Getting Started with Deep Learning and TensorFlow

By Jeff Heaton

Deep learning is a rapidly evolving machine learning technology. The world's largest technology companies are investing heavily in deep learning. They are sharing this investment with the world by open sourcing their deep learning technologies. Currently the tech titans of the world have open-sourced the following deep learning frameworks:

- Amazon – Deep Scalable Sparse Tensor Network Engine (DSSTNE)[1]
- Baidu – PArallel Distributed Deep Learning (PADDLE)[2]
- Google – TensorFlow[3]
- Microsoft – Computational Network Toolkit (CNTK)[4]

All of these frameworks have their complete source code available on GitHub, which is a web platform that allows everyone from individual programmers to Fortune 500 companies to share source code and collaborate. As of the late 2016 writing of this article, DSSTNE and PADDLE both only work with the Linux operating system. TensorFlow works both with Macintosh and Linux. Not too surprisingly, Microsoft's CNTK is the only one of the four to support Microsoft Windows. The platforms supported by these frameworks will increase in the future. Work is already underway for Windows support in TensorFlow.

GOOGLE'S TENSORFLOW

Since its recent introduction in 2015, TensorFlow has taken the world of deep learning by storm. Though typically associated with deep learning, TensorFlow is actually a mathematics package specifically designed to leverage machine learning across CPU, GPU, and grid computing. Many machine learning models can be adapted to TensorFlow. It works best with neural network-like models, such as deep belief neural networks, generalized linear regression (GLM), support vector machines (SVM), and Long Short Term Memory (LSTM). While it might be possible to adapt TensorFlow to tree-based models, such as Random Forests or Gradient Boosting Machines (GBMs), these are not a focus for current versions of TensorFlow.

Python is the most widely supported language for TensorFlow. TensorFlow itself is implemented in C++, so it is also possible to directly access TensorFlow from a less widely known C++ based application programming interface (API). Typically, models are defined as TensorFlow compute graphs that define the order that calculations must occur in order to calculate a model. For example, a GLM would be defined as dot products feeding into a link function. The graph ensures that the link function is not calculated until the precursor dot products have been calculated. Similarly, neural networks are layers of dot product calculations and activation functions. The graph defines the exact order of these calculations. Python code is used to define this compute graph. However, Python would be too slow to calculate and fit these models in any acceptable timeframe. Rather, TensorFlow transforms these compute graphs into highly efficient C++ and Graphical Processing Unit (GPU) code. Deep learning is very well adapted to GPUs, and TensorFlow contains extensive support for GPUs.

INSTALLING TENSORFLOW

Because performance is paramount in deep learning, every reasonable optimization has been employed in its design. Many of these optimizations are platform specific. Currently, TensorFlow officially supports the Macintosh OSX and Linux operating systems. Windows is not currently supported. Google suggests using a virtual machine or Docker (a software containerization platform) if you must make use of the Windows operating system. At some point, TensorFlow might support windows natively. However, that time has not yet arrived. Google provides installation instructions for TensorFlow for Mac, Linux, and Windows (using an emulator).[5]

It is also possible to use TensorFlow entirely from the cloud in a web browser. This frees you from the complexities of installing binary Python packages. Jupyter notebooks provide a convenient web-hosted environment to program Python. IBM provides a free Jupyter notebook that can be used directly from the web. The Data Scientist Workbench[6] is a free and open Jupyter notebook that can be used to run the examples provided in this article.

USING TENSORFLOW

To make use of TensorFlow you must import it into Python. The following two lines of code import TensorFlow and report what version of it you are using.

```
import tensorflow as tf

print("Tensor Flow Version: {}".
format(tf.__version__))
```

The above code should report that you are using TensorFlow 0.8 or higher. The examples provided with this article were all created with 0.8 of TensorFlow. These examples are stored at GitHub and will likely be updated for future versions of TensorFlow.[7] The reference link refers to a deep learning class at Washington University in St. Louis that is taught by the author of this article.

ENCODING CATEGORICAL DATA FOR A DEEP NEURAL NETWORK

Neural networks function similarly to traditional classification and regression models. An input vector (x) of predictors is provided to the neural network and a result (y) is returned from the network. The data provided to the neural network's input vector must be encoded to numeric form. If a categorical value is a predictor, meaning it is part of the information given to the neural network to make a prediction, then it should be encoded into dummy variables. The following Python function can be used to encode a dummy variable:

```
def encode_text_dummy(df,name):
    dummies = pd.get_dummies(df[name])
    for x in dummies.columns:
        dummy_name = "{}-{}".format(name,x)
        df[dummy_name] = dummies[x]
    df.drop(name, axis=1, inplace=True)
```

For example, to encode a dummy variable named "state," in the dataframe "df," use the following call:

```
encode_text_dummy( df, "state")
```

This will remove the column "state" from your dataframe and replace it with 50 dummy variables that represent each of the 50 U.S. states (assuming all 50 were present in your dataset). Dummy variables replace a categorical variable with a number of 1/0 (true/false) columns that represent the categorical value. For each row, there would be 50 such fields, all of which would be zero, except for the state that the row corresponds to.

If we were predicting the state, and it were on the y side of the model, then we must encode this categorical value to an index. The following code accomplishes this:

```
def encode_text_index(df,name):
    le = preprocessing.LabelEncoder()
    df[name] = le.fit_transform(df[name])
    return le.classes_
```

The following code would encode the state to an index:

```
encode_text_index(df, "state")
```

Encoding to an index removes textual state abbreviations and assigns an index to each. Rather than getting 50 dummy variables, you have a single column variable. It is important that you not encode predictors as indexes. This will introduce bias. For example, two states might have indexes that are very close to each other. The distance between state indexes would convey undesired bias information to the network. This limitation does not exist for the output (y) values as TensorFlow simply treats each output as a separate independent category.

