Proposal for Collaboration between Brocade and the Poznan Supercomputing and Network Center (PSNC)

David Meyer, Brian Levy and Andrew Coward[1]

[1]{dmm,blevy,acoward}@brocade.com

January 25, 2016

Abstract

Recent advances in machine learning, coupled with the onslaught of data being collected from a wide variety of sensors has rekindled interest in using machine learning as a method to uncover hidden structure in these ever growing data sets. In particular, advances in the design of multi-layer deep artificial neural networks (DNNs) combined with effective approaches for training DNNs has opened up the opportunity to use DNNs for novel applications ranging from speech recognition and generation to self-driving vehicles. DNNs are multiple-layer architectures (deep architectures) which extract inherent features in data and discover important hidden structure in diverse data sets. Given that the factors contributing to traffic flow, congestion, and queuing delay in a data center result from the non-obvious interaction of complex factors, DNNs represent a novel and powerful method for learning how these factors interact and for predicting a wide variety of complex network behaviors. These same DNNs can be used to detect various Advanced Persistent Threats in the security domain. This document introduces a specific form of DNN, and provides a framework for an ongoing joint project between Brocade and the Poznan Supercomputing and Network Center.

1 Introduction

Recent advances in machine learning, coupled with the onslaught of data being collected from a wide variety of sensors has rekindled interest in using machine learning as a method to uncover hidden structure in these ever growing data sets[1]. In particular, advances in the design of multi-layer deep artificial neural networks (DNNs) combined with effective approaches for training DNNs has opened up the opportunity to use DNNs for novel applications ranging from speech recognition and generation to self-driving vehicles. DNNs are multiple-layer architectures (deep architectures) which extract inherent features in data

[1]This phenomena is evidenced by the explosive growth in the number of "data analytics" startups [5].

and discover important hidden structure in diverse data sets. Given that the factors contributing to traffic flow, congestion, and queuing delay in a data center result from the non-obvious interaction of complex factors, DNNs represent a novel and powerful method for learning how these factors interact and for predicting a wide variety of complex network behaviors.

Neural networks had traditionally been trained with an algorithm called back propagation [9], which is so named because the algorithm propagates the error in the neural network's estimate *backward* from the output layer towards the input layer. Back propagation also requires labeled data sets; these *training* sets have elements of the form $(\mathbf{x}^{(i)}, \mathbf{t}^{(i)})$, where the $\mathbf{x}^{(i)}$ are the inputs and the $\mathbf{t}^{(i)}$ are the targets (the targets tell what the data is, for example, "cat"). The DNN computes an output value, sometimes called (largely for historical reasons) the hypothesis $h_\theta(\mathbf{x}^{(i)})$. $h_\theta(\mathbf{x}^{(i)})$ is then compared to the target $\mathbf{t}^{(i)}$ and the difference $h_\theta(\mathbf{x}^{(i)}) - \mathbf{t}^{(i)}$ is taken as an estimate of the model's error. This error is then "back propagated" (with the help of additional algorithmic machinery) down the DNN from output to input, adjusting the model parameters along the way. Back propagation is an instance of a *supervised learning* algorithm since it requires labeled data. Training algorithms that use unlabeled data are referred to as *unsupervised learning* algorithms.

There were, however, several weaknesses with the back propagation algorithm which essentially limited the utility of DNNs. These included the fact that back propagation really didn't work well in deep networks (for technical reasons relating to the computation of what are called gradients) and the tendency for the algorithm to fall into poor local minima when the DNN was initialized with random weights[2]. The requirement for labeled data sets was also a problem since most data is unlabeled. These two problems with DNNs, the need for labeled training sets and ineffective training via back propagation, were largely overcome by the groundbreaking work of Geoffrey Hinton and his colleagues in 2006 [3]. Hinton's breakthrough was to show that unsupervised, greedy, layerwise training of DNNs was effective in overcoming the problems with traditional back propagation training. This is discussed in more detail in Section 2.2.

2 So What Is An AutoEncoder?

In this section we introduce our basic methodology which is based on a deep-learning based prediction model. A stacked *autoencoder* [10] model is used to learn generic features, and as such is part of a *representation learning* system. This section reviews basic autoencoder and stacked autoencoder technology.

[2]The problem of non-optimal minima is a property of non-convex optimization, where local minima aren't necessarily global minima when the some of the DNNs parameters were initialized with random values [1].

