Applying Deep Learning Algorithms to Alarm/Anomaly Detection for Grid Jobs

Erik Torres
University of Illinois at Urbana-Champaign
Fermi National Accelerator Laboratory
FNAL SIST Program, August 17th, 2017
Supervisor: Dr. Michael Kirby

Abstract

Within the complex system of distributed computing offered by the Fermilab General Purpose Grid (GPGrid) and the Open Science Grid (OSG), failures within the infrastructure can be difficult to recognize and distinguish because of the rapidly changing dynamics of job scheduling and misbehaving infrastructure. The Fermilab GPGrid is a computing cluster with more than 19,000 cores and running computing processes from more than 20 particle physics experiments and workflows from the OSG. With the technological advancements there have been in the field of machine learning, particularly in the subfield of deep learning, I looked into applying deep learning algorithms to aid in developing an alarm/anomaly detection program for determining if the continuous state change of the system is part of normal operations or an abnormal situation. This project utilized deep learning algorithms from tensorflow and monitoring data of the Fermilab GPGrid to develop an alarm system to identify abnormal behavior within the infrastructure and system for a better operation of the resources offered by the Fermilab GPGrid and the OSG.

Introduction

With the growing interests in deep learning and notable breakthroughs that have been accomplished in the past several years, deep learning offers ways in developing recognition of patterns for outstanding classification of the patterns which are fed in as inputs and to be trained on for analyzing and classifying to a target or output. Before going in depth with the discussion of deep learning, it is essential to understand the fields above deep learning.

Artificial intelligence, being the broadest term, is the application of any technique that enables computers to mimic human intelligence using logic, if-then rules, decision trees, and machine learning (includes deep learning). With the effort to automate intellectual tasks normally performed by humans, symbolic AI (programmers handcrafting a sufficiently large set of explicit rules for manipulating knowledge) has proved to offer solutions to well-defined logic problems (like a game of chess). However, it turned out to be that symbolic AI had problems in figuring out rules for solving complex problems like image classification, speech recognition, language translation, etc.

Machine learning, being a subfield of AI, includes statistical techniques that enable machines to improve at tasks with experience. In essence, a machine learning system is “trained” rather than explicitly programmed; the system is presented with many “examples” relevant to a task and it finds a statistical structure in these examples, allowing the system to come up with rules for automating the task. Within machine learning, a computer can be programmed to learn one of two ways. The computer can either be programmed to learn via supervised learning or unsupervised learning. In supervised learning, there are output datasets (targets of given inputs)

that are provided which are used to train the machine and get the desired outputs. However, in unsupervised learning, no labels (targets to given inputs) are provided, instead the data is structured into different classes on its own. In technicality, machine learning is the search for useful representations of some input data, within a pre-defined space of possibilities, receiving assistance for making decisions based on some sort of feedback signal. Speaking in terms of analyzing representations of data, this leads to a discussion of deep learning.

Deep learning is a subfield within machine learning; its take on learning representations from data comes from setting up successive "layers" of meaningful representations of the data. Deep learning is merely a mathematical framework for learning representations from data. The layered representations are learned via models called "neural networks" that are structured in literal layers stacked on after the other. Since machine learning deals with mapping inputs to targets by observing many examples of those inputs and targets, deep learning does this input-to-target mapping via the sequence of layers. An artificial neural network has many layers of nodes between the input and output layer. The "deep" in deep learning comes from the idea of the number of successive layers (also known as hidden layers) of representations there are in a neural network; the number of layers that contribute to a deep learning model of the data is called the "depth" of the model. The specifications of what a layer does to its input data is stored in the layer's "weights." These weights are adjusted as the loss score (an error measurement of how wrong the neural network is classifying the input data) is decreased from the loss function. The lost function (also known as the cost function) measures how far off the neural network is with the current weights. The "learning" comes from the updates of the values for the weights of all the layers in the neural network so that the network will correctly classify the example inputs to the associated targets. In particular, each node in the neural network has a set activation

function that allows the mapping of the input to output via a non-linear transform function for neural networks to make complex boundary decisions for features at various levels of abstraction. Examples of common activation functions used in a NN are: logistic, hyperbolic (tanh), exponential, softmax, unit sum, square root, sine, ramp and step. The means by how exactly this is done will be discussed in the sections of Tensorflow and Training of the RNN.

More specifically, why is deep learning so attractive to use now? The reason why deep learning is picking up interests by data scientists and researchers is because of the revolution of remarkable results on perceptual problems like “seeing” and “hearing” problems, which involve skills that seem natural and intuitive to humans. There are even some models that are trained by deep learning algorithms, for example, in image recognition, that outperform humans in recognizing images. Other notable breakthroughs in areas of machine learning that have been achieved by deep learning: near-human level image classification, speech recognition & handwriting transcription, improved machine translation & text-to-speech conversion, near-human level autonomous driving, improved ad targeting and improved search results on the web. The technical forces that have drove advancements in deep learning is better hardware, more datasets & benchmarks and algorithmic advancements.

