Artificial Neural Networks and Deep Learning

Neural networks are leading the current machine learning trend. Whether it's Tensorflow, Keras, CNTK, PyTorch, Caffee, or any other package, they are currently achieving results that few other algorithms have achieved, especially in domains such as image processing. With the advent of fast computers and big data, the neural network algorithms designed in the 1970s are now usable. The big issue, even a decade ago, was that you needed lots of training data that was just not available, and, at the same time, even when you had enough data, the time required to train the model was just too much. This problem is now more or less solved.

The main improvement over the years has been the neural network architecture. The backpropagation algorithm used to update the neural networks is more or less the same as before, but the structure has seen numerous improvements, such as convolutional layers instead of dense layers, or, **Long Short Term Memory** (**LSTM**) for regular recurrent layers.

Here is the plan that we will follow: first a deep dive inside TensorFlow and its API, then we will apply it on convolutional neural networks for image processing, and finally we will tackle recurrent neural networks (specifically the flavor known as LSTM) for image processing and text processing.

Talks about machine learning speed are mainly about neural network speed. Why? Because neural networks are basically matrix multiplications and parallel math functions—blocks that GPUs are very good at.

Using TensorFlow

We already saw some examples of using TensorFlow, and it's now time to understand more about how it works.

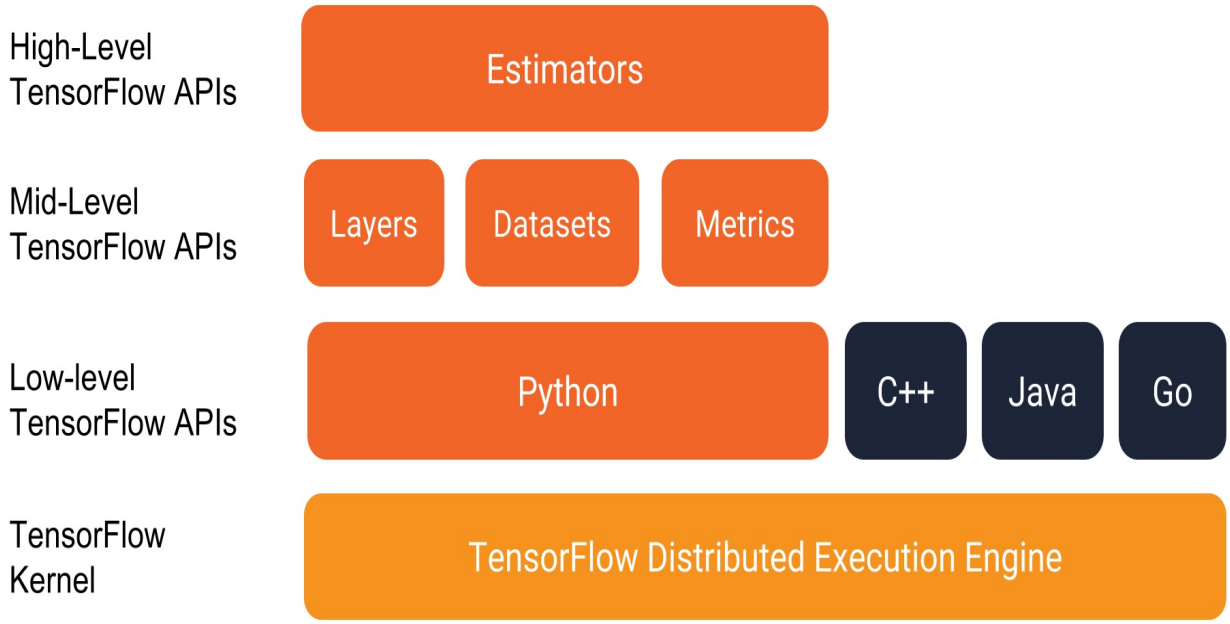
First things first, the name comes of the fact that TensorFlow uses tensors (matrices with more than two dimensions) for all computations. All functions work on these objects, returning either tensors or operations that behave like tensors, with new names defined for all of them. The second part of the name comes from the graph that underlies the data flowing between tensors.

Neural networks were inspired by how the brain works, but it doesn't work as the model use for neural networks. Yes, each neuron is connected to lots of other neurons, but the output is not a product of the input times a transition matrix plus a bias fed inside an activation function. Also, neural networks have layers (deep learning refers to neural networks with more than one so-called hidden layer, meaning neither the input nor the output) with a strict progression in their architecture. A brain has connections all over the place and continuous evolution, whereas a neural network always has a stable output for a given input and a given moment in time (until we get a new tick, as we will see in recurrent networks).