2.1 The Basic Autoencoder

The traditional autoencoder is an artificial neural network that attempts to reproduce its input, i.e., the target output is the input. More formally (and following the notation of [7]), an autoencoder takes an input vector $\mathbf{x} \in [0,1]^d$ and maps it to a hidden representation $\mathbf{y} \in [0,1]^d$ through a deterministic mapping $\mathbf{y} = f_\theta(\mathbf{x}) = s(\mathbf{W}\mathbf{x} + b)$, parameterized by $\theta = \{\mathbf{W}, \mathbf{b}\}$. $\mathbf{W}$ is a $d' \times d$ weight matrix, $\mathbf{b}$ is a bias vector and s is the *sigmoid*[3] activation function, $s(\mathbf{x}) = \frac{1}{1+e^{-\mathbf{x}}}$. The hidden representation $\mathbf{y}$, sometimes called the *latent* representation, is then mapped back to a reconstructed vector $\mathbf{z} \in [0,1]^d$, where $\mathbf{z} = g_{\theta\prime}(\mathbf{y}) = s(\mathbf{W}'\mathbf{y} + \mathbf{b}')$, with $\theta' = \{\mathbf{W}', \mathbf{b}'\}$. This scenario is depicted in cartoon form in Figure 1. Thus each training $\mathbf{x}^{(i)}$ is thus mapped to a corresponding $\mathbf{y}^{(i)}$ and a reconstruction (of $\mathbf{x}^{(i)}$) $\mathbf{z}^{(i)}$. Finally, note that the weight matrix $\mathbf{W}'$ may optionally be constrained by $\mathbf{W} = \mathbf{W}^T$, in which case the autoencoder is said to have *tied* weights. Each training example $\mathbf{x}^{(i)}$ is thus mapped to a corresponding $\mathbf{y}^{(i)}$ which is then mapped to a reconstruction $\mathbf{z}^{(i)}$ such that $\mathbf{z}^{(i)} \approx \mathbf{x}^{(i)}$.

The basic idea here is that the autoencoder is constructed in such a way that the mapping $\mathbf{x}^{(i)} \mapsto \mathbf{y}^{(i)}$ reveals essential structure in the input vector $\mathbf{x}^{(i)}$ that is not otherwise obvious. For example, if the autoencoder has fewer hidden units than input units it must find a representation that essentially compresses the input in such a way that it can be efficiently reconstructed. The compressed representation has lower dimensionality than the input and is represents an *abstraction* of the input. In the case of image recognition $\mathbf{x}^{(i)}$ might be an image (pixels) while $\mathbf{y}^{(i)}$ might consist of edges in various orientations.

The parameters θ and θ' of the model are optimized to minimize the *average reconstruction error* as shown in Equation 1:

$$\begin{aligned} \theta^*, \theta'^* &= \arg\min_{\theta,\theta'} \frac{1}{n} \sum_{i=1}^{n} L\left(\mathbf{x}^{(i)}, \mathbf{z}^{(i)}\right) \\ &= \arg\min_{\theta,\theta'} \frac{1}{n} \sum_{i=1}^{n} L\left(\mathbf{x}^{(i)}, g_{\theta'}(f_\theta(\mathbf{x}^{(i)}))\right) \end{aligned} \tag{1}$$

Here L is a loss function such as the traditional squared error $L(\mathbf{x}, \mathbf{z}) = \| \mathbf{x} - \mathbf{z} \|_2^2$. Note that if $\mathbf{x}$ and $\mathbf{z}$ can be interpreted as either bit vectors or vectors or probabilities (i.e., they are Bernoulli probability vectors), then the *reconstruction cross-entropy*, as defined in Equation 2, can be used.

$$\begin{aligned} L_{\mathcal{H}}(\mathbf{x}, \mathbf{z}) &= \mathcal{H}(\mathcal{B}_{\mathbf{x}}, \mathcal{B}_{\mathbf{z}}) \\ &= -\sum_{k=1}^{d} [\mathbf{x}_k \log \mathbf{z}_k + (1 - \mathbf{x}_k) \log(1 - \mathbf{z}_k)] \end{aligned} \tag{2}$$

[3]Note that the sigmoid activation function s "squashes" its input into the range $[0,1]$