ENCODING CONTINUOUS DATA FOR A DEEP NEURAL NETWORK

Neural networks prefer their input columns to be centered near zero. They do not need to be normally distributed, but the zero

centering has been shown to help with neural network accuracy. Statistical z-scores are a great way to accomplish this. The following function will normalize a column to z-scores:

```
def encode_numeric_
zscore(df,name,mean=None,sd=None):
    if mean is None:
        mean = df[name].mean()

    if sd is None:
        sd = df[name].std()

    df[name] = (df[name]-mean)/sd
```

This function allows the mean and standard deviation to either be passed in or calculated. If you are training the initial dataset you should not provide mean and standard deviation, as you have enough data to calculate them. However, later you might have only a few values to generate predictions on, so it is helpful to provide the mean and standard deviations from the original training set. To convert the column "income" to z-scores use the following call:

```
encode_numeric_zscore(df,”income”)
```

Later, if you wanted to normalize just a few rows, and you already knew the mean and standard deviation were 50,000 and 15,000, you would call:

```
encode_numeric_zscore(df,”income”, 50000,
15000)
```

Once all of the input columns have been normalized correctly, you are ready to train a neural network.

TRAINING A DEEP NEURAL NETWORK

A TensorFlow network is trained using two sets of data named x and y. The dataframe (df) must be separated into these predictors (x) and the expected output (y). The following function can be used to do this:

```
def to_xy(df,target):
    result = []
    for x in df.columns:
        if x != target:
            result.append(x)
    return df.as_matrix(result),df[target]
```

If you wanted to predict (y) the income column for the dataframe (df) you would use the following call to separate into x and y:

```
x, y = to_xy(df,”income”)
```

Now that the dataframe is separated, the neural network can be created and then trained.

To create and train a neural network for classification, use the following code:

```
classifier = skflow.TensorFlowDNNClassifier(
    hidden_units=[30, 20, 10],
    n_classes=3,steps=200)
classifier.fit(x, y)
```

The above neural network would have hidden layers with 30, 20 and 10 hidden neurons. The number of hidden neurons can affect the accuracy of the neural network. Usually you will start with a larger number of hidden neurons (30) and add layers working down to a smaller number (10). This network would be able to classify three classes, and 200 steps would be used to train it.

To create and train a neural network for regression, use the following code:

```
regressor = skflow.TensorFlowDNNRegressor(
    hidden_units=[10, 20, 10],
    steps=200)
regressor.fit(x, y)
```

Fitting the neural network may take a while, depending on how many steps you have specified. The more steps, the more accurate the neural network will become.

EVALUATING A DEEP NEURAL NETWORK

There are a variety of ways to evaluate a neural network. Two of the most simple are root mean square error (RMSE) for regression and accuracy for classification. The following calculates the RMSE score:

```
score = \
  np.sqrt(metrics.mean_squared_
error(regressor.predict(x),y))
print("Final score (RMSE): {}".
format(score))
```

The RMSE error simply measures the magnitude of the average difference between the expected outcome and the actual outcome. Lower RMSE scores are better.

Accuracy is measured similarly:

```
score = metrics.accuracy_score(y,
classifier.predict(x))
print("Final score: {}".format(score))
```

Accuracy is simply the percent of data items that were classified correctly. Higher accuracy scores are better.

OTHER APPLICATIONS OF DEEP LEARNING

Deep neural networks can accomplish the same type of classification and regression tasks that other models like support vector machines, GLMs and decision trees are used for. While deep neural networks might sometimes provide better results than other model types, there are two important areas where deep neural networks really shine: computer vision and time series prediction.

Computer vision might seem like a technology more suited to a Google self-driving car than an insurance company. However, there are cases where computer vision can be very useful to an insurance company. Two recent Kaggle competitions highlighted these areas. The first was the Kaggle Diabetic Retinopathy Detection.[8] This challenge used predictive models to look at retinopathy images and predict if an individual had diabetes. Additionally, State Farm ran a Kaggle competition to analyze images and detect distracted drivers.[9] Both of these computer vision applications could help insurers to determine risk.

Time series is another area where neural networks are particularly adept. This is because neural networks can be recurrent. By allowing connections backwards through the neural network they are able to learn to predict patterns in a series of inputs, not just patterns within individual input. A neural network could have two inputs to read the systolic and diastolic blood pressures. However, a traditional model would always output the same for a given reading. A recurrent neural network could detect a pattern in a series of readings. Two of the most current types of recurrent neural networks are the LSTMand gated recurrent unit (GRU) networks. Time series and neural networks will be the topic of a future article for this newsletter. ■

Jeff Heaton is a lead data scientist at RGA Reinsurance Company in Chesterfield, Mo. He can be reached at *jheaton@rgare.com*

ENDNOTES

1 *https://github.com/amznlabs/amazon-dsstne*

2 *https://github.com/baidu/Paddle*

3 *https://github.com/tensorflow/tensorflow*

4 *https://github.com/Microsoft/CNTK*

5 *https://www.tensorflow.org/versions/r0.10/get_started/os_setup.html*

6 *https://datascientistworkbench.com/*

7 *https://github.com/jeffheaton/t81_558_deep_learning/blob/master/t81_558_class2_tensor_flow.ipynb*

8 *https://www.kaggle.com/c/diabetic-retinopathy-detection*

9 *https://www.kaggle.com/c/state-farm-distracted-driver-detection*