CISC 867 Lab 1

TensorFlow 2 Tutorial: Get Started in Deep Learning With tf.keras **(adopted from https://machinelearningmastery.com/tensorflow-tutorial-deep-learning-with-tf-keras/)**

TensorFlow is the premier open-source deep learning framework developed and maintained by Google. Although using TensorFlow directly can be challenging, the modern tf.keras API beings the simplicity and ease of use of Keras to the TensorFlow project.

Using tf.keras allows you to design, fit, evaluate, and use deep learning models to make predictions in just a few lines of code. It makes common deep learning tasks, such as classification and regression predictive modeling, accessible to average developers looking to get things done.

In this tutorial, you will discover a step-by-step guide to developing deep learning models in TensorFlow using the tf.keras API.

After completing this tutorial, you will know:

- The difference between Keras and tf.keras and how to install and confirm TensorFlow is working.
- The 5-step life-cycle of tf.keras models and how to use the sequential and functional APIs.
- How to develop MLP models with tf.keras for regression and classification, .

TensorFlow Tutorial Overview

It is a large tutorial and as such, it is divided into five parts; they are:

1. Install TensorFlow and tf.keras
    1. What Are Keras and tf.keras?
    2. How to Install TensorFlow
    3. How to Confirm TensorFlow Is Installed
2. Deep Learning Model Life-Cycle
    1. The 5-Step Model Life-Cycle
    2. Sequential Model API (Simple)
    3. Functional Model API (Advanced)
3. How to Develop Deep Learning Models
    1. Develop Multilayer Perceptron Models

1. Install TensorFlow and tf.keras

In this section, you will discover what tf.keras is, how to install it, and how to confirm that it is installed correctly.

1.1 What Are Keras and tf.keras?

Keras is an open-source deep learning library written in Python.

The project was started in 2015 by Francois Chollet. It quickly became a popular framework for developers, becoming one of, if not the most, popular deep learning libraries.

During the period of 2015-2019, developing deep learning models using mathematical libraries like TensorFlow, Theano, and PyTorch was cumbersome, requiring tens or even hundreds of lines of code to achieve the simplest tasks. The focus of these libraries was on research, flexibility, and speed, not ease of use.

Keras was popular because the API was clean and simple, allowing standard deep learning models to be defined, fit, and evaluated in just a few lines of code.

A secondary reason Keras took-off was because it allowed you to use any one among the range of popular deep learning mathematical libraries as the backend (e.g. used to perform the computation), such as TensorFlow, Theano, and later, CNTK. This allowed the power of these libraries to be harnessed (e.g. GPUs) with a very clean and simple interface.

In 2019, Google released a new version of their TensorFlow deep learning library (TensorFlow 2) that integrated the Keras API directly and promoted this interface as the default or standard interface for deep learning development on the platform.

This integration is commonly referred to as the *tf.keras* interface or API ("*tf*" is short for "*TensorFlow*"). This is to distinguish it from the so-called standalone Keras open source project.

- **Standalone Keras**. The standalone open source project that supports TensorFlow, Theano and CNTK backends.
- **tf.keras**. The Keras API integrated into TensorFlow 2.

The Keras API implementation in Keras is referred to as "*tf.keras*" because this is the Python idiom used when referencing the API. First, the TensorFlow module is imported and named "*tf*"; then, Keras API elements are accessed via calls to *tf.keras*; for example:

```
# example of tf.keras python idiom
import tensorflow as tf
# use keras API
model = tf.keras.Sequential()
...
```

Given that TensorFlow was the de facto standard backend for the Keras open source project, the integration means that a single library can now be used instead of two separate libraries. Further, the standalone Keras project now recommends all future Keras development use the *tf.keras* API.

1.2 How to Install TensorFlow

Before installing TensorFlow, ensure that you have Python installed, such as Python 3.6 or higher.

There are many ways to install the TensorFlow open-source deep learning library.

The most common, and perhaps the simplest, way to install TensorFlow on your workstation is by using *pip*.

For example, on the command line, you can type:

```
sudo pip install tensorflow
```

1.3 How to Confirm TensorFlow Is Installed

Once TensorFlow is installed, it is important to confirm that the library was installed successfully and that you can start using it.

Don't skip this step.

If TensorFlow is not installed correctly or raises an error on this step, you won't be able to run the examples later.

