High Level Libraries for TensorFlow

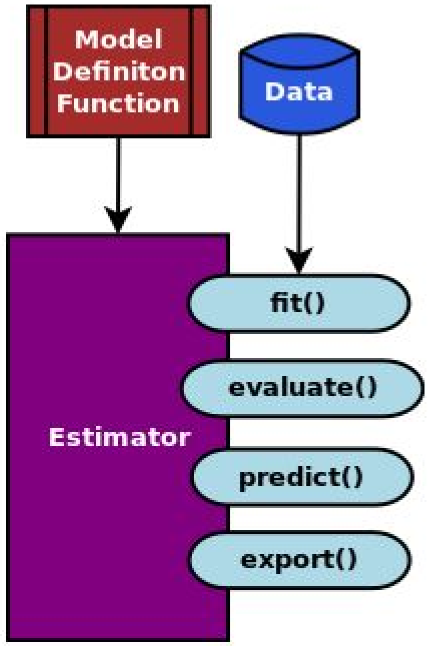
There are several high-level libraries and interfaces (API) for TensorFlow that allow us to build and train models easily and with less amount of code such as TF Learn, TF Slim, Sonnet, PrettyTensor, Keras and recently released TensorFlow Estimators.
We will go through:

- TF Estimator - previously TF Learn
- TF Slim
- TFLearn
- PrettyTensor
- Sonnet

We will build the models for MNIST dataset using all of the five libraries.

1. TF Estimator - previously TF Learn
    a. TF Estimator is a high-level API that makes it simple to create and train models by encapsulating the functionalities for training, evaluating, predicting and exporting. TensorFlow recently re-branded and released the TF Learn package within TensorFlow under the new name **TF Estimator**, probably to avoid confusion with TFLearn package from tflearn.org.
    b. TF Estimator interface design is inspired from the popular machine learning library SciKit Learn, allowing to create the estimator object from different kinds of available models, and then providing four main functions on any kind of estimator:
        i. **estimator.fit()**
        ii. **estimator.evaluate()**
        iii. **estimator.predict()**
        iv. **estimator.export()**

c. The estimator object represents the model, but the model itself is created from the model definition function provided to the estimator.



d. Using the Estimator API instead of building everything in core TensorFlow has the benefit of not worrying about graphs, sessions, initializing variables or other low-level details.
e. TensorFlow provides following pre-built estimators:
   i. **tf.contrib.learn.KMeansClustering**
   ii. **tf.contrib.learn.DNNClassifier**
   iii. **tf.contrib.learn.DNNRegressor**
   iv. **tf.contrib.learn.DNNLinearCombinedRegressor**
   v. **tf.contrib.learn.DNNLinearCombinedClassifier**
   vi. **tf.contrib.learn.LinearClassifier**
   vii. **tf.contrib.learn.LinearRegressor**
   viii. **tf.contrib.learn.LogisticRegressor**
f. The simple workflow in TF Estimator API is as follows:
   i. Find the pre-built Estimator that is relevant to the problem you are trying to solve.
   ii. Write the function to import the dataset.
   iii. Define the columns in data that contain features.
   iv. Create the instance of the pre-built estimator that you selected in step 1.
   v. Train the estimator.
   vi. Use the trained estimator to do evaluation or prediction.
g. Keras library discussed in the next chapter, provides a convenience function to convert Keras models to Estimators: **keras.estimator.model_to_estimator()**.
h. Example - [Click here]

2. TF Slim
    a. TF Slim is a lightweight library built on top of TensorFlow core for defining and training models. TF Slim can be used in conjunction with other TensorFlow low level and high-level libraries such as TF Learn. The TF Slim comes as part of the TensorFlow installation in the package: **tf.contrib.slim**
    b. Run the following command to check if your TF Slim installation is working:

```
python3 -c 'import tensorflow.contrib.slim as slim; eval =
slim.evaluation.evaluate_once'
```

    c. TF Slim provides several modules that can be picked and applied independently and mixed with other TensorFlow packages.

