The Gentlest Introduction to Tensorflow

Summary: Tensorflow (TF) is Google's attempt to put the power of Deep Learning into the hands of developers around the world. It comes with a beginner & an advanced tutorial, as well as a course on Udacity. However, the materials attempt to introduce both ML and TF concurrently to solve a multi-feature problem—character recognition, which albeit interesting, unnecessarily convolutes understanding. In this series of articles, we present the gentlest introduction to TF that starts off by showing how to do linear regression for a single feature problem, and expand from there.

This is part of a series:

- Part 1 (this article): Linear regression with Tensorflow for single feature single outcome model
- Part 2: Tensorflow training illustrated in diagrams/code, and exploring training variations

Introduction

We are going to solve an overly simple, and unrealistic problem, which has the upside of making understanding the concepts of ML and TF easy. We want to predict a single scalar outcome, house price (in $) based on a single feature, house size (in square meters, sqm). This eradicates the need to handle multi-dimensional data, enabling us to focus solely on defining a model, implementing, and training it in TF.

Machine Learning (ML) In Brief

We start with a set of data points that we have collected (chart below), each representing the relationship between two values—an outcome (house price) and the influencing feature (house size).
However, we cannot predict values for features that we don't have data points for (chart below)

We can use ML to discover the relationship (the 'best-fit prediction line' in the chart below), such that given a feature value that is not part of the data points, we can predict the outcome accurately (the intersection between the feature value and the prediction line).
Step 1: Choose a Model

Model Types

To do prediction using ML, we need to choose a model that can best-fit the data that we have collected.

We can choose a linear (straight line) model, and tweak it to match the data points by changing its steepness/gradient and position.

We can also choose an exponential (curve) model, and tweak it to match the same set of data points by changing its curvature and position.
Cost Function

To compare which model is a better-fit more rigorously, we define best-fit mathematically as a **cost function** that we need to minimize. An example of a cost function can simply be the absolute sum of the differences between the actual outcome represented by each data point, and the prediction of the outcome (the vertical projection of the actual outcome onto the best-fit line). Graphically the cost is depicted by the sum of the length of the blue lines in the chart below.

**NOTE**: More accurately the cost function is often the squared of the difference between actual and predicted outcome, because the difference can sometimes be negative; this is also known as **min least-squared**.

Linear Model In Brief

In the spirit of keeping things simple, we will model our data points using a linear model. A linear model is represented mathematically as:
\[ y = W \cdot x + b \]

Where:

- \( x \): house size, in sqm
- \( y \): predicted house price, in $

To tweak the model to best fit our data points, we can:

- Tweak \( W \) to change the gradient of the linear model
  
  \[ W_1 < W_2 < W_3 \]

- Tweak \( b \) to change the position of the linear model
  
  \[ b_1 < b_2 < b_3 \]

By going through many values of \( W, b \), we can eventually find a best-fit linear model that minimizes the cost function. Besides randomly trying different values, is there a better way to explore the \( W, b \) values quickly?
Gradient Descent

If you are on an expansive plateau in the mountains, when trying to descent to the lowest point, your viewpoint looks like this.

The direction of descent is not obvious! The best way to descend is then to perform gradient descent:

- Determine the direction with the steepest downward gradient at current position
- Take a step of size X in that direction
- Repeat & rinse; this is known as training

Minimizing the cost function is similar because, the cost function is undulating like the mountains (chart below), and we are trying to find the minimum point, which we can similarly achieve through gradient descent.

With the concepts of linear model, cost function, and gradient descent in hand, we are ready to use TF.