Create a new file called *versions.py* and copy and paste the following code into the file.

```
# check version
import tensorflow
print(tensorflow.__version__)
```

Save the file, then open your command line and change directory to where you saved the file. Then type:

```
python versions.py
```

You should then see the installed version. This confirms that TensorFlow is installed correctly and that we are all using the same version.

2. Deep Learning Model Life-Cycle

In this section, you will discover the life-cycle for a deep learning model and the two tf.keras APIs that you can use to define models.

2.1 The 5-Step Model Life-Cycle

A model has a life-cycle, and this very simple knowledge provides the backbone for both modeling a dataset and understanding the tf.keras API.

The five steps in the life-cycle are as follows:

1. Define the model.
2. Compile the model.
3. Fit the model.
4. Evaluate the model.
5. Make predictions.

Let's take a closer look at each step in turn.

Define the Model

Defining the model requires that you first select the type of model that you need and then choose the architecture or network topology.

From an API perspective, this involves defining the layers of the model, configuring each layer with a number of nodes and activation function, and connecting the layers together into a cohesive model.

Models can be defined either with the Sequential API or the Functional API, and we will take a look at this in the next section.

```
...
# define the model
model = ..
```

Compile the Model

Compiling the model requires that you first select a loss function that you want to optimize, such as mean squared error or cross-entropy.

It also requires that you select an algorithm to perform the optimization procedure, typically stochastic gradient descent, or a modern variation, such as Adam. It may also require that you select any performance metrics to keep track of during the model training process.

From an API perspective, this involves calling a function to compile the model with the chosen configuration, which will prepare the appropriate data structures required for the efficient use of the model you have defined.

The optimizer can be specified as a string for a known optimizer class, e.g. '*sgd*' for stochastic gradient descent, or you can configure an instance of an optimizer class and use that.

For a list of supported optimizers, see this:

- tf.keras Optimizers

```
# compile the model
opt = SGD(learning_rate=0.01, momentum=0.9)
model.compile(optimizer=opt, loss='binary_crossentropy')
```

The three most common loss functions are:

- '*binary_crossentropy*' for binary classification.
- '*sparse_categorical_crossentropy*' for multi-class classification.
- '*mse*' (mean squared error) for regression.

```
...
# compile the model
model.compile(optimizer='sgd', loss='mse')
```

For a list of supported loss functions, see: tf.keras Loss Functions

Metrics are defined as a list of strings for known metric functions or a list of functions to call to evaluate predictions.

For a list of supported metrics, see: tf.keras Metrics

```
...
# compile the model
model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['accuracy'])
```

Fit the Model

Fitting the model requires that you first select the training configuration, such as the number of epochs (loops through the training dataset) and the batch size (number of samples in an epoch used to estimate model error).

Training applies the chosen optimization algorithm to minimize the chosen loss function and updates the model using the backpropagation of error algorithm.

Fitting the model is the slow part of the whole process and can take seconds to hours to days, depending on the complexity of the model, the hardware you're using, and the size of the training dataset.

From an API perspective, this involves calling a function to perform the training process. This function will block (not return) until the training process has finished.

```
...
# fit the model
model.fit(X, y, epochs=100, batch_size=32)
```

While fitting the model, a progress bar will summarize the status of each epoch and the overall training process. This can be simplified to a simple report of model performance each epoch by setting the "*verbose*" argument to 2. All output can be turned off during training by setting "*verbose*" to 0.

```
...
# fit the model
model.fit(X, y, epochs=100, batch_size=32, verbose=0)
```

Evaluate the Model

Evaluating the model requires that you first choose a holdout dataset used to evaluate the model. This should be data not used in the training process so that we can get an unbiased estimate of the performance of the model when making predictions on new data.

The speed of model evaluation is proportional to the amount of data you want to use for the evaluation, although it is much faster than training as the model is not changed.

From an API perspective, this involves calling a function with the holdout dataset and getting a loss and perhaps other metrics that can be reported.

```
...
# evaluate the model
loss = model.evaluate(X, y, verbose=0)
```

Make a Prediction

Making a prediction is the final step in the life-cycle. It is why we wanted the model in the first place.

It requires you have new data for which a prediction is required, e.g. where you do not have the target values.

