A Comparative Study of Deep Learning Approaches in Off-Line Network Intrusion Detection

Jiaqi Yan
Department of Computer Science
Illinois Institute of Technology
Chicago, Illinois 60616
Email: jyan31@hawk.iit.edu

Jin Dong
Department of Computer Science
Illinois Institute of Technology
Chicago, Illinois 60616
Email: dong.jin@iit.edu

***Abstract*—[Network intrusion detection is complex and needs to be A.I.] Recently a handful of novel deep neural networks bundled with more advanced and smarter training algorithms have achieved unprecedentedly good performance on image classification, natural language processing, speech recognition and many other research branches. Motivated by these impressive improvements in the field of artificial intelligence, this paper tries to answer the following questions: Can we transfer hall-of-fame deep learning approaches to network intrusion detection task? If yes, how much improvement can be expected? If no, what are the reasons? We answer these questions in three steps. Firstly, we introduce deep learning models and techniques and why they may better solve the network intrusion detection problem from a high-level point of view. Besides, we also survey and discuss the status quo of available network intrusion detection datasets and its implication on applying deep learning models to network intrusion detection. Then we briefly review the existing traditional machine learning approaches, some of which do but some of which don't fit into the category of deep learning. Nevertheless, these existing works provide the state-of-the-art detection performance. In contrast to these approaches, we describe several groups of the cutting-edge deep learning models in concisely mathematical languages. We conduct a quantitatively comparative study of them with the off-line network intrusion detection datasets, on the basis of our own TensorFlow-based deep learning library, NetLearner, that contains multi-layer perceptrons, restricted Boltzmann machine, autoencoders and generative adversarial nets. Apart from making NetLearner publicly available, we also share the hacks and tricks used during the training phase so that future researchers can easily reproduce and extend our work. [To the best of our knowledge, our feed-forward neural network achieves the best 5-classification result, with accuracy of 81.42% and F1-Score of 80.44%.]**

1. Introduction

Network intrusion detection system is the essential security technology that aims to protect a computer network intelligently and automatically. As either a hardware device or software application, it monitors a network for malicious activities or policy violations. By intercepting and analyzing the bi-direction traffics through the network, it raises alarm if intrusion, attack or violation are observed. There are two general approaches to detect intrusions. In signature based intrusion detection, e.g. SNORT [1], rules for specific attacks are pre-installed in the system. It report suspicious traffic when the traffic matches any signature of known attacks. The major drawback of signature matching approach is that it is only effective for previously detected attacks that have an identifiable signature. As a result, signature database needs to be manually updated whenever a new type of attack is discovered, with significant effort, by the network administrator. Anomaly detection based approach overcomes these limitations by adopting a certain type of machine learning technique to model the trustworthy network activities. Traffics that significantly deviates from the built model are treated as malicious. This idea have been shown to be able to detect unknown or novel attacks [2], [3]. However, if the built model for normal traffics are not generalized enough, anomaly based approach will treat unforeseen normal traffic as malicious, suffering from high false positive.

In this project, we follow the anomaly based idea, and tries to enhance it with the state-of-art machine learning technology, e.g. several deep learning architectures. Specifically, we have made the following contributions in this paper. We take the task of offline network intrusion detection for the well-known NSL-KDD dataset [2]. Several deep learning approaches are first investigated and discussed, including multi-layer perceptrons, restricted Boltzmann machine and two types of autoencoders. We then present NetLearner, an implementation of all of the investigated approaches, on the basis of TensorFlow. We not only make the codebase of NetLearner publicly available to research community, but also share the deep learning related hacks and tricks used during the training phase, so that future researches can easily reproduce and extend our work. Finally the detection performance is measured in accuracy, precision, recall and F-Score, with detailed confusion matrix.

2. Deep Learning Background

2.1. Learning/Training Techniques

2.2. Unsupervised Generative Models

The amount of network traffic data is enormously large, usually in the order of terabytes per day in a large monitored network. Such 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 or a group of them to review, e.g., to find patterns and label anomalies. Generative model which can be trained unsupervised comes to rescue in that

- It utilizes the large amount of unlabeled data to learning useful and hierarchical features from the data itself;
- It is equivalently a way to initialize the weights of the hidden layers 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.

