ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUCPA03

EXPERIENCE WITH MACHINE LEARNING IN ACCELERATOR CONTROLS *

K.A. Brown†, S. Binello, T. D'Ottavio, P. S. Dyer, S. Nemesure, D. J. Thomas,
Collider-Accelerator Department, BNL, Upton, NY, USA

Abstract

The repository of data for the Relativistic Heavy Ion Collider and associated pre-injector accelerators consists of well over half a petabyte of uncompressed data. Some of this data is viewed and analyzed in the course of accelerator operations. Other data has been retrospectively analyzed offline. However, a large fraction of that data has never been analyzed. Even data that has been analyzed may contain additional useful information that did not come to the surface during initial processing. We will describe in this paper our efforts to use machine learning techniques to pull out new information from existing data. Our focus has been to look at simple problems, such as associating basic statistics on certain data sets and doing predictive analysis on single array data. The tools we have tested include unsupervised learning using TensorFlow™, multimode neural networks, and hierarchical temporal memory techniques using NuPIC.

INTRODUCTION

Statistical machine learning uses automated techniques for predictive modeling [1]. What distinguished these approaches from classical statistical methods is they are data driven. No linear or specific structure is imposed on the interpretation of the data. Machine learning is focused on developing efficient algorithms to optimize a predictive model.

In our business, we put much effort into real-time processing of data, in order to present to operators, engineers, and scientists results that allow them to either diagnose the health of the system or have a signal on which to perform some optimization process. For example, most of us monitor (on some comfort display in the main control room) the beam current or intensity in the accelerator in real time. People become "trained" in recognizing when these signals are doing the wrong thing. For example, operators get to be extremely good at making the very abstract connection of the behavior in the beam current signal to particular fault conditions in the accelerator.

Our job, here, is to ask whether we can use machine learning to recognize (or train, if you like) in software what a person is able to detect visually. An advantage to such an approach is the computer can be looking all the time, while people tend to get distracted. Also, a computer can, possibly, react much more quickly than a person, if a learned response is given to the algorithm.

We break up machine learning for accelerator controls into two domains; recognizing anomalies (true anomalies and outlier values) and developing learned "responses". One example of a learned response is to consider the use of machine learning for correcting the beam trajectory in a beam line. This highlights well how machine learning uses no model but just learns the statistics of a signal and adapts a response based on setting target statistical values (e.g., bring a given signal to within n sigma of a target value of x by adjusting parameter a).

RELATED WORK

Machine learning, as a field, has grown out of advances in Artificial Intelligence research, particularly in the areas of pattern recognition and computational learning theory. The term, coined by Arthur Samual, IBM, goes back to 1959, where the idea was to give "computers the ability to learn without being explicitly programmed" [2]. The *Machine Learning* journal has been in publication since 1986. So there is a long and interesting history to this field.

For particle accelerators, the use of Machine Learning techniques goes back as far as 1987, when T. Higo, H. Shoaee, and J. E. Spencer, SLAC, discussed applications of artificial intelligence to problems in accelerators [3]. In 1989, J. E. Spencer, SLAC, discussed using Neural Networks techniques in accelerator controls [4]. Two years later, D. Nguyen, M. Lee, R. Sass, and H. Shoaee, SLAC, used Neural Network techniques to develop a dynamic feedback system for beam line controls [5].

More recently, A. L. Edelen, et al., have been experimenting with the use of machine learning techniques for RF gun temperature control [6]. At the SwissFEL, A. Rezaeizadeh, T. Schilcher, and R. Smith, PSI, used a model-free iterative learning approach to produce flat-topped RF pulses. The method iteratively updated the input pulse shape to generate the desired output pulse shapes in the RF system [7].

At Los Alamos, Sung-il Kwon, et al., used iterative learning techniques for modeling the SNS SRF cavity feedback controls [8].

At TRIUMF, M. Laverty and K. Fong used an iterative learning feedforward LLRF controller to improve the beam stability in the e-linac [9].

The use of machine learning for orbit control goes back to the work at SLAC, but has been investigated by many others over the years. In 2012, E. Meier, Australian Synchrotron, studied the use of neural networks for orbit correction [10]. Going back further in time, in 1994 E. Bozoki and A. Friedman, BNL, studied the use of neural networks for orbit control in the National Synchrotron Light Source [11].

