Can Congestion in Data Center Networks Be Predicted By Of Time Of Day?

Draft 0.0

David Meyer

dmm@brocade.com

December 17, 2014

Abstract

The recent explosive 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 the general scaling of data center capacity. These challenges share a common implication: network congestion (and hence network delay) within a data center has is strongly negatively correlated with user quality of experience and operator efficiency. Hence congestion and the closely related problem of network delay are among the key concerns for data center and service operators. Much of the literature in this area generally puts a finer point on the problem: a user's quality of experience can be badly affected when even a single flow suffers from a large latency [2]. The work described in this document demonstrates a novel approach to the congestion control problem, namely, the use of artificial neural networks to predict nascent congestion (and hence queuing delay) in data center networks. This capability will allow operators to both operational and capital expenditures while at the same time optimizing users' quality of experience. 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 explosive 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 the general scaling of data center capacity. These challenges share a common implication: network congestion (and hence network delay) within a data center has is strongly negatively correlated with user quality of experience and operator efficiency. Hence congestion and the closely related problem of network delay are among the key concerns for data center and service operators. Much of the literature in this area generally puts a finer point on the problem: a user's quality of experience can be badly affected when even a single flow suffers from a large latency [2].

The largest part of network delay in today's data centers comes from the queueing delay at router and switch interfaces. Since the propagation delay within a data center is in most cases 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), managing delay is largely about reducing congestion and hence queuing delay at switch interfaces. For example, DCTCP [9] uses ECN marking to slow down flows before the relevant queues become full, while HULL [10] takes a further step and gives up a small amount bandwidth for even lower latency. Our work is also motivated by the need 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.

1.1 Why Artificial Neural Networks?

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 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 [15], 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

[1]This phenomena is evidenced by the explosive growth in the number of "data analytics" startups [7].

"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 [5]. 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 3.2.

In this work we introduce the novel use of a specific form of DNN, the Stacked Autoencoder [12], 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 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 Spatio-Temporal Prediction of Traffic Flow Problem

The Spatio-Temporal Prediction of Traffic Flow Problem is a formal description of the question asked in the title of this document and 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. An observation location can be an interface counter, switch cpu load or memory utilization, or other relevant sensor value (note here that we consider traditional network interface counters to be *sensors*). The network topology is not explicitly represented but is rather encoded in the observation locations. Now, 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. We also want

[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 [3].

to consider the temporal relationships inherent in traffic flows, so in order 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).

3 Methodology

In this section we introduce our basic methodology which is based on a deep-learning based prediction model. A stacked *autoencoder* [16] 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 [12]), 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 $\mathbf{z} = g_{\theta'}(\mathbf{y}) = s(\mathbf{W}'\mathbf{y} + \mathbf{b}')$, 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 recon-*

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

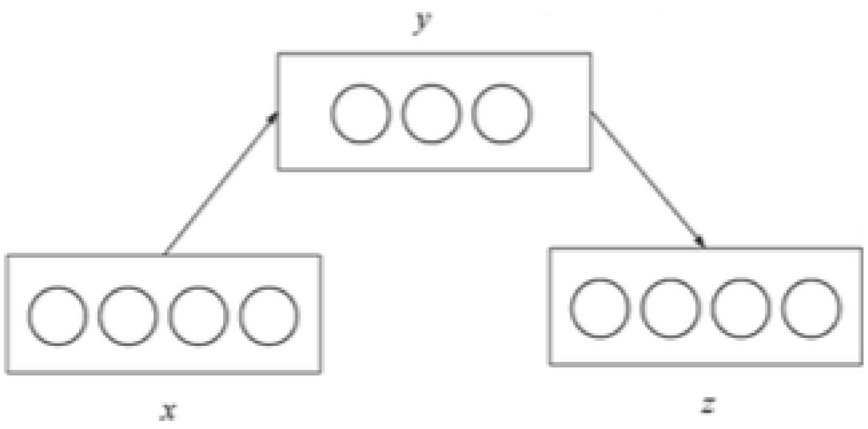

Figure 1: Classic Autoencoder

struction 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 [18]. 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 [4]. 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 [11] 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 [13]. 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 $\hat{\rho}_j = \frac{1}{n} \sum\limits_{i=1}^{n} s(\mathbf{W}\mathbf{x}_j^{(i)} + \mathbf{b})$, 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 Autoencoders (SAEs) are, as the name implies, a stack of single-level autoencoders; hence the SAE is a deep learning model [17]. SAEs use the autoencoders described above as building blocks to create a deep network [16]. While deep architectures can be more expressive and can extract more sophisticated features from input data, until relatively 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 [5]. This result revitalized the machine learning community and deep architectures been successfully applied to a wide variety of classification and prediction problems [6].