2.3. Datasets

3. Existing Works

Due to the large number of network intrusion detection systems that adopt machine learning or data mining approaches we only review some of them that achieved state-of-the-art detection performance. There are very limited number of existing network intrusion detection systems that adopt deep learning approaches. Their details can be found in section 4.

3.1. State-of-Art Machine Learning Approaches

Prior researchers modeled the intrusion detection task as an unsupervised anomaly detection problem, and proposed a series of approaches. Examples include Mahalanobis-distance based outliner detection [4], density-based outliner detection [4], [5], evidence accumulation for ranking outliner [6], etc. One of the advantage of these unsupervised approaches is to tackle the problem of the unavailability of labeled traffic data.

Alternatively, prior researchers made a lot of effort to obtain meaningful attacking data and to convert them into large amount of labeled data [2], [7], [8]. Such efforts make it possible to apply supervised machine learning algorithms to the intrusion detection problem. Successfully applied approaches include decision trees [9], linear and non-linear support vector machines [10], NB-Tree [11] and so on.

3.2. Deep Learning Flavor Approaches

There are some pioneer works that introduced deep learning approaches to intrusion detection. For example, [3] adopts sparse autoencoder and the self-taught learning scheme [12] to handle the problem that there is limited amount of labeled data available for training supervised model. Similar semi-supervised approach have also been applied to Discriminative restricted Boltzmann machine [13].

4. Deep Learning Neural Networks

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

4.1. Multilayer Perceptron

Multilayer perceptron (MLP) is a classic deep learning classifier with simple design of the connectivity between neurons. It is a fully connected feed-forward neural network, as shown in Figure 1. By introducing non-linear neural units (perceptrons), it can distinguish data that are not linearly separable. However, the non-linearity also make it very hard to train a deep MLP of more than three layers, even if people have proposed the efficient back-propagation learning algorithm [14]. Recently it revived due to the various new training techniques designed by deep learning community, including Stochastic Gradient Descent (SGD), batch normalization [15] and Dropout [16]. Except 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 back-propagation:

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

Recently, most deep neural networks adopt rectifier neural unit and achieved very good performance [17]. 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.

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

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

4.2. Restricted Boltzmann Machine

Restricted Boltzmann machine (RBM) [18] is a type of energy-based model, which associate a scalar energy to each configuration vector of the variables in the network. In

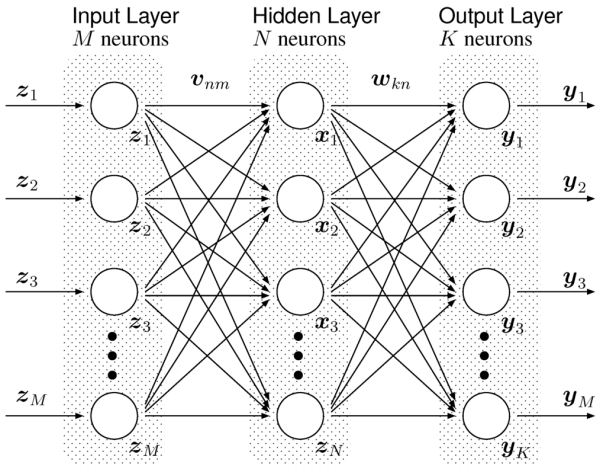

Figure 1: A multilayer perceptron neural network with 1 hidden layer. Figure courtesy of Teijiro Isokawa, Haruhiko Nishimura and Nobuyuki Matsui.

energy-based model, learning is the process of configuring the network weights so that the average energy over training data is 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 makes its training to be faster than Boltzmann machine and makes it feasible to stack multiple separately trained RBM together to form deep architecture. A joint configuration, $(\mathbf{v}, \mathbf{h})$, of the visible and hidden units has the energy of

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

where $a = \{a_i\}$ and $b = \{b_j\}$ are biases in visible and hidden layer respectively, and $W = \{w_{ij}\}$ is the weights between them. The network assigns 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 to raise the probability of a training example and it can be done by adjusting the weights biases to lower the energy of the considered example. Meanwhile, we can let other examples make a big contribution to the partition function Z by raising their energy. Both insights can be translated to the following formula:

