SAND2019-5702C

Packet2Vec: Utilizing Word2Vec for Feature Extraction in Packet Data

Eric L. Goodman, Chase Zimmerman, and Corey Hudson

Sandia National Laboratories, Albuquerque, NM, USA

Abstract. One of deep learning's attractive benefits is the ability to automatically extract relevant features for a target problem from largely raw data, instead of utilizing human engineered and error prone hand-crafted features. While deep learning has shown success in fields such as image classification and natural language processing, its application for feature extraction on raw network packet data for intrusion detection is largely unexplored. In this paper we modify a Word2Vec approach, used for text processing, and apply it to packet data for automatic feature extraction. We call this approach *Packet2Vec*. For the classification task of benign versus malicious traffic on a 2009 DARPA network data set, we obtain an area under the curve (AUC) of the receiver operating characteristic (ROC) between 0.988-0.996 and an AUC of the Precision/Recall curve between 0.604-0.667.

1 Introduction

An appealing aspect of many deep learning approaches is the ability to automatically extract features from largely unprocessed data. In Krizhevsky et al. [13], one of the seminal works that started the popularization of convolutional neural networks applied to images, they show that the learned early convolutional kernels displayed a range of image filters, similar to hand-crafted features from more traditional vision processing approaches such as SIFT [15] and SURF [4].

For text processing, Word2Vec approaches [17, 18] create a vectorized representation of words, called embeddings, where similar words (e.g. King and Queen) are close distance-wise in the embedded space. Vector operations also make intuitive sense, such as *King - Man + Woman = Queen*, meaning that the vector representation of *King* minus the vector for *Man* plus the vector for *Woman* creates a vector where the closest word embedding is the one for *Queen*. This feat is achieved on a large corpus of raw text with little to no-preprocessing. The deep learning approach is able to create these word embeddings just based on the text itself without human-engineered feature extraction.

Cyber data and intrusion detection is an area ripe for exploration of how deep learning can automatically extract features from raw packet data. However, most of the current work applying deep learning to intrustion detection relies upon the features already being extracted from packet data [12, 27]. Many researchers choose to use data sets such as NSL-KDD [24, 26] or the original 1999 KDD data set, both of which have 41 features to represent the network packet data.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Instead of creating hand-crafted features for each packet, the approach we take is to pass the raw packet data through a Word2Vec approach to create a vectorized representation for each packet, and then perform classification of the packet based on that representation.

Specifically, our approach has the following steps:

- **N-grams:** Word2Vec requires a sequence of tokens. Packet data has no clear analog. To address this, we take each packet and transform it into a sequence of n-grams. This forms our sequence of words, similar to the presentation of text. We purposefully throw out IP and port information, as we want the representation of the packet to be based on content, not who sent it.
- **Embeddings:** Once we have a sequence of n-grams, applying Word2Vec is straightforward, and we create a vectorized representation for each frequent n-gram (vocab size is a hyperparameter).
- **Feature Vectors:** To perform classification on each packet, we need a fixed-size vector representation for each packet. We take the simple approach of averaging the word embeddings for all of the n-grams in a packet, i.e.

$$v(p) = \frac{\sum_{t \in p} e(t)}{|p|} \tag{1}$$

  where p is a packet, $t \in p$ are the n-grams of p, $|p|$ is the number of n-grams found in p, and $e(t)$ returns the embedding for n-gram t.
- **Learning and Classification:** Once we have each packet translated into fixed-size feature vectors, we then pass those feature vectors to a supervised machine learning approach for training and then testing on unseen data.

Intrusion detection is an important area of research, vital for protection of national infrastructures, intellectual property, financial systems, privacy, and safety; however, the problem is a moving target, an arms race between defenders and attackers, along with constant evolution of the underlying technologies. There is evidence of growing sophistication among malicious actors. Symantec reports that the number of targetted attack groups, i.e. groups that are professional, highly organized, and target specifically rather than indiscriminately, grew at a rate of 29 groups a year between the years of 2015 to 2017, from a total of 87 to 140 [25]. Also, as evidence of constant change in the cyber arena, the number of IoT (Internet of Things) attacks grew by 600%, an increase of 54% of mobile malware variants, and an 80% increase in Mac malware.

We view our contribution as a way to increase the rate that defenders can evolve their methods to protect networks and infrastructure. Instead of manually hand-crafting features, which is error prone and difficult to determine impact, we can rely upon our Packet2Vec approach to automatically calculate features of interest.