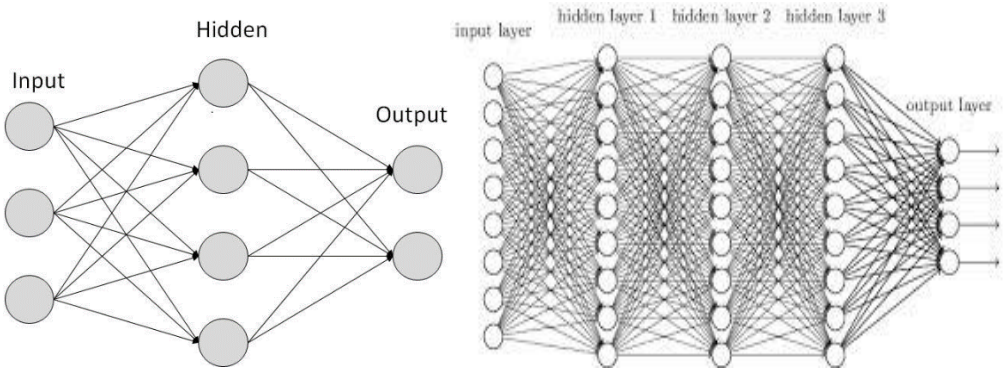

Figure 1: An example of a simple early Neural Network

Figure 2: An example of a complex developed Neural Network

TensorFlow

To apply the deep learning algorithms for the project, I had to familiarize myself with using TensorFlow with Python. TensorFlow is a Python library that allows users to express arbitrary computation as a graph of data flows. TensorFlow was developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research.

Tensorflow works to apply deep learning by setting up graphs that represent computations. Nodes that are created in the graph are called ops (short for operations). These nodes can take zero or more tensors as input into these nodes and produce (return) zero or more tensors. A tensor, in TensorFlow, is a typed multi-dimensional array that contain data which flows throughout the computation graphs, hence the name TensorFlow. The tensor data structure represents all data which are passed between operations in the computation graph. A tensor has a static type, a rank (how many dimensions the tensor describes) and a shape (essential for specific inputs that must match for batches of training data, weights or biases).

Initially, to build a graph, ops need to be set up for the graph. After the nodes are initialized, the graph must be launched to execute the computations in these nodes. In order to launch a graph, a session object has to be created. Once the session is created, since the session constructor implicitly launches the default graph, the nodes in the graph can be ran explicitly by

using the run function within the session class on the session object created by using the specific node that needs to be executed as an input into the run function.

Lastly, another essential step for understanding the usage of TensorFlow is initializing and maintaining the states of variables used across executions of the graph that is launched. These variables are used to provide shapes, constants and placeholders to make the running of a graph work. However, it is imperative to know that all the variables must be initialized with an initializer function that must be ran on a session like discussed earlier to make sure that the variables used correctly in the graph. Some of the important variables that are set up for the neural network are mainly for initializing the weights and biases (set randomly because they will be updated as the lost function is calculated) and setting placeholders. Placeholders also play a pivotal role in the training of a neural network as they will take inputs from a batch of training data and target data for the neural network. There will be more discussion about this aspect in Training of the RNN section.

Input Data & Target Data

To train a Neural Network (NN) in deep learning, a critical step in starting the process is to gather up the training inputs and targets that will be used by the NN to train on. To see how well the NN is learning on the training input and target data, there also has to be a set of testing input and target data to evaluate how well the NN is classifying its prediction to its associated output, given any input data.

In order to gather data about the system as a whole, I had to access data from FIFE Batch Monitoring. FIFE stands for the Fabric for Frontier Experiments group; they provide central tools and services to address common challenges in distributed computing (resources from

Fermilab GPGrid and resources from OSG) that are used by experiments to run their computer processes. In addition to providing services to address challenges in distributed computing, they also provide a means by which they gather data about grid job submissions and place it in to two different places that can be accessed by the FIFE monitoring group or users of the experiments that are using the distributed computing. The FIFE Batch Monitoring Pipeline (as shown in Figure 3) first starts with HTCondor, which manages information about job statuses and provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Probes then collect data from the grid and job details as a whole. Raw documents are then placed into Kibana through elasticsearch and time series data is placed into Grafana through graphite. Elasticsearch stores fully indexed JSON which could be accessed through a python script that queries the graphical web frontend to elasticsearch. Graphite stores time-series data that is used to be shown on Grafana. I worked on developing a script for querying and getting data back from elasticsearch as I thought that the most important data would come from having individual details on the status of the jobs provided for the hour time slices, after running the script. I ran the script to gather about 4200 hours of monitoring data (around 5 months of data). The data accessed from elasticsearch provided details on an hour time slice (to form a sequence of data) on: the number of jobs that were executed (ExecuteEvents), the number of jobs that were aborted (either because of user or because they are using more memory on the node than necessary – JobAbortedEvent), number of jobs that were disconnected from the job sub server (job sub server provides tools that manages grid submission), the number of jobs that were evicted from a Gliden site because the site allows priority to its users, the number of jobs that were held because of a user's fault in that running time or memory usage exceeded, the number of head nodes that were updated (JobImageSizeEvent), the number of jobs that failed to

reconnect again after being disconnected and the number of jobs that were able to reconnect again, and several other types of jobs.