* Work performed under Contract Number DE-SC0012704 with the auspices of the US Department of Energy.
† kbrown@bnl.gov

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUCPA03

A number of other examples showing how people have made use of Machine Learning techniques can be found in the various conference proceedings on the JACoW site [12]. A few we found are listed here [13–19]

A LITTLE ON THEORY

The probability that the next observation in some data will be a certain value, given some set of previously observed data, can be derived from our knowledge of the data up to this time and how it impacts the probability of a given observation. This may sound like a circular definition, but it is simply a statement on how we make predictions based on what we already know. This is better known as Bayes' theorem and can be written as,

$$P(H|E) = \frac{P(E|H)}{P(E)} \cdot P(H) \tag{1}$$

With this notation, you read *(H|E)* as a conditional. So *P(H|E)* is the probability of H given E and $P(E|H)/P(E)$ is a way of saying the impact of E on the probability of H.

Why are we starting out with Bayes' theorem?

There are basically two ways to make predictions based on prior knowledge. One is known as the Frequentists approach, which is what we know as a historical way of analyzing data and focuses on the frequency or proportion of the data. The basic idea here is that any two repetitions of the same experiment should produce statistically independent results. This is known as statistical inference. Bayesian inference, the second approach, uses Bayes' theorem to update the probability as more data is collected.

In the introduction we introduced *Statistical machine learning*, which, as we stated, is a technique for predictive modeling. Here we are discussing pure *Machine Learning*, which is really about algorithms that can learn from data. Machine learning, some parts of pattern recognition, and some aspects of neurocomputing are all subtopics of artificial intelligence. Without going too deep in this direction, the basic point here is, machine learning uses analysis of prior data to learn the characteristics of that data and enable predictions to be made about future data. Artificial intelligence takes that data and through some algorithm devises a basis for taking a decision. Now pattern recognition and neurocomputing are also subtopics of machine learning. They also use prior data to make predictions about future data.

Statistical Machine Learning

Let's take some vector of values, called a record, and classify that record with how similar records are classified. To do this we will use ensemble learning and form a decision tree. In ensemble learning we use many models to form a prediction. What we are after is to form a predictive model and we will focus on the statistics in the data to look at the underlying structure of the model. The simplest method to do this is called K-Nearest neighbors. Neighbors are records that have similar predictor values. A predictor value is basically a feature of the record (e.g., the ratio of two values or an average of some set of values).

A simple example might be to take a series of accelerator cycles where the record contains the beam current, the measured vertical tune, and whether the beam was successfully extracted or not (e.g., beam was lost before extraction). We can create predictor values from just the beam current and the vertical tune. We can predict whether the next cycle will succeed based on the new beam current and tune value, by comparing those values to a handful of similar values from the past (the nearest neighbors). Our prediction can be improved if we collect more data and average over records (i.e., binning the data). Since we likely want to achieve the highest beam current, we would look for nearest neighbors at high currents that are more likely to succeed. The process is not much different from what an operator would do to keep the operation of the machine stable but at the highest level of performance.

Neural Nets

The traditional way of viewing a neural net is as an interconnected group of artificial neurons in which each neuron receives weighted values from other neurons and an activation function defines when the neuron 'fires'. The weights are learned during a training period. Training periods can be supervised or unsupervised (self-learning).

Consider, for example, a simple feedforward network. A layer of input neurons receive one input signal per neuron. All other layers of neurons can receive an arbitrary number of input signals, which are outputs of other neurons. For each neuron, the inputs are weighted. How the neurons are interconnected defines the topology of the network and there can be hidden layers or signals can be cascaded. Typically, there is a desired output of the network (e.g., a beam position at a given point is 0). The training processes learn the weights that best achieve the desired result. However, there can be many parameters to optimize, depending on the network model.

Modern neural networks are used to model complex relationships in data, for example in pattern recognition or to untangle the statistics in overlapping probability distributions. They are naturally good at modeling data that has non-linear statistical properties. We usually represent neural networks as directed graphs, but the graph models can be highly complex and contain non-linearity's. There are many kinds of networks; Bayesian, Markov random field, bipartite (e.g., restricted Boltzmann machine), chain graphs (e.g., a Markov or Bayesian chain), and so on. The traditional feedforward neural network is a form of a Bayesian network, which is a directed acyclic graph.

The beam current and vertical tune example from the previous section is an example that can be described with a Bayesian network. Given values from the beam current and the tune we can either have or not have successful extraction. The joint probability function is;

$$P_J(S, B, T) = P(S|B, T)P(B|T)P(T), \tag{2}$$

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUCPA03

where S = Successful extraction, B represents the beam current safe level, and T is the tune safe level. Statistical learning could be used to determine these values. A safe level could be thought of as a probability for that variable to be in a safe or unsafe range. We can now ask what the probability will be that we have successful extraction, using a conditional probability function. As a neural network we want to have B and T to be learned such that S is always successful, while maximizing, if you will, both B and T.

