Inference of Stacked Sparse Autoencoder

Shengke Xue, 11531015

Abstract

An autoencoder, which is an unsupervised learning algorithm, can learn features from unlabeled data autonomously. This paper focuses on introduction to the architecture of the autoencoder and its training approach with gradient descent method. During the derivation, the sparse representation and backpropagation algorithm are also considered. After that, This paper extends results to the stacked autoencoder (i.e. one of deep neural networks).

Index Terms

autoencoder, unsupervised, sparse, backpropagation, stacked.

I. Introduction

Supervised learning is one of the most powerful tools of Artificial Intelligence, and has led to automatic zip code recognition, speech identification, self-driving cars, and continually improving comprehension of the human genome. Despite its significant successes, supervised learning present is still severely limited. More specifically, most applications of it require that we manually specify the input features x given to the algorithm. Once a good feature representation is obtained, a supervised learning algorithm can work well. But in such domains as computer vision, audio processing, and natural language processing, there are hundreds or perhaps thousands of researches who have spent years of their efforts slowly and laboriously in hand-engineering vision, audio or text features. While much of this feature-engineering work is extremely clever, one may wonder if we can do better.

Ideally we would like to have algorithms that can automatically learn even better feature representations than the hand-engineered ones. That is the reason why this paper describes the **sparse autoencoder** learning algorithm, which is one approach to learn features from unlabeled data autonomously. In some

Shengke Xue is a PhD. candidate with the College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China. e-mail: xueshengke@zju.edu.cn.

This report is updated on April 19, 2016.

domains, such as computer vision, this approach is not competitive by itself with the best hand-engineering features, but the features it can learn do turn out to be practical for a range of situations.

II. Related Work

Though several classical representations of statistics information described in [1] have already existed, we declare that our approach is quite distinct from those. The concept "autoencoder" was first proposed in [2] but with only a short introduction, in 2006. Afterwards, [3] and [4] proclaimed the effectiveness of unsupervised learning algorithm in deep neural networks. In 2013, it also was applied by [5] to denoise corrupted images to obtain the latent representation with a better classification performance. Significant improvement of autoencoder with bilingual sparse features was suggested by [6] for statistical machine translation in 2014. Moreover, sparse autoencoder was employed as an useful tool for the signature verification that reduced error and increased accuracy by [7] in 2015.

III. Derivation of Autoencoder

We skip the description of feedforward neural network with supervised learning but begin with deducing autoencoder directly. Suppose that we have only unlabeled training examples set $\{x^{(1)}, x^{(2)}, x^{(3)}, \cdots\}$, where $x^{(i)} \in \mathbb{R}$. An autoencoder neural network (see Fig. 1) is an unsupervised learning algorithm that applies backpropagation, setting the outputs to be equivalent to the inputs, i.e., $y^{(i)} = x^{(i)}$.

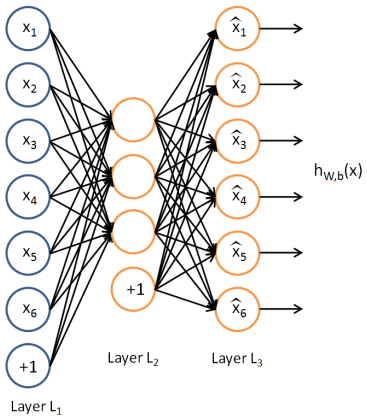

Fig. 1. An autoencoder architecture

The autoencoder tries to learn a function $h_{W,b}(\boldsymbol{x}) \approx x$. In other words, it desires to learn an approximation to the identity function, so as to output $\hat{\boldsymbol{x}}$ that is similar to $\boldsymbol{x}$. The identity function seems a particularly trivial function to be done pointlessly; but by placing constraints on the network, such as by

limiting the number of hidden units, we can discover potential structure about the data. As a concrete instance, assume the inputs $\boldsymbol{x}$ are the pixel intensity values from a 10×10 image (100 pixels) so $n = 100$, and there are $s_2 = 50$ hidden units in layer L_2. Note that we also have $\boldsymbol{y} \in \mathbb{R}^{100}$ as the output. Since there are only 50 hidden units, the network is forced to perceive a **compressed** representation of the input, i.e., given only the vector of activations of hidden units $\boldsymbol{a}^{(2)} \in \mathbb{R}^{50}$, it must endeavor to **reconstruct** the 100-pixel input $\boldsymbol{x}$. If the input were completely stochastic—say, each x_i comes from an i.i.d. Gaussian distribution independent of the other features—then this compression task would be quite difficult. But if there is underlying structure in the data, e.g., if some of the input features are correlated, then this algorithm will be able to discover some of those correlations.