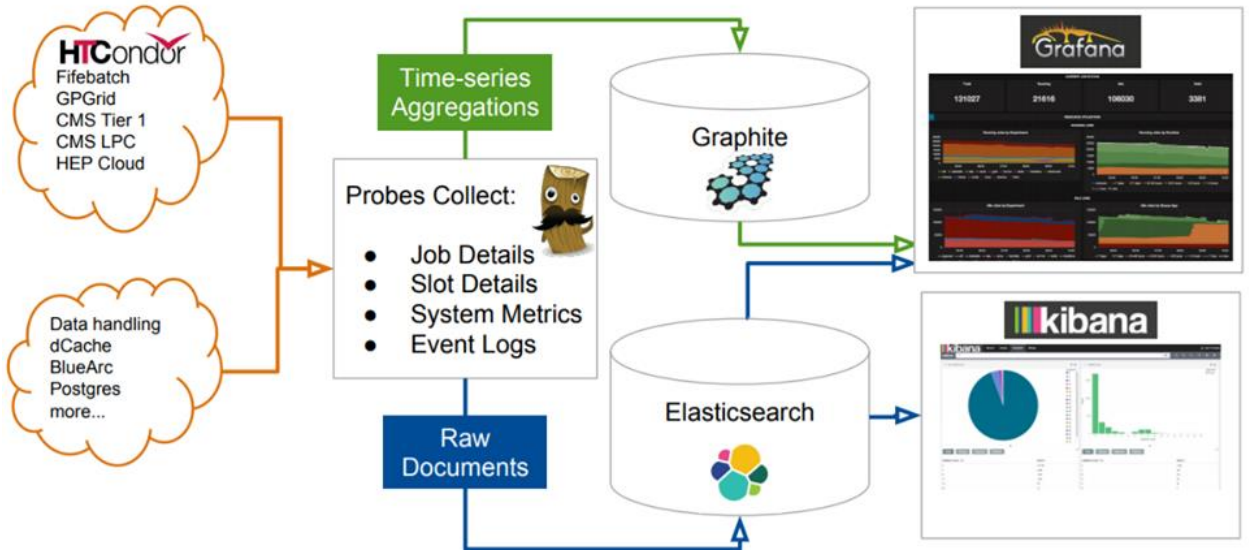

Figure 3: FIFE Batch Monitoring Pipeline

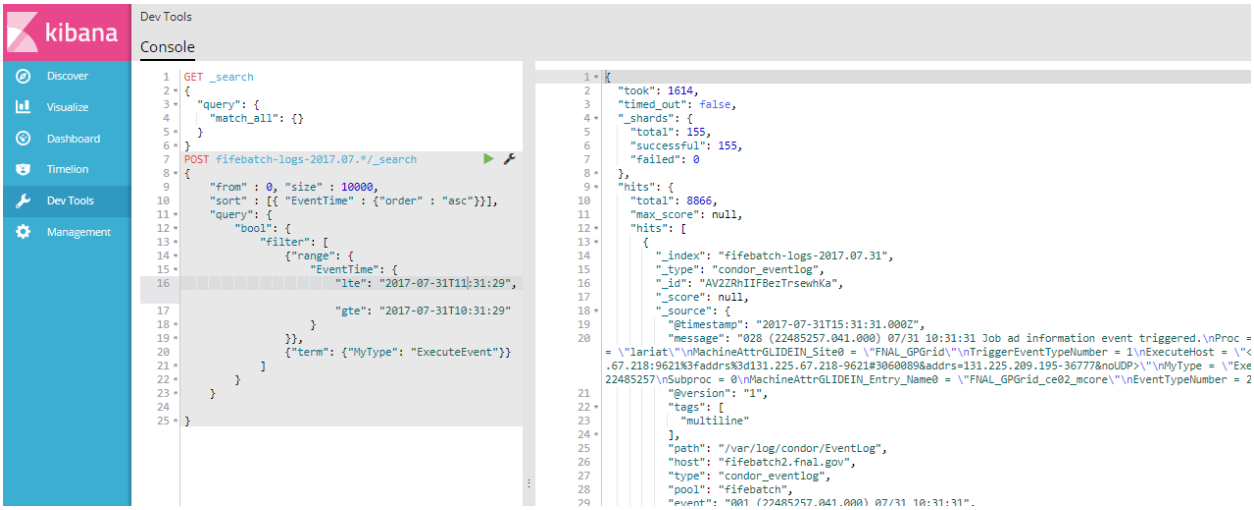

Figure 4: Example of a query through elasticsearch on Kibana

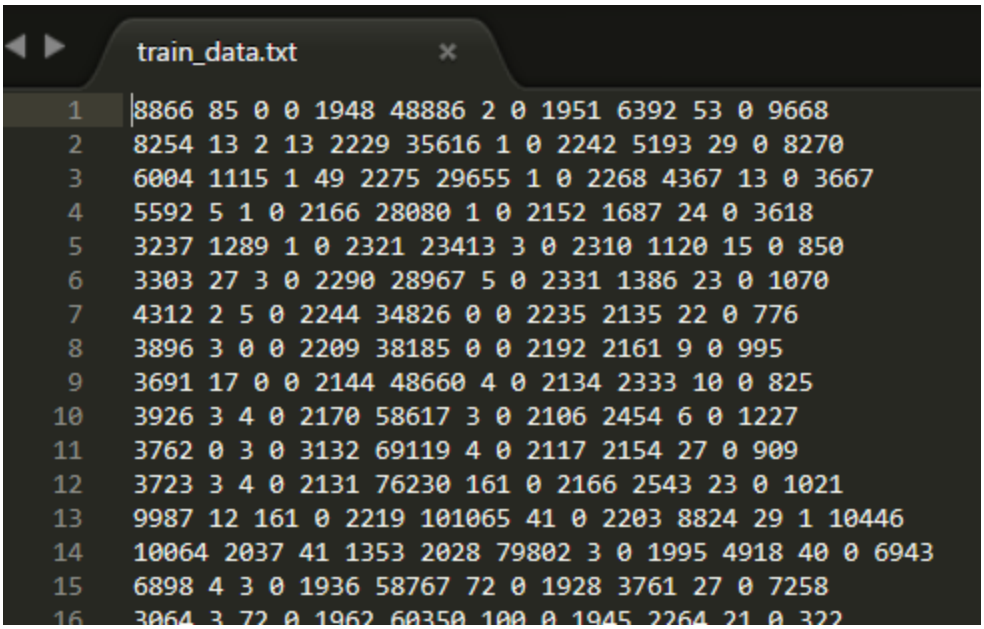

Figure 5: This is how the data from Kibana looked after creating the script for querying elasticsearch

Using this information about the status of job grid submissions, it served as a good start to have as input data into the NN. However, with initially having the intentions of practicing supervised learning, there had to be target data gathered to be associated with this input data. In order to classify an anomaly as a target with the hour slice of input data, I had to interface with Grafana and manually classify anomalies based on the outage timeline. The anomalies that were classified by some of the outages had to deal with nodes that had collapsed, fifebatch being unresponsive and several number of jobs that had been disconnected and reconnected. Overall, the targets were classified into a 0 being normal operations and 1 being an anomaly.