From an API perspective, you simply call a function to make a prediction of a class label, probability, or numerical value: whatever you designed your model to predict.

You may want to save the model and later load it to make predictions. You may also choose to fit a model on all of the available data before you start using it.

Now that we are familiar with the model life-cycle, let’s take a look at the two main ways to use the tf.keras API to build models: sequential and functional.

```
...
# make a prediction
yhat = model.predict(X)
```

2.2 Sequential Model API (Simple)

The sequential model API is the simplest and is the API that I recommend, especially when getting started.

It is referred to as “*sequential*” because it involves defining a Sequential class and adding layers to the model one by one in a linear manner, from input to output.

The example below defines a Sequential MLP model that accepts eight inputs, has one hidden layer with 10 nodes and then an output layer with one node to predict a numerical value.

```
# example of a model defined with the sequential api
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
# define the model
model = Sequential()
```

```
model.add(Dense(10, input_shape=(8,)))
model.add(Dense(1))
```

Note that the visible layer of the network is defined by the "*input_shape*" argument on the first hidden layer. That means in the above example, the model expects the input for one sample to be a vector of eight numbers.

The sequential API is easy to use because you keep calling *model.add()* until you have added all of your layers.

For example, here is a deep MLP with five hidden layers.

```
# example of a model defined with the sequential api
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
# define the model
model = Sequential()
model.add(Dense(100, input_shape=(8,)))
model.add(Dense(80))
model.add(Dense(30))
model.add(Dense(10))
model.add(Dense(5))
model.add(Dense(1))
```

2.3 Functional Model API (Advanced)

The functional API is more complex but is also more flexible.

It involves explicitly connecting the output of one layer to the input of another layer. Each connection is specified.

First, an input layer must be defined via the *Input* class, and the shape of an input sample is specified. We must retain a reference to the input layer when defining the model.

```
...
# define the layers
x_in = Input(shape=(8,))
```

Next, a fully connected layer can be connected to the input by calling the layer and passing the input layer. This will return a reference to the output connection in this new layer.

```
...
x = Dense(10)(x_in)
```

We can then connect this to an output layer in the same manner.

```
...
x_out = Dense(1)(x)
```

Once connected, we define a Model object and specify the input and output layers. The complete example is listed below.

```
# example of a model defined with the functional api
from tensorflow.keras import Model
from tensorflow.keras import Input
from tensorflow.keras.layers import Dense
# define the layers
x_in = Input(shape=(8,))
x = Dense(10)(x_in)
x_out = Dense(1)(x)
# define the model
model = Model(inputs=x_in, outputs=x_out)
```

As such, it allows for more complicated model designs, such as models that may have multiple input paths (separate vectors) and models that have multiple output paths (e.g. a word and a number).

The functional API can be a lot of fun when you get used to it.

For more on the functional API, see: The Keras functional API in TensorFlow

Now that we are familiar with the model life-cycle and the two APIs that can be used to define models, let's look at developing some standard models.

3. How to Develop Deep Learning Models

In this section, you will discover how to develop, evaluate, and make predictions with standard deep learning models, including Multilayer Perceptrons (MLP), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs).

3.1 Develop Multilayer Perceptron Models

A Multilayer Perceptron model, or MLP for short, is a standard fully connected neural network model.

It is comprised of layers of nodes where each node is connected to all outputs from the previous layer and the output of each node is connected to all inputs for nodes in the next layer.

An MLP is created by with one or more *Dense* layers. This model is appropriate for tabular data, that is data as it looks in a table or spreadsheet with one column for each variable and one row for each variable. There are three predictive modeling problems you may want to explore with an MLP; they are binary classification, multiclass classification, and regression.

Let's fit a model on a real dataset for each of these cases.

Note, the models in this section are effective, but not optimized. See if you can improve their performance. Post your findings in the comments below.

MLP for Binary Classification

We will use the Ionosphere binary (two-class) classification dataset to demonstrate an MLP for binary classification.

This dataset involves predicting whether a structure is in the atmosphere or not given radar returns.

The dataset will be downloaded automatically using Pandas, but you can learn more about it here.

- Ionosphere Dataset (csv).
- Ionosphere Dataset Description (csv).

We will use a LabelEncoder to encode the string labels to integer values 0 and 1. The model will be fit on 67 percent of the data, and the remaining 33 percent will be used for evaluation, split using the train_test_split() function.