A. Sparsity

One argument above relies on the number of hidden units s_2 being small. But even when the number of hidden units is large (perhaps even greater than the number of input pixels), we can still discover latent structure, by imposing other constraints on the network. In particular, if we impose a **sparsity** constraint on the hidden units, then the autoencoder will still detect potential structure in the data, even if the number of hidden units is large.

Informally, we will think of a neuron as being "active" (or "firing") if its output value is close to 1, or as being "inactive" if its output value is close to 0[1]. We would like to constrain the neurons to be inactive most of the time.

Recall that $a_j^{(2)}$ denotes the activation of hidden unit j in the autoencoder. However, this notation does not make explicit what is the input $\boldsymbol{x}$ that leads to this activation. Thus, we rewrite $a_j^{(2)}(\boldsymbol{x})$ to signify the activation of this hidden unit when the network is given a specific input $\boldsymbol{x}$. Further, let

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} [a_j^{(2)}(x^{(i)})] \tag{1}$$

be the average activation of hidden unit j (averaged over the training set). We would like to approximately enforce the constraint

$$\hat{\rho}_j = \rho, \tag{2}$$

where ρ is a **sparsity parameter**, typically a small value close to zero (e.g. $\rho = 0.05$). In other words, we would like the average activation of each hidden neuron j to be close to 0.05. To satisfy this constraint, the hidden unit's activations must mostly be near 0.

[1]This statement assumes a sigmoid activation function. As for a tanh activation function, we think of a neuron as being inactive when its output values near to -1.

To achieve this, we will add an extra penalty term to our optimization objective given in (5) that penalizes $\hat{\rho}_j$ deviating from ρ considerably. Many choices of the penalty term will give reasonable results. We will choose the following:

$$\sum_{j=1}^{s_2} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}. \tag{3}$$

Here, s_2 is the number of neurons in the hidden layer, and the index j is summing over the hidden units in our network. If you are familiar with the concept of KL divergence, this penalty term is based on it, and can also be written

$$\sum_{j=1}^{s_2} \mathrm{KL}(\rho \parallel \hat{\rho}_j), \tag{4}$$

where $\mathrm{KL}(\rho \parallel \hat{\rho}_j)$ is the Kullback-Leibler (KL) divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean $\hat{\rho}_j$. KL divergence is a standard function for measuring how distinct two different distributions are.

This penalty function has the property that $\mathrm{KL}(\rho \parallel \hat{\rho}_j) = 0$ if $\hat{\rho}_j = \rho$, and otherwise it increases monotonically as $\hat{\rho}_j$ deviates from ρ. For instance, in Fig. 2, we have set $\rho = 0.2$, and plotted $\mathrm{KL}(\rho \parallel \hat{\rho}_j)$ for $\hat{\rho}_j$ ranging from $[0, 1]$.

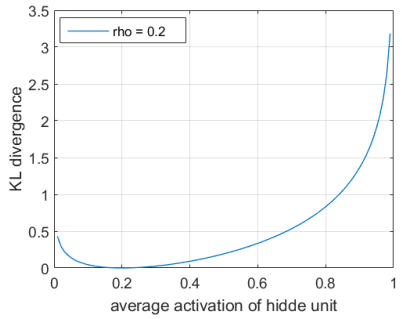

Fig. 2. Plot KL divergence with $\rho = 0.2$

We find that the KL divergence reaches its minimum of 0 at $\hat{\rho}_j = \rho$, and rises promptly (actually approaches ∞) as $\hat{\rho}_j$ closes to 0 or 1. Thus, minimizing this penalty term has the effect of causing $\hat{\rho}_j$ to approximate to ρ.

B. Optimization

Then we formulate our problem to the standard form yields

$$\text{minimize} \quad \frac{1}{m}\sum_{t=1}^{m}\frac{1}{2} \parallel h_{W,b}(x^{(t)}) - x^{(t)} \parallel^2 \tag{5}$$

$$\text{subject to} \quad \parallel \boldsymbol{W} \parallel^2 \leqslant \epsilon, \tag{6}$$

$$\hat{\rho}_j = \rho, \ j = 1, 2, \cdots, s_2. \tag{7}$$

Clearly we will minimize the objective as mean-square errors. It is accompanied by two constraints: (1) The magnitude of weight $\boldsymbol{W}$ should be small; (2) Average activation of each hidden unit $\hat{\rho}_j$ ought to be sparse. Note that $h_{W,b}(\boldsymbol{x})$, which is regraded as a re-constructive function of the input $\boldsymbol{x}$, will be influenced by $\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}$ significantly. But we do not comprehend how weight $\boldsymbol{W}$ works on the objective. In short, we can **not** prove whether the problem proposed in (5) is **convex** or not. Thus, the close form of weight $\boldsymbol{W}$ will hardly be expressed, directly.

