end CLI.*
- see the debugger tutorial for commands available with tfdbg.
- you can use the following command to let the debugger repeatedly execute `Session.run()` calls without stopping at the run-start or run-end prompt, until the first `nan` or `inf` value shows up in the graph.

```
tfdbg> run -f has_inf_or_nan
```

- To view the value of the tensor, click the underlined tensor name `cross_entropy/Log:0` or enter the equivalent command:

```
tfdbg> pt cross_entropy/Log:0
```

- Currently, `tfdbg` can debug the `fit()` `evaluate()` methods of tf-learn `Estimator`s. To debug `Estimator.fit()`, create a `LocalCLIDebugHook` and supply it in the `monitors` argument to `fit()` and `evaluate()`
- Often, your model is running on a remote machine or a process that you don't have terminal access to. To perform model debugging in such cases, you can use the `offline_analyzer` binary of `tfdbg` (described below). It operates on dumped data directories. This can be done to both the lower-level `Session` API and the higher-level `Estimator` and `Experiment` APIs.
- Sessions can own resources, such as `tf.Variable`, `tf.QueueBase`, and `tf.ReaderBase`; and these resources can use a significant amount of memory. These resources (and the associated memory) are released when the session is closed, by calling `tf.Session.close`.
- The intermediate tensors that are created as part of a call to `Session.run()` will be freed at or before the end of the call.
- If a TensorFlow operation has both CPU and GPU implementations, the GPU devices will be given priority when the operation is assigned to a device.
- To find out which devices your operations and tensors are assigned to, create the session with `log_device_placement`configuration option set to `True`

- If you would like a particular operation to run on a device of your choice instead of what's automatically selected for you, you can use `with tf.device` to create a device context such that all the operations within that context will have the same device assignment.