A Comparative Study of Deep Learning Approaches in Network Intrusion Detection

[Midterm Project Report] [*]

Jiaqi Yan
Illinois Institute of Technology
10 West 31st Street
Chicago, Illinois, 60616
jyan31@hawk.iit.edu

Dong Jin
Illinois Institute of Technology
10 West 31st Street
Chicago, Illinois, 60616
dong.jin@iit.edu

ABSTRACT

Recently, a handful of novel deep neural networks have achieved unprecedentedly good performance on image classification, natural language processing, speech recognition and many other artificial intelligence related research fields. In this project we investigate the possibility of applying the cutting edge deep learning techniques to the classic network intrusion detection system. We specifically look at a well-known public network intrusion dataset and compare the performance of several popular deep neural network architectures on this particular NSL-KDD dataset. At this stage, we

- report how we processing the NSL-KDD dataset so that it can be fed to deep neural networks;
- briefly introduce each deep neural network considered in this project

Keywords

Network Intrusion Detection; Deep Learning; Machine Learning

1. INTRODUCTION

More background information will be available in the final report, for example, introducing back-propagation [9].

2. DATASET AND PREPROCESSING

Among various available datasets [2,7,11], we choose NSL-KDD dataset [11] to evaluate the performance of applying various deep neural networks to the network intrusion detection problem. NSL-KDD dataset originates from the KDDCup 99 dataset [2], which was used for the third International Knowledge Discovery and Data Mining Tool Competition. However, NSL-KDD dataset addresses two issues of the KDDCup 99 dataset. First, it eliminates the redundant records existing in KDDCup 99, which takes up 78% and 75% of the records in train and test set, respectively. Second, it adds another label, classification difficulty, to each distinct record, and samples the dataset such that the fraction of the record from a difficulty level is inversely proportional to its difficulty. Both enhancements make NSL-KDD dataset more suitable for evaluation of intrusion detection systems.

The train dataset consists of 125,973 TCP connection records, while the test dataset consists of 22,544 ones. A record is defined by 41 features, including 9 basic features of individual TCP connections, 13 content features within a connection and 9 temporal features computed within a two-second time window, and 10 other features. Connections in the train dataset are labeled as either normal or one of the 24 attack types. There are addtional 14 types of attacks in the test dataset, specifically designed to test the classifier's ability to handle novel attacks. The task of the classifier is to identify whether a connection is normal or one of the 4 categories of attacks, namely denial of service (DoS), remote to local (R2L), user to root (U2R) and probing. Alternatively, we can configure the classifier to report a connection to be either normal or attack. The former is called 5-class problem, whereas the later 2-class problem, throughout this text.

Before feeding the NSL-KDD to neural networks, we need to take some preprocessing steps. First and foremost, we need to convert symbolic features and the labels to one-hot encoding format. For example, if feature f can take n possible values from 1 to n, feature value x will be converted to a n-dimention binary vector with the xth dimention set to 1 and others set to zero. This can be done for the entire dataset (both train and test set) using the `OneHotEncoder` provided by scikit-learn [3]. Then we shuffled the data, along with its label, so that later in the stochastic gradient descent learning phase batch data are already randomized. At last, we perform the min-max normalization so that data values are all in the range of [0, 1].

3. DEEP NEURAL NETWORKS

In this section, we give a brief review of the deep learning architectures that we used in network intrusion detection problem.

3.1 Multilayer Perceptron

[*]code for our project available on www.github.com

CS 558 Spring '17, March 10th, 2017, Chicago.
© 2017 ACM. ISBN xxx-x-xxxx-xxxx-x/xx/xx...$15.00
DOI: http://dx.doi.org/xx.xxxx/xxxxxxxxxx.xxxxxx

