TENSORFLOW

Natural Language Processing & Sentiment Analysis

Justin Johnson | Z23136514 | jjohn273@fau.edu

COT6930 - Natural Language Processing | Florida Atlantic University | Spring 2018

Table of Contents

Introduction

TensorFlow is a library for data flow programming that was made open source by Google in 2015. The library provides extreme flexibility and high performance across a range of domains and is fine-tuned for large scale machine learning tasks. The goal of this document is to introduce TensorFlow and apply it to a Natural Language Processing task.

After TensorFlow's core concepts are introduced, it will be used to construct a Linear Regression model which predicts housing prices. This may seem strange, as predicting housing costs is not an NLP problem, but the Linear Regression model will allow us to get comfortable with TensorFlow's API[7] and Tensorboard (TensorFlow's visualization tool). When learning something new, it's better to walk before running.

Finally, TensorFlow will be used to perform Sentiment Analysis on an IMDb movie review data set. A Long Short-Term Memory network, a type of recurrent neural network that excels in NLP tasks, will be trained to predict movie reviews as positive or negative.

TensorFlow Overview

Inception

According to the release article published by Google[1] in November of 2015, TensorFlow is Google's 2nd generation machine learning infrastructure, a successor to DistBelief, which overcomes some of the limitations of DistBelief. Although heavily used in machine learning and specifically deep learning, TensorFlow is a general data flow programming library that can be applied to a variety of problems. Since it has become open source in November of 2015 it has quickly gained great momentum in the machine learning community. Originally created for internal use, the TensorFlow library is heavily used by Google in both research and in production.

Advantages

TensorFlow has many advantages that led to its fast growth in the community. Being both supported and used internally by Google provides great testament to the library. Several advantages are listed below:

- Runs on any platform - Windows, macOS, Linux, Android, and iOS
- Easy to use APIs for Python, C++, Java, and Go
- Extremely flexible - capable of producing any imaginable network architecture
- Out of the box GPU and distributed computing support, allowing developers to train large models on massive data sets efficiently
- Automatic differentiating - they provide a number of optimizer nodes for minimizing cost functions
- Very popular, large community, a lot of support, easy to find resources
- Comes with Tensorboard - a tool for visualizing network architectures along with any training / testing outputs (such as loss function or accuracy)
- Can be run on TPUs - Google's very own ASIC designed specifically for machine learning
    - TPUs are available for use on Google's Cloud Service: Cloud TPU[2]
    - Google claims their TPUs are 15 - 30 times faster and 30 - 80 times more energy efficient than traditional GPU or CPU clusters[3]

High Level APIs

With great flexibility and functionality comes an increased learning curve, as is true with any software package. Getting started with TensorFlow's low level API was not as easy an experience as was getting started with other machine learning libraries, but the payoff is much greater.

Several high-level layers have been designed to work with TensorFlow, simplifying the development process by providing easy to use abstractions. Keras, for example, is a high level neural network API that interfaces with TensorFlow and several other learning frameworks[4]. Another high-level API, provided by TensorFlow, is the Estimator API[5]. TensorFlow created the Estimators API to greatly simplify machine learning tasks, making the technology available to more people.

General Workflow

TensorFlow projects can be broken up into two primary stages. The first stage is the computation graph creation stage. The TensorFlow computation graph defines how data will flow through the network, from inputs, through various nodes, down to the outputs. The second stage is the graph execution stage, which is done via a TensorFlow session. It is important to always remember that these two stages are separate, that no graph nodes will contain values or perform operations on data until a session is created and run on the graph.

This section of the report does not contain any specific code examples, but examples can be found in the Linear Regression and Sentiment Analysis sections. I suggest checkout out the Linear Regression[9] and Sentiment Analysis[8] Jupyter Notebooks made available on GitHub, as they provide specific implementation details and are well commented.

TensorFlow Graph

The TensorFlow Graph defines variable, placeholder, and operation nodes along with their interconnections. Placeholders simply output the values they are fed, ideal for input nodes. All the placeholder nodes will do is feed the input data into the data flow when the session is run. Variables are subject to change throughout session execution. A variable node is a good candidate for a model's weights, as weights need to be adjusted during the training process. Finally, operations perform transformations on the data flowing through them.

Data flows through the nodes of the computation graph in the form of tensors. Put simply, a tensor is a n-dimensional array or vector object with various built in methods. The tensors flow through the computation graph to produce an output, hence the name TensorFlow.

In both examples to follow, Tensorboard will be used to visualize the computation graphs that have been defined. Being able to visualize computation graphs can help improve network architecture design and debugging.

TensorFlow Session

A TensorFlow Session creates a run time environment to execute TensorFlow operations, like the operations defined in the computation graph. The Session will map the computation to the devices available hardware resources, taking advantage of GPU or distributed clusters if available to the session. Once a tf.Session object is instantiated, it's run method is used to execute TensorFlow operations. The actual execution of the data flow and operations is carried out by the Session using C++. When the Session's run method is called, it will first determine the operation's dependencies, and it will evaluate those dependencies in the required order.

The tf.Session class's run method takes an optional parameter, a dictionary of feeds (feed_dict). The feed dictionary is used to pass data to the graph's placeholder nodes. Recall that placeholder nodes simply output the data that is fed to them during execution. In the examples to follow, placeholders are used to feed input features and labels to the network. When the graph is defined, these placeholder nodes do not store any values or data. It is not until they are evaluated by the session that data will flow through them.

TensorBoard

TensorBoard is TensorFlow's visualization toolkit which can be used to visualize computation graph's or any metrics that are generated during a session's execution. In the following examples TensorBoard will be used to visualize the computation graphs, training loss, training accuracy, and test set evaluation accuracy.