The rest of this paper is organized as follows: Section 2 describes our approach in detail, including the steps we took to parallelize our solution. Section 3 presents the results of using our approach on a large cyber data set. Section 4 covers related work. Section 5 concludes.

2 Approach

In the introduction, we presented our approach at a high-level. However, applying Word2Vec on cyber data is challenging due to amount of information. In particular, we examined the DARPA 2009 data set [10]. This data set spans a period of 10 days, from November 3rd to November 12th, 2009. It is broken up into files that are just over 1 billion bytes (954 MBs), where each file represents 1-6 minutes worth of traffic. In this work we examined the first day, which is roughly 15.5 hours (it starts after 8:30 am) and comprises 558.8 GBs in total packet data. Due to the size of the data, we needed to create an iterative process for training our model.

Our solution is a combination of C++ code that is then exposed to python using Boost python [2]. We developed most of our implementation in C++ for performance, but then exposed it to python so that we could integrate with the Tensorflow library [1] for creating the embeddings for the n-grams, and the Scikit-learn library [6] for the classifier models to make predictions on whether the packets are benign or malicious. We also took efforts to parallelize the code using standard C++ features such as std::thread to manually instrument the code. As we discuss the implementation, we will highlight the parallelization. Also, in Section 3.1, we will discuss the parallel performance of the code.

Figure 1 gives an overview of the iterative approach. The first phase (pseudocode found in Algorithm 1), creates a dictionary, mapping n-grams to integer identifiers. The first phase begins by iterating through all pcap files used for training, n-gramming each packet, and incrementing the counter for each n-gram. After obtaining counts for each n-gram found in all the training files, identifiers are assigned for the top $|V|$ n-grams, where $|V|$ is the size of the vocabulary, a hyperparameter. Concerning memory utilization, we only load one pcap file at a time. Also, the dictionary is limited by the number of found n-grams. We used 2 byte n-grams, which at most has 2^{16} possible values.

The actual implementation of Algorithm 1 is a bit more nuanced as we structured it in such a way to enable parallelization. We first iterate in parallel over all packets and n-gram them. This is embarrassingly parallel and requires no inter-thread coordination. The end result is a vector of vector of n-grams. Then we flatten the vector of vector of n-grams into a single vector of n-grams, again in parallel. Finally, we hand the single vector of all n-grams to the dictionary, which updates the frequency counts for each n-gram. This is the only loop that requires coordination between threads, as two threads can potentially try to update the count for the same n-gram; however, adding mutexes around the update routine makes it thread safe. After all files have been processed, we also parallelize the implementation of lines 15 - 18. We need the dictionary for later phases, so we write it out to disk on line 19.

The second phase (Algorithm 2) utilizes the dictionary created from the first phase to translate the pcap files into integers. We iterate through each pcap file (line 1), creating two data structures for each pcap file. One data structure is a list of integers (line 2), which is the pcap file translated into integers using the dictionary. There is also a vector of vector of integers (line 3), which is the

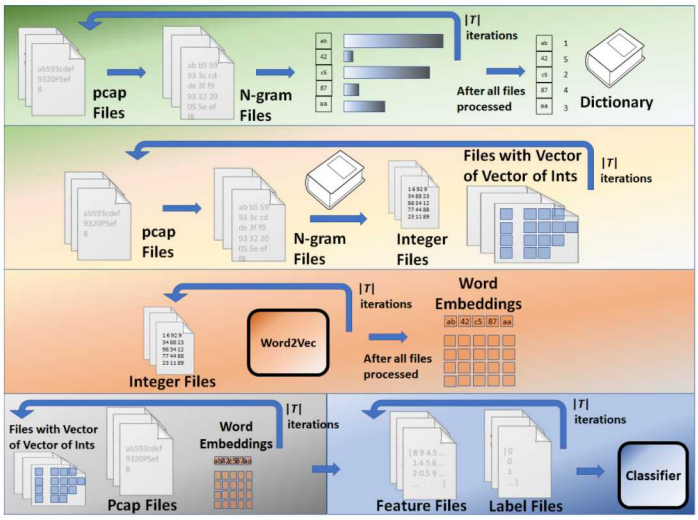

Fig. 1: Implementation of iterative pcap processing approach. The first phase creates a dictionary, mapping n-grams to integer identifiers. The dictionary is utilized in the second phase to transform the raw pcap data into integer vectors which are saved on disk. In the third phase, a Word2Vec approach is applied to the 1D integer vectors to create the n-gram embeddings. These embeddings are used in conjunction with the 2D integer vectors to create feature vectors (fourth phase) which are then used for training in the final phase.