Let's now dive into the TensorFlow graph API.

TensorFlow API

The best way to start is by having a look at the programming environment:



We are obviously interested in the Python stack, and we will mainly focus on layers and metrics. Datasets are interesting, but lots of them are from external contributions, and some are targeted for removal. The scikit learn API is considered as more future-proof, so we won't look at it.

Estimators are the higher-level APIs, but they are not as well developed as the one from scikit learn. As we develop new networks, being able to debug them and check what they have inside their gut is easier in the middle API than the top one, although the fact that all tensors have names makes it possible to get this information outside of the Estimator API.

Lots of online tutorials are still directly using the lower API, and we used it in our regression example by calling `tf.matmult` directly. We think it is better to use the middle- or the high-level API than the others, even if they may sometimes seem more flexible and closer to what you think you need.

Graphs

Graphs are central to TensorFlow, as we saw by the definition of TensorFlow. The default graph contains the structure between objects (placeholders, variables, or constants) as well as the type of these objects (variables are, for instance, trainable variables and all the trainable variables can be retrieved by calling `tf.trainable_variables()`).

It is possible to change the default graph by using the `with` construct:

```
g = tf.Graph()
with g.as_default():
    c = tf.constant("Node in g")
```

So each time we call a TensorFlow function, we add nodes to the default graph (whether we are in a block or not). A graph on its own doesn't do anything. No operation is actually done when we create a new layer, use a metric, or create a placeholder. The only thing we do is add nodes on the graph.

Once we have an interesting graph, we need to execute in what is called a session. This is the only place where TensorFlow will actually execute code and where values can be retrieved from the graph.

Just as for the graph, variables, placeholders, and so on will be put on the best possible device. This device will be the CPU for all platforms, at the time of writing, that's Linux for HIP-capable AMD GPUs or nVidia GPU for Linux and Windows.

They can be pinned to a specific device with the command:

```
with tf.device("/device:CPU:0"):
```

It is sometimes interesting to have the same name for different parts of a graph. To allow this, we can use `name_scope` to prefix the name with a path. Of course, they can be used recursively:

```
var = tf.constant([0, 1, 2, 3])
with tf.name_scope("section"):
    mean = tf.reduce_mean(var)
```

Sessions

It is time to learn a little bit more about sessions. As we saw earlier, TensorFlow only executes the operations inside a session.

The simplest usage of a session is the following:

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    sess.run([mean], feed_dict={})
```

This will initialize the variables by calling their initializer inside the session. In a neural network, we cannot have all variables starting with a zero value, especially the weights in the different layers (some, such as bias, can be initialized to 0). The initializer is then of prime importance and is either set directly for explicit variables, or is implicit when calling mid-level functions (see the different initializers available with TensorFlow and play with them in all our examples).

TensorFlow uses also an external way to get reports, called summaries. They live in the `tf.summary` module, and can track tensors, and preprocess them:

```
tf.summary.histogram(var)
tf.summary.scalar('mean', tf.reduce_mean(var))
```

All these summary reports can then be written and retrieved during a session run and saved by a special object:

```
merged = tf.summary.merge_all()
writer = tf.summary.FileWriter(path/to/log-directory)
with tf.Session() as sess:
    summary, _ = sess.run([merged, train_step], feed_dict={})
    writer.add_summary(summary, i)
```

Tensorboard is a tool provided with TensorFlow that allows us to display these summaries. It can be launched with `tensorboard --logdir=path/to/log-directory`.

If we don't want to use a session directly, the **Estimator** class can be used.

From an estimator, we can call its method train, which takes as an argument a

data generator and the number of steps that we want to run. For instance, this could be:

```
def input_fn():
    features = {'SepalLength': np.array([6.4, 5.0]),
        'SepalWidth': np.array([2.8, 2.3]),
        'PetalLength': np.array([5.6, 3.3]),
        'PetalWidth': np.array([2.2, 1.0])}
    labels = np.array([2, 1])
    return features, labels
estimator.train(input_fn=input_fn , steps=STEPS)
```

Similarly, we can use test to get results from the model:

```
estimator.test(input_fn=input_train)
```

If you want to use more than the simple Estimators already provided with TensorFlow, we suggest to follow the tutorial on the TensorFlow website: https://www.tensorflow.org/get_started/custom_estimators.

Useful operations

In all the previous TensorFlow models, we encountered functions that create layers in TensorFlow. There are a few layers that are more or less inescapable.

The first one is `tf.dense`, connecting all input to a new layer. We saw them in the auto-encoder example, and they take as an `inputs` parameter a tensor (variable, placeholder...) and then `units` the number of output units. By default, it also has bias, meaning that the layer computes `inputs * weights + bias`.

Another important layer that we will see later is `conv2d`. It computes a convolution on an image, and this times it takes the `filters` that will indicate the number of nodes in the output layer. It is what defines convolutional neural networks. Here is the usual formula for the convolution:

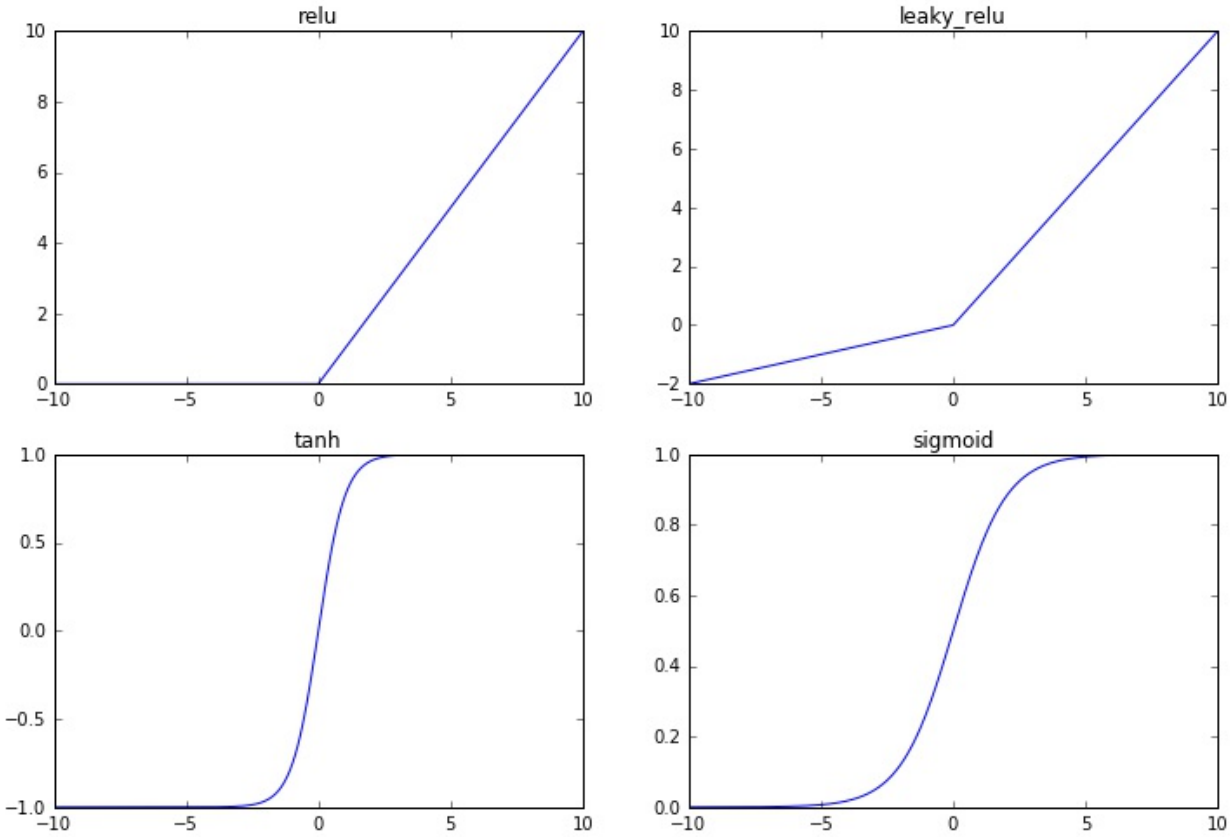
$$conv2d[i,j] = \sum_{k=-filters/2}^{k=filters/2} \sum_{l=-filters/2}^{l=filters/2} kernel[k,l] * input[i-k, j-l]$$

The standard name for the tensor of coefficients in the convolution is called a kernel.

Let's have a look at a few other layers:

- `dropout` will randomly put some weights to zero during the training phase. This is very important in a complex deep-learning network to prevent it from overfitting. We will also see it later.
- `max_pooling2d` is a very important complement to the convolution layer. It selects the maximum of the input on a two-dimensional shape. There is also a one-dimensional version that works after dense layers.

All layers have an `activation` parameter. This activation transforms a linear operation to a nonlinear one. Let's have a look at the most useful ones from the `tf.nn` module:



As we saw earlier, scikit learn provides lots of metrics to compute accuracy, curves, and more. TensorFlow provides similar operations in the `tf.metrics` module.

Saving and restoring neural networks

There are two ways of storing a trained neural network for future use and then restoring it. We will see that they enable this in the convolutional neural network example.

The first one lives in `tf.train`. It is created with the following statement:

```
saver = tf.train.Saver(max_to_keep=10)
```

And then each training step can be saved with:

```
saver.save(sess, './classifier', global_step=step)
```

Here the full graph is saved, but it is possible to only save part of it. We save it all here, and only keep the last 10 saves, and we postfix the name of the save with the step we are at.

Let's say that we saved the final training step with `saver.save(sess, './classifier-final')`. We know we first have to restore the graph state with:

```
new_saver = tf.train.import_meta_graph("classifier-final.meta")
```

This didn't restore the variable state, for this we have to call:

```
new_saver.restore(sess, tf.train.latest_checkpoint('./'))
```

Be aware that only the graph is restored. If you have Python variables pointing to nodes in this graph, you need to restore them before you can use them. This is true for placeholders and operations.

We also have to restore some of our variables:

```
graph = tf.get_default_graph()
training_tf = graph.get_tensor_by_name('is_training:0')
```

This is also a good reason to use proper names for all tensors (placeholders, operations, and so on) as we need to use their name to get a reference to them

again when we restore the graph.

The other mechanism builds on this one and is far more powerful, but we will present the basic usage that mimics the simple one. We first create what is usually called a `builder`:

```
builder = tf.saved_model.builder.SavedModelBuilder(export_dir)
```

The `export_dir` folder is created by the builder here. If it already exists, you have to remove it before creating a new saved model.

Now after the training, we can call it to save the state of the network:

```
builder.add_meta_graph_and_variables(sess, [tf.saved_model.tag_constants.TRAINING])
```

Obviously, we can save more than one network in this object, with far more attributes, but, in our case, we just need to call one function to restore the state:

```
tf.saved_model.loader.load(sess, [tf.saved_model.tag_constants.TRAINING], export_dir
```

Training neural networks

We haven't talked about training neural networks that much. Basically, all optimizations are a gradient descent, the questions are what step length are we going to use, and should we take the previous gradients into account or not?

When computing one gradient, there is also the question of whether we do this for just one new sample or we do it for a multitude of samples at the same time (the batch). Basically, we almost never feed only one sample at a time (as the size of a batch varies, all the placeholders have a first dimension set to `None` indicating that it will be dynamic).

This also imposes the creation of a special layer, `batch_normalization`, that scales the gradient (up or down, so that the layers can be updated in a meaningful manner, so the batch size is important here), and in some network architectures, it will be mandatory. This layer also has two learning parameters, which are the mean and the standard deviation. If they are not important, a simpler batch-normalization layer can be implemented and will be used in an example in `Chapter 12`, *Computer Vision*.

The optimizer we used before is `GradientDescentOptimizer`. It is a simple gradient descent with a fixed step. This is very fragile, as the step length is heavily dependent on the dataset and the model we are using.

Another very important one is `AdamOptimizer`. It is currently one of the most efficient optimizers because it scales the new gradient based on the previous one (trying to mimic the hessian scaling of the Newton approach of cost-function reduction).

Another one that is worth mentioning is `RMSPropOptimizer`. Here, the additional trick is the momentum. Momentum indicates that the new gradient uses a fraction of the previous gradient on top of the new gradient.

The size of the gradient step, or learning rate, is crucial. The selection of an adequate for

it often requires some know-how. The rate must be small enough so that the optimizations makes the network better, but big enough to have efficient first iterations. The improvement of the training is supposed to be fast for the first iterations and then improve slowly (it is globally fitting an often requires some know-how. The rate must be small enough so that the optimizations makes the network better, but big enough to have efficient first iterations. The improvement of the training is supposed to be fast for the first iterations and then improve slowly (it is globally fitting an e^{-t} *curve).*
To avoid over-generalization, it is sometimes advised to stop optimization early (called early stopping), when the improvements are slow. In this context, using collaborative filtering can also achieve better results.
Additional information can be found at `http://ruder.io/optimizing-gradient-descent/`.