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

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 the sampling from the data and it is easy to compute since there is no directed connection between hidden units. The sampling of h_j is based on the probability

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

Similarly, v_i can be sampled with the following distribution

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

The term $\langle v_i h_j \rangle_{model}$ can be obtained by performing alternative Gibbs sampling for a long time. The sampling starts 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. Instead of doing alternating Gibbs sampling for a large number of iterations, [19] proposed contrastive divergence (CD) as a faster learning procedure. The training also start with a training vector to compute the states of the hidden units using Equation 12. Then, with the chosen hidden states, we reconstruct the visible states by sampling each v_i with probability given in Equation 13. The change of weight is then computed by

$$\Delta w_{ij} = \varepsilon(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{reconstruct}) \tag{14}$$

This is called contrastive divergence using one full step of alternating Gibbs sampling. Contrastive divergence with n rounds of alternating Gibbs sampling is usually denoted as CDn.

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 RBM in the second layer and this process can be repeated to learn as many hidden layers as desired. As data passing through the RBMs, we 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.

4.3. 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 identity function, we need to place extra constraints on the

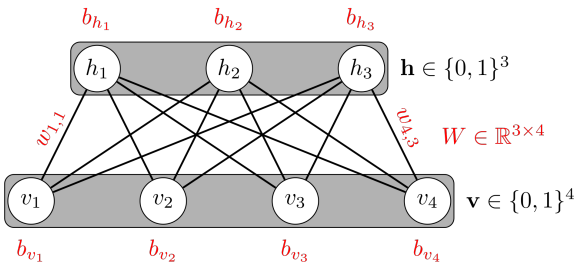

Figure 2: Restricted Boltzmann Machine. Figure courtesy of https://commons.wikimedia.org/wiki/File:Restricted-boltzmann-machine.svg

network, giving birth to different flavors of autoencoders. In this project we consider two most popular types of autoencoder, sparse autoencoder and denoising autoencoder.

The **denoising autoencoder** algorithm is proposed by [20] and illustrated in Figure 4. To prevent learning identity 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 $\mathbf{z} = g'_\theta(\mathbf{y})$. The training needs to learn 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 $\mathbf{x}$.

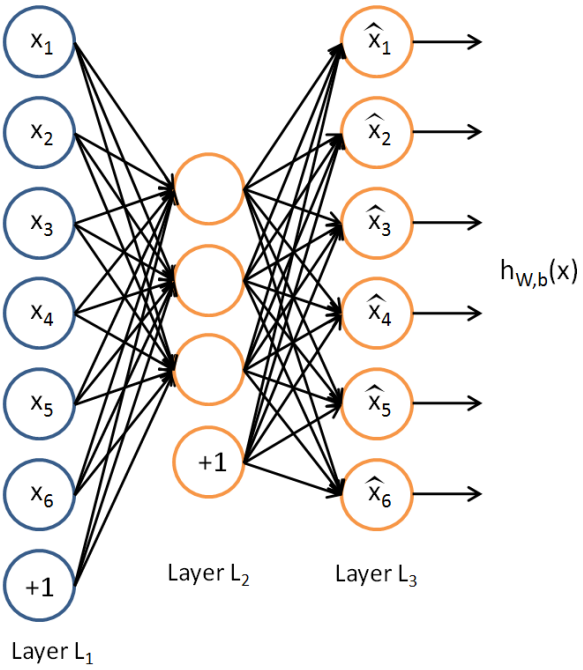

Figure 3: General Architecture of Autoencoders. Figure courtesy of [21].