TF Slim module	Module description
arg_scope	Provides a mechanism to apply elements to all graph nodes defined under a scope.
layers	Provides several different kinds of layers such as `fully_connected` , `conv2d` , and many more.
losses	Provides loss functions for training the optimizer
learning	Provides functions for training the models
evaluation	Provides evaluation functions
metrics	Provides metrics functions to be used for evaluating the models
regularizers	Provides functions for creating regularization methods
variables	Provides functions for variable creation
nets	Provides various pre-built and pre-trained models such as VGG16, InceptionV3, ResNet

    d. The simple workflow in TF Slim is as follows:
        i. Create the model using slim layers.
        ii. Provide the input to the layers to instantiate the model.
        iii. Use the logits and labels to define the loss.
        iv. Get the total loss using convenience function **get_total_loss()**.
        v. Create an optimizer.
        vi. Create a training function using convenience function **slim.learning.create_train_op()**, **total_loss** and **optimizer**
        vii. Run the training using the convenience function **slim.learning.train()** and training function defined in the previous step.
    e. Example - Click here
    f. The convenience function **slim.learning.train()** saves the output of the training in checkpoint files in the specified log directory. If you restart the training, it will first check if the checkpoint exists and will resume the training from the checkpoint by default.

3. **TFLearn**

a. TFLearn is a modular library in Python that is built on top of core TensorFlow.
b. TFLearn is different from the TensorFlow Learn package which is also known as TF Learn (with one space in between TF and Learn).
c. TFLearn can be installed in Python 3 with the following command:

```
pip3 install tflearn
```

d. The simple workflow in TFLearn is as follows:
    i. Create an input layer first.
    ii. Pass the input object to create further layers.
    iii. Add the output layer.
    iv. Create the net using an estimator layer such as **regression**.
    v. Create a model from the net created in the previous step.
    vi. Train the model with the **model.fit()** method.
    vii. Use the trained model to predict or evaluate.
e. Creating the TFLearn Layers
    i. Create an input layer first:

```
input_layer = tflearn.input_data(shape=[None,num_inputs]
```

    ii. Pass the input object to create further layers:

```
layer1 = tflearn.fully_connected(input_layer,10,
                                 activation='relu')
layer2 = tflearn.fully_connected(layer1,10,
                                 activation='relu')
```

    iii. Add the output layer:

```
output = tflearn.fully_connected(layer2,n_classes,
                                 activation='softmax')
```

    iv. Create the final net from the estimator layer such as **regression**:

```
net = tflearn.regression(output,
                         optimizer='adam',
                         metric=tflearn.metrics.Accuracy(),
                         loss='categorical_crossentropy'
                        )
```

    v. The TFLearn provides several classes for layers
        1. TFLearn core layers
            a. TFLearn offers the following layers in the **tflearn.layers.core** module:

Layer class	Description
`input_data`	This layer is used to specify the input layer for the neural network.
`fully_connected`	This layer is used to specify a layer where all the neurons are connected to all the neurons in the previous layer.
`dropout`	This layer is used to specify the dropout regularization. The input elements are scaled by `1/keep_prob` while keeping the expected sum unchanged.
`custom_layer`	This layer is used to specify a custom function to be applied to the input. This class wraps our custom function and presents the function as a layer.
`reshape`	This layer reshapes the input into the output of specified shape.
`flatten`	This layer converts the input tensor to a 2D tensor.
`activation`	This layer applies the specified activation function to the input tensor.
`single_unit`	This layer applies the linear function to the inputs.
`highway`	This layer implements the fully connected highway function.
`one_hot_encoding`	This layer converts the numeric labels to their binary vector one-hot encoded representations.
`time_distributed`	This layer applies the specified function to each time step of the input tensor.
`multi_target_data`	This layer creates and concatenates multiple placeholders, specifically used when the layers use targets from multiple sources.

