### Step 1) Import the data

First of all, you need to import the necessary library. You can import the MNIST dataset using scikit learn as shown in the TensorFlow Neural Network example below.

The MNIST dataset is the commonly used dataset to test new techniques or algorithms. This dataset is a collection of 28×28 pixel image with a handwritten digit from 0 to 9. Currently, the lowest error on the test is 0.27 percent with a committee of 7 convolutional neural networks.

import numpy as np

import tensorflow as tf

np.random.seed(1337)

You can download scikit learn temporarily at this address. Copy and paste the dataset in a convenient folder. To import the data to python, you can use fetch\_mldata from scikit learn. Paste the file path inside fetch\_mldata to fetch the data.

from sklearn.datasets import fetch\_mldata

mnist = fetch\_mldata(' /Users/Thomas/Dropbox/Learning/Upwork/tuto\_TF/data/mldata/MNIST original')

print(mnist.data.shape)

print(mnist.target.shape)

After that, you import the data and get the shape of both datasets.

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(mnist.data, mnist.target, test\_size=0.2, random\_state=42)

y\_train = y\_train.astype(int)

y\_test = y\_test.astype(int)

batch\_size =len(X\_train)

print(X\_train.shape, y\_train.shape,y\_test.shape )

### Step 2) Transform the data

In the previous tutorial, you learnt that you need to transform the data to limit the effect of outliers. In this Neural Networks tutorial, you will transform the data using the min-max scaler. The formula is:

(X-min\_x)/(max\_x - min\_x)

Scikit learns has already a function for that: MinMaxScaler()

## resclae

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

# Train

X\_train\_scaled = scaler.fit\_transform(X\_train.astype(np.float64))

# test

X\_test\_scaled = scaler.fit\_transform(X\_test.astype(np.float64))

### Step 3) Construct the tensor

You are now familiar with the way to create tensor in Tensorflow. You can convert the train set to a numeric column.

feature\_columns = [tf.feature\_column.numeric\_column('x', shape=X\_train\_scaled.shape[1:])]

### Step 4) Build the model

The architecture of the neural network contains 2 hidden layers with 300 units for the first layer and 100 units for the second one. We use these value based on our own experience. You can tune theses values and see how it affects the accuracy of the network.

To build the model, you use the estimator DNNClassifier. You can add the number of layers to the feature\_columns arguments. You need to set the number of classes to 10 as there are ten classes in the training set. You are already familiar with the syntax of the estimator object. The arguments features columns, number of classes and model\_dir are precisely the same as in the previous tutorial. The new argument hidden\_unit controls for the number of layers and how many nodes to connect to the neural network. In the code below, there are two hidden layers with a first one connecting 300 nodes and the second one with 100 nodes.

To build the estimator, use tf.estimator.DNNClassifier with the following parameters:

* feature\_columns: Define the columns to use in the network
* hidden\_units: Define the number of hidden neurons
* n\_classes: Define the number of classes to predict
* model\_dir: Define the path of TensorBoard

estimator = tf.estimator.DNNClassifier(

feature\_columns=feature\_columns,

hidden\_units=[300, 100],

n\_classes=10,

model\_dir = '/train/DNN')

### Step 5) Train and evaluate the model

You can use the numpy method to train the model and evaluate it

# Train the estimator

train\_input = tf.estimator.inputs.numpy\_input\_fn(

x={"x": X\_train\_scaled},

y=y\_train,

batch\_size=50,

shuffle=False,

num\_epochs=None)

estimator.train(input\_fn = train\_input,steps=1000)

eval\_input = tf.estimator.inputs.numpy\_input\_fn(

x={"x": X\_test\_scaled},

y=y\_test,

shuffle=False,

batch\_size=X\_test\_scaled.shape[0],

num\_epochs=1)

estimator.evaluate(eval\_input,steps=None)

**Output:**

{'accuracy': 0.9637143,

'average\_loss': 0.12014342,

'loss': 1682.0079,

'global\_step': 1000}

The current architecture leads to an accuracy on the the evaluation set of 96 percent.

### Step 6) Improve the model

You can try to improve the model by adding regularization parameters.

We will use an Adam optimizer with a dropout rate of 0.3, L1 of X and L2 of y. In TensorFlow Neural Network, you can control the optimizer using the object train following by the name of the optimizer. [TensorFlow](https://www.guru99.com/what-is-tensorflow.html) is a built-in API for the Proximal AdaGrad optimizer.

To add regularization to the deep neural network, you can use tf.train.ProximalAdagradOptimizer with the following parameter

* Learning rate: learning\_rate
* L1 regularization: l1\_regularization\_strength
* L2 regularization: l2\_regularization\_strength

estimator\_imp = tf.estimator.DNNClassifier(

feature\_columns=feature\_columns,

hidden\_units=[300, 100],

dropout=0.3,

n\_classes = 10,

optimizer=tf.train.ProximalAdagradOptimizer(

learning\_rate=0.01,

l1\_regularization\_strength=0.01,

l2\_regularization\_strength=0.01

),

model\_dir = '/train/DNN1')

estimator\_imp.train(input\_fn = train\_input,steps=1000)

estimator\_imp.evaluate(eval\_input,steps=None)

**Output:**

{'accuracy': 0.95057142,

'average\_loss': 0.17318928,

'loss': 2424.6499,

'global\_step': 2000}

The values chosen to reduce the over fitting did not improve the model accuracy. Your first model had an accuracy of 96% while the model with L2 regularizer has an accuracy of 95%. You can try with different values and see how it impacts the accuracy.