The **sparse autoencoder** works by placing a sparsity constraint on the hidden units [12]. First, we make the

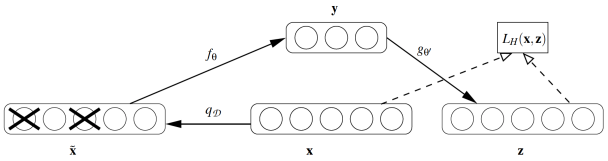

Figure 4: The denoising autoencoder algorithm. Input example $\mathbf{x}$ is randomly corrupted via $q_\mathcal{D}$ and then is mapped via encoder f_θ to $\mathbf{y}$. The decoder g'_θ attempts to reconstruct $\mathbf{x}$ and produces $\mathbf{z}$. Reconstruction error is measured by loss $L_H(\mathbf{x}, \mathbf{z})$, to be minimized during the training phase. Figure courtesy of [20].

autoencoder's hidden layer size to be over-complete, that is, of 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 define the average activation of hidden unit j over the m-size training set

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m} a_j^2(\mathbf{x}) \tag{15}$$

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

$$\hat{\rho}_j = \rho \tag{16}$$

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

$$\sum_j^{n_2} KL(\rho||\hat{\rho}_j) = \sum_j^{n_2}[\rho \log\frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j}] \tag{17}$$

The sparsity penalty term is integrated into the cost function by adding another hyper-parameter β

$$L(W,b) = \frac{1}{2}||h_{W,b}(\mathbf{x}) - \mathbf{x}||^2 + \beta\sum_j^{n_2} KL(\rho||\hat{\rho}_j) \tag{18}$$

Denoising autoencoder and sparse autoencoder, surprisingly, have different application domains. Vincent et al. [20] have shown that stacked denoising 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 denoising autoencoder with MNIST dataset, it is able to re-synthesize a variety of similarly good quality digits. Raina et al. [12] have compared sparse encoding with principle component analysis (PCA) and argue that transferring raw features with a well unsupervised trained sparse autoencoder can be beneficial to supervised learning algorithms, for example support vector machines (SVM).

4.4. Generative Adversarial Nets

As another generative model, Generative Adversarial Nets (GAN) [22] adopts a novel training framework, in

which two models are trained simultaneously and adversarially. The generative model $G(z;\theta_g)$ aims to capture the probability distribution of the available unlabelled dataset, where its input is a noise variable z following a prior distribution p_z. The discriminative model $D(x;\theta_d)$ output the probability that whether the its input source S comes from training dataset ($x \sim data$) or the generative model ($x \sim G(z)$):

$$D(X) = P(S|X) \tag{19}$$

Models G and D can be as simple as multilayer perceptrons, or as complex as deep convolutional nets when training dataset is images. The two models are trained in opposition to one another, with respect to the log-likelihood function

$$\begin{aligned} V(D,G) =& \mathbb{E}_{\boldsymbol{x}\sim data}[\log P(S = real|X = \boldsymbol{x})] + \\ & \mathbb{E}_{\boldsymbol{x}\sim G(\boldsymbol{z})}[\log P(S = fake|X = \boldsymbol{x})] \\ =& \mathbb{E}[\log D(\boldsymbol{x})] + \mathbb{E}[\log(1 - D(G(\boldsymbol{z})))] \end{aligned} \tag{20}$$

With $V(D,G)$ properly defined, the training procedure is a two-player minimax game. First we make D maximize the log-likelihood that it correctly recognize both the training examples and samples generated from G; at the same time, we make G to generate samples that trick D to make mistake about the generated samples. This two-phase minmax optimization can be summarized as:

$$\min_G \max_D V(D,G) \tag{21}$$

Powerful though GAN is, large amount of efforts are needed in training. One way to make the training stable and fast is to augment an auxiliary classifier so that the training phase can employ the labels if available in the dataset [23]. In auxiliary classifier GAN (AC-GAN), the discriminator D now gives both a probability distribution over the sources (whether $\boldsymbol{x}$ is real or fake) and a probability distribution over the class labels:

$$D(X) = P(S|X), P(C|X) \tag{22}$$

Accordingly, the log-likelihood function $V(D,G)$ is augmented with the log-likelihood of the correct class:

$$\begin{aligned} V(D,G) &= L_S + L_C \\ L_S =& \mathbb{E}_{\boldsymbol{x}\sim data}[\log P(S = real|X = \boldsymbol{x})] + \\ & \mathbb{E}_{\boldsymbol{x}\sim G(\boldsymbol{z})}[\log P(S = fake|X = \boldsymbol{x})] \\ L_C =& \mathbb{E}_{\boldsymbol{x}\sim data}[\log P(C = c|X = \boldsymbol{x})] + \\ & \mathbb{E}_{\boldsymbol{x}\sim G(\boldsymbol{z})}[\log P(C = c|X = \boldsymbol{x})] \end{aligned} \tag{23}$$

The training procedure for AC-GAN is similar to GAN: we train D to maximize $V(D,G)$; while at the same time we train G to minimize $L_S - L_C$.