TensorBoard visualizations are created with the tf.summary.FileWriter class. The class's constructor is provided two input parameters: a path to a directory where it will write content to, and a pointer to the session's graph. TensorFlow will create the directory and write the graph details to a file. When executing a session, metrics can also be written to the TensorBoard log directory for visualization. This is done using the FileWriter's add_summary method.

Recap

TensorFlow's low level Python API will be used to construct computation graphs for both prediction with a Linear Regression model and classification with a Recurrent Neural Network model. TensorFlow sessions will be created, feeding the input features and labels to the network. The FileWriter class will be used to create Tensorboard visualizations.

Getting Started with Linear Regression

To become more familiar with TensorFlow's API, we will first construct a Linear Regression model for the purpose of predicting house prices. A prepared data set from Scikit-Learn's datasets module, load_boston, will be used. The data set contains 506 instances with 14 features per instance. The 14th feature, median value, is the target feature that we will be training our model to predict. Below is a complete description of the data's features:

```
:Attribute Information (in order):
    - CRIM     per capita crime rate by town
    - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
    - INDUS    proportion of non-retail business acres per town
    - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
    - NOX      nitric oxides concentration (parts per 10 million)
    - RM       average number of rooms per dwelling
    - AGE      proportion of owner-occupied units built prior to 1940
    - DIS      weighted distances to five Boston employment centres
    - RAD      index of accessibility to radial highways
    - TAX      full-value property-tax rate per $10,000
    - PTRATIO  pupil-teacher ratio by town
    - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
    - LSTAT    % lower status of the population
    - MEDV     Median value of owner-occupied homes in $1000's
```

Preparing Data

To keep things simple and stay focused on TensorFlow, no feature selection will be applied. We will just normalize the data, add a bias column with value of 1, and split our data into train and test sets (80/20).

Training a Linear Regression Model with TensorFlow

In [14]:
```
# start with fresh copy of our data
data = bostonData
labels = bostonTarget
```

In [15]:
```
# helper function for normalizing data
def normalize(data):
    mu = np.mean(data,axis=0)
    sigma = np.std(data,axis=0)
    return (data - mu)/sigma
```

Data Pre-Processing

In [16]:
```
# store shape of data set
rows, cols = np.shape(data)

# normalize data
data = normalize(data)

# add bias to data
bias = np.ones((rows, 1), dtype=np.float32)
data = np.column_stack([bias, data])

# re-calc shape now that we've added new column
rows, cols = np.shape(data)

# reshape labels
labels = np.reshape(labels, (rows, 1))

# split into test and train sets
from sklearn.model_selection import train_test_split
trainData, testData, trainLabels, testLabels = train_test_split(
    data, labels, test_size=0.2, random_state=42)
```

Define Learning Parameters

For this example, we will just define two training parameters. We will set the learning rate to 0.01 and the number of training iterations to be 1000. These can be tuned to realize model performance.

Define Learning Parameters

In [17]:
```
# Defining Learning Params

# larger learning_rate converges quicker but may get stuck in local minimums
learning_rate = 0.01

# number of training iterations
epochs = 1000

# array to store cost history for visualizing
cost_history = np.empty(shape=[1], dtype=float)
```

Define TensorFlow Graph

Our first two graph nodes should be obvious, placeholders for X and Y which will be fed our input features and labels. Next, we define a variable W for weights. This is defined as a variable because it will need to change throughout the training process. The weights are adjusted after each iteration in order to minimize error. We define the prediction to be the matrix multiplication of the input features and the weights vector. The cost function is defined as the mean squared error. Note the tf.summary.scalar which is passed the value for cost. This scalar value will be used during our session to write the cost metric to the TensorBoard directory, allowing us to visualize the change in cost over time. Finally, the training operation is defined using TensorFlow's build in GradientDescentOptimizer.

In [18]:
```
# Define TensorFlow graph to carry out linear regression

import tensorflow as tf


# Placeholders for feature and label inputs
with tf.name_scope('Input'):
    X = tf.placeholder(tf.float32,[None, cols], name='Features')
    Y = tf.placeholder(tf.float32,[None, 1], name='Labels')


# Variable Weights will be adjusted during training to minimize loss function
# summary operation used to plot weights distribution
with tf.name_scope('Weights'):
    W = tf.Variable(tf.ones([cols,1]), name='W')
    tf.summary.histogram('weights', W)


# Prediction = W * X
with tf.name_scope('Predictions'):
    y_ = tf.matmul(X, W)


# Cost Function = mean squared error
# summary operation used to record cost during training
with tf.name_scope('Cost'):
    cost = tf.reduce_mean(tf.square(y_ - Y))
    tf.summary.scalar('cost', cost)


# Training Step
# We use TensorFlow's Gradient Descent optimizer to minimize cost!
with tf.name_scope('Train'):
    training_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Name Scopes

Note how the previous graph construction uses tf.name_scope to combine like nodes and operations. This is not required ,but it will help to clean up the graph for visualization purposes. TensorBoard will group node's in the graph by name scope, making the graph much easier to interpret.

Creating TensorFlow Session

We will now create a TensorFlow session which will carry out the computations of the previously defined graph. Once the session is created, it's run method is called on the variable initializer, which will initialize all variables defined in above graph construction. Then a FileWriter is created in order to take advantage of TensorBoard's visualizations.

Create TensorFlow Session

In [19]:
```
# initialize session and global graph variables
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)