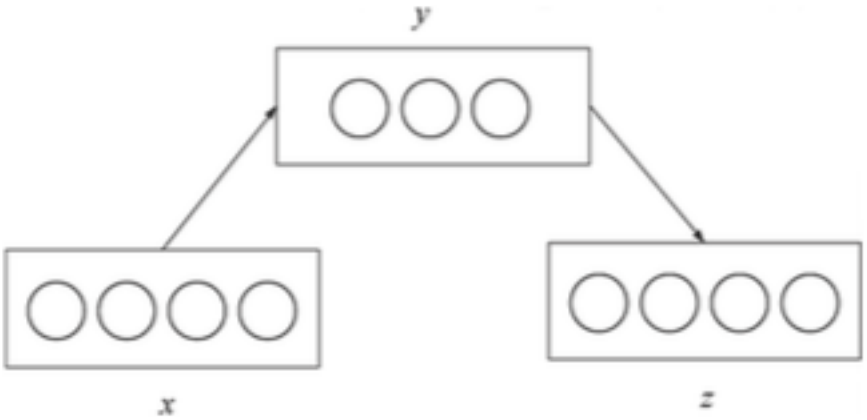

Figure 1: Classic Autoencoder

More generally, this methodology casts learning as *optimization* using Empirical Risk Minimization [11]. The Empirical Risk $\hat{R}$ is defined as

$$\hat{R}(f_\theta, D_n) = \sum_{i=1}^{n} L\left(f_\theta(\mathbf{x}^{(i)}), \mathbf{z}^{(i)}\right) \tag{3}$$

In some cases it may be necessary to induce a preference for some values of the parameters to avoid overfitting [2]. To avoid overfitting we can define a *Regularized Empirical Risk*, where the regularization imposes a degree of sparseness on the derived encodings. Suppose we have a training set $D_n = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(n)}\}$. Then the *Regularized Empirical Risk* is defined as follows:

$$\hat{R}_\lambda(f_\lambda, D_n) = \left(\sum_{i=1}^{D_n} L\left(f_\theta(\mathbf{x}^{(i)}), \mathbf{z}^{(i)}\right)\right) + \lambda\Omega(\theta) \tag{4}$$

where Ω penalizes more or less certain parameter values and $\lambda \geq 0$ controls the amount of regularization. Regularizers generally perform two basic functions: First, a regularizer can keep the autoencoder from learning the identity function (in which case the autoencoder would be useless). The second function that a regularizer provides is that it enforces certain properties on the weights. For example, using the L1-norm[4] as a regularizer, where the L1-norm $\| \mathbf{x} \|_1 = \sum_{i}^{n} | x_i |$, enforces spareness on the vector $\boldsymbol{x}$. This is because in order to

[4]The L1-norm is sometimes called the *taxicab* norm

minimize $\| \mathbf{x} \|_1$, some (or most) of the x_i have to be zero. On the other hand, using the L2-norm[5] is defined as follows: Let $\mathbf{x}$ be an n-dimensional vector in $\mathbb{R}^n$. Then the L2-norm $\| \mathbf{x} \|_2 = \left(\sum_{i=1}^{n} | x_i |^2 \right)^{\frac{1}{2}}$, which forces the x_i (i.e., the weights) to be small [6].

Thus *learning* in this setting amounts to finding optimal parameters θ^* that satisfy $\theta^* = \arg\min_\theta \hat{R}(f_\theta, D_n)$. However, one of the risks of using autoencoders, as mentioned above, is that the autoencoder can potentially learn the identity function and thereby not extract useful features from the input. This problem is especially acute if the size of the hidden layer has the same number of units as the input layer (or more). One way to train an autoencoders that has more hidden units than input units to learn useful features is to impose sparsity constraints such as the regularizations described above on the minimization problem [6] described in Equation 1. The effect is to force the representations found by the hidden layers to be sparse. Such an autoencoder is referred to as a *sparse autoencoder*. A popular sparsity constraint is based on the Kullback-Leibler divergence [8]. The Kullback-Leibler Divergence $\mathcal{D}_{KL}(P \parallel Q)$ can be thought of as a measure of the information lost when probability distribution Q is used to approximate P. For our purposes we define $\mathcal{D}_{KL}(\rho \parallel \hat{\rho})$ as follows

$$\mathcal{D}_{KL}(\rho \parallel \hat{\rho}) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \hat{\rho}_j} \tag{5}$$

where ρ is a sparsity parameter who's value is close to zero and $\hat{\rho}_j = \frac{1}{n} \sum_{i=1}^{n} s(\mathbf{W}\mathbf{x}_j^{(i)} + \mathbf{b})$, the average activation of hidden unit j. Putting this together with Equation 1 we get the following optimization problem:

$$\arg\min_{\theta,\theta'} \frac{1}{n} \sum_{i=1}^{n} L\left(\mathbf{x}^{(i)}, g_{\theta'}(f_\theta(\mathbf{x}^{(i)}))\right) + \gamma \sum_{j=1}^{H_D} \mathcal{D}_{KL}(\rho \parallel \hat{\rho}) \tag{6}$$

where H_D is th number of hidden units and γ is a sparsity weighting term. Kullback-Leibler Divergence has the nice property that $\mathcal{D}_{KL}(\rho \parallel \hat{\rho}) = 0$ if $\rho = \hat{\rho}$ (and so weighting these cases improves the sparsity of the encoding; this is the job of the γ parameter).