4.5. Wide and Deep Learning with Embeddings

5. Implementation

5.1. TensorFlow 101

TensorFlow [24] is an open-source software library for machine learning developed by the Google Brain Team. The library models the computations in machine learning as data flow graphs. Multidimensional data arrays are called tensors in TensorFlow. Nodes in the graph represent mathematical operations between tensors, such as add, multiply, softmax and dropout. Graph edges represent the flow of tensors between nodes. The computation graph based architecture allow researchers to run or train neural networks on one or more CPUs or GPUs with unified API.

For general classification task, input X and label Y are defined as **placeholder**s and feed into the computation graph at running time using a dictionary. The graph of a typical deep learning model have three parts. The **inference** graph should be built so that output predictions are returned as tensor. For example, in the multilayer perceptron case, inference graph contains all iterative computation 4-5 in the feed-forwarding steps. The **loss** graph should compute the loss function defined by specific models or algorithms. Usually it is either cross-entropy or mean-squared error averaged across the batch data. The loss graph will be optimized, usually minimized, by the **train** part. This optimization can be conducted by various optimizing algorithms, such as gradient descent, Momentum, RMSProp. After sufficient steps of batch training, we evaluate the trained model with inference graph and compare the predictions with the test dataset labels. TensorFlow also provides various useful utilities for training models and running experiments. Using a Saver, we are able to checkpoint the training process so as to restoring the model for further training or evaluation. Users create Summary nodes to log the snapshot of interest variables, which can be automatically visualized by TensorBoard.

5.2. NetLearner

We provide a Python library NetLearner [25] that wraps up several deep learning models on the basis of TensorFlow. NetLearner modularizes multilayer perceptron, restricted Boltzmann machine, sparse autoencoder and masking-noise autoencoder, all of which are used to perform the 5-class classification on the NSL-KDD dataset.

5.3. Hacks and Tricks

For the multiple layer perceptron, we tried a 4-hidden-layer network with very wide size in each layer, several hundreds for each layer. The accuracy on training set is very exiting, usually more than 96%. However, its performance on test dataset is not satisfactory. Instead we found out that a single hidden layer with only sixteen neurons has good accuracy. It is trained with stochastic gradient descent (SGD) for 20 epochs and batch size 100. During the training, learning rate decays from 0.1 exponentially with the base of 0.32. We did not include regularization in the model, but did apply dropout of keep probability 0.8. We denote this approach as MLP and show its detailed results in the later section.

We build a RBM with 200-hidden units to perform unsupervised feature learning first on the dataset. The learned

features are then fed into a simple softmax regression classifier. We trained the RBM using CD1 (contrastive divergence using one full step to get the negative data), with batch size 10 for 40 epochs. The learning rate is initialized at 0.01 and decay exponentially with the base of 0.64. We denote this combination of RBM and softmax regression as RBM in the later section.

We also implemented the self-taught learning architecture proposed in [3], [12], adopting sparse autoencoder as the unsupervised feature learner. The learned features will then be used for classification by a Softmax regression classifier. We contact the author of [3] so that we can reproduce their implementation with the same hyper-parameters. For example, the hidden layer size of the sparse autoencoder is 64; the sparsity value ρ is 0.25. Different from [3], we found that using regularization in neither autoencoder nor softmax regression is helpful. So we didn't include regularization term in the both autoencoder and softmax cost function. The autoencoder is trained with SGD for 1000 epochs and batch size 5000. Different from MLP, we used Adam optimizer during the training. The learning rate starts at 0.01 and decay exponentially with base of 0.6. We denote this approach as SAE and report its performance in the later section.

