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. This paper provides an overview of the basic building block of deep learning networks, the auto-encoder. After describing the basics of auto-encoders, we describe how deep networks are built by stacking auto-encoders to build deep learning artificial neural networks.

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. 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

\footnote{This phenomena is evidenced by the explosive growth in the number of "data analytics" startups.}
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 [12], 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 \((x^{(i)}, t^{(i)})\), where the \(x^{(i)}\) are the inputs and the \(t^{(i)}\) 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 \(h_\theta(x^{(i)})\). \(h_\theta(x^{(i)})\) is then compared to the target \(t^{(i)}\) and the difference \(h_\theta(x^{(i)}) - t^{(i)}\) 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 [4]. 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 [13] 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 [2].
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 [9]), an autoencoder takes an input vector \( x \in [0, 1]^d \) and maps it to a hidden representation \( y \in [0, 1]^d \) through a deterministic mapping \( y = f_\theta(x) = s(Wx + b) \), parameterized by \( \theta = \{W, b\} \). \( W \) is a \( d' \times d \) weight matrix, \( b \) is a bias vector and \( s \) is the sigmoid activation function, \( s(x) = \frac{1}{1+e^{-x}} \). The hidden representation \( y \), sometimes called the latent representation, is then mapped back to a reconstructed vector \( z \in [0, 1]^d \), where \( z = g_{\theta'}(y) = s(W'y + b') \), with \( \theta' = \{W', b'\} \). This scenario is depicted in cartoon form in Figure [1]. Thus each training \( x^{(i)} \) is thus mapped to a corresponding \( y^{(i)} \) and a reconstruction (of \( x^{(i)} \)) \( z^{(i)} \). Finally, note that the weight matrix \( W' \) may optionally be constrained by \( W = W^T \), in which case the autoencoder is said to have tied weights. Each training example \( x^{(i)} \) is thus mapped to a corresponding \( y^{(i)} \) which is then mapped to a reconstruction \( z^{(i)} \) such that \( z^{(i)} \approx x^{(i)} \).

The basic idea here is that the autoencoder is constructed in such a way that the mapping \( x^{(i)} \mapsto y^{(i)} \) reveals essential structure in the input vector \( x^{(i)} \) 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 \( x^{(i)} \) might be an image (pixels) while \( y^{(i)} \) might consist of edges in various orientations.

The parameters \( \theta \) and \( \theta' \) of the model are optimized to minimize the average reconstruction error as shown in Equation [1].

\[
\theta^*, \theta'^* = \arg\min_{\theta, \theta'} \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, z^{(i)})
\]

\[= \arg\min_{\theta, \theta'} \frac{1}{n} \sum_{i=1}^{n} L\left(x^{(i)}, g_{\theta'}(f_{\theta}(x^{(i)}))\right)\]

(1)

Here \( L \) is a loss function such as the traditional squared error \( L(x, z) = \|x - z\|^2 \). Note that if \( x \) and \( z \) 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.

\[
L_{\mathcal{H}}(x, z) = \mathcal{H}(B_x, B_z)
\]

\[= - \sum_{k=1}^{d} x_k \log z_k + (1 - x_k) \log(1 - z_k)]
\]

(2)

\(^3\text{Note that the sigmoid activation function } s \text{ "squashes" its input into the range } [0, 1] \)
More generally, this methodology casts learning as *optimization* using Empirical Risk Minimization \cite{empirical_risk_minimization}. The Empirical Risk $\hat{R}$ is defined as

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

(3)

In some cases it may be necessary to induce a preference for some values of the parameters to avoid overfitting \cite{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 $D_n = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$. Then the *Regularized Empirical Risk* is defined as follows:

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

(4)

where $\Omega$ 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 \footnote{The L1-norm is sometimes called the *taxicab norm*} as a regularizer, where the L1-norm $\| x \|_1 = \sum_i | x_i |$, enforces spareness on the vector $x$. This is because in order to
minimize $\| \mathbf{x} \|_1$, some (or most) of the $x_i$ have to be zero. On the other hand, using the L2-norm 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. In general, the p-norm is defined to be $\| \mathbf{x} \|_p = \left( \sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}$.

Thus learning in this setting amounts to finding optimal parameters $\theta^*$ 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 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 [10]. The Kullback-Leibler Divergence $D_{KL}(P \parallel Q)$ 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 $D_{KL}(\rho \parallel \hat{\rho})$ as follows

$$D_{KL}(\rho \parallel \hat{\rho}) = \rho \log \frac{\rho}{\hat{\rho}} + (1 - \rho) \log \frac{1 - \rho}{1 - \hat{\rho}}$$

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

$$\arg \min_{\theta, \theta'} \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, g_{\theta'}(f_{\theta}(x^{(i)}))) + \gamma \sum_{j=1}^{H_D} D_{KL}(\rho \parallel \hat{\rho})$$

where $H_D$ is the number of hidden units and $\gamma$ is a sparsity weighting term. Kullback-Leibler Divergence has the nice property that $D_{KL}(\rho \parallel \hat{\rho}) = 0$ if $\rho = \hat{\rho}$ (and so weighting these cases improves the sparsity of the encoding; this is the job of the $\gamma$ 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 [14]. SAEs use the autoencoders described above as building blocks to create a deep network [13]. 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 [4]. This result revitalized the machine learning community and deep architectures been successfully applied to a wide variety of classification and prediction problems [5].

![Layerwise training of a Stacked Autoencoder](image)

**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 $y^{(i)}$) 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 [12].
3 Conclusions and Future Work
4 Acknowledgements
References