# create FileWriter object to record results to TensorBoard
writer = tf.summary.FileWriter("tf_logs/run2")
merged_summary = tf.summary.merge_all()
writer.add_graph(sess.graph)
```

Training the Model with TensorFlow Session

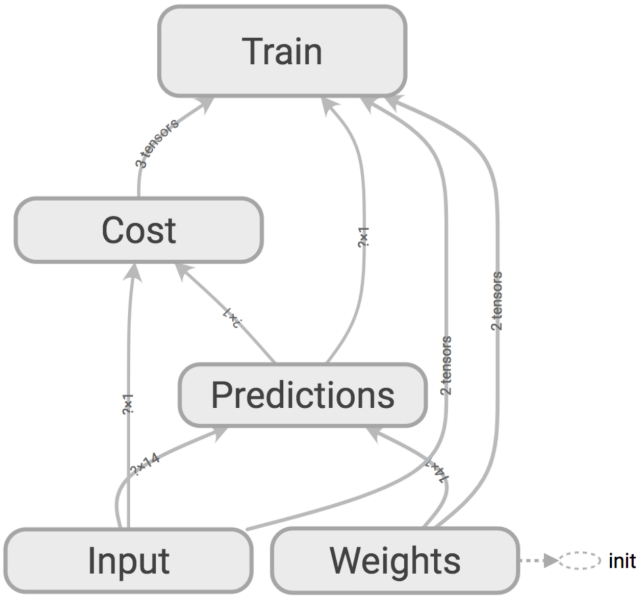
Our last step is to use the session object to run the training step. The 2nd parameter to the session's run method is the feed_dict, this is where the input features and labels are passed to the placeholder nodes and made available to the network. The training step's GradientDescentOptimizer will use the error to update the weights, and this is repeated for 1000 iterations. The FileWriter instance will also write the loss to the TensorBoard log directory, enabling us to visualize loss over time.

Execute Session and Train Model

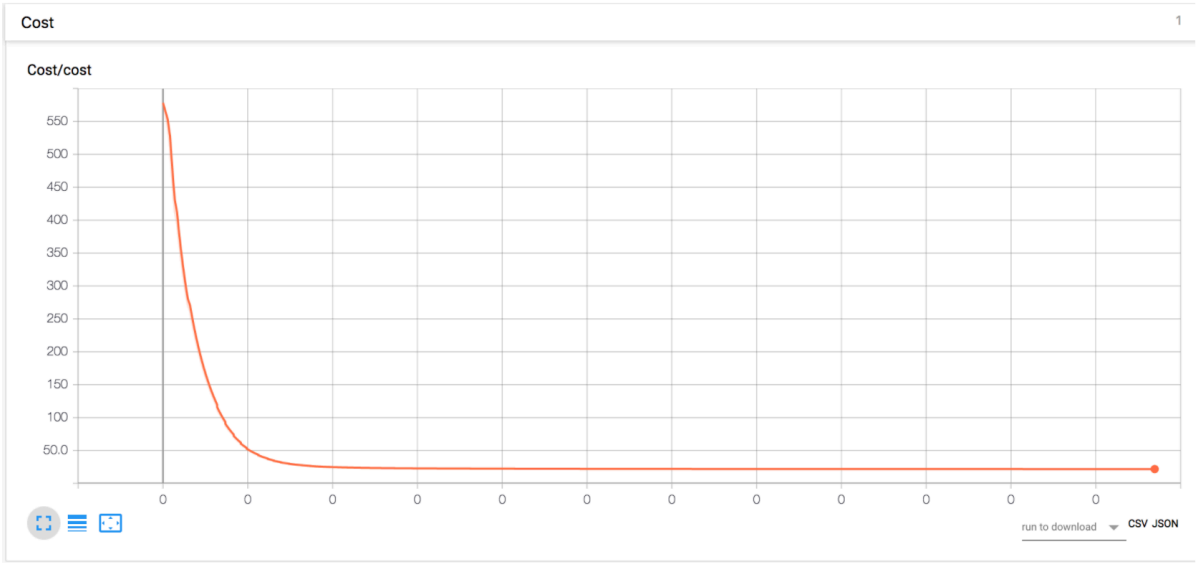
In [20]:
```
for epoch in range(epochs):

    # run training step
    sess.run(training_step, feed_dict={X: trainData, Y: trainLabels})

    # write summary to TensorBoard
    summary = sess.run(merged_summary, feed_dict={X: trainData, Y: trainLabels})
    writer.add_summary(summary)

    # record cost for local plot
    cost_history = np.append(cost_history, sess.run(cost, feed_dict={X: trainData, Y: trainLabels
}))
```

Visualize with Tensorboard

Below images display the computation graph that was created followed by the change in error over time, calculated after each iteration.



This graph would appear significantly more complex if name scopes were not used during the graph construction phase. Each node can be expanded, allowing users to view the individual nodes and operations that make up the node. This can be helpful when debugging.



As expected, the cost (MSE) reduces with each iteration as the GradientDescentOptimizer adjusts the weights to reduce error.

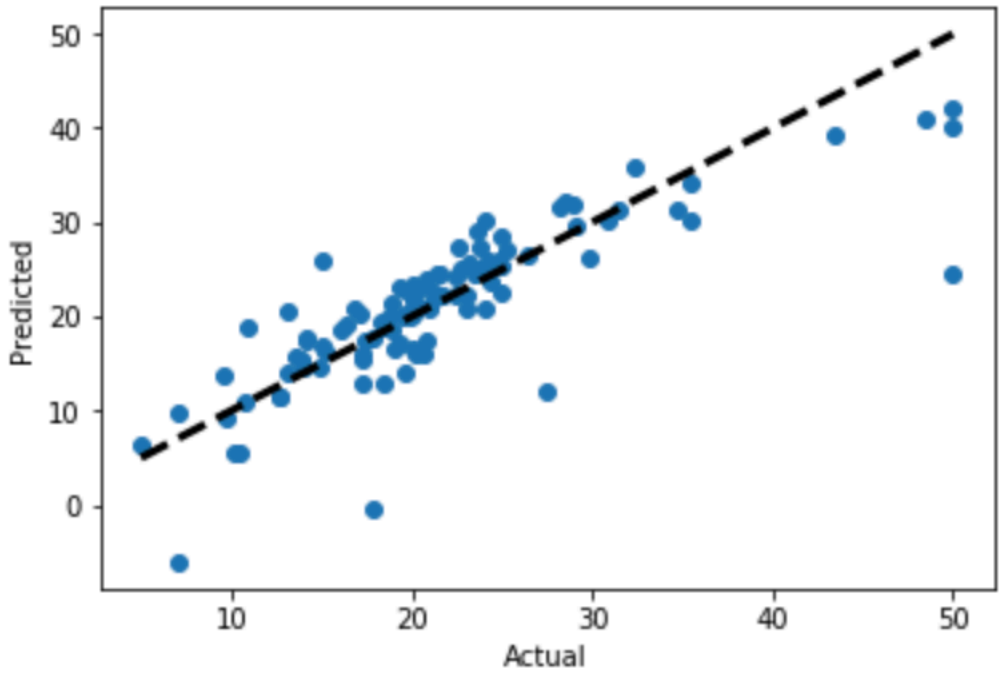
Evaluating the Model

Finally, we will evaluate the trained Linear Regression model with the test data set. The TensorFlow session is used to run the prediction operation, _y, on the test data set. The mean squared error is then calculated using the prediction and the test set labels.

Evaluating Model With Test Set

In [22]:
```
# calculate predictions and mean squared error
pred_y = sess.run(y_, feed_dict={X: testData})
mse = tf.reduce_mean(tf.square(pred_y - testLabels))
print("MSE: %.4f" % sess.run(mse))
```