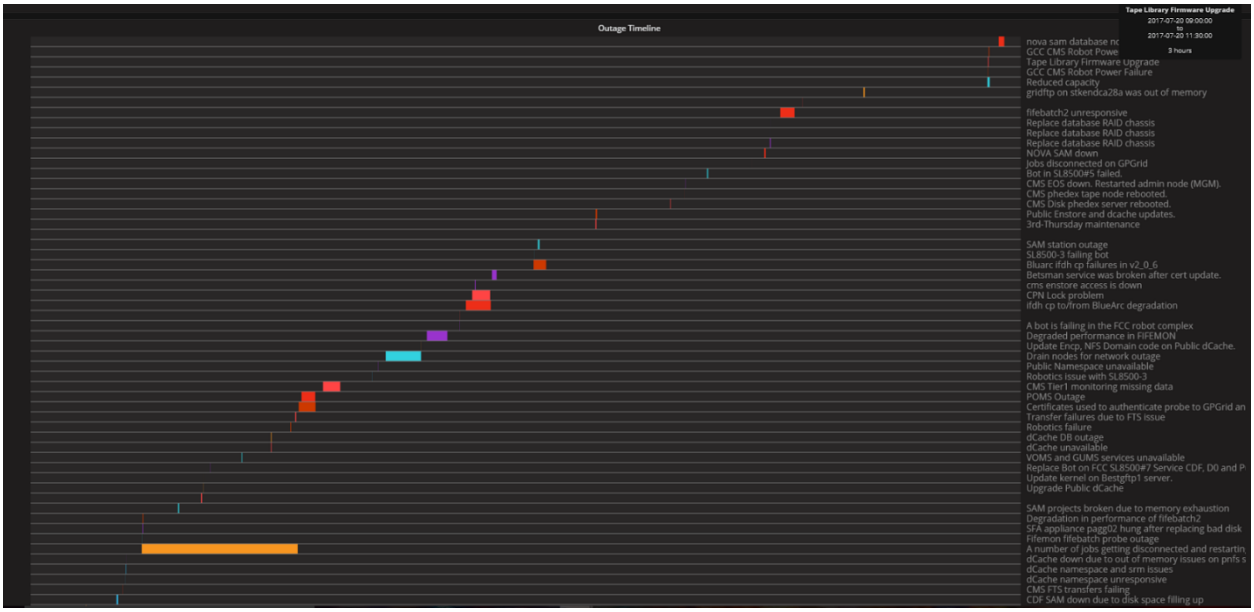

Figure 6: Outage timeline on Grafana

Recurrent Neural Network

Another important step in the process for training a NN is to choose they type of NN that would be beneficial to use for dealing with the specific goal at hand. There are several types of NNs, but the main ones to pay attention to are feed-forward neural networks, convolutional neural networks and recurrent neural networks (RNNs). A feed-forward net is built to be a general-purpose model with a very basic NN having an input layer, an output layer and one or more hidden layers. A convolutional neural network is similar to a feed-forward neural net because of how data in passed throughout the network. However, a convolutional net is different in the way they learn compared to a feed-forward net. For example, using a convolutional net with image recognition is very common because of the filters that are passed over an underlying image to recognize features in each section. In comparison to a feed-forward NN, a feed-forward net may not recognize that feature if it were to show up in an uncommon position because it

would analyze and learn the image as a whole, instead of processing it in pieces. An example of how convolutional nets works is shown in Figure 7.

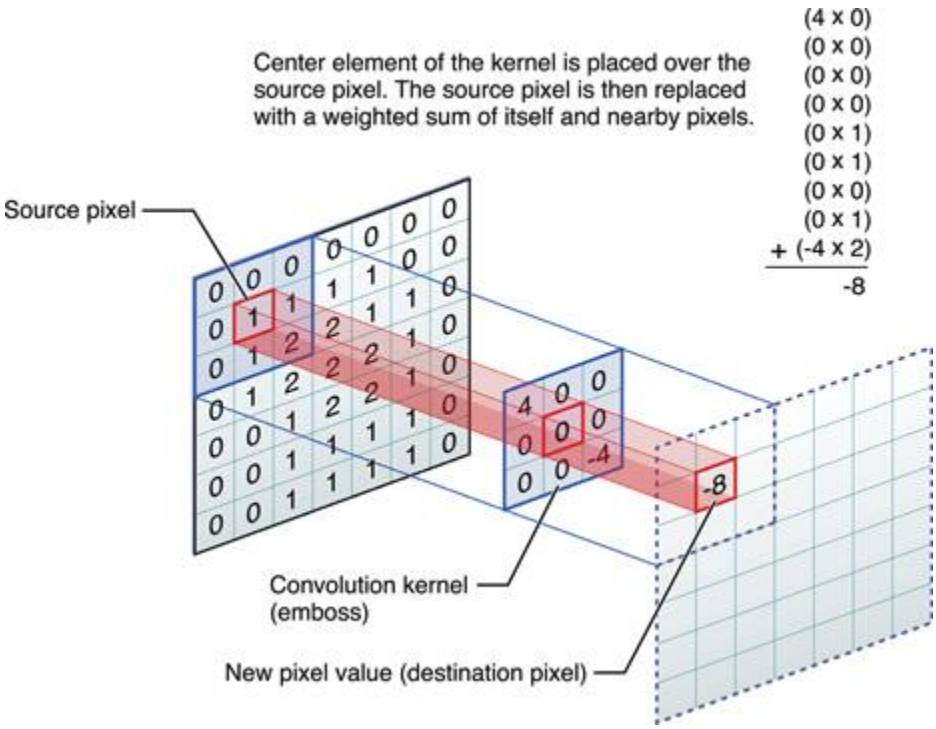

Figure 7

For dealing with a data that comes in arbitrary sequences, like time series data, RNNs work well for processing this type of data and make a good start for developing an anomaly detection system. RNNs work different than feed-forward nets as RNN hidden layer nodes maintain an internal state (sort of memory) that is updated when there are new inputs into the NN. In essence, the nodes (also known as cells) make decisions based on current input and what has come before. As an RNN traverses the input sequence of data that is fed in, the output for every input also becomes a part of the input for the next item of the sequence. This is where the 'recurrent' property of the network comes in, where the previous output for an input item becomes a part of the current input item in the sequence and the last output. Its tracking of

dependencies and correlation within data over many time steps require that its current state and some number of previous states be known. Figures 8 and 9 show how a RNN works.