Step 2: Create the Model in TF

Linear Model in TF
The 2 basic TF components are:

**Placeholder**: Represent an entry point for us to feed actual data values into the model when performing gradient descent, i.e., the house sizes \(x\), and the house prices \(y\).

\[x = \text{tf.placeholder(tf.float32, [None, 1])}\]

**Variable**: Represent a variable that we are trying to find ‘good’ values that minimizes the cost function, e.g., \(W\), and \(b\).

\[W = \text{tf.Variable(tf.zeros([1, 1]))}\]
\[b = \text{tf.Variable(tf.zeros([1]))}\]

The linear model \(y = Wx + b\) in TF then becomes:

\[y = \text{tf.matmul}(x, W) + b\]

**Cost Function in TF**

Similarly to feed actual house prices \(y\) of the data points into the model, we create a placeholder.

\[y_\_ = \text{tf.placeholder(tf.float32, [None, 1])}\]

Our cost function of least-min squared becomes:
Data
Since we do not have actual data points for house price \( y_\) , house size \( x \) , we generate them.

```python
for i in range(100):
    // Create fake data for actual data
    xs = np.array([i])
    ys = np.array([2*i])
```

We set the house price (ys) to always be 2 times the house size (xs) for simplicity.

Gradient Descent
With the linear model, cost function, and data, we can start performing gradient descent to minimize the cost function, to obtain the ‘good’ values for \( W, b \).

```python
train_step = tf.train.GradientDescentOptimizer(0.00001).minimize(cost)
```

The 0.00001 is the size of the ‘step’ we take in the direction of steepest gradient each time perform a training step; this is also called learning rate.

Step 3: Train the Model
Training involves performing gradient descent a pre-determined number of times or until the cost is below a pre-determined threshold.

TF Quirks
All variables needs to be initialize at the start of training otherwise they may
hold remnant values from previous execution.

```
init = tf.initialize_all_variables()
```

## TF Session

Although TF is a python library, and python is an interpreted language, TF operations, by default are NOT interpreted for performance reasons. Thus the `init` above is NOT executed. Instead TF executes within a session; create a session (`sess`) and then execute stuff using `sess.run()`.

```
sess = tf.Session()
sess.run(init)
```

Similarly we execute the `train_step` above within a loop by calling it within `sess.run()`.

```python
for i in range(steps):
    # Create fake data for y = W.x + b where W = 2, b = 0
    xs = np.array([[i]])
    ys = np.array([[2*i]])
    # Train
    feed = { x: xs, y_: ys }
    sess.run(train_step, feed_dict=feed)

    print("After %d iteration:" % i)
    print("W: %f" % sess.run(W))
    print("b: %f" % sess.run(b))
```

The reason why you need to feed actual data points into `feed`, which is composed of `x, y` is that TF resolves the `train_step` into its dependencies:
At the bottom of the dependencies are the placeholders $x$, $y$; and as we learned earlier `tfplaceholders` are used to indicate where we will feed actual data point values house price ($y$), and house size ($x$).

**Result**

The print statement in the loop will show how TF learn the ‘good’ values for $W$, and $b$ over each iteration.

```
After 0 iteration:
W: 0.000000
b: 0.000000
After 1 iteration:
W: 0.000040
b: 0.000040
...
...
After 98 iteration:
W: 1.997181
b: 0.051121
After 99 iteration:
W: 1.997832
b: 0.051126
```

**Wrapping Up**

We have learned about Machine Learning in its simplest form; predict an outcome from a single feature. We chose a linear model (for simplicity) to fit our data points, define a cost function to represent best-fit, and train our model by repeatedly tweaking its gradient variable, $W$, and position variable
b to minimize the cost function.

**Coming Up Next**

In the next article, we will:

- Set up Tensor Board to visualize TF execution to detect problems in our model, cost function, or gradient descent
- Feed data points in batches into the model during each training step (instead of just one data point at a time) to understand how it affects training

**Resources**

- The code on Github
- The slides on SlideShare
- The video on YouTube
Gentlest Introduction to Tensorflow (Part 2)

Summary: We show in illustrations how the machine learning ‘training’ process happens in Tensorflow, and tie them back to the Tensorflow code. This paves the way for discussing ‘training’ variations, namely stochastic/mini-batch/batch, and adaptive learning rate gradient descent. The ‘training’ variation code snippets presented serve to reinforce the understanding of the role of Tensorflow placeholders.

This is part of a series:

- Part 1: Linear regression with Tensorflow for single feature single outcome model
- Part 2 (this article): Tensorflow training illustrated in diagrams/code, and exploring training variations

Quick Review

In the previous article, we used Tensorflow (TF) to build and learn a linear regression model with a single feature so that given a feature value (house size/sqm), we can predict the outcome (house price/$).

