Can Congestion in Data Center Networks Be Predicted As A Function Of Time Of Day?

David Meyer

dmm@brocade.com

Abstract

The recent growth in data center structure, function and scale has brought a myriad of new challenges to network operators, including virtualization, development of new services, and scaling data center capacity in both overlay and underlay networks. These challenges share a common characteristic: network congestion and hence network delay within a data center has a strongly negative impact on user quality of experience and operator efficiency. Hence congestion and the closely related problem of network delay are among the the key concerns for data center and service operators. Previous work [**?**] puts a finer point on this: a user's quality of experience can be badly affected when even a single flow suffers from a large latency.

The largest part of network delay in today's data centers comes from the queueing delay at switch interfaces. Since the propagation delay within a data center should be almost negligible (in the ideal case a 100 meters of network cabling between two nodes adds only 0.5 μs of propagation delay, while a single 1500 byte packet queued at a 10Gbps port already costs 1.2 μs), much effort has been put into reducing congestion and hence queuing delay at switch interfaces. For example, DCTCP [**?**] uses ECN marking to slow down flows before the relevant queues become full., while HULL [**?**] takes a further step and gives up a little bandwidth to have even lower latency than DCTCP. Our work is also motivated to reduce the queueing delay in data centers, but here we focus on predicting nascent congestion on important data center links as a method to minimize queuing delay. In this work we demonstrate a novel use of artificial neural networks to predict nascent congestion (and hence queuing delay) in data center networks, which will allow operators to further optimize both operational and capital expenditures. Our longer term goal is to build general framework for predicting various data center network parameters that effect both operator efficiency and user quality of experience.

1 Introduction

The recent growth in data center structure, function and scale has brought a myriad of new challenges to network operators, including virtualization, development of new services,

and scaling data center capacity in both overlay and underlay networks. These challenges share a common characteristic: network congestion and hence network delay within a data center has a strongly negative impact on user quality of experience and operator efficiency. Hence congestion and the closely related problem of network delay are among the the key concerns for data center and service operators. Previous work [**?**] puts a finer point on this: a user's quality of experience can be badly affected when even a single flow suffers from a large latency.

The largest part of network delay in today's data centers comes from the queueing delay at switch interfaces. Since the propagation delay within a data center should be almost negligible (in the ideal case a 100 meters of network cabling between two nodes adds only 0.5 μs of propagation delay, while a single 1500 byte packet queued at a 10Gbps port already costs 1.2 μs), much effort has been put into reducing congestion and hence queuing delay at switch interfaces. For example, DCTCP [**?**] uses ECN marking to slow down flows before the relevant queues become full., while HULL [**?**] takes a further step and gives up a little bandwidth to have even lower latency than DCTCP. Our work is also motivated to reduce the queueing delay in data centers, but here we focus on predicting nascent congestion on important data center links as a method to minimize queuing delay. In this work we demonstrate a novel use of artificial neural networks to predict nascent congestion (and hence queuing delay) in data center networks, which will allow operators to further optimize both operational and capital expenditures. Our longer term goal is to build general framework for predicting various data center network parameters that effect both operator efficiency and user quality of experience.

Traffic flow and hence congestion prediction in data centers depends heavily on both historical and real-time data collected from various sensor sources, including interface counters, cpu and memory utilization provides, as well as various sensor networks and social media, to name a few. Note in addition to traditional data sources, emerging sensor technologies (e.g., Internet of Things), traffic data are exploding and as such new techniques for exploring the structure of this data are needed. Note also that we consider traditional network data sources to be *sensors*; these includes data sources like switch and router telemetry such as interface, frame, and error counters as well as switch CPU and memory utilization statistics.

1.1 Why Artificial Neural Networks?

Recent advances in machine learning, coupled with the onslaught of data being collected from various types of network telemetry, Internet of Things, social networks, and the web has rekindled interest in using machine learning as a method to uncover hidden structure in these vast 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

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

up the opportunity to use DNNs for novel applications. DNNs have recently been successfully applied to classification tasks, natural language processing, dimensionality reduction, object detection, motion modeling, and many other tasks. Deep learning algorithms use multiple-layer architectures (deep architectures) to extract inherent features in data from the lowest level to the highest level, and they can discover important hidden structure in diverse data sets. Given that the factors contributing to traffic flow, congestion, and queuing delay in a data center can result from the non-obvious interaction complex factors, deep learning algorithms represent a novel method for learning these interactions and features in the data center context.

That said, until recently DNNs were thought to be ineffective in uncovering latent structure data sets, largely due to the deficiencies in training algorithms. Neural networks had typically been trained with an algorithm called back propagation [**?**], which is so named since the algorithm propagates the error in the DNNs estimates backwards from the output layer of the DNN towards the input layer. In addition, back propagation 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 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 is an estimate of the model's error. This error is then "back propagated" down the DNN from output to input, adjusting the different parameters of the model along the way.