2.2 Stacked Autoencoders

Stacked Autoencoders (SAEs) are, as the name implies, a stack of single-level autoencoders; hence the SAE is a deep learning model [**?**]. SAEs use the autoencoders described above as building blocks to create a deep network [10]. While deep architectures can be more expressive and can extract more sophisticated features from input data, until relatively

[5] The L2-norm is sometimes called the *Euclidean* norm

[6] In general, the p-norm is defined to be $\| \mathbf{x} \|_p = \left(\sum_{i=1}^{n} | x_i |^p \right)^{\frac{1}{p}}$.

recently deep networks were thought to be too difficult to train and as such of limited utility. As mentioned above, the breakthrough came when Geoffrey Hinton and his colleagues showed how fast, layerwise greedy and unsupervised algorithms can be used to initialize a slower algorithm that fine tunes the learned weights and provides very good results on deep networks [3]. This result revitalized the machine learning community and deep architectures been successfully applied to a wide variety of classification and prediction problems [4].

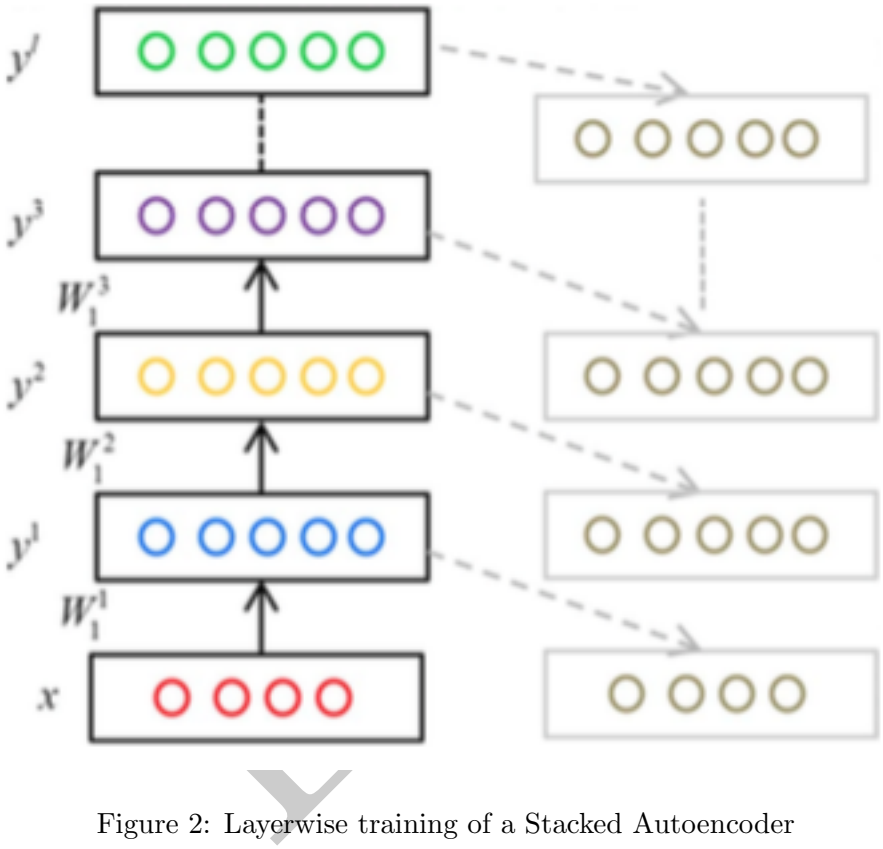

Figure 2: Layerwise training of a Stacked Autoencoder

The basic idea behind layerwise training is shown in Figure 2. The idea here is to train each layer as described in Equation 6. After a layer is trained, the autoencoder output layer is discarded and the features (the $\mathbf{y}^{(i)}$) are used as the input to the next layer. Hence the training is greedy and layerwise. Finally, the last layer in the network, usually either a linear regression layer (if the output values are continuous) or logistic regression layer (if the output is discrete). The final step is to fine tune the network in a supervised fashion using the back propagation algorithm [9]. The key idea here is that this approach can be used to *automatically* learn the important features in network datasets such as might be

collected with Netflow or IPFIX.