TensorFlow ™

TensorFlow™, an open-source library developed by Google [20], is a framework for creating deep learning models. Deep learning models are basically multi-layered neural networks. In TensorFlow™a computational graph is built, which is basically a network of computational nodes.

The difference between what has been described so far, machine learning with neural networks, and what is done in more general, with deep learning tools, is to break up the learning into a process. In a TensorFlow ™model, a neural network model is constructed by the user, which then gets compiled into a dataflow graph. There are many neural network models to choose from; feedforward models such as autoencoders, probabilistic, or time delay, and recurrent neural network models, such as fully recurrent, long short-term memory (LSTM), hierarchical, or stochastic. For the work we have done, we have focused on the use of (unsupervised) autoencoders and LSTM, both of which map a set of input signals to a set of output signals with the intent to reconstruct both sets of signals in such a way that [Input ≈ Output]. LSTM has the added ability to learn context in time series data through the use of remember/forget gates.

In the neural network discussion, we described how the training process can be supervised or unsupervised. In supervised learning, we want to learn weights such that we construct a map from some set of inputs to a desired set of outputs. In unsupervised learning, we are basically looking for hidden patterns in the data (to find anomalies, for example). There is also reinforcement learning, in which some kind of feedback is provided to direct the learning (e.g., self-driving vehicles obtain constant feedback as to what is a correct or incorrect response.) For deep learning, tasks are often categorized (e.g., is an image a car or a cat? If those are the only two choices, does a dog get categorized as a car?) So in classification there is a model for the learner to follow. Supervised learning often uses regression to converge on the best map representation.

Hierarchical Temporal Memory

Hierarchical temporal memory (HTM) is an example of a neurocomputing approach to machine learning. The approach is described in the book, "On Intelligence", by Jeff Hawkins and Sandra Blakeslee [21]. Basically, they built a library that models the way the human cortex learns and by using this library methods for machine learning can be developed. To be clear, HTM is not a deep learning or machine learning technology. It is a machine intelligence framework, based on models of how animal brains function. This library is an open source project and there are various implementations (C++, Python, Java, Clojure) [22]. The developers of this system are mostly focused on researching ways to model the brain. The use of these libraries for things like machine learning is a spinoff of their research.

Deep Learning

For deep learning there are many tools. TensorFlow™is currently one of the most popular of these tools, especially since it is built to run on multiprocessor and GPU systems. Since machine learning is being used in so many areas, including speech recognition, image recognition, natural language processing, web searching, world wide web data mining, etc., many methods have been developed to turn a simple neural network into a hierarchical learning machine. In a hierarchical system there are many hidden layers and systems can combine supervised, unsupervised, and reinforcement learning into a single tool.

We will introduce just two deep learning tools, Theano and Keras. In our investigations we have used Theano and Keras, but we are still learning how these tools may or may not be useful in processing accelerator data.

Theano [23] is a Python library for defining and evaluating computations on multi-dimensional arrays. It was designed to support the rapid development of machine learning algorithms, with a focus on processing large amounts of data.

Keras [24] is a Python deep learning library meant to be easy to use. It is also designed to work on top of TensorFlow™or Theano, with a focus on enabling exploration without having to be an expert in machine learning.

TENSORFLOW™

TensorFlow™is, at its core, written in a combination of highly optimized C++ and CUDA as a library known as Eigen [25], as well as NVIDIA™'s cuDNN [26], an optimized DNN (deep neural network) library for NVIDIA™GPU's. Python is used by the user for neural network model expression, which TensorFlow™compiles to a dataflow graph in C++ and CUDA. It should be noted that TensorFlow™has out of the box neural network structures, but combining those into a usable model of multiple layers along with training and optimization algorithms is done by hand. If the user is willing to sacrifice some degree of versatility and learning ability/speed, there are third party libraries that can be used on top of TensorFlow™to construct models for you. Two of these being Theano and Keras. Both support TensorFlow™'s ability to take advantage of NVIDIA™GPU's.