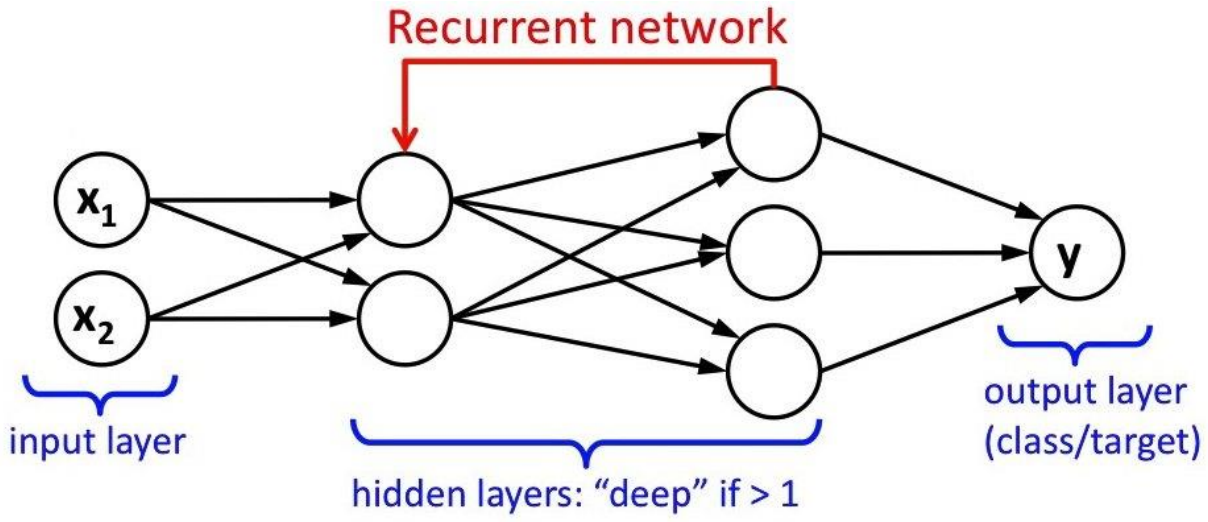

Figure 8: Recurrent Neural Network

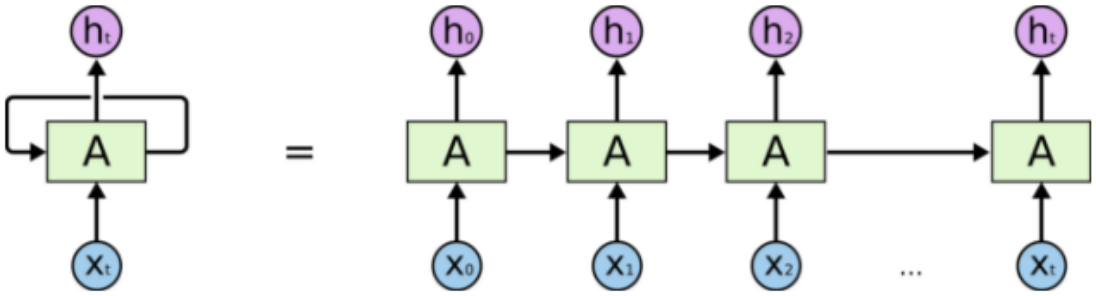

Figure 9: An unrolled Recurrent Neural Network shows how a cell/node behaves with having an input (x_t) and an output (h_t)

In particularly, the type of RNN that would be beneficial to use is a Long Short Term Memory network (LSTM). LSTMs are just a special kind of RNN which works, in many tasks, better than the standard RNN because of its capability to learn long-term dependencies. Like the

RNN, it still retains important data from the previous inputs and uses that data to modify the current output. The difference between a RNN and a LSTM is the structure in which the neural networks take. They are both still a chain of repeating modules of neural networks, however, a LSTM differs from a RNN because of the structure inside the cell/node in which four layers (four layers amounts to using four activation functions) interact in special way, whereas the structure inside a RNN cell only has 1 single layer to make the decision for the signal that is passed through the cell/node. Overall, LSTMs solve the problem of training over long sequences and retraining memory by adding a few more gates that control access to the cell state. Figures 10 and 11 show how RNNs and LSTMs interact as a repeating module of cells.