Multilayer perceptron (MLP) is the simplest deep learning classifier in terms of design of the architecture. It is a fully connected feed-forward neural network, as shown in Figure 1. By introducing non-linear neural units (perceptron), it can distinguish data that are not linearly separable. However, the non-linearity also make it very hard to train a deep MLP, even if people have proposed the efficient back-propagation learning algorithm [9]. Recently it revived due to various new training techniques designed by deep learning community, including Stochastic Gradient Descent (SGD), batch normalization [5] and Dropout [10]. Expect for the number of neurons in each layer and number of layers, MPL can also be tuned with different activation functions, or neural types. The most popular two, which are used in this project, are logistic function and rectifier linear unit. Logistic function is written as

$$f(x) = \frac{1}{1+e^{-x}} \tag{1}$$

It has a very useful property when we applying backpropagation:

$$f'(x) = f(x)(1-f(x)) \tag{2}$$

Recently, most deep neural networks adopt rectifier neural unit and achieved very good performance [6]. Rectifier linear unit is defined as

$$f(x) = \max(0, x) \tag{3}$$

Let $\mathbf{a}^{(l)}$ be the activation of layer l, $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ be layer l's parameter. With activation function defined, we have the following recursive formula that describes the feed-forward step of the perceptron network.

$$\mathbf{z}^{(l+1)} = \mathbf{W}^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l)} \tag{4}$$

$$\mathbf{a}^{(l+1)} = f(\mathbf{z}^{(l+1)}) \tag{5}$$

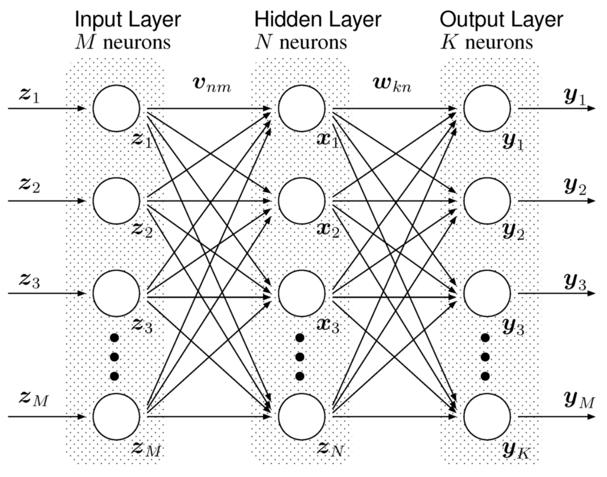

Figure 1: A perceptron neural network with 1 hidden layer. Figure courtesy of Teijiro Isokawa, Haruhiko Nishimura and Nobuyuki Matsui.

3.2 Generative Models

The amount of network traffic data is enormously large, usually in the order of terabytes per day in a large monitored network. This available big data makes deep learning techniques a promisingly better solution to traffic classification. In practice, however, the amount of data is impossible for a human security analyst to review, e.g., to find patterns and label anomalies. Generative model which can be trained unsupervisedly comes to rescue in that

- It utilizes the large amount of unlabeled data to learning useful and hierarchical features;
- It is actually a way to initialize the weights in a deep neural network, which can be further fine-tuned to be a high performance classifier.

In this project we propose to try two generative models: restricted Boltzmann machine and autoencoders.

3.2.1 Restricted Boltzmann Machine

Restricted Boltzmann machine (RBM) [4] is a type of energy-based model, which associate a scalar energy to each configuration of the variables in the network. In energy-based model, learning is the process of configure the network weights so that the energy over training data are minimized. RBM consists of a layer of hidden units (H) and a layer of visible units (V). Here "restricted" means that connections are just between hidden and visible layer, but not within hidden layers or visible layers. This make training RBM faster than Boltzmann machine and feasible for stacking together to form deep architecture. A joint configuration, $(\mathbf{v}, \mathbf{h})$ of the visible and hidden units has an energy given by

$$E(\mathbf{v},\mathbf{h}) = -\sum_{i \in visible} a_i v_i - \sum_{j \in hidden} b_j h_j - \sum_{i,j} v_i h_j w_{ij} \tag{6}$$

where $a = \{a_i\}$ and $b = \{b_j\}$ are biases in visible and hidden layer respectively, $W = \{w_{ij}\}$ is the weights between them. The network assign a probability to every possible pair of $(\mathbf{v}, \mathbf{h})$ via this energy function

$$p(\mathbf{v},\mathbf{h}) = \frac{1}{Z} e^{-E(\mathbf{v},\mathbf{h})} \tag{7}$$

$$p(\mathbf{v}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{8}$$

where Z is the partition function that equals to the summation over all possible hidden and visible vector pairs

$$Z = \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{9}$$

Based on the "maximizing log likelihood" idea, we want raise the probability of a training example and it can be done by adjusting the weights and biases to lower the energy of the considered example and raise the energy of the other examples, so that they make big contribution to the partition function Z:

$$\frac{\partial \log p(v)}{\partial w_{ij}} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model} \tag{10}$$