Despite the convexity of our problem being unknown, that is, we can not figure out the global optimal solution for $\boldsymbol{W}$ straightforwardly, it is still feasible to employ the **gradient descent method**, one of the effective but not the best techniques, to solve the problem.

With a variety of elements in $\boldsymbol{W}$, we do not ensure that $\boldsymbol{W}$ converges eventually to its global optimal solution instead of local optimal solution. Practically, it is pretty probable to obtain the latter situation but usually performs fairly well.

C. Backpropagation

Our overall cost function is defined as

$$J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m}\sum_{t=1}^{m}\frac{1}{2} \parallel h_{W,b}(x^{(t)}) - x^{(t)} \parallel^2 + \frac{\lambda}{2}\sum_{\ell=1}^{n_\ell - 1}\sum_{i=1}^{s_\ell}\sum_{j=1}^{s_{\ell+1}}\left(W_{ji}^{\ell}\right)^2 + \beta\sum_{j=1}^{s_2}\mathrm{KL}(\rho \parallel \hat{\rho}_j). \tag{8}$$

The first term in the definition $J(\boldsymbol{W}, \boldsymbol{b})$ is an average sum-of-squares error term over m training samples, followed by a regularization term (also called a **weight decay** term) that tends to decrease the magnitude of the weights, and assists prevent over-fitting. The parameters λ and β control the relative significance of three terms. The term $\hat{\rho}_j$ implicitly depends on $\boldsymbol{W}, \boldsymbol{b}$, for it is the average activation of hidden unit j, which is acquired through $\boldsymbol{W}, \boldsymbol{b}$.

One iteration of gradient descent updates the parameters $\boldsymbol{W}, \boldsymbol{b}$ given by,

$$W_{ji}^{\ell} := W_{ji}^{\ell} - \alpha \frac{\partial J}{\partial W_{ji}^{\ell}}, \tag{9}$$

$$b_{j}^{\ell} := b_{j}^{\ell} - \alpha \frac{\partial J}{\partial b_{j}^{\ell}}. \tag{10}$$

Here α denotes the learning rate. From that, The key step is computing the partial derivatives above. In consideration of multi-layer architecture (will be introduced in Section IV), we adopt **error backpropagation algorithm** uniformly to simplify our inference.

The "error" which propagates backwards through the network can be thought of as "sensitivities" with respect to the current total input of each unit, yields

$$\boldsymbol{\delta}^{\ell} = \frac{\partial J}{\partial \boldsymbol{z}^{\ell}}, \ \boldsymbol{a}^{\ell} = f(\boldsymbol{z}^{\ell}). \tag{11}$$

Let us consider the autoencoder network as Fig. 3 demonstrates. We derive the sensitivity with respect

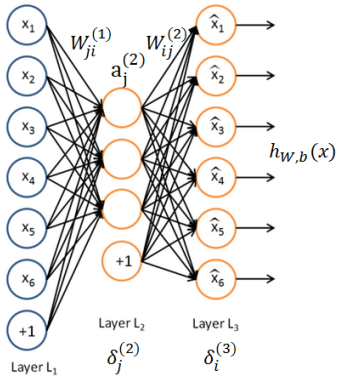

Fig. 3. Train an autoencoder

to each unit from last layer to the hidden layer,

$$\delta_i^{(3)} = \frac{\partial J}{\partial z_i^{(3)}} = \frac{1}{m} \sum_{t=1}^{m} (h_{W,b}(x^{(t)}) - x^{(t)}) f'(z_i^{(3)}), \tag{12}$$

$$\delta_j^{(2)} = \frac{\partial J}{\partial z_j^{(2)}} = \left(\left(\sum_{i=1}^{s_2} W_{ij}^{(2)} \delta_i^{(3)} \right) + \beta \left(-\frac{\rho}{\hat{\rho}_j} + \frac{1-\rho}{1-\hat{\rho}_j} \right) \right) f'(z_j^{(2)}). \tag{13}$$

Note that we will need to know $\hat{\rho}_j$ to compute this term. Thus, we have to go through a forward pass on all training examples first to calculate the average activations on the entire training set, before computing backpropagation on any sample.