Figure 1 shows the machine learning processing data flow model using TensorFlow™modules. The main modules are *neuralframe.py*, *generic.py*, *analysis.py*, and *procdata.py*. *Userscript.py* is how an application would process using these modules.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUCPA03

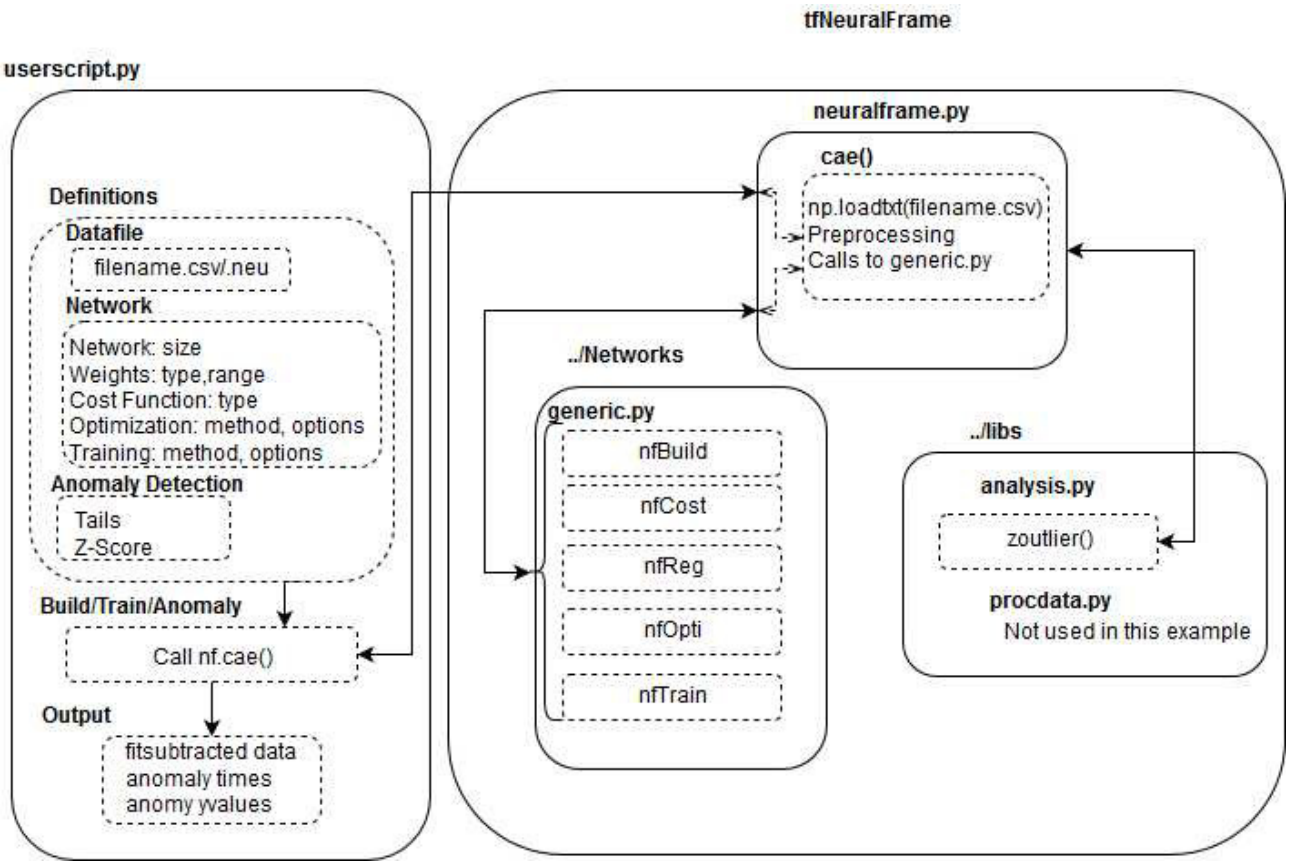

Figure 1: TensorFlow™based ML processing for C-AD data.

Neuralframe.py contains functions that represent and are used to construct and train specific types of networks. *Neuralframe.py* replaces a set of predefined neural networks. In this specific example a call is being made to *cae()*, a constructor and trainer for a Contractive Autoencoder [27]. This function takes arguments from *userscript.py*. A data file is read and converted to a numpy ndarray with the function *np.loadtxt()*. Preprocessing is then done inside of *cae()*. This may include altering the shape of the ndarray and other needed corrections. While still inside *cae()*, multiple calls are made to *generic.py* to build and train the network. Anomaly detection is also done inside *cae()* with a call to *analysis.py*. *cae()* returns anomaly results along with the fit-subtracted data to *userscript.py* so that the user can make use of the results.