```
MSE: 24.7463
```



The above diagram compares the model's predicted values to the actual value. The Linear Regression model has successfully been trained to produce accurate predictions. The MSE on the test set is 24.75.

Linear Regression Conclusion

To summarize, we have used TensorFlow's Python API to construct a computation graph that implements linear regression. A TensorFlow session was created and run against our training step for 1000 iterations, adjusting the input weights at each iteration to minimize MSE. TensorBoard was used to visualize the computation graph and the MSE over time. Finally, the model was evaluated using a test data set. This concludes the introduction to TensorFlow for Linear Regression.

Sentiment Analysis with IMDb Movie Reviews

Now that we have a basic understanding of working with TensorFlow, we will apply it to a Natural Language Processing task, Sentiment Analysis. For this example, an IMDb movie review data set[10] containing 25K movie reviews will be used to train a LSTM network. The LSTM network will be trained to classify movie reviews as positive or negative.

The foundation of this portion of the project originated from a tutorial written by Adit Desphande[12]. In his tutorial, he provides an excellent overview of word vectors, RNNs, and LSTMs.

LSTM Network Introduction

Before implementation, the LSTM network is introduced at a high level. For more information on Recurrent Neural Networks or LSTM networks, I recommend reading "Understanding LSTM Networks"[11].

A LSTM network is a type of recurrent neural network. Recurrent neural networks contain feedback loops which allow the output of a neuron to be a function of both the current input and previous inputs. For problems which involve sequences, like text or speech, recurrent neural networks outperform traditional feed forward networks. There has been a lot of success in the area of NLP with recurrent neural networks, because language is dependent on previous input. Traditional bag of words models are unable to take into consideration the position of text in a sequence. When RNNs are tuned to handle longer and longer distance relationships, they experience a "vanishing gradient" problem, which stops the model from making improvements as the gradients approach zero. The LSTM networks are able to retain longer distance dependencies than traditional RNNs by their implementation, they do not suffer from the same vanishing gradient problem. LSTM units contain gates which allow the unit to remember or forget input as it passes through. LSTM networks have many applications in NLP.

Data Set

The IMDb data set[10] contains 25,000 movie reviews separated into two directories, positive and negative. The data is completely balanced, with 12,500 reviews of each type. Each movie review is stored in its own text file within its corresponding positive/negative folder.

Several steps must be taken before the movie reviews can be fed to the neural network. Punctuation and special characters are removed from the documents, and all text is case folded to lowercase. There is still one problem, the neural network requires an input of scalar values, we cannot feed text directly into the network. In the next section, we will solve this problem with word vectors.

Word Vectors

Word Vectors are a popular method for text representation. By representing each term as a vector, it is possible to store additional information about the word. Word vectors are able to capture meaning, context, part of speech, semantics, and more. For example, the words 'cat' and 'dog' will have vector representations with high similarity as both are nouns, animals, and pets. Since the words are vectors of scalar values, it is also possible to perform operations on them. Adding a vector for the word 'king' to another vector for the word 'woman' should produce a new vector that has high similarity with the vector for 'queen'. One can easily see the power in Word Vectors and their application to natural language processing.

Creating word vectors is another task in itself, and TensorFlow provides a tutorial on training word vectors using Word2Vec[13]. For this project, we use an existing word vector lexicon provided by GloVe (Global Vectors for Word Representation)[14]. The lexicon provided by GloVe contains 400K words, and each word is represented by a vector of length 50.

Next, we will need to map each movie review to a list of word vectors.

Mapping Movie Reviews to Word Vectors

First, we will construct a word indices matrix for every movie review. This means that we will be replacing each word of the movie review with that word's index, as it is listed in the Word Vector lexicon. If the GloVe lexicon has the word 'movie' stored at index 145, then the word 'movie' will be replaced in all of the movie reviews with the number 145. This process is straight forward. A 2-dimensional array is created such that each row will represent a movie review and the columns will represent the sequence of words in that review. Each movie review is opened and read, the movie review word indices are looked up in the GloVe lexicon, and the indices are stored in the 2-dimensional output array. Once complete, the 2-dimensional array of movie review word indices is saved so that we do not need to re-compute these values.

With the movie reviews converted to word indices, we can now use TensorFlow's tf.nn.embedding_lookup to look up the word vectors during run time. Storing just the word indices and then using TensorFlow's embedding lookup at run time is a more efficient than converting all 25K reviews to word vectors and storing them in memory.

Now that we have our movie reviews in the proper format, we can begin defining our LSTM network and constructing our computation graph.

LSTM Network Parameters

The screenshot below displays one parameter selection that was used in this project. Batch size, LSTM units, and epochs were varied over several runs. Their results were compared in an attempt to improve accuracy. Unfortunately, we were unable to perform better than 85% accuracy on the test set, regardless of the parameter selection. I will comment on this further in the conclusions section, but this is primarily due to the size of our data set. In general, neural networks perform better when they have seen more data.