Here is the review with illustration below:

1. We have some data of house sizes & house prices (the gray round points)
2. We model the data using linear regression (the red dash line)
3. We find the ‘best’ model by training W, and b (of the linear regression model) to minimize the ‘cost’ (the sum of the length of vertical blue lines, which represent the differences between predictions and actual outcomes)
4. Given any house size, we can use the linear model to predict the house size (the dotted blue lines with arrows)
In machine learning (ML) literature, we come across the term ‘training’ very often, let us literally look at what that means in TF.

**Linear Regression Modeling**

Linear Model (in TF notation): $y = tf.matmul(x, W) + b$

The goal in linear regression is to find $W, b$, such that given any feature value $(x)$, we can find the **prediction** $(y)$ by substituting $W, x, b$ values into the model.

However to find $W, b$ that can give accurate predictions, we need to ‘train’ the model using available data (the multiple pairs of actual feature $(x)$, and actual outcome $(y_\_)$, note the *underscore*).

**Training’ Illustrated**

To find the best $W, b$ values, we can initially start with any $W, b$ values. We also need to define a cost function, which is a measure of the **difference** between the **prediction** $(y)$ for given a feature value $(x)$, and the **actual**
outcome \(_y_\) for that same feature value \(_x\). For simplicity, we use least minimum squared error (MSE) as our cost function.

Cost function (in TF notation): \(\text{tf.reduce_mean(tf.square(y_ - y))}\)

By minimizing the cost function, we can arrive at good \(W, b\) values.

Our code to do training is actually very simple and it is labelled with \([A, B, C, D]\), which we will refer to later on. The full source is on Github.

```python
# ... (snip) Variable/Constants declarations (snip) ... 

# [A] TF.Graph
y = tf.matmul(x,W) + b 
cost = tf.reduce_mean(tf.square(y_-y))

# [B] Train with fixed 'learn_rate'
learn_rate = 0.1 
train_step = 
    tf.train.GradientDescentOptimizer(learn_rate).minimize(cost)

for i in range(steps):
    # [C] Prepare datapoints
    # ... (snip) Code to prepare datapoint as xs, and ys (snip) ... 

    # [D] Feed Data at each step/epoch into 'train_step'
    feed = { x: xs, y_: ys }
    sess.run(train_step, feed_dict=feed)
```

Our linear model and cost function equations \([A]\) can be represented as TF graph as shown:
Next, we select a datapoint \((x, y)\) [C], and feed [D] it into the TF Graph to get the prediction \((y)\) as well as the cost.

To get better \(W, b\), we perform gradient descent using TF’s \texttt{tf.train.GradientDescentOptimizer} [B] to reduce the cost. In non-technical terms: given the current cost, and based on the graph of how cost varies with other variables (namely \(W, b\)), the optimizer will perform small tweaks (increments/decrements) to \(W, b\) so that our prediction becomes better for that single datapoint.
The final step in the training cycle is to update the $W, b$ after tweaking them. Note that ‘cycle’ is also referred to as ‘epoch’ in ML literature.

In the next training epoch, repeat the steps, but use a different datapoint!
Using a variety of datapoints generalizes our model, i.e., it learns $W, b$ values that can be used to predict any feature value. Note that:

In most cases, the more datapoints, the better your model can learn and generalize.

If you train more epochs than datapoints you have, you can re-use datapoints, which is not an issue. The gradient descent optimizer always use both the datapoint, **AND** the cost (calculated from the datapoint, with $W, b$ values of that epoch) to tweak $W, b$; the optimizer may have seen that datapoint before, but not with the same cost, thus it will learn something new, and tweak $W, b$ differently.

You can train the model a fixed number of epochs or until it reaches a cost threshold that is satisfactory.