As a variation to the sparse autoencoder based self-taught learning architecture, we explore what will the performance be if we replace sparse autoencoder with denosing autoencoder. We simply use dropout to emulate the masking noise and build masking noise autoencoder, in which input is randomly masked out with keep probability of 0.4. The size of the denoising autoencoder is 100. The autoencoder is trained with SGD for 1000 epochs and batch size 5000. We trained denoising autoencoder in the same way as we trained sparse autoencoder. The result of this approach is labeled as DAE in the later section.

One thing to notice is that we use the same seed to randomly initialized the weights and biases of the softmax regression classifier such that the learned features from RBM, sparse autoencoder and denoising autoencoder are comparable. For the same reason, all the softmax regression classifiers used by RBM, sparse autoencoder and denoising autoencoder are trained with Adam optimizer of batch size 100 for 100 epochs, with exponentially decay learning rate starting at 0.01, with dropout technique of keep probability 0.8.

6. Experiment Results

6.1. Dataset and Preprocessing

Among various available datasets [2], [7], [8], we choose NSL-KDD dataset [2] to evaluate the performance of applying various neural networks in the network intrusion detection. NSL-KDD dataset originates from the KDDCup 99 dataset [8], which was used for the third International Knowledge Discovery and Data Mining Tool Competition. 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 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 evaluating 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 additional 14 types of attacks in the test dataset, intentionally 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, also known as 5-class classification problem. Our data preprocessing starts with converting symbolic features and the labels to one-hot encoding format. For example, if feature f can take n possible values from 1 to n, a feature of value x will be converted to a n-dimensional binary vector with the xth dimension set to 1 and others set to zero. 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].

6.2. Evaluation Metrics

We evaluate the classification performance of our proposed deep learning approaches with the following metrics.

- **Accuracy** is the percentage of correctly classified connections over the total number of connections in the dataset:

$$A = \frac{\text{Correct Predictions}}{\text{Number of Records}} \tag{24}$$

  Accuracy is not suitable for evaluating biased dataset where the number of records of some class is extremely larger than the number of records of another class. In NSL-KDD dataset, the number of available U2R records (67) is in two degrees of magnitude less than the other classes of traffic (9711, 7458, 2887, 2121 respectively). Therefore we also consider the following metrics.
- **Precision** is the percentage of the correctly classified positives over the total number of positives predicted by the classifier:

$$P = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{25}$$

- **Recall** is the percentage of the correctly classified positives over the total number of relevant elements:

$$R = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{26}$$

- **F1-Score** represents a balance between precision and recall and is calculated as their harmonic mean:

$$F = \frac{2PR}{P + R} \tag{27}$$

In the 5-class classification, we calculate the precision, recall and F1-Score for each traffic class. Additionally, we report the weighted average of these metrics as a single value for comparing various approaches. The weight for each class is determined by its proportion in the test dataset. The weight vector for class [Normal, Probe, DoS, U2R, R2L] is [0.431, 0.107, 0.339, 0.018, 0.105]. Besides, we also provide the confusion matrix of the classification results when applying different approaches on the test dataset. In our confusion matrix table, the row represents the instance in an actual class, while the column represents the instance in a predicted class. It is called confusion matrix because it is useful for visualizing how a classifier is confusing one class with other classes.

6.3. Performance of Deep Learning Approaches

First we report the classification accuracy of each considered approach in Figure 5. Surprisingly, the most "accurate" approach is the simple 16-neuron perceptron (81.42%). Sparse autoencoder based self-taught leaner achieved second best accuracy of 79.15%. This number coincides with the previously reported results in [3] (79.10%). RBM and denoising autoencoder have similar accuracy results (77.58% and 76.93%).