It is a good practice to use '*relu*' activation with a '*he_normal*' weight initialization. This combination goes a long way to overcome the problem of vanishing gradients when training deep neural network models. For more on ReLU, see the tutorial:

- A Gentle Introduction to the Rectified Linear Unit (ReLU)

The model predicts the probability of class 1 and uses the sigmoid activation function. The model is optimized using the adam version of stochastic gradient descent and seeks to minimize the cross-entropy loss.

The complete example is listed below.

```
# mlp for multiclass classification
from numpy import argmax
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
# load the dataset
path = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv'
df = read_csv(path, header=None)
# split into input and output columns
X, y = df.values[:, :-1], df.values[:, -1]
# ensure all data are floating point values
X = X.astype('float32')
# encode strings to integer
y = LabelEncoder().fit_transform(y)
# split into train and test datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
# determine the number of input features
n_features = X_train.shape[1]
# define model
model = Sequential()
model.add(Dense(10, activation='relu', kernel_initializer='he_normal',
input_shape=(n_features,)))
model.add(Dense(8, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(3, activation='softmax'))
# compile the model
```

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
# fit the model
model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=0)
# evaluate the model
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print('Test Accuracy: %.3f' % acc)
# make a prediction
row = [5.1,3.5,1.4,0.2]
yhat = model.predict([row])
print('Predicted: %s (class=%d)' % (yhat, argmax(yhat)))
```

Running the example first reports the shape of the dataset, then fits the model and evaluates it on the test dataset. Finally, a prediction is made for a single row of data.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

What results did you get? Can you change the model to do better?
Post your findings to the comments below.

In this case, we can see that the model achieved a classification accuracy of about 98 percent and then predicted a probability of a row of data belonging to each class, although class 0 has the highest probability.

```
(100, 4) (50, 4) (100,) (50,)
Test Accuracy: 0.980
Predicted: [[0.8680804 0.12356871 0.00835086]] (class=0)
```

MLP for Regression

We will use the Boston housing regression dataset to demonstrate an MLP for regression predictive modeling.

This problem involves predicting house value based on properties of the house and neighborhood.

The dataset will be downloaded automatically using Pandas, but you can learn more about it here.

- Boston Housing Dataset (csv).
- Boston Housing Dataset Description (csv).

This is a regression problem that involves predicting a single numerical value. As such, the output layer has a single node and uses the default or linear activation function (no activation function). The mean squared error (mse) loss is minimized when fitting the model.

Recall that this is a regression, not classification; therefore, we cannot calculate classification accuracy. For more on this, see the tutorial:

- Difference Between Classification and Regression in Machine Learning

The complete example of fitting and evaluating an MLP on the Boston housing dataset is listed below.

```python
# mlp for regression
from numpy import sqrt
from pandas import read_csv
from sklearn.model_selection import train_test_split
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
# load the dataset
path = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.csv'
df = read_csv(path, header=None)
# split into input and output columns
X, y = df.values[:, :-1], df.values[:, -1]
# split into train and test datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
# determine the number of input features
n_features = X_train.shape[1]
# define model
model = Sequential()
model.add(Dense(10, activation='relu', kernel_initializer='he_normal',
input_shape=(n_features,)))
model.add(Dense(8, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(1))
# compile the model
model.compile(optimizer='adam', loss='mse')
# fit the model
model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=0)
# evaluate the model
error = model.evaluate(X_test, y_test, verbose=0)
print('MSE: %.3f, RMSE: %.3f' % (error, sqrt(error)))
# make a prediction
row = [0.00632,18.00,2.310,0,0.5380,6.5750,65.20,4.0900,1,296.0,15.30,396.90,4.98]
yhat = model.predict([row])
print('Predicted: %.3f' % yhat)
```

Running the example first reports the shape of the dataset then fits the model and evaluates it on the test dataset. Finally, a prediction is made for a single row of data.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

What results did you get? Can you change the model to do better?
Post your findings to the comments below.

In this case, we can see that the model achieved an MSE of about 60 which is an RMSE of about 7 (units are thousands of dollars). A value of about 26 is then predicted for the single example.

```
(339, 13) (167, 13) (339,) (167,)
MSE: 60.751, RMSE: 7.794
Predicted: 26.983
```