Generic.py contains functions for constructing networks, choosing cost functions and optimization routines, and training the network. The network constructor *nfBuild()* is very generic as there are no predefined networks. *nfBuild()* is responsible for feeding TensorFlow™information regarding the shape of the network along with information about what type of distribution the weights/bias' will be initialized with and their range. *nfCost()* and *nfOpti()* contain routines for constructing different types of cost and optimization functions respectively. *nfTrain* is used to initialize all TensorFlow™variables, start the actual TensorFlow™session, and train the neural network. The type of cost function being used is the Mean Squared Error [28]. The Frobenius norm [27] is added to this term using *nfReg()*, which is what defines this autoencoder as contractive.

Analysis.py contains routines for data analysis. An important function in this script, *zoutlier()*, is used by *cae()* in *neuralframe.py*. *zoutlier()*, when given fit-subtracted data along with arguments for tail number and significant z-score, will detect anomalies in a data set using z-score=(x-mean)/(standard deviation). A common value that indicates a significant deviation from a normal distribution is a z-score of 3.0.

Figures 2,3, and 4 show results of finding anomalies using this TensorFlow™approach. In the first case we see how the system is able to detect two quench events. In the second case it is used to detect anomalous server activity.

NUPIC

The Numenta Platform for Intelligent Computing (NuPIC) is an implementation of Hierarchical Temporal Memory (HTM), a memory-prediction framework with the goal of creating general intelligence by mimicking the learning processes of the human neocortex. It employs Sparse Distributed Representations (SDRs) along with a network of Spatial Pooler and Temporal Memory regions to learn patterns in temporal data. One typical use-case for an HTM system is anomaly detection; HTM will learn

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUCPA03

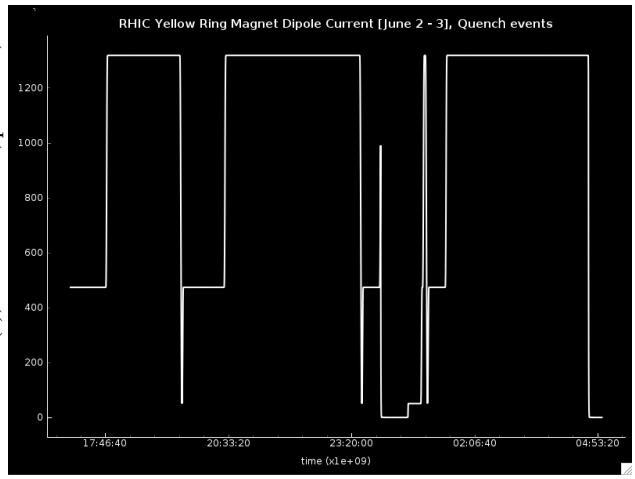

Figure 2: RHIC Yellow Ring Main Dipole Current, with two quench events.

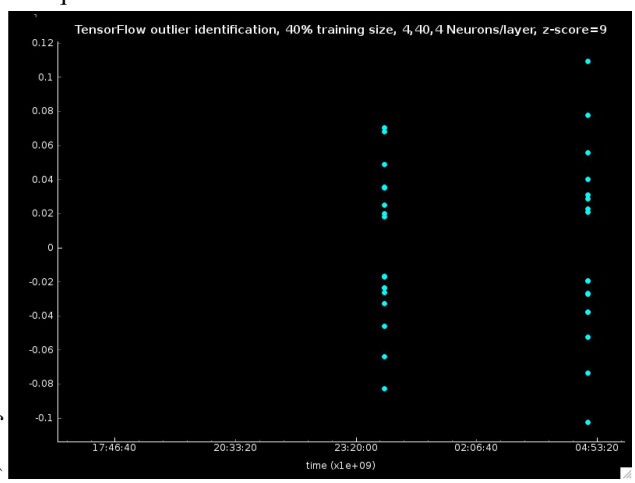

Figure 3: TensorFlow™anomalies identifying the quench events.

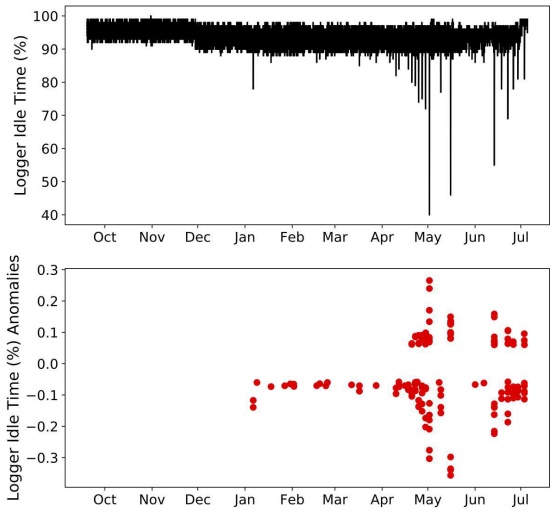

Figure 4: TensorFlow™Analysis of server activity over nine months. Almost every transient dip gets flagged as an anomaly.

what is "normal" for a signal and flag significant deviations as anomalies.