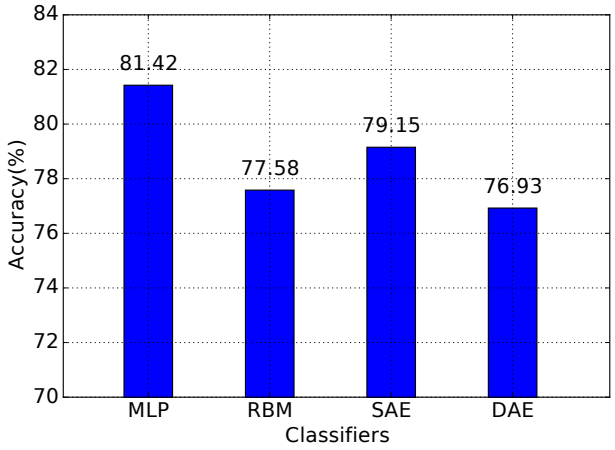

Figure 5: Classification Accuracy of Proposed Approaches

Table 1 – 4 summarize the confusion matrices of each approach and their weighted average metrics (Precision, Recall and F1-Score). Apart from best accuracy, MLP also

TABLE 1: Confusion Matrix of MLP on Test Dataset

		Prediction				
		Normal	Probe	DoS	U2R	R2L
Actual	Normal	9329	230	70	25	57
	Probe	164	1914	271	10	66
	DoS	1358	82	6146	47	3
	U2R	345	4	0	41	6
	R2L	1247	30	2	171	926
Precision(%)		74.97	84.69	94.71	13.95	87.52
Wtd. Avg.(%)						82.96
Recall(%)		96.07	78.93	80.49	10.35	38.97
Wtd. Avg.(%)						81.42
F1-Score(%)		84.22	81.71	87.02	11.88	53.93
Wtd. Avg.(%)						**80.44**

TABLE 2: Confusion Matrix of RBM on Test Dataset

		Prediction				
		Normal	Probe	DoS	U2R	R2L
Actual	Normal	8903	318	428	11	51
	Probe	232	2015	159	2	17
	DoS	1879	143	5613	0	1
	U2R	356	3	1	27	9
	R2L	1550	8	1	8	809
Precision(%)		68.91	81.02	90.50	56.25	91.21
Wtd. Avg.(%)						79.65
Recall(%)		91.68	83.09	73.51	6.82	34.05
Wtd. Avg.(%)						77.04
F1-Score(%)		78.68	82.04	81.12	12.16	49.59
Wtd. Avg.(%)						75.63

achieved the best F1-Score (80.44%) among all of the considered approaches. However, we can see that for U2R attacks, MLP still has very poor results of both precision (13.95%) and recall (10.35%). The second highest F1-Score is achieved by sparse autoencoder combined with softmax regression (78.05%). This value is actually a little bit higher than the result (75.76%) reported in [3]. We believe this is partly due to the reason that we did not introduce regularization for both sparse autoencoder and softmax regression classifier, and partly due to the dropout technique we used in training the softmax regression classifier. RBM and DAE again achieve very similar classification performance, with F1-Score of 75.63% and 75.65% respectively. Considering both the high accuracy and best F1-Score, we conclude that in the competition of 5-class classification, MLP is the winner.

Confusion matrices here tell us something interesting about the classification performance for each type of traffics. MLP correctly recognized the most number of normal traffics (9329 out of 9711). For the attacking traffics, the winner classifier MLP correctly predicted the most number of DoS attacks (6146 out of 7636), U2R attacks (41 out of 396) and R2L attacks (926 out of 2376). RBM is the best classifier in predicting Probe attacks (2015 out of 2425).

7. Conclusion

In this project we conducted a comparative study on the deep learning approaches for the network intrusion detection problem. We take the off-line network intrusion detection dataset NSL-KDD for evaluation. In this paper, multilayer

TABLE 3: Confusion Matrix of SAE on Test Dataset

		Prediction				
		Normal	Probe	DoS	U2R	R2L
Actual	Normal	8864	696	92	11	48
	Probe	179	2001	164	2	79
	DoS	1542	39	6054	0	1
	U2R	357	1	1	30	7
	R2L	1444	6	5	26	895
Precision(%)		71.56	72.95	95.85	43.48	86.89
Wtd. Avg.(%)						81.06
Recall(%)		91.28	82.52	79.28	7.58	37.67
Wtd. Avg.(%)						79.15
F1-Score(%)		80.23	77.44	86.78	12.90	52.55
Wtd. Avg.(%)						78.05