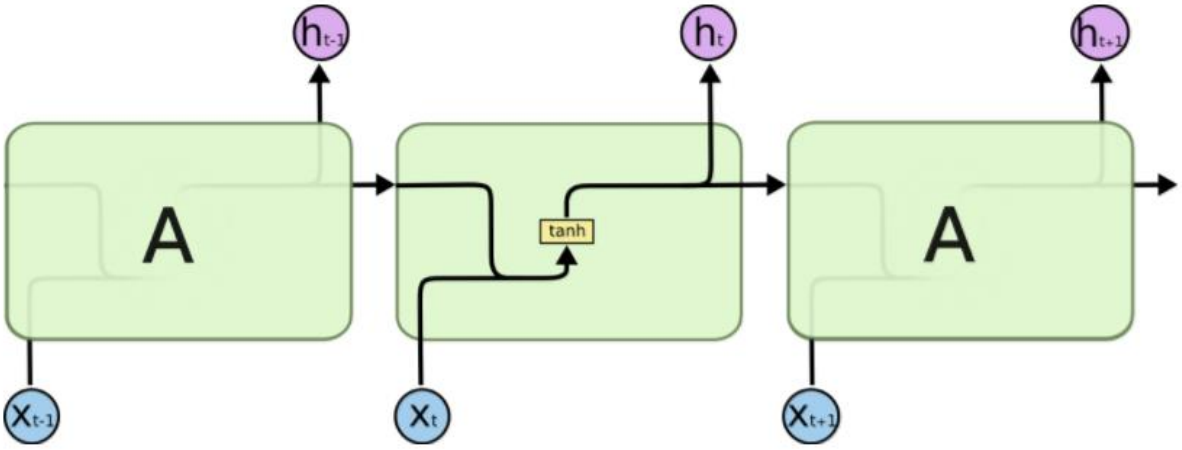

Figure 10: RNN repeating modules

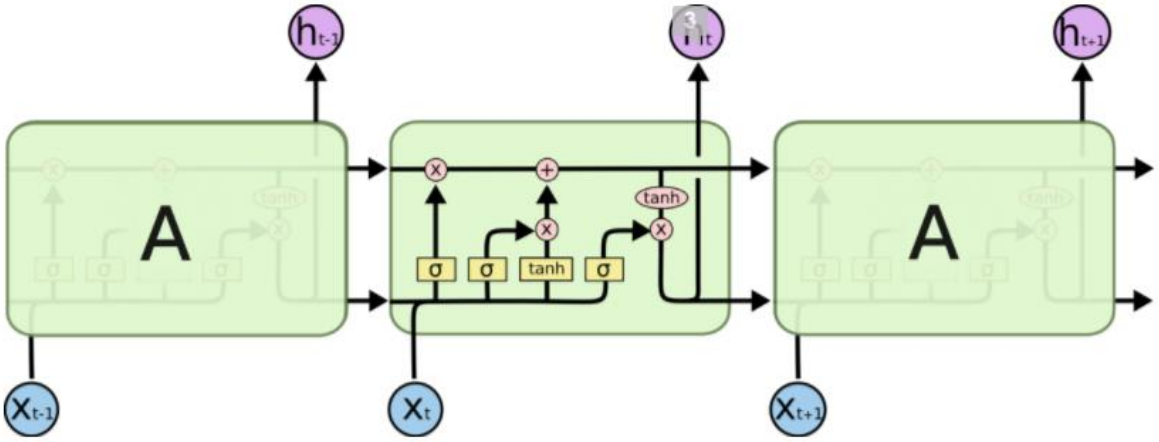

Figure 11: LSTM repeating modules

Training of the RNN

In order to train the RNN, the input data had to be set up correctly in a file to be fed as batches into the RNN. To make this possible, placeholders had to be initialized to be able to take the values of the input and target batches. The input for the RNN expects a tensor of batch size x feature size. Batch size is the number of datasets that I would like the RNN to evaluate at a time. Feature size is the numbers that describe the dataset that I used. In my case, I used a batch size of 20 with a feature size of 13 because that was the number of events that I analyzed to be a dataset from elasticsearch. After setting up these placeholders (not shown on code below, but made in the main function of the program), the RNN was made and initialized through the given functions in tensorflow and variables for the weights and biases. The RNN function was used to return the prediction that the neural network had made, given the input data in "RNN_inputs." The RNN function was called from a function that I made to train and run the model; this function launched the graph made of the initialization of nodes by creating a session object, like discussed earlier. In this function, the predictions of the RNN was compared to the labels that