```
In [73]: # Define RNN and trianing params
         batchSize = 50
         lstmUnits = 64
         numClasses = 2
         epochs = 50000
```

There is no learning rate listed here, because the TensorFlow optimizer's default learning rate of 0.001 was used for most training sessions. Increasing the learning rate can speed up time to convergence, but if the learning rate is too high the model will likely bounce around and diverge. The only downfall to using a smaller learning rate is that more iterations will be required for the model to converge to a minimum. Dynamic or decaying learning rates are another strategy that can help to improve optimization.

Increasing LSTM units will allow the model to learn more complex problems at the cost of increase in training time. Both 64 and 128 LSTM units were tested in this project with no improvement to performance.

Total number of epochs was selected through inspection of previous results. It became clear that with a learning rate of 0.001, approximately 20,000 iterations were required to reach an optimal model. To be safe and to examine how the model trained with additional iterations, epochs was fixed at 50,000.

Constructing the Graph

We define the networks inputs and variables as follows:

In [70]:
```
tf.reset_default_graph()


# placeholder for input features
with tf.name_scope('Features'):
    input_data = tf.placeholder(tf.int32, [batchSize, maxSeqLength], name="Features")


# placeholder for input labels
with tf.name_scope("Labels"):
    labels = tf.placeholder(tf.float32, [batchSize, numClasses], name="Labels")


# placeholder for word vectors previously loaded from GloVe
with tf.name_scope("WordVectors"):
    word_vectors = tf.placeholder(tf.float32, [400000, 50], name="WordVectors")


# operation - lookup word vector embedding for input features
with tf.name_scope("EmbeddingLookup"):
    data = tf.Variable(tf.zeros([batchSize, maxSeqLength, wordVecLength]),dtype=tf.float32, name=
"Embedding")
    data = tf.nn.embedding_lookup(word_vectors, input_data)


# variable - weights initialized as random values from a truncated normal distribution
with tf.name_scope("Weights"):
    weight = tf.Variable(tf.truncated_normal([lstmUnits, numClasses]), name="Weights")


# variable - Bias initialized to 0.1
with tf.name_scope("Bias"):
    bias = tf.Variable(tf.constant(0.1, shape=[numClasses]), name="Bias")
```

Most of this should look familiar from the linear regression problem. Features, labels, and word vectors are defined as placeholders which will be fed to the network at run time. The ‘EmbeddingLookup’ will take the input movie reviews and the GloVe word vectors as inputs, and it will map each movie review from word indices to word vectors. The weights and bias nodes are defined as variables as they will be adjusted during training.

One of the biggest sources of error that I experienced in this process is in defining nodes of the correct shape. Take note of each tensors shape and be sure that nodes which handle these tensors have matchings shapes.

The next image displays the construction of the LSTM layer:

```
# operations related to prediction
# grabs output from RNN, multiplies by weights and adds bias
with tf.name_scope("RNN"):
    lstmCell = tf.contrib.rnn.BasicLSTMCell(lstmUnits)
    lstmCell = tf.contrib.rnn.DropoutWrapper(cell=lstmCell, output_keep_prob=0.75)
    value, _ = tf.nn.dynamic_rnn(lstmCell, data, dtype=tf.float32)
    value = tf.transpose(value, [1, 0, 2])
    last = tf.gather(value, int(value.get_shape()[0]) - 1)
    prediction = (tf.matmul(last, weight) + bias)
    softmax = tf.nn.softmax_cross_entropy_with_logits(logits=prediction, labels=labels)


# operation - accuracy calculation
with tf.name_scope("Accuracy"):
    correctPred = tf.equal(tf.argmax(prediction,1), tf.argmax(labels,1))
    accuracy = tf.reduce_mean(tf.cast(correctPred, tf.float32))
    tf.summary.scalar('Accuracy', accuracy)
    test_acc_summary = tf.summary.scalar('Accuracy', accuracy)


# operation - loss calculation
with tf.name_scope("Loss"):
    loss = tf.reduce_mean(softmax)
    tf.summary.scalar('Loss', loss)


# define optimization strategy to be executed during training
with tf.name_scope("Optimizer"):
    optimizer = tf.train.AdamOptimizer().minimize(loss)
```

The LSTM layer is composed of LSTM units, provided to us by TensorFlow's tf.contrib.rnn module. We first compose a layer of 64 LSTM units and then wrap this in TensorFlow's RNN DropoutWrapper. Dropout is a regularization technique that helps to prevent overfitting. The output of the RNN is fed through a softmax layer to compute the predicted probability of each class.

tf.Summary operators are used to output results to TensorBoard using the FileWriter. The Accuracy and Loss nodes include tf.Summary.scalar operations, allowing us to visualize these metrics over time.

The training step in this example uses TensorFlow's AdamOptimizer. This is a popular optimizer that offers advantages over the more traditional Gradient Descent Optimizer. The Gradient Descent Optimizer was used for one training session in order to compare results, and the AdamOptimizer performed significantly better.

Creating a TensorFlow Session

A TensorFlow session is created in the same manner as was created in the Linear Regression example. The only difference here is that we are creating two FileWriter objects so that we can write training metrics and test metrics to two separate directories. Doing this will allow TensorBoard to plot training accuracy and test accuracy on the same graph, so that accuracy can be compared, and overfitting can be detected.

In [71]:
```
%%time