This implies the following learning rule for performing stochastic gradient ascent on training data

$$\Delta w_{ij} = \varepsilon(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}) \tag{11}$$

The first term $\langle v_i h_j \rangle_{data}$ is a sample from the data and it is easy to compute since there is not directed connection between hidden units. The sampling of h_j is based on the probability

$$Prob(h_j = 1|\mathbf{v}) = sigmoid(b_j + \sum_i v_i w_{ij}) \quad (12)$$

Similarly, v_i can be sampled with the following distribution

$$Prob(v_i = 1|\mathbf{h}) = sigmoid(a_j + \sum_j h_i w_{ij}) \quad (13)$$

The term $\langle v_i h_j \rangle_{model}$ can be obtained by performing alternative Gibbs sampling for a long time. The sampling start from a random visible state. Then we update the hidden units in parallel with Equation 12, followed by updating the visible units in parallel with Equation 13.

The layer-by-layer training algorithm for stacking RBMs goes in a greedy fashion. After learning the first layer RBM, the activity vector of the hidden units can be used as "data" for training the second layer RBM and this process can be repeated to learn as many hidden layers as desired. As data passing through the RBMs, we can obtain the highest level features which are typically fed into a classifier. The entire deep network (RBMs plus the classifier) can be fine-tuned to improve the classification performance.

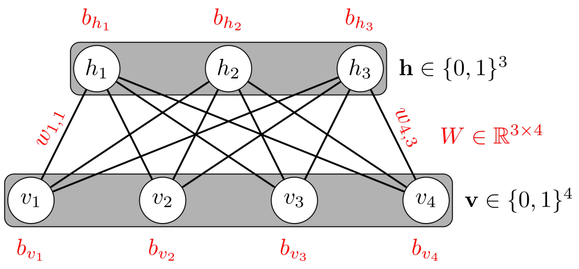

Figure 2: Restricted Boltzmann Machine. Figure courtesy of https://commons.wikimedia.org/wiki/File:Restricted-boltzmann-machine.svg

3.2.2 *Autoencoders*

An autoencoder neural network is an unsupervised model with typically one hidden layer that tries to set the output layer to be equal to the input. As shown in Figure 3, we want the network to learn a function $h_{W,b}(x) \approx x$. However, to prevent the network from learning the meaningless identify function, we need to place extra constraints on the network, which introduces many different flavors of autoencoders. In this project we consider two most popular type of autoencoders, sparse autoencoder and denoise autoencoder.

The **denoising autoencoder** algorithm is proposed by [12] and illustrated in Figure 4. To prevent learning identify function, an example $\mathbf{x}$ is first corrupted, either by adding Gaussian noise or by random masking a fraction of items in $\mathbf{x}$ to zero. The autoencoder then maps corrupted $\tilde{\mathbf{x}}$ to a hidden representation $\mathbf{y} = sigmoid(\mathbf{W}\tilde{\mathbf{x}} + \mathbf{b})$. From $\mathbf{y}$ we reconstruct a $\mathbf{z} = g'_\theta(\mathbf{y})$. The training needs to learning the parameters θ and θ' so that average reconstruction error is minimized over training set. For binary input $\mathbf{x}$, usually cross entropy is adopted as $L_H(\mathbf{x}, \mathbf{z})$; while mean squared error is used for real-valued $\mathbf{x}$.

The **sparse autoencoder** works by placing a sparsity constraint on the hidden units [8]. First, we make the autoencoder's hidden layer size to be over-complete, that is of