Having gained the sensitivities in each layer of an autoencoder, we write the gradients for $\boldsymbol{W}^{\ell}$ and $\boldsymbol{b}^{\ell}$ of each layer, respectively, as a concise form given by

$$\frac{\partial J}{\partial W_{ij}^{(2)}} = \frac{\partial J}{\partial z_i^{(3)}} \frac{\partial z_i^{(3)}}{\partial W_{ij}^{(2)}} = \delta_i^{(3)} a_j^{(2)} + \lambda W_{ij}^{(2)}, \quad \frac{\partial J}{\partial b_i^{(2)}} = \frac{\partial J}{\partial z_i^{(3)}} \frac{\partial z_i^{(3)}}{\partial b_i^{(2)}} = \delta_i^{(3)}. \tag{14}$$

$$\frac{\partial J}{\partial W_{ji}^{(1)}} = \frac{\partial J}{\partial z_j^{(2)}} \frac{\partial z_j^{(2)}}{\partial W_{ji}^{(1)}} = \delta_j^{(2)} x_i + \lambda W_{ji}^{(1)}, \quad \frac{\partial J}{\partial b_j^{(1)}} = \frac{\partial J}{\partial z_j^{(2)}} \frac{\partial z_j^{(2)}}{\partial b_j^{(1)}} = \delta_j^{(2)}. \tag{15}$$

So far we may train the parameters of an autoencoder network in an effective way through gradient descent method.

IV. Stacked Autoencoder

Stacked autoencoder neural network is composed of multi-layer sparse autoencoders, see Fig. 4. It takes the outputs from the previous autoencoder as the inputs for the next autoencoder. A fair way to

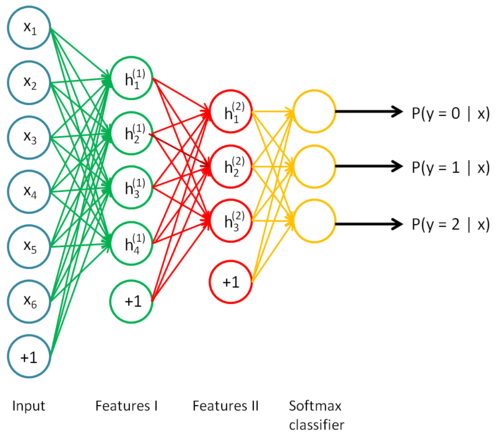

Fig. 4. A stacked autoencoder neural network

update parameters for a stacked autoencoder is to use **greedy layer-wise** training. To do this, we train the first layer on the raw input to get fine parameters $\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}$. Note that the vector of activation of hidden units $\boldsymbol{a}^{(2)}$ is a kind of sparse representation, interpreted above, of the raw input. Therefore, we apply this as the input to train parameters $\boldsymbol{W}^{(2)}, \boldsymbol{b}^{(2)}$ in second layer. Repeat this procedure for subsequent layers. This method trains the parameters of each layer individually while freezing parameters for the remainder of the network. To acquire better results, after this stage of training is accomplished, **fine-tuning** using backpropagation can be employed to improve the results by tuning parameters of all layers that are changed simultaneously. If the only interest lies on the purpose of classification, the common

practice is to then discard the "decoding" layers of the stacked autoencoder and link the last hidden layer $\boldsymbol{a}^{(n)}$ to the softmax classifier, as Fig. 4 illustrates. The gradients from the (softmax) classification error will be propagated backwards to the "encoding" layers to guarantee the efficiency of training.

V. Discussion

A stacked autoencoder enjoys all the benefits of any deep network of greater expressive power. Further, it often captures an useful "hierarchical combination" or "part-whole decomposition" of the input. To view this, recall that an autoencoder endeavors to learn features that form an appropriate representation of its input. The first layer of a stacked autoencoder tries to learn first-order features in the raw input (such as edges in an image). The second layer of a stacked autoencoder tries to learn second-order features corresponding to patterns in the appearance of first-order features (e.g. in terms of what edges tend to occur together—for instance, to form contour or corner detectors). Higher layers of the stacked autoencoder attempt to learn even higher-order features.

References

[1] A. Hyvärinen, J. Hurri, and P. O. Hoyer, *Natural Image Statistics: A Probabilistic Approach to Early Computational Vision.* Springer Science & Business Media, 2009, vol. 39.

[2] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[3] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle *et al.*, "Greedy layer-wise training of deep networks," *Advances in neural information processing systems*, vol. 19, p. 153, 2007.

[4] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?" *The Journal of Machine Learning Research*, vol. 11, pp. 625–660, 2010.

[5] K. Cho, "Simple sparsification improves sparse denoising autoencoders in denoising highly corrupted images," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, S. Dasgupta and D. Mcallester, Eds., vol. 28, no. 3. JMLR Workshop and Conference Proceedings, May 2013, pp. 432–440.

[6] B. Zhao, Y.-C. Tam, and J. Zheng, "An autoencoder with bilingual sparse features for improved statistical machine translation," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on.* IEEE, 2014, pp. 7103–7107.

[7] M. Fayyaz, M. H. Saffar, M. Sabokrou, M. Hoseini, and M. Fathy, "Online signature verification based on feature representation," in *Artificial Intelligence and Signal Processing (AISP), 2015 International Symposium on*. IEEE, 2015, pp. 211–216.