The HTM paradigm postulates that the SDR is the natural data type of the human brain. SDRs can be visualized as sparse arrays filled with 1's and 0's, wherein less than 20% of the cells are 1's. SDRs provide a huge storage capacity for encoding input data, as well as a large tolerance for noise. In a simple HTM network, the input space contains input data encoded as binary arrays, which is connected to a Spatial Pooler. The Spatial Pooler is a stack of SDRs, in which each "column" of cells connects to a subsample of cells in the input space. These connections, or synapses, are modified during the learning process to pool spatial patterns in the input. Each column will learn to be "active" when it detects its specific pooled pattern in the input.

The Temporal Memory region learns sequences of the active columns in the Spatial Pooler. While the Spatial Pooler learns to identify patterns, Temporal Memory learns the sequential contexts of those patterns. For example, suppose the Spatial Pooler has learned the patterns A-D, and the input signal varies between one of two sequences: ABCD or DBCA. Temporal Memory learns that C can occur in two different contexts and subsequently what pattern should follow C in each context. This is achieved by using the individual cells of the Spatial Pooler's SDR stack to represent the contexts. One combination of cells across specific columns will represent C in the AB context while a different combination across the same columns will represent C in the DB context. A specific set of active columns represents C, but the specific active cells in those columns determines the context.

While Spatial Pooling relies on proximal synapses from cells in lower levels of the network hierarchy (e.g. input/sensor regions), Temporal Memory relies on distal synapses from cells in the same hierarchical level. During learning, these synapses are modified to strongly connect context cells to their sequential successors. In the example mentioned, the cells representing B in the Spatial Pooler will create strong distal synapses to the cells representing C. Thus, when B occurs those B cells will pre-activate C cells, in effect predicting that C should occur next.

Anomaly detection compares the predicted cells against the next set of active cells. The non-overlap, or number of predicted cells that do not become active on the next step, is used to calculate an anomaly score (little overlap means poor prediction and implies anomalous behavior). These anomaly scores are collected over time to build a distribution. Once a sufficient number are collected, new anomaly scores are compared against the distribution; if the score is well outside the norm, the observation is said to have a high anomaly likelihood and flagged as anomalous (the standard threshold is >0.9999 on a scale of 0 to 1). This process of anomaly detection implies that subsequent similar anoma-

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUCPA03

lous events will not be flagged as anomalies because the incorporation of the anomaly scores into the distribution will eventually define a new norm (i.e. if you see the same kind of anomaly multiple times, by definition it becomes normal).

Getting started with NuPIC can be an involved process, but Numenta provides three different ways of using their code: the Online Prediction Framework (OPF), the Network API, and the source code. The OPF is the easiest to use but the least customizable, while the source code is the opposite. The Network API provides a middle ground of customizability and ease of use. We chose to work with the OPF for our exploration.

To use the OPF, one has to specify many, many different parameters to setup the different regions (e.g. number of cells per column, the rate at which synapses grow and die, etc). Fortunately, Numenta provides a straightforward tool that "swarms" the parameter space and determines what parameters are best for modeling the input data. One can choose swarms of different sizes, the idea being that larger swarms will find better parameters than smaller swarms. In our experience, we found no huge improvements between the two. Additionally, since we were specifically interested in anomaly detection, we used a set of model parameters provided by NuPIC that was reported to work well detecting temporal anomalies from streams of scalar data.

To explore how HTM performs, we applied it to two cases: learning an electromagnet's current during a hysteresis ramp, and learning a data logger server's activity over time. In both cases, we were most interested in evaluating HTM's ability to predict anomalies.