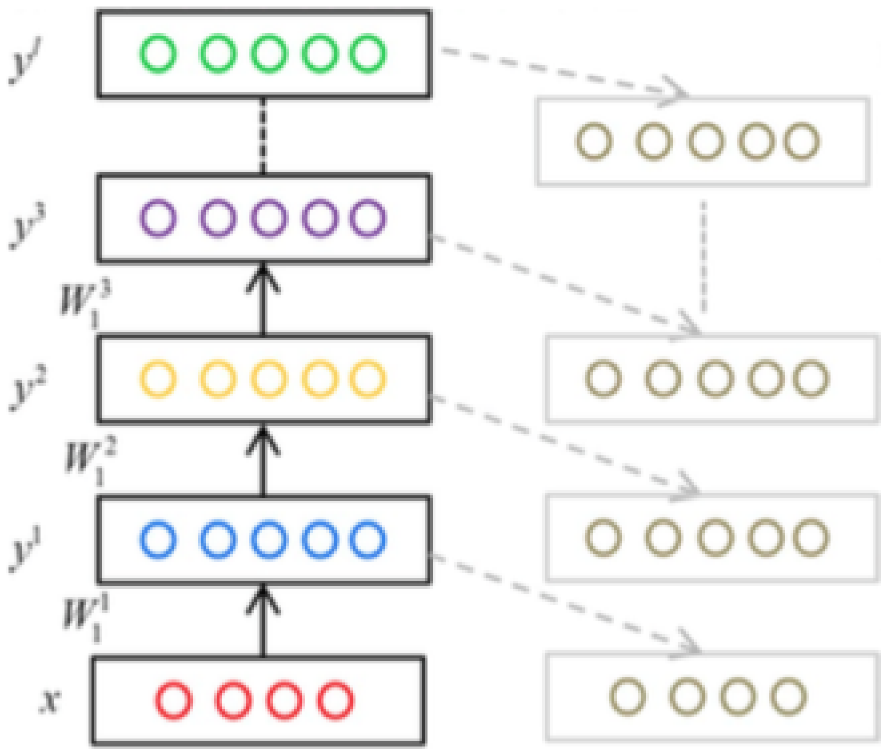

Figure 2: Layerwise training of a Stacked Autoencoder

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 [15].

4 Data Sets, Performance Metrics and Model Performance

In the initial study we collect data from Ψ sensors (interfaces)[4] 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 [1] 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.

4.1 Congestion Computation

For purposes of this study it would be useful to have a sensor that indicated output queue length, such as the RFC 2863 [8] ifOutQLen counter which in theory could directly 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.

Note: need to compute a robust congestion statistic if I can't get it directly e.g., ifOutQLen ; there is also a question as to L2 vs. L3 queues

Finally, more complex methods of estimating congestion (e.g., [14]) could be used in future versions.

4.2 Data Dimensionality

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) , $i = 1, 2, ..., m$ 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; the dimension d of the input data is $\Psi * r * (15 + 1)$.

Note: I want to train the top layer (linear regression) on specific links of interest in the network; we use all the data to train the SAE layers (this provides the model with both temporal and spatial correlations (if they exist) inherently

[4]Ψ is dependent both on network topology and instrumentation.

4.3 Performance Metrics

In order to evaluate the effectiveness of the proposed model, we use three performance indexes: Mean Absolute Error (MAE), Mean Relative Error (MRE), and the Root Mean Squared Error (RMSE). These are defined as follows:

$$\begin{aligned}
&\text{MAE} = \tfrac{1}{n}\sum_{i=1}^{n} |f_i - \hat{f}_i| \\
&\text{MRE} = \tfrac{1}{n}\sum_{i=1}^{n} \tfrac{|f_i - \hat{f}_i|}{f_i} \\
&\text{RMSE} = \left[\tfrac{1}{n}\sum_{i=1}^{n} \left(|f_i - \hat{f}_i|\right)^2\right]^{\frac{1}{2}}
\end{aligned}$$

where where f_i is the observed traffic flow and $\hat{f}$ is the predicted traffic flow.

4.4 Results

5 Conclusions and Future Work

6 Acknowledgements

References

[1] M. Bjorklund. RFC 7223: A YANG Data Model for Interface Management.

[2] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better. Never than late: Meeting deadlines in datacenter networks. *Proceedings of the ACM SIGCOMM*, 2011.

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

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

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

[6] http://deeplearning.net.

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

[8] K. McCloghrie and F. Kastenholz. RFC 2863: The Interfaces Group MIB.

[9] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). *Proceedings of the ACM SIGCOMM*, 2010.

[10] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. *Proceedings of the USENIX NSDI*, 2012.

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

[12] 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.

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

[14] Shao Liu, Mung Chiang, Mathias Jourdain, and Jin Li. Congestion location detection: Methodology, algorithm, and performance. *17th IEEE International Workshop on Quality of Service*, 2007.

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

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

[17] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *arXiv.org*, 2012.

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