2. TFLearn convolutional layers
    a. TFLearn offers the following layers in the **tflearn.layers.conv** module:

Layer class	Description
`conv_1d`	This layer applies 1D convolutions to the input data
`conv_2d`	This layer applies 2D convolutions to the input data
`conv_3d`	This layer applies 3D convolutions to the input data
`conv_2d_transpose`	This layer applies transpose of conv2_d to the input data
`conv_3d_transpose`	This layer applies transpose of conv3_d to the input data
`atrous_conv_2d`	This layer computes a 2-D atrous convolution
`grouped_conv_2d`	This layer computes a depth-wise 2-D convolution
`max_pool_1d`	This layer computes 1-D max pooling
`max_pool_2d`	This layer computes 2D max pooling
`avg_pool_1d`	This layer computes 1D average pooling
`avg_pool_2d`	This layer computes 2D average pooling
`upsample_2d`	This layer applies the row and column wise 2-D repeat operation
`upscore_layer`	This layer implements the upscore as specified in http://arxiv.org/abs/1411.4038
`global_max_pool`	This layer implements the global max pooling operation
`global_avg_pool`	This layer implements the global average pooling operation
`residual_block`	This layer implements the residual block to create deep residual networks
`residual_bottleneck`	This layer implements the residual bottleneck block for deep residual networks
`resnext_block`	This layer implements the ResNeXt block

3. TFLearn recurrent layers
    a. TFLearn offers the following layers in the **tflearn.layers.recurrent** module:

Layer class	Description
`simple_rnn`	This layer implements the simple recurrent neural network model
`bidirectional_rnn`	This layer implements the bi-directional RNN model
`lstm`	This layer implements the LSTM model
`gru`	This layer implements the GRU model

4. TFLearn normalization layers
    a. TFLearn offers the following layers in the tflearn.layers.normalization module:

Layer class	Description
`batch_normalization`	This layer normalizes the output of activations of previous layers for each batch
`local_response_normalization`	This layer implements the LR normalization
`l2_normalization`	This layer applies the L2 normalization to the input tensors

5. TFLearn embedding layers
    a. TFLearn offers only one layer in the **tflearn.layers.embedding_ops** module:

Layer class	Description
`embedding`	This layer implements the embedding function for a sequence of integer IDs or floats

6. TFLearn merge layers
    a. TFLearn offers the following layers in the **tflearn.layers.merge_ops** module:

Layer class	Description
`merge_outputs`	This layer merges the list of tensors into a single tensor, generally used to merge the output tensors of the same shape
`merge`	This layer merges the list of tensors into a single tensor; you can specify the axis along which the merge needs to be done

7. TFLearn estimator layers
    a. TFLearn offers only one layer in the **tflearn.layers.estimator** module:

Layer class	Description
`regression`	This layer implements the linear or logistic regression

    b. While creating the regression layer, you can specify the optimizer and the loss and metric functions.
    c. TFLearn offers the following optimizer functions as classes in the **tflearn.optimizers** module:
        i. SGD
        ii. RMSprop
        iii. Adam
        iv. Momentum
        v. AdaGrad
        vi. Ftrl
        vii. AdaDelta
        viii. ProximalAdaGrad
        ix. Nesterov
    d. We can create custom optimizers by extending the **tflearn.optimizers.Optimizer** base class.
    e. TFLearn offers the following metric functions as classes or ops in the **tflearn.metrics** module:
        i. Accuracy or accuracy_op
        ii. Top_k or top_k_op
        iii. R2 or r2_op
        iv. WeightedR2 or weighted_r2_op
        v. Binary_accuracy_op

f. We can create custom metrics by extending the **tflearn.metrics.Metric** base class.
g. TFLearn provides the following loss functions, known as objectives, in the **tflearn.objectives** module:
    i. softymax_categorical_crossentropy
    ii. categorical_crossentropy
    iii. binary_crossentropy
    iv. weighted_crossentropy
    v. mean_square
    vi. hinge_loss
    vii. roc_auc_score
    viii. Weak_cross_entropy_2d

8. While specifying the input, hidden, and output layers, we can specify the activation functions to be applied to the output. TFLearn provides the following activation functions in the **tflearn.activations** module:
    a. linear
    b. tanh
    c. sigmoid
    d. softmax
    e. softplus
    f. softsign
    g. relu
    h. relu6
    i. leaky_relu
    j. prelu
    k. elu
    l. crelu
    m. selu

vi. Creating the TFLearn Model

1. Create the model from the net created in the previous step:

```
model = tflearn.DNN(net)
```

2. Types of TFLearn models
    a. **DNN** (Deep Neural Network) model: This class allows you to create a multilayer perceptron from the network that you have created from the layers
    b. **SequenceGenerator** model: This class allows you to create a deep neural network that can generate sequences

vii. Training the TFLearn Model

1. After creating, train the model with the **model.fit()** method:

```
model.fit(X_train,
```

```
          Y_train,
          n_epoch=n_epochs,
          batch_size=batch_size,
          show_metric=True,
          run_id='dense_model')
```