There were several weaknesses with the back propagation algorithm which essentially limited the utility of DNNs. These included the reduced significance of the error terms as you get further away from the output layer (basically the *gradient* gets smaller as you move down the DNN from output to input layers), and the tendency of the algorithm to fall into poor local minima when the DNN was initialized with random weights[2]. In addition, back propagation required labeled data sets for training; this form of training is generally referred to as *supervised* learning, whereas training models with unlabeled data sets is referred to as *unsupervised* learning. These two problems with DNNs, namely, 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 [**?**]. Hinton's breakthrough involved, among other things, methods for training DNNs. In particular, Hinton and his colleagues showed 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 3.2.

In this work we introduce the novel use of a specific DNN, the Stacked Autoencoder [**?**], as a platform for predicting parameters of interest for data center and service operators, starting with congestion of important links in the data center. To demonstrate the technique, we attack the initial problem from the perspective of modeling congestion as a

[2]This is really a problem with non-convex optimization, and doesn't occur if the objective function (of the optimization problem) is convex [**?**].

function of time of day. We call this problem the Spatial-Temporal Prediction of Traffic Flows in Data Centers problem. The remainder of this paper is organized as follows: Section 2 describes the Spatial-Temporal Prediction of Traffic Flows Problem in a data center network. Section 3 describes our methodology and reviews both autoencoder and stacked autoencoder technology. Section 4 outlines our data sets, evaluation metrics and results. Finally, Section 5 discusses conclusions and future work.

2 The Spatial-Temporal Prediction of Traffic Flow in Data Centers Problem

The Spatial-Temporal Prediction of Traffic Flow in Data Centers Problem can be stated as follows: Let X_i^t denote the the observed traffic flow during the t^{th} time interval at the i^{th} observation location[3]. Given a sequence $\{X_i^t\}$ of observed traffic flow data , $i = 1, 2, ..., m$ and $t = 1, 2, ..., T$, the problem is to predict the traffic flow (and hence potential congestion) at the time $(t + \Delta)$ for some prediction horizon Δ. Note that we also want to consider the temporal relationships inherent in traffic flows, so to predict the traffic flow at time interval t, we also use the traffic flow data at previous time intervals, i.e., $X^{t-1}, X^{t-2}, ..., X^{t-r}$ for some value of r (its not clear how far back in time you need to go to get valuable predictions). Thus while the the exact network topology is a hyper-parameter in this study, the model described here incorporates both temporal and spatial correlations.

3 Methodology

In this section we introduce our basic methodology which is based on a deep-learning based prediction model. A stacked *autoencoder* [**?**] model is used to learn generic traffic flow features. This section reviews basic autoencoder and stacked autoencoder technology.

3.1 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 [**?**]), 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*[4] 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

[3] An observation location can be an interface counter, switch cpu load or memory utilization, or other relevant sensor value; here we consider interface counters (etc) to be *sensors*.

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

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 $\mathbf{x}^{(i)}$ is thus mapped to a corresponding $\mathbf{y}^{(i)}$ and a reconstruction (of $\mathbf{x}^{(i)}$) $\mathbf{z}^{(i)}$. The idea here is that the autoencoder is constructed in such a way that the mapping $\mathbf{x}^{(i)} \rightarrow \mathbf{y}^{(i)}$ reveals essential structure in $\mathbf{x}^{(i)}$ that is not otherwise obvious.

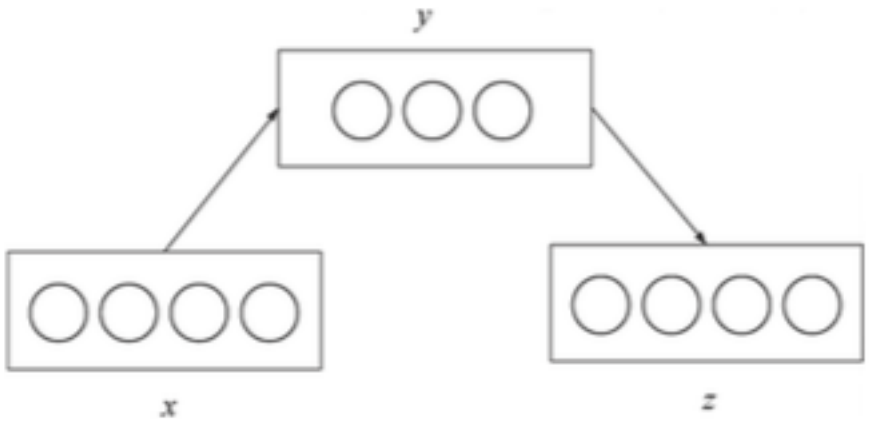

Figure 1: Classic Autoencoder

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.

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

More generally, this methodology casts learning as optimization using Empirical Risk Minimization [**?**]. 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 [**?**]. 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.

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 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 on the minimization problem [**?**] 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 [**?**]. 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).