3 Proposed Collaboration

The proposed collaboration here is to build an anomaly detection system[7] based on the workflow shown in Figure 3. As showing in the figure, such a system is comprised of a data collection/ingestion phase, an extract, transform and load (ETL) phases, a model development and evaluation phase, followed by feedback from both evaluation by both technical analysts and the target customer.

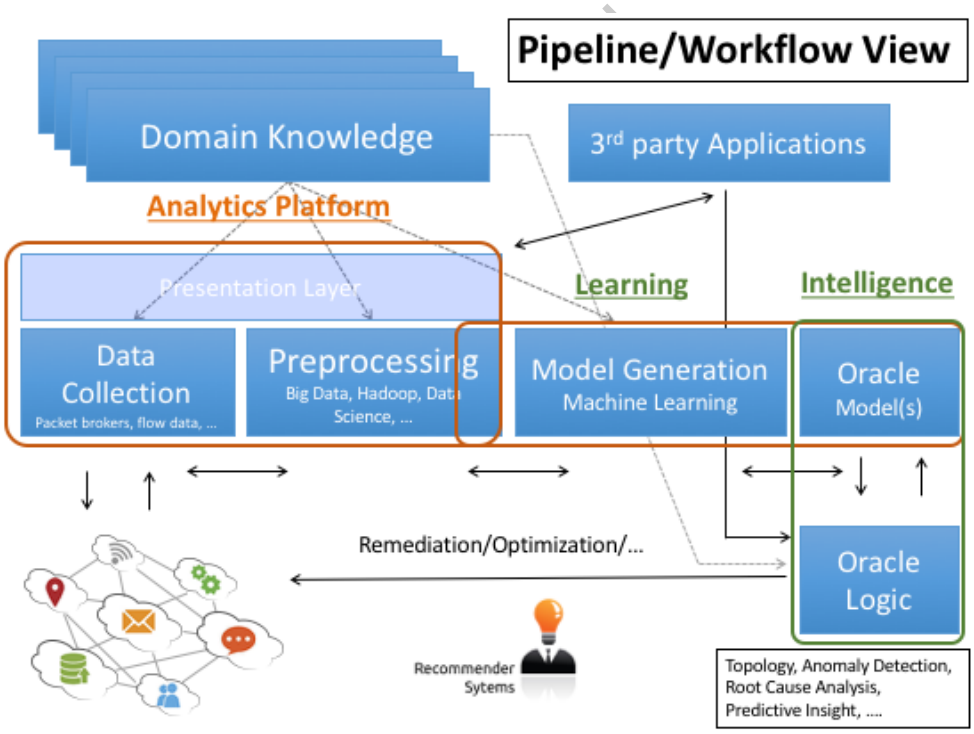

Figure 3: Machine Learning Workflow for Network Environments

[7] "Anomaly" here is being used in the most general way, i.e., anomaly detection as a classification problem. Anomalies can be, among other things, deviations from normal traffic patterns, routing updates, configurations, etc.

One of the novel aspects of the proposed work is the idea of using DNNs to automatically detect features of interest in network data and the understanding of such features for use in classification systems such as described above. Of course, there are many other "open" issues in the construction of such a system, such as how to make the system "on-line" so that incremental training is possible within reasonable computational bounds.

4 Milestones and Deliverables

- M1: Meet with PSNC to vet the proposal
- M2: Build data collection infrastructure
- M3: Work with PSNC data scientists to develop new models for network data[8]
- M4: Deploy new models on targeted PSNC network segments
- M5: Feedback and refinement

5 Acknowledgements

[8] Does the PSNC have data scientists?

References

[1] Mung Chiang. Nonconvex optimization for communication systems. Technical report, Princeton University, 2011.

[2] Tom Dietterich. Overfitting and undercomputing in machine learning overfitting and undercomputing in machine learning. *ACM Comuting Surveys*, 1995.

[3] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 2006.

[4] http://deeplearning.net.

[5] https://angel.co/big-data analytics.

[6] R. B. Palm. Prediction as a candidate for learning deep hierarchical models of data,. *Technical Univ. Denmark, Palm, Denmark*, 2012.

[7] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol . Extracting and composing robust features with denoising autoencoders. Technical Report 1316, Universite de Montreal, 2008.

[8] Fernando Perez-Cruz. Kullback-leibler divergence estimation of continuous distributions. Technical report, Princeton University, 2011.

[9] B Widrow. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 2002.

[10] Y. Bengio, P. Lamblin, D. Popovici and H. Larochelle. Greedy layerwise training of deep networks. *Proc. Adv. NIPS*, pages 153–160, 2007.

[11] Yoshua Bengio, Aaron Courville and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 2013.