TABLE 4: Confusion Matrix of DAE on Test Dataset

		Prediction				
		Normal	Probe	DoS	U2R	R2L
Actual	Normal	9249	319	85	10	48
	Probe	576	1504	226	2	117
	DoS	1842	128	5665	0	1
	U2R	353	1	0	38	4
	R2L	1469	3	1	17	886
Precision(%)		68.57	76.93	94.78	56.72	83.90
Wtd. Avg.(%)						79.75
Recall(%)		95.24	62.02	74.19	9.60	37.29
Wtd. Avg.(%)						76.93
F1-Score(%)		79.73	68.68	83.23	16.41	51.63
Wtd. Avg.(%)						75.65

perceptron, restricted Boltzmann machine, sparse autoencoder and denoising autoencoder are briefly described. The main contribution lies on sharing the hacks and tricks used in training these neural networks and the results of comparable evaluation of them. From our experiment results, we conclude that for the NSL-KDD test dataset, multilayer perceptron has relatively best performance since both its accuracy and F1-Score are outstanding among compared neural networks.

Acknowledgment

The authors would like to thank...

References

[1] "SNORT," https://www.snort.org/, accessed: 2017-4-15.

[2] 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*, July 2009, pp. 1–6.

[3] A. Javaid, Q. Niyaz, W. Sun, and M. Alam, "A deep learning approach for network intrusion detection system," in *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies, New York, NY, USA*, vol. 35, 2015, pp. 2126–2132.

[4] A. Lazarevic, L. Ertoz, V. Kumar, A. Ozgur, and J. Srivastava, *A Comparative Study of Anomaly Detection Schemes in Network Intrusion Detection*, pp. 25–36.

[5] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: Identifying density-based local outliers," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00. New York, NY, USA: ACM, 2000, pp. 93–104.

[6] P. Casas, J. Mazel, and P. Owezarski, "Unsupervised network intrusion detection systems: Detecting the unknown without knowledge," *Computer Communications*, vol. 35, no. 7, pp. 772 – 783, 2012.

[7] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das, "The 1999 DARPA Off-line Intrusion Detection Evaluation," *Computer Networks*, vol. 34, no. 4, pp. 579–595, Oct. 2000.

[8] "KDD Cup 1999 Data," http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, accessed: 2017-3-10.

[9] J. R. Quinlan, *Learning Efficient Classification Procedures and Their Application to Chess End Games*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 463–482.

[10] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep 1995.

[11] R. Kohavi, "Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96. AAAI Press, 1996, pp. 202–207.

[12] 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*, ser. ICML '07. New York, NY, USA: ACM, 2007, pp. 759–766.

[13] U. Fiore, F. Palmieri, A. Castiglione, and A. D. Santis, "Network anomaly detection with the restricted boltzmann machine," *Neurocomputing*, vol. 122, pp. 13–23, 2013.

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Neurocomputing: Foundations of Research," J. A. Anderson and E. Rosenfeld, Eds. Cambridge, MA, USA: MIT Press, 1988, ch. Learning Representations by Back-propagating Errors, pp. 696–699.

[15] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[16] 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.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014.

[17] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

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

[19] G. E. Hinton, "Training products of experts by minimizing contrastive divergence," *Neural Comput.*, vol. 14, no. 8, pp. 1771–1800, Aug. 2002.

[20] 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*, vol. 11, no. Dec, pp. 3371–3408, 2010.

[21] "Autoencoders," http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/, accessed: 2017-3-3.

[22] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," *ArXiv e-prints*, Jun. 2014.

[23] A. Odena, C. Olah, and J. Shlens, "Conditional Image Synthesis With Auxiliary Classifier GANs," *ArXiv e-prints*, Oct. 2016.

[24] "TensorFlow: An open-source software library for Machine Intelligence," https://www.tensorflow.org/, accessed: 2017-4-15.

[25] "NetLearner," https://github.com/littlepretty/NetLearner, accessed: 2017-7-14.