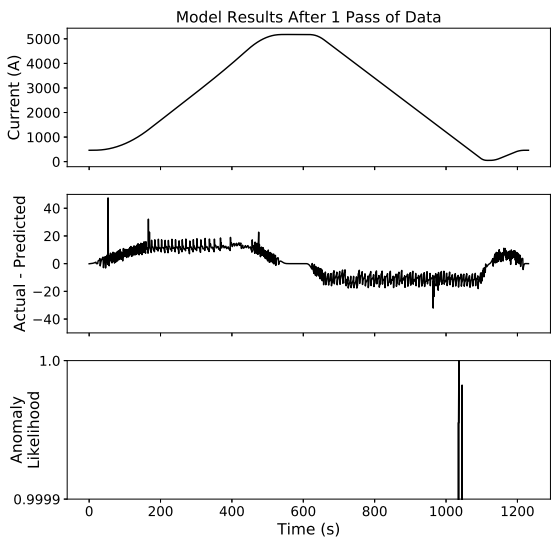

Figure 5: After one pass on magnet data, the prediction lags the actual value.

Using the OPF, we trained a network to learn how a electromagnet's current behaves during a hysteresis ramp. The input data included seconds elapsed from the beginning of the ramp and the magnet's current at that time. Because the data set is so small, we ran multiple passes of the data through the network. At the end of each pass, we reset the Temporal Memory sequence states to prevent the system learning that hysteresis ramps are followed by hysteresis ramps. As seen in Fig. 5, after one pass of the data the prediction is poor (the prediction lags the actual value). After 60 passes of the data (Fig. 6), the prediction is much better (the prediction now generally leads the actual value). In addition, no anomalies are detected using the standard threshold of 0.9999, as expected.

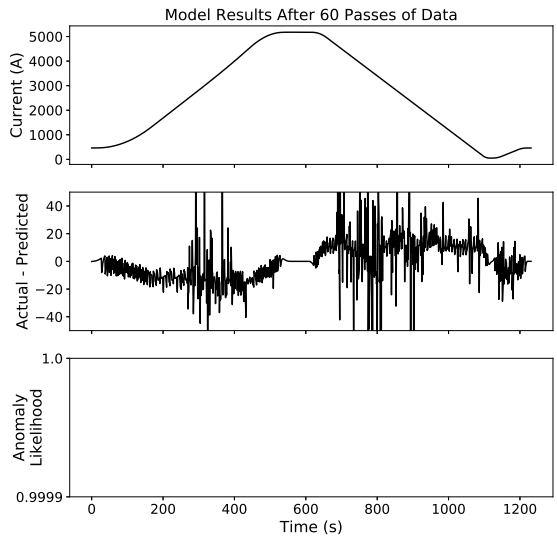

Figure 6: After 60 passes of the data, the prediction now leads the actual value.

To simulate an anomaly, we presented the network with a hysteresis ramp during which the magnet quenches, causing an interlock which drops the current to zero (aka a QLI). Figure 7 shows this registered as an anomaly. Post-anomaly, the prediction quickly began lagging the actual current as the system began learning the new norm. Interestingly, even though there were several (10) significant prediction spikes, the system did not report these as anomalous.

In our second application, we trained a network on a data logger server's activity over the course of nine months. Here we were not concerned with prediction; knowing what the idle time of a logger will be is not useful for us. What is useful is knowing if the server is experiencing anomalous behavior. To get this information, we trained a network using model parameters that NuPIC provides for general anomaly detection on streams of scalar data. Unlike the magnet current application, we did not run the data multiple times through the network.

Figure 8 shows that the network did very well flagging as anomalies almost every transient dip in the signal. Note that the decrease in the DC component just prior to Decem-

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUCPA03

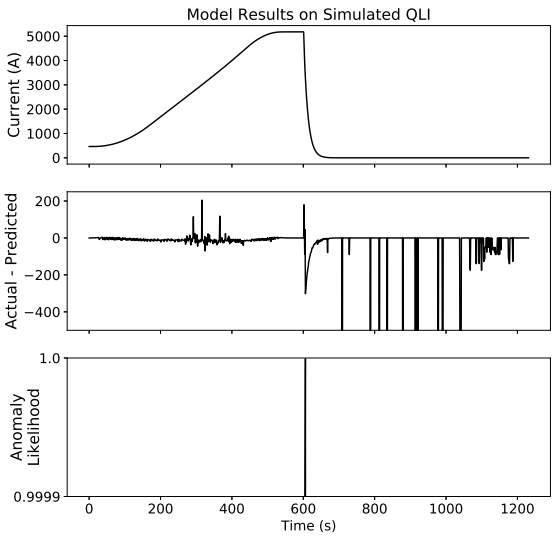

Figure 7: Simulated anomaly, magnet ramp with a quench.