# Init new session and saver to store checkpoints
init = tf.global_variables_initializer()
sess = tf.Session()
saver = tf.train.Saver()
sess.run(init)


# create summary writers to store graph and chart metrics
merged = tf.summary.merge_all()
trainLogDir = "tensorboard/graph_e50K_b50_lstm64_lr001_train"
writer = tf.summary.FileWriter(trainLogDir, sess.graph)


# we create separate summary writer for test data evaluations
# this is workaround to let us view train vs test accuracy on same plot
testLogDir = "tmpboard/graph_e50K_b50_lstm64_001)test"
testWriter = tf.summary.FileWriter(testLogDir)
```

Training the Model

Once again, training our model is very similar to the Linear Regression example. We iterate over the total number of epochs, and for each epoch we grab the next batch of training data and feed it to the network. The session's run method is called on the optimizer node, passing a feed dictionary that includes the word vectors, features, and labels. TensorFlow then carries out the necessary computations as they are defined in the computation graph, and once the batch is complete the error is calculated, and the weights and biases are adjusted. This is repeated for all 50,000 epochs.

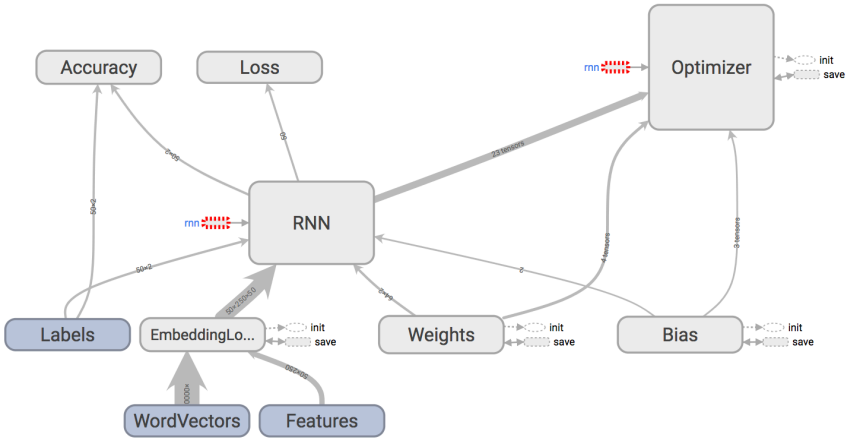
There are a few extra lines of code in this example that handle saving the model after every tenth percentile is completed. TensorFlow's Saver class, exposed through the tf.train module, allows us to save a model and restore it at a later time. Saving the model every once in a while is encouraged when performing long tasks, as it allows us to easily recover from a crash. We should also save the model when training is complete, so that we can use it for making predictions in the future.

```
for i in range(epochs):
   # Train Batch of reviews
   nextBatch, nextBatchLabels = getTrainBatch();
   sess.run(optimizer, {word_vectors: wordVectors, input_data: nextBatch, labels: nextBatchLabels
})


   # Write summary to Tensorboard
   if (i % 50 == 0):
         summary = sess.run(merged, {word_vectors: wordVectors, input_data: nextBatch, labels: next
BatchLabels})
         writer.add_summary(summary, i)

         testBatch, testLabels = getTestBatch()
         test_summary_str = sess.run(test_acc_summary,{word_vectors: wordVectors, input_data: testB
atch, labels: testLabels})

         testWriter.add_summary(test_summary_str, i)