viii. Using the TFLearn Model

1. Use the trained model to predict or evaluate:

```
score = model.evaluate(X_test, Y_test)
print('Test accuracy:', score[0])
```

f. Example - Click here

4. PrettyTensor
   a. PrettyTensor provides a thin wrapper on top of TensorFlow. The objects provided by PrettyTensor support a chainable syntax to define neural networks.
   b. For example, a model could be created by chaining the layers as shown in the following code:

```
model = (X.
         flatten().
         fully_connected(10).
         softmax_classifier(n_classes, labels=Y))
```

   c. PrettyTensor can be installed in Python 3 with the following command:

```
pip3 install prettytensor
```

   d. PrettyTensor offers a very lightweight and extensible interface in the form of a method named **apply()**. Any additional function can be chained to PrettyTensor objects using the **.apply(function, arguments)** method. PrettyTensor will call the **function** and supply the current tensor as the first argument to the **function**.
   e. User-created functions can be added using the **@prettytensor.register** decorator. Details can be found at *https://github.com/google/prettytensor*.
   f. The workflow to define and train models in PrettyTensor is as follows:
      i. Get the data.
      ii. Define hyperparameters and parameters.
      iii. Define the inputs and outputs.
      iv. Define the model.
      v. Define the evaluator, optimizer, and trainer functions.
      vi. Create the runner object.
      vii. Within a TensorFlow session, train the model with the **runner.train_model()** method.
      viii. Within the same session, evaluate the model with the **runner.evaluate_model()** method.
   g. Example - Click here

5. Sonnet

a. Sonnet is an object-oriented library written in Python. It was released by DeepMind in 2017.
b. Sonnet intends to cleanly separate the following two aspects of building computation graphs from objects:
    i. The configuration of objects called modules
    ii. The connection of objects to computation graphs
c. Sonnet can be installed in Python 3 with the following command:

```
pip3 install dm-sonnet
```

d. The modules are defined as sub-classes of the abstract class **sonnet.AbstractModule**. The following modules are available in Sonnet:

Basic modules	`AddBias`, `BatchApply`, `BatchFlatten`, `BatchReshape`, `FlattenTrailingDimensions`, `Linear`, `MergeDims`, `SelectInput`, `SliceByDim`, `TileByDim`, and `TrainableVariable`
Recurrent modules	`DeepRNN`, `ModelRNN`, `VanillaRNN`, `BatchNormLSTM`, `GRU`, and `LSTM`
Recurrent + ConvNet modules	`Conv1DLSTM` and `Conv2DLSTM`
ConvNet modules	`Conv1D`, `Conv2D`, `Conv3D`, `Conv1DTranspose`, `Conv2DTranspose`, `Conv3DTranspose`, `DepthWiseConv2D`, `InPlaneConv2D`, and `SeparableConv2D`
ResidualNets	`Residual`, `ResidualCore`, and `SkipConnectionCore`
Others	`BatchNorm`, `LayerNorm`, `clip_gradient`, and `scale_gradient`

e. We can define our own new modules by creating a subclass of **sonnet.AbstractModule**. An alternate non-recommended way of creating a module from a function is to create an object of the **sonnet.Module** class by passing the function to be wrapped as a module.
f. The workflow to build a model in the Sonnet library is as follows:
    i. Create classes for the dataset and network architecture which inherit from **sonnet.AbstractModule**. In our example, we create an MNIST class and an MLP class.
    ii. Define the parameters and hyperparameters.
    iii. Define the test and train datasets from the dataset classes defined in the preceding step.
    iv. Define the model using the network class defined. As an example, **model = MLP([20, n_classes])** in our case creates an MLP network with two layers of 20 and the **n_classes** number of neurons each.
    v. Define the **y_hat** placeholders for the train and test sets using the model.
        1. Define the loss placeholders for the train and test sets.

2. Define the optimizer using the train loss placeholder.
3. Execute the loss function in a TensorFlow session for the desired number of epochs to optimize the parameters.

vi. Example - Click here

vii. The **__init__** method in each class initializes the class and the related superclass. The **_buildmethod** creates and returns the dataset or the model objects when the class is called.