### Training Variation

**Stochastic, Mini-batch, Batch**

In the training above, we feed a single datapoint at each epoch. This is known as *stochastic* gradient descent. We can feed a bunch of datapoints at each epoch, which is known as *mini-batch* gradient descent, or even feed all the datapoints at each epoch, known as *batch* gradient descent. See the graphical comparison below and note the 2 differences between the 3
The number of datapoints (upper-right of each diagram) fed to TF.Graph at each epoch

The number of datapoints for the gradient descent optimizer to consider when tweaking W, b to reduce cost (bottom-right of each diagram)

Stochastic gradient descent

Mini-batch gradient descent
The number of datapoints used at each epoch has 2 implications. With more datapoints:

- Computational resource (subtractions, squares, and additions) needed to calculate the cost and perform gradient descent increases
- Speed at which the model can learn and generalize increases

The pros and cons of doing stochastic, mini-batch, batch gradient descent can be summarized in the diagram below:
To switch between stochastic/mini-batch/batch gradient descent, we just need to prepare the datapoints into different batch sizes before feeding them into the training step [D], i.e., use the snippet below for[C]:

```python
# * all_xs: All the feature values
# * all_ys: All the outcome values
# datapoint_size: Number of points/entries in all_xs/all_ys
# batch_size: Configure this to:
#             1: stochastic mode
#             integer < datapoint_size: mini-batch mode
#             datapoint_size: batch mode
# i: Current epoch number

if datapoint_size == batch_size:
    # Batch mode so select all points starting from index 0
    batch_start_idx = 0
elif datapoint_size < batch_size:
    # Not possible
    raise ValueError("datapoint_size: %d, must be greater than
    batch_size: %d" % (datapoint_size, batch_size))
else:
    # stochastic/mini-batch mode: Select datapoints in batches
    # from all possible datapoints
    batch_start_idx = (i * batch_size) % (datapoint_size - batch_size)
    batch_end_idx = batch_start_idx + batch_size
    batch_xs = all_xs[batch_start_idx:batch_end_idx]
    batch_ys = all_ys[batch_start_idx:batch_end_idx]

# Get batched datapoints into xs, ys, which is fed into
# 'train_step'
xs = np.array(batch_xs)
ys = np.array(batch_ys)
```

Learn Rate Variation

Learn rate is how big an increment/decrement we want gradient descent to tweak W, b, once it decides whether to increment/decrement them. With a small learn rate, we will proceed slowly but surely towards minimal cost, but with a larger learn rate, we can reach the minimal cost faster, but at the risk of ‘overshooting’, and never finding it.

To overcome this, many ML practitioners use a large learn rate initially (with the assumption that initial cost is far away from minimum), and then decrease the learn rate gradually after each epoch.
TF provides 2 ways to do so as wonderfully explained in this StackOverflow thread, but here is the summary.

**Use Gradient Descent Optimizer Variants**

TF comes with various gradient descent optimizer, which supports learn rate variation, such as `tf.train.AdagradientOptimizer`, and `tf.train.AdamOptimizer`.

**Use `tf.placeholder` for Learn Rate**

As you have learned previously, if we declare a `tf.placeholder`, in this case for learn rate, and use it within the `tf.train.GradientDescentOptimizer`, we can feed a different value to it at each training epoch, much like how we feed different datapoints to `x, y_`, which are also `tf.placeholders`, at each epoch.

We need 2 slight modifications:

```python
# Modify [B] to make 'learn_rate' a 'tf.placeholder'
# and supply it to the 'learning_rate' parameter name of
# tf.train.GradientDescentOptimizer
learn_rate = tf.placeholder(tf.float32, shape=[])
train_step = tf.train.GradientDescentOptimizer(
    learning_rate=learn_rate).minimize(cost)

# Modify [D] to include feed a 'learn_rate' value,
# which is the 'initial_learn_rate' divided by
# 'i' (current epoch number)
# NOTE: Oversimplified. For example only.
feed = { x: xs, y_: ys, learn_rate: initial_learn_rate/i }
sess.run(train_step, feed_dict=feed)
```

**Wrapping Up**

We illustrated what machine learning ‘training’ is, and how to perform it using Tensorflow with just model & cost definitions, and looping through the training step, which feeds datapoints into the gradient descent optimizer. We also discussed the common variations in training, namely changing the size of datapoints the model uses for learning at each epoch, and varying the learn rate of gradient descent optimizer.
Coming Up Next

Set up Tensor Board to visualize TF execution to detect problems in our model, cost function, or gradient descent

Perform linear regression with multiple features

Resources

The code on Github

The slides on SlideShare

The video on YouTube