Food for Thought
Is it possible to efficiently train machine learning algorithms on huge datasets?
How can you implement a complex neural network using TensorFlow?
What high-level APIs exist that work with the architecture of TensorFlow?
What operating systems and programming languages are compatible with TensorFlow?
How do you use TensorFlow and TensorBoard to create data visualizations?

Background
The TensorFlow library, developed by the Google Brain team, is a powerful resource for extremely large-scale computation and machine learning applications (on the order of billions of instances and millions of features). On the front end, a user can create graphs of computations to perform, and on the back end, TensorFlow will turn those graphs into optimized C++ operations. TensorFlow supports parallel processing and distributed computing to run on multiple processors or a network of servers, and it can run on Windows, Linux, and macOS. The name TensorFlow derives from the nature of inputs and outputs to its operations, which are multidimensional arrays called tensors (in the Python API, tensors are represented as NumPy ndarrays).

Installing TensorFlow
If you have a normal installation of Python, you can use pip to install TensorFlow, like so:

```
$ pip install --upgrade tensorflow
```

If you are using a virtualenv environment, you have to activate it first, then use the pip command as above:

```
$ cd $ML_PATH     #ML_PATH is your machine learning working directory
$ source env\bin\activate
```

If you are using Anaconda, you can create and activate a TensorFlow environment in an Anaconda command prompt (or a terminal window if your OS is macOS or Linux):

```
$ conda create -n tf tensorflow  #tf stands in for what you want to name your environment
$ conda activate tf
```

Creating a Computation Graph
Now that everything is installed and activated, you can get started by simply creating a blank Python file and writing some code similar to the following few lines. I've created two TensorFlow Variables called x and y, given them values of 2 and 5, respectively, and computed the value of a function f which equals $3x^2 - 4xy + y^2$.

```
import tensorflow as tf

x = tf.Variable(2, name="x")
y = tf.Variable(5, name="y")
f = 3*x*x - 4*x*y + y*y
```

Running a Graph in a Session
While the above code might look like it does some variable creation, initialization, and evaluation, you'll need to run a TensorFlow session in order to actually evaluate the result of a computation graph. You can do this in the same program as you wrote the above code in.

```
init = tf.global_variables_initializer()
with tf.Session() as sess:
```

```
   init.run()
   result = f.eval()
   print(result)
```

Managing Multiple Graphs
By default, every node you create using TensorFlow will be added to the same (default) graph. If you want to add a node to a different graph, you can create a Graph, set it as the default, and make the changes you want. If you want to erase all nodes from a graph, you can use the reset_default_graph function.

```
x1 = tf.Variable(19.92)
my_graph1 = tf.Graph()
with my_graph1.as_default():
   x2 = tf.Variable(-34.119)
tf.reset_default_graph() #resets the graph containing the x1 node
```

Linear Regression (Normal Equation)
I'm going to use TensorFlow to train three linear regression algorithms on the California housing data. The first one is more of a test computation than a machine learning algorithm because it makes use of the closed form of the linear regression model—the normal equation.

```
import numpy as np
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing()
m, n = housing.data.shape
housing_bias = np.c_[np.ones((m,1)), housing.data]
X = tf.constant(housing_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1,1), dtype=tf.float32, name="y")
XT = tf.transpose(X)
theta = tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)

with tf.Session() as sess:
   theta_value = theta.eval()
```

Linear Regression (Batch Gradient Descent)
The next linear regression method I am going to implement, and really the first test of TensorFlow's processing capabilities, is a batch gradient descent algorithm. It is important to note that using any gradient descent algorithm requires that all input vectors be on the same scale, so I will first scale the data using the StandardScaler class from sklearn.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_housing_bias = scaler.fit_transform(housing_bias)

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1,1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
```

```
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()
with tf.Session() as sess:
   sess.run(init)
   for epoch in range(n_epochs):
      if epoch % 50 == 0:
         print("Epoch", epoch, "MSE =", mse.eval())
      sess.run(training_op)
   best_theta = theta.eval()
```

Linear Regression (**Mini-batch Gradient Descent**)
To implement mini-batch gradient descent, we need to make a few changes to the code we have for batch gradient descent. Specifically, we need to replace the declarations of X and y with placeholders, define the batch size, create a function to fetch each mini-batch, and use the feed_dict parameter to get the values of X and y whenever we need them during evaluation.

```
#replacing the declarations of X and y
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")

#added somewhere before the declaration of init and the execution section
batch_size = 100
n_batches = int(np.ceil(m / batch_size))

#the new execution section (the declaration of init is omitted here, but still necessary)
def fetch_batch(epoch, batch_index, batch_size):
   np.random.seed(epoch * n_batches + batch_index)
   indices = np.random.randint(m, size=batch_size)
   X_batch = scaled_housing_bias[indices]
   y_batch = housing.target.reshape(-1, 1)[indices]
   return X_batch, y_batch

with tf.Session() as sess:
   sess.run(init)
   for epoch in range(n_epochs):
      for batch in range(n_batches):
         X_batch, y_batch = fetch_batch(epoch, batch, batch_size)
         sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
   best_theta = theta.eval()
```

Resources
This code is adapted from chapter 9 of the textbook and the corresponding Jupyter notebook, which can be found here:
https://nbviewer.jupyter.org/github/ageron/handson-ml/blob/master/09_up_and_running_with_tensorflow.ipynb
TensorFlow tutorials and guide: https://www.tensorflow.org/tutorials | https://www.tensorflow.org/guide

Creating visualizations with TensorBoard: https://www.tensorflow.org/tensorboard/