were the targets of whether the input data was considered normal operations (0) or an anomaly (1). The lost function, which measures the difference between a neural net's guess and the ground truth, computed a softmax cross entropy between the labels and the predictions. This error calculation was then minimized by an optimizer function called AdamOptimizer, which applies the learning rate for how quickly the neural network can learn to make adjustments to the weight values. Epochs (one epoch is a complete pass through all of the training data) are ran so that the RNN is trained until the error rate is acceptable.

```
"""This function is in charge of creating the RNN with LSTM cells"""
    # RNN_inputs - batch of data of shape: [20,13] used for inputs into a layer of the RNN
    # config - configuration
def recurrent_neural_network(RNN_inputs,config):
    # Weights & biases made for a layer
    Weights = tf.Variable(tf.random_normal([config.LSTM_units,config.dataset_size]),dtype=tf.float32)
    Biases = tf.Variable(tf.random_normal([config.dataset_size]),dtype=tf.float32)

    # initialize the number of cells that will be used from set number of layers
    LSTMlayers = []
    for _ in range(config.num_layers):
        temp_cell = tf.contrib.rnn.LSTMCell(num_units=config.LSTM_units)
        LSTMlayers.append(temp_cell)

    # creates an RNN cell composed sequentially of multiple LSTM simple cells
    RNNcell = tf.contrib.rnn.MultiRNNCell(LSTMlayers)

    # set up first state for the cell
    initial_cell_state = RNNcell.zero_state(config.batch_size,dtype=tf.float32)

    outputs = []
    state = initial_cell_state
    # set up for loop for feeding in a batch into RNNCell
    for i in range(config.num_steps):
        cell_ouput, state = RNNcell(RNN_inputs,state)
        outputs.append(cell_ouput)

    output = tf.matmul(outputs[-1],Weights + Biases)
    return output
```

Figure 12: RNN code

```
"""This function will run the model and train the RNN"""
    #   session - current Session for running Graph Ops
    #   anomaly_model - object which has methods and properties that were trained
    #   is_training - Not sure if needed, but this bool will tell function if training (true)
def run_model(raw_input_data,raw_target_data,config,RNN_input_placeholder,RNN_target_placeholder):
    # create a name scope: context manager for which is used when defining a python op
    with tf.Session() as sess:
        with tf.name_scope("Train"):
            # prediction made my RNN on set of 13 events -> 1 = anomaly; 0 -> normal
            prediction = recurrent_neural_network(RNN_input_placeholder,config)
            # cost function/ loss function
            cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=RNN_target_placeholder,logits=prediction))
            # optimizer that implements the Adam algorithm
            optimizer = tf.train.AdamOptimizer(config.learning_rate).minimize(cost)
            # initializes all variables created in the graph
            tf.global_variables_initializer().run()
            # run 10 epochs on training data
            for epoch in range(config.num_epochs):
                epoch_loss = 0
                for batch_iteration in range(config.num_batches):
                    # get current input & target batch
                    curr_input_batch = normal_batch(True,raw_input_data,batch_iteration)
                    curr_target_batch = normal_batch(False,raw_target_data,batch_iteration)
                    # run the optimizer and cost function for the specific inputs and outputs to evaluate for the RNN
                    op, cost_loss = sess.run([optimizer,cost],feed_dict={RNN_input_placeholder: curr_input_batch, RNN_target_placeholder: curr_target_batch})
                        # NOTE: get error about feeding in tensor objects; so I will feed in a numpy array
                    epoch_loss += cost_loss

                print('Epoch ', epoch, ' completed out of ', config.num_epochs, ' loss: ', epoch_loss)
```

Figure 13: Code for training the Model

Conclusion & Future Directions

In conclusion, I collected 4200 hours of monitoring data and formatted the data to be prepare to feed into the RNN as input and target data. The code for the scripts and programs that I wrote are in a code repository created on github. Within the program, the architecture of the network is set up, but still needs some finishing touches with a couple of bugs that need to be fixed. Going forward, there are several aspects that can be completed to make the anomaly detection work well. One of those aspects is to use a bigger dataset with more information of the current infrastructure (background) data that is on Grafana like the number of Queued jobs, Slots claimed by a Virtual Organizaton (users or experiments) in the GPGrid, Grid Health and Grid Utilization. In addition, applying a dropout layer for the RNN will also help in the training of the RNN as it creates more generalizable representations of data. This is useful as it prevents overfitting with neural nets that have a large number of parameters. The project can also head into the possibility of applying unsupervised learning after supervised learning has taken place to give the computer a chance to classify and recognize certain patterns within the data that seem

abnormal, in addition to having a classification of what the input data looks like when an anomaly has taken place.

Acknowledgements

I would like to thank my supervisor, Dr. Michael Kirby for helping throughout this project. I would also like to thank Kevin Retzke, Jod Boyd and Shreyas Bhat for helping me carry out my project during the summer. Finally, I would like to thank the SIST committee for making this summer possible.

References

Anomaly Detection for Time Series Data with Deep Learning. (2017, February 17). Retrieved from https://www.infoq.com/articles/deep-learning-time-series-anomaly-detection

Chollet, F. (2017). What is Deep Learning? Deep Learning with Python. (pp. 1-16). Manning Publications

Recurrent Neural Network Regularization. (2014, September 8). Retrieved from https://arxiv.org/pdf/1409.2329v5.pdf

The Unreasonable Effectiveness of Recurrent Neural Networks. (2015, May 21). Retrieved from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Understanding LSTM Networks. (2015, August 27). Retrieved from shttp://colah.github.io/posts/2015-08-Understanding-LSTMs/