   # Save the network every 10% completed
   if (i % tenPercent == 0 and i != 0):
       save_path = saver.save(sess, "models/pretrained_lstm.ckpt", global_step=i)
       print("saved to %s" % save_path)
```

Note that accuracy metrics are only written to the TensorBoard directory every 50 iterations. It is not necessary to write metrics after each iteration and doing so would slow down the training process and increase the total content written to disk.

Visualizing the Computation Graph

To visualize the computation graph, TensorBoad must be activated from the terminal. From the project's root directory, enter "tensorboard --logdir <dirname>". This will start up a local server that can be accessed at localhost:6006. Opening the browser to this URL will display our TensorBoard interface.



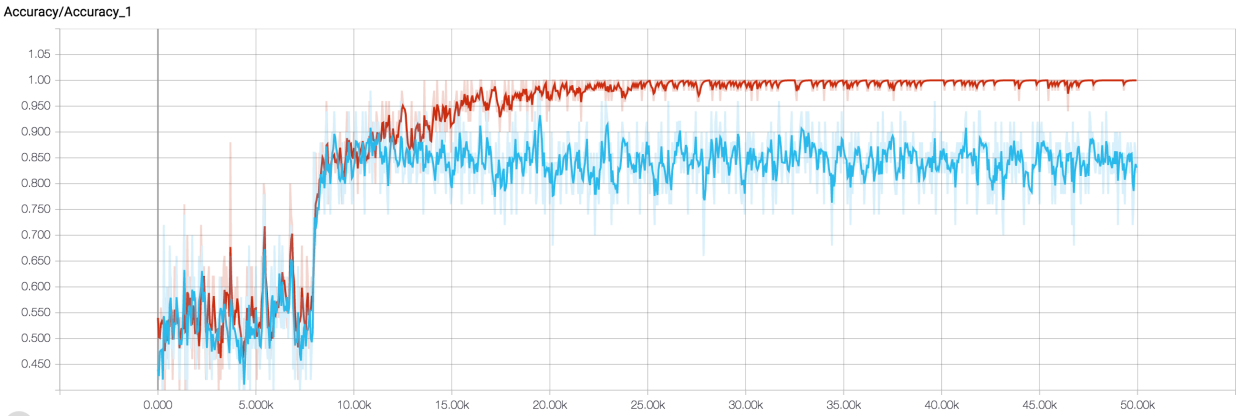
We can see in the above diagram the various nodes that were defined in the graph construction phase. Drilling down deeper into the RNN and Optimizer nodes reveals what's going on under the hood, At the lowest level a network of basic tensor operations such as matrix multiplication will be found.

It is clear by the visual that the word vectors and features are fed to the EmbeddingLookup node, and that the EmbeddingLookup node then feeds the movie review embeddings into the RNN network. The RNN network also takes the labels, weights, and biases as inputs. The RNN makes a prediction and then uses the optimizer to update the weights and biases according to the calculated prediction error.

Visualizing Training and Test Accuracy

From the TensorBoard console, we can select individual runs and then visualize the accuracy metrics recorded for the given run. This allows for easy comparison between runs and their model performances.

Our first image displays the results of using the AdamOptimizer with default learning rate of 0.001, 64 LSTM units, and 50,000 epochs.

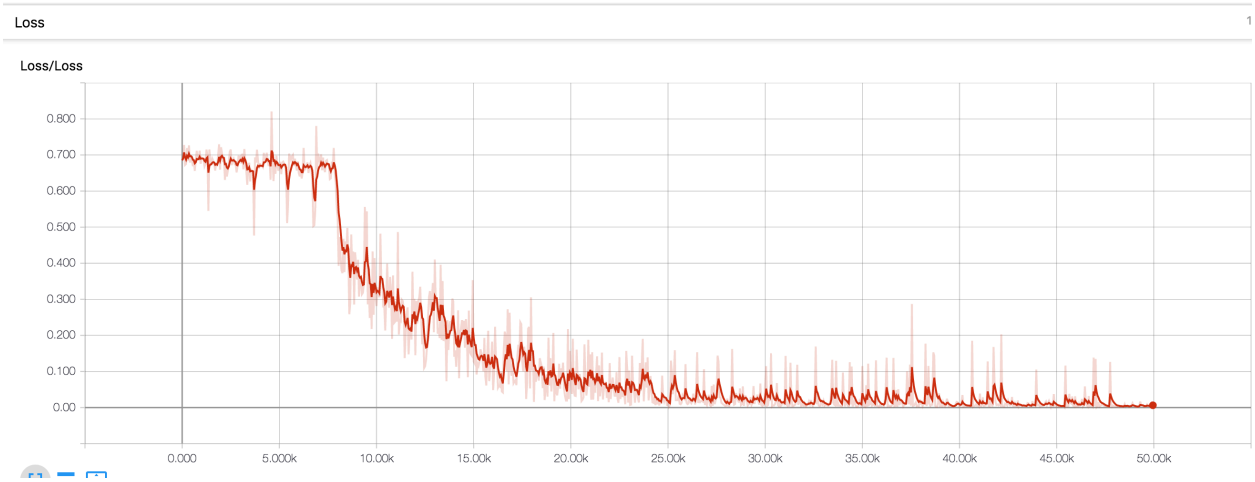


The red trace is the training accuracy - note how it reaches approximately 100% right around 20,000 iterations. The blue trace is the test set accuracy, and it settles right around 85%. The results for the majority of our models were very similar - with the model getting near perfect predictions on the training set and roughly 85% accuracy on the test set. In the conclusions section, we will outline other run's parameters and their corresponding results for comparison.

The reason we have plotted the test accuracy against the train accuracy is to check for overfitting. I was concerned to see the training accuracy hit 100%, I imagined that the model must be overfitting the training set. After plotting the test set accuracy, we can conclude that the model is not overfitting, because if it was we would see degradation in the test set accuracy at the time of overfitting.

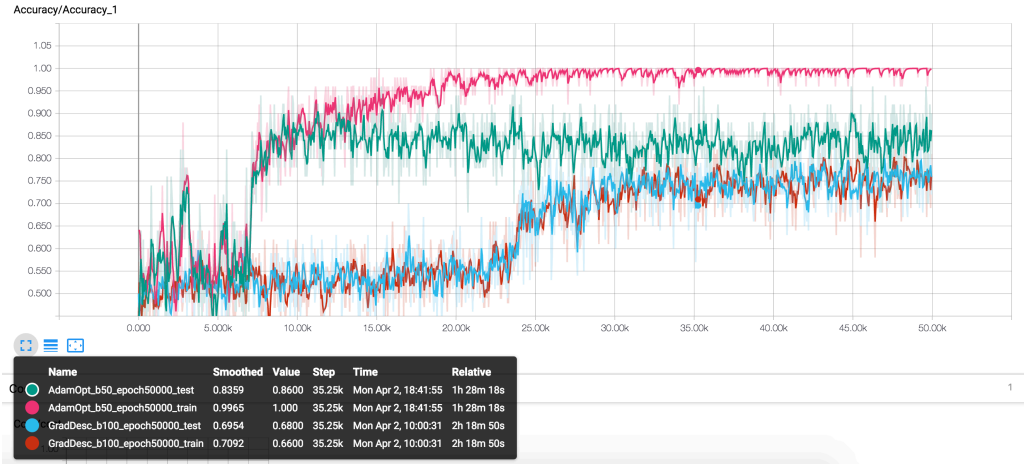
Visualizing Training Loss

In the same manner as accuracy, we've also recorded the loss over time during training. As expected, the loss decreases and approaches 0.



Comparing Models

TensorFlow's GradientDescentOptimizer was used to compare results with the AdamOptimizer:

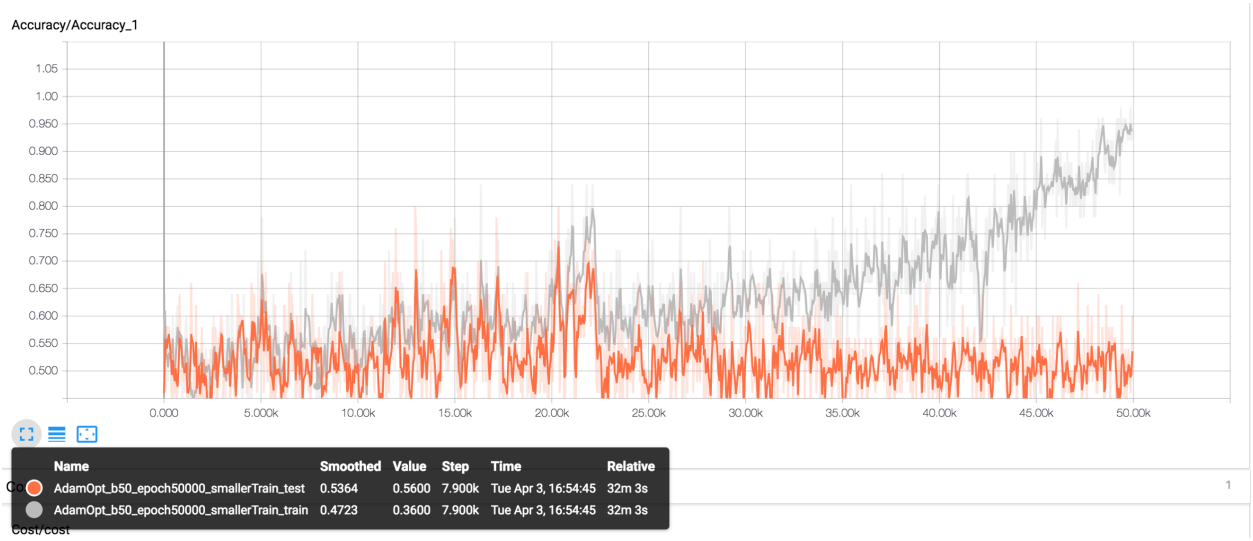


The pink and green traces are the train and test accuracies, respectively, generated with the AdamOptimizer using default learning rate of 0.001, batch sizes of 50, and 50,000 epochs. The blue and brown traces are the respective train and test accuracies as recorded with the Gradient Descent optimizer. The AdamOptimizer was able to score near 100% accuracy on the training set and roughly 85% accuracy on the test set, while the Gradient Descent optimizer was not able to score above 75% on the training or test data.