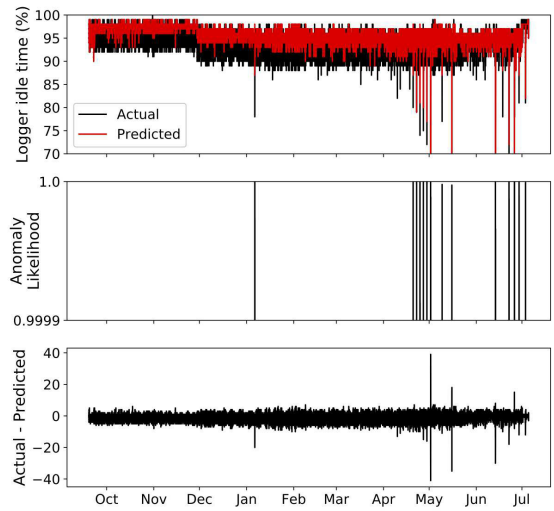

Figure 8: Analysis of server activity over nine months. Almost every transient dip gets flagged as an anomaly.

ber was not detected, nor was the increase just after July. These changes may be simply too subtle for the model to detect, and different model parameters may yield better detection. Regardless, this performance is satisfactory as these changes are not of the type needing to be addressed.

Overall, the NuPIC implementation is fairly easy to use and produces acceptable results using pre-packaged models with minimal adjustments. More investigation is needed to see if similar performance can be achieved with other signals in the accelerator physics domain (e.g. orbit position evolution, emittance growth).

FUTURE WORK

Our main focus remains on exploring how we can use machine learning techniques with different kinds of accelerator data sets. However, we are discussing tools that can be built for operations and for the control systems. For operations, we believe these techniques can be used to improve data mining and may be useful in building early warning systems. There is still more research needed on the practicality of such systems.

REFERENCES

[1] P. Bruce and A. Bruce, *Practical Statistics for Data Scientists*, Sebastopol, CA, USA, O'Reilly Media, Inc., 2017

[2] A. Munoz, `https://www.cims.nyu.edu/~munoz/files/ml_optimization.pdf`

[3] T. Higo, H. Shoaee, and J. E. Spencer, in *Proc. PAC'87*, pp.701-703

[4] J. E. Spencer, in *Proc. PAC'89*, pp.1642-1644

[5] D. Nguyen, M. Lee, R. Sass, H. Shoaee, in *Proc. PAC'91*, pp. 1437-1439

[6] A. L. Edelen, *et al.*, in *Proc. IPAC'15*, MOPWI028

[7] A. Rezaeizadeh, T. Schilcher, and R. Smith, in *Proc. IPAC'15*, MOPTY060

[8] S. Kwon, *et al.*, in *Proc. LINAC'00*, TUC13

[9] M. Laverty and K. Fong, in *Proc. LINAC'16*, TUPLR009

[10] E. Meier, Y. .E. Tan, and G. S. LeBlanc, in *Proc. IPAC'12*, WEPPP057

[11] E. Bozoki and A. Friedman, in *Proc. EPAC'94*, pp.1589-1591

[12] JACoW, `http://www.jacow.org`

[13] L. M. Kegelmeyer, *et al.*, in *Proc. ICALEPCS'13*, THMIB04

[14] R. Flora, *et al.*, in *Proc. PAC'95*, pp.2172-2174

[15] E. Meier, *et al.*, in *Proc. PAC'09*, TU5RFP050

[16] Y. Kijima, *et al.*, in *Proc. EPAC'92*, pp.1155-1157

[17] A. Gokhale, A. Sharma, and B. P. Dubey, in *Proc. APAC'07*, THPMA073

[18] D. Schirmer, *et al.*, in *Proc. EPAC'06*, WEPCH013

[19] Y. B. Kong, *et al.*, in *Proc. Cyclotrons'16*, TUP19

[20] TensorFlow™, `https://www.tensorflow.org`

[21] J. Hawkins and S. Blakeslee, *On Intelligence*, New York, NY, USA, Times Books, 2007

[22] Numenta.org, `https://numenta.org`

[23] Theano, `http://deeplearning.net/software/theano/`

[24] Keras, `https://keras.io`

[25] Eigen3, `http://eigen.tuxfamily.org/`

[26] NVIDIA™cuDNN, `https://developer.nvidia.com/cudnn`

[27] S. Rifai, *et al.*, in *Proc. 28th International Conference on Machine Learning, Bellevue, WA, USA, 2011*

[28] MSE, `https://en.wikipedia.org/wiki/Mean_squared_error`