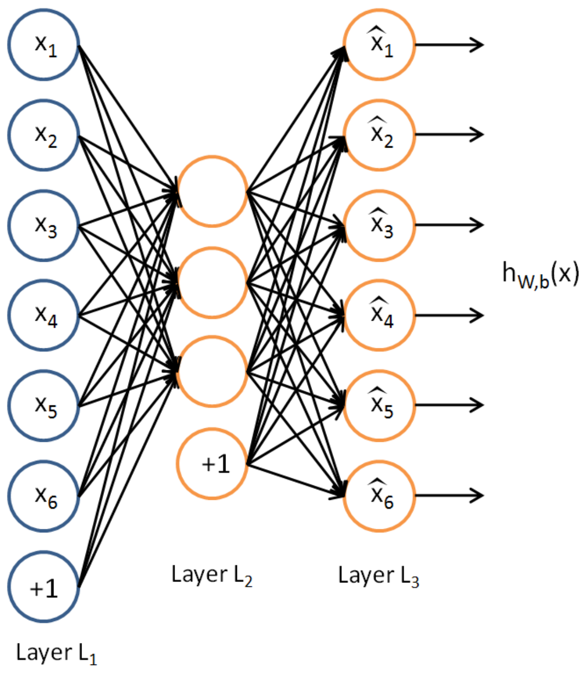

Figure 3: General Architecture of Autoencoders. Figure courtesy of [1].

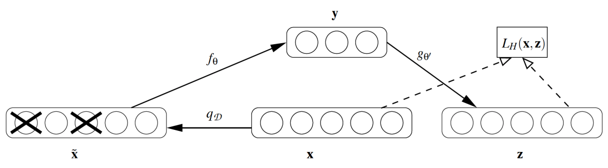

Figure 4: The denoising autoencoder algorithm. Input example $\mathbf{x}$ is randomly corrupted via $q_{\mathcal{D}}$ and then is mapped via encoder f_θ to $\mathbf{y}$. The decoder g'_θ attempts to reconstruct $\mathbf{x}$ and produces $\mathbf{z}$. Reconstruction error is measured by loss $L_H(\mathbf{x}, \mathbf{z})$, to be minimized during the training phase. Figure courtesy of [12].

larger size comparing to the dimension of the input. Let's denote the activation of hidden unit j of layer 2 in Figure 3 to be $a_j^2(\mathbf{x})$ given input example $\mathbf{x}$. With that, we can define the average activation of hidden unit j over the m-size training set

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} a_j^2(\mathbf{x}) \quad (14)$$

The sparsity constraint is enforcing, $\forall$ hidden unit j,

$$\hat{\rho}_j = \rho \quad (15)$$

where ρ is a sparsity parameter that approximates zero (say 0.05). This constraint can be vectorized over the hidden layer, say of size n_2, with the KL divergence based penalty term

$$\sum_j^{n_2} KL(\rho||\hat{\rho}_j) = \sum_j^{n_2} [\rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}] \quad (16)$$

The sparsity penalty term is integrated into the cost function by adding another hyperparameter β

$$L(W,b) = \frac{1}{2}||h_{W,b}(\mathbf{x}) - \mathbf{x}||^2 + \beta \sum_{j}^{n_2} KL(\rho||\hat{\rho}_j) \tag{17}$$

Denoise autoencoder and sparse autoencoder, surprisingly, have different application domains. Vincent et al. [12] have shown that stacked denoise autoencoder can be used to initialize a deep neural network's weight parameter, achieving similar and sometimes better performance than stacked RBM. They also show that training stacked denoise autoencoder with MNIST dataset, it is able to resynthesize a variety of similarly good quality digits. Raina et al. [8] have compared sparse encoding with principle component analysis (PCA) and argue that transfer raw features with a well unsupervisely trained sparse audoencoder can be beneficial to supervised learning algorithm, for example support vector machine (SVM).

4. IMPLEMENETION

5. EXPERIMENT RESULTS

6. CONCLUSION

7. REFERENCES

[1] Autoencoders. http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/. Accessed: 2017-3-3.

[2] KDD Cup 1999 Data. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. Accessed: 2017-3-10.

[3] sklearn.preprocessing.OneHotEncoder. http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html. Accessed: 2017-3-10.

[4] G. Hinton. A practical guide to training restricted boltzmann machines. Technical Report UTML TR-2010-003, Department of Computer Science, University of Toront, 6 King's College Rd, Toronto, August 2010.

[5] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[6] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[7] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 DARPA Off-line Intrusion Detection Evaluation. *Computer Networks*, 34(4):579–595, Oct. 2000.

[8] R. Raina, A. Battle, H. Lee, B. Packer, and A. Y. Ng. Self-taught learning: Transfer learning from unlabeled data. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 759–766, New York, NY, USA, 2007. ACM.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Neurocomputing: Foundations of Research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, Jan. 2014.

[11] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani. A detailed analysis of the KDD CUP 99 data set. In *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, pages 1–6, July 2009.

[12] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.