Next, the default learning rate was changed from 0.001 to 0.01:

With the increased learning rate, the model was able to reach approximately 95% accuracy on the training set much faster, in under 10,000 iterations. Unfortunately, the model eventually diverged. I predict that some form of dynamic learning rate which decays over time would enable the model to converge more quickly and prevent it from diverging by reducing the learning rate over time.

Finally, the training set size was reduced to see how a smaller training set would impact the network's performance:



The grey trace, the training accuracy, is much slower to reach 95%, and it never reaches 100% in the 50,000 iterations as the previous examples did. In addition, the orange trace denoting the test set accuracy is unable to exceed beyond 60% accuracy.

This suggests that the more data given to the recurrent neural network, the better the results.

Conclusions

TensorFlow is a powerful data flow programming library that is well equipped for complex machine learning tasks. Programming in TensorFlow has two primary stages. The first stage consists of constructing a computation graph using TensorFlow placeholders, variables, operations. The second stage uses a TensorFlow session to map the computation graph to hardware resources and carry out the graph's operations. TensorFlow's visualization toolkit, TensorBoard, enables users to visualize computation graphs and plot training or test metrics over time.

Two examples using TensorFlow were presented, predicting housing prices with a Linear Regression model, and predicting movie review sentiment using a LSTM network. For each example, code snippets were illustrated to present the application of TensorFlow to solve machine learning problems. To view the complete working code for these examples, please refer to the Jupyter Notebooks available on GitHub: Sentiment Analysis with TensorFlow and LSTM Network[8] and Linear Regression with TensorFlow[9].

References

1. Google Research Blog, https://research.googleblog.com/2015/11/tensorflow-googles-latest-machine_9.html

2. Google Cloud TPU, https://cloud.google.com/tpu/

3. TPU is 15x to 30x Faster Than GPUs and CPUs According To Google, https://www.zdnet.com/article/tpu-is-15x-to-30x-faster-than-gpus-and-cpus-google-says/

4. Keras - The Python Deep Learning Library, https://keras.io/

5. TensorFlow Estimators, https://www.tensorflow.org/programmers_guide/estimators

6. TensorFlow Python API, https://www.tensorflow.org/api_docs/

7. TensorFlow Graphs and Sessions, https://www.tensorflow.org/programmers_guide/graphs

8. Sentiment Analysis with TensorFlow and LSTM Network, https://github.com/johnsonj561/Introduction-To-TensorFlow/blob/master/TensorFlow-Sentiment-Analysis-IMDb-Final.ipynb

9. Linear Regression with TensorFlow, https://github.com/johnsonj561/Introduction-To-TensorFlow/blob/master/Linear%20Regression%20TensorFlow.ipynb

10. IMDb Movie Review Data Set, https://keras.io/datasets/#imdb-movie-reviews-sentiment-classification

11. Understanding LSTM Networks, http://colah.github.io/posts/2015-08-Understanding-LSTMs/

12. Performing Sentiment Analysis with LSTMs, Using TensorFlow, https://www.oreilly.com/learning/perform-sentiment-analysis-with-lstms-using-tensorflow

13. Vector Representations for Words, https://www.tensorflow.org/tutorials/word2vec

14. GloVe: Global Vectors for Word Representation, https://nlp.stanford.edu/projects/glove/