3.2 Stacked Autoencoders

Stacked Autoencoder (SAE) are, as the name implies, a stack of single-level autoencoders. As such the SAE is a deep learning model [**?**]. SAEs use the autoencoders described above as building blocks to create a deep network [**?**]. While deep architectures can be more expressive and can extract more sophisticated features from into data, until relatively recently deep networks were thought to be too difficult to train. As mentioned above, the breakthrough came when Geoffrey Hinton and his colleagues showed how fast, greedy and unsupervised algorithms can be used to initialize a slower algorithm that fine tunes the learned weights provides very good results on deep networks [**?**]. This result revitalized the machine learning community and deep architectures have been growing in popularity since [**?**].

The basic idea behind the layerwise is shown in 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 [**?**].

4 Data Sets, Evaluation Metrics and Model Performance

In the initial study we collect data from Ψ sensors (interfaces)[5] both in the aggregation and core layers of the data center network on 5 minute intervals for one year, yielding 105,120 samples. These sensors can be thought of as the following RFC 7223 [**?**] counters: speed,discontinuity-time,in-octets, in-unicast-pkts, in-broadcast-pkts, in-multicast-pkts,in-discards,in-errors, in-unknown-protos, out-octets, out-unicast-pkts, out-broadcast-pkts, out-multicast-pkts, out-discards,out-errors which are collected on a per-interface basis. For purposes of this study it would be useful to have a sensor that indicated output queue length, such as the RFC 2863 [**?**] ifOutQLen counter which in theory could directly

[5]Note that Ψ is a hyper-parameter that encodes spatial information about the network's topology.

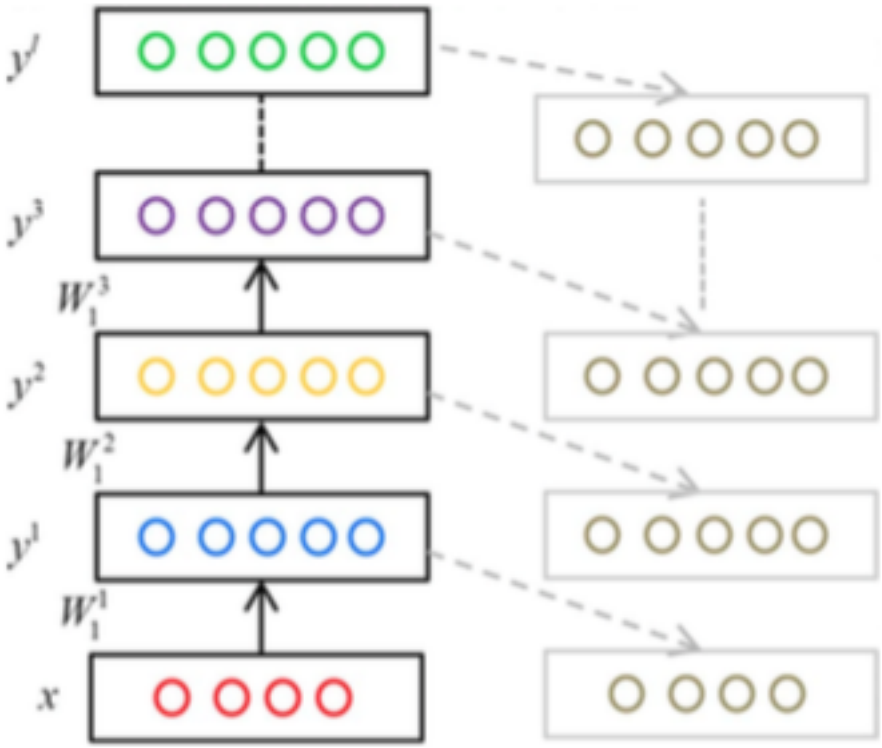

Figure 2: Layerwise training of a Stacked Autoencoder

measure congestion as a function of output queue length. Given that RFC 7223 counters are not universally implemented, RFC 2863 counters are used where RFC 7223 counters are not available. Finally, more complex methods of estimating congestion such as described in [**?**]could be used in future versions.

Recall that the the problem is to measure the sequence $\{X_i^t$ of observed traffic flow data (the 15 parameters described above) plus a timestamp, for $i = 1, 2, ..., m$ and $t = 1, 2, ..., T$ and predict the traffic flow (and hence potential congestion) at the time $(t + \Delta)$ for some prediction horizon Δ. Here $T = (60 * 24 * 365)/5 = 105120$. Note that we also want to consider the temporal relationships inherent in traffic flows, so to predict the traffic flow at time interval t, we also use the traffic flow data at previous time intervals, i.e., $X^{t-1}, X^{t-2}, ..., X^{t-r}$ for some value of r. Thus while the the exact network topology is a hyper-parameter in this study, the model described here incorporates both temporal and spatial correlations. So the dimension of the input data is $\Psi * r * 16$.

4.1 Metrics

Metrics include Mean Absolute Error (MAE), Mean Relative Error (MRE) and the RMS Error (RMSE).

4.2 Results

5 Conclusions and Future Work

6 Acknowledgements