Algorithm 1 Training Phase 1: Creating the Dictionary

T, the set of pcap files used for training.
D, a dictionary mapping from n-grams to integers.
$|V|$, the size of the vocabulary.
for all $f \in T$ **do**
    for all $packet \in f$ **do**
        $ngrams \leftarrow ngram(packet)$
        for all $ngram \in ngrams$ **do**
            $D[ngram] \leftarrow D[ngram] + 1$
        end for
    end for
end for
$keys \leftarrow sort(D)$ ▷ Keys sorted by decreasing frequency
$D.clear()$ ▷ Clear out counts
$i \leftarrow 1$
while $i \leq |V|$ **do**
    $D[key[i]] \leftarrow i$
    $i \leftarrow i + 1$
end while
$write(D)$

same as the integer list, but now indexed by packet. After processing a pcap file, we write out the list of integers (line 13) and the vector of vector of integers to disk (line 14). This again allows us to process all of the large pcap files without exceeding memory limits. We also parallelize the for loop of line 1. Each of the packets can be handled independently, so it is embarrasingly parallel.

The third phase is where we create the word embeddings, i.e. vectorized representations for each n-gram in the vocabulary. The process is described in Algorithm 3 in high level pseudocode. We iterate over all the integer files (pcap files translated by the dictionary into a single sequence of integers). On the first iteration we create an embedding model based on the first integer file using a standard word2vec approach. This creates a matrix of size $|V| \times embedding\_size$, where each row corresponds to the learned vector representation of an n-gram. This first embedding matrix serves as the starting point for the next iteration of applying word2vec to another integer file. We continue in this manner until all integer files have been processed.

Algorithm 2 Training Phase 2: Translating Pcap Files

```
1: for all f ∈ T do
2:     l, a list of integers
3:     vv, a 2D vector of integers
4:     for all packet ∈ f do
5:         v, a vector of integers
6:         ngrams ← ngram(packet)
7:         for all ng ∈ ngrams do
8:             l.push_back(D[ng])
9:             v.push_back(D[ng])
10:        end for
11:        vv.push_back(v)
12:    end for
13:    write(l)
14:    write(vv)
15: end for
```

Algorithm 3 Training Phase 3: Creating Word Embeddings

```
1: L, the set of files with lists of integers
2: first_run ← True
3: for all l ∈ L do
4:     if first_run then
5:         E ← create_model(l)
6:         first_run = False
7:     else
8:         E ← update_model(l, E)
9:     end if
10: end for
11: write(E)
```

The method for training the model is a standard word2vec approach. We use the skip-gram model [18] with noise constrastive estimation [11]. The basis of this approach is for the network to predict the context given a target word. However, with noise constrastive estimation, it becomes a logistic regression problem where the network is making a binary classification for each word in the vocabulary of whether or not it came the distribution of context words or from the noise distribution (unrelated words). The hyperparameters associated with this approach include the following. In paranthesis we specify the value we used in our experiments. *Batch size* (128): The number of words considered at one time. *Skip window* (1): How big of a context window to consider. A value of one selects words to the left and right of the target word. *Num skips* (2): The *batch size* is divided by *num skips* to determine the number of skip windows. *Embedding size* (128): The size of each embedding vector. *Num negative* (64):

The number of negative examples used per batch. *Num steps* (100000): How many batches to create and from which to train.

The fourth phase utilizes the word embeddings in conjunction with the two dimensional integer vectors to create the feature files. Each feature file is a matrix where each row represents the features derived for a packet. On line 3 we iterate over the two dimensional integer vector files, VV. On line 6 we iterate over each vector, v, within the two dimensional integer vector, vv. v is a vector of integers, representing the n-grams of the original packet translated using D, the dictionary from Algorithm 1. To create a single representation for the entire packet, we use the simple strategy of averaging the embeddings (lines 9 - 12). In the end, we write out each feature matrix, X, to disk (line 15).

There is also another process for producing labels for the data. The DARPA-2009 dataset has a spreadsheet with labels; however, the labeling is not at the individual packet level. It lists times, IP addresses, and ports used by malicious traffic. Thus, to create labels, we read in the original pcap files and evaluate each packet, checking if the parameters of the packet match those of an entry in the label spreadsheet.

Algorithm 4 Training Phase 4: Create Feature Vectors

```
1: VV, set of files with 2D vector of integers
2: E, the word embeddings indexed by integer identifier
3: for i ← 1 to |VV| do
4:     vv ← VV[i]
5:     X, a matrix of features
6:     for j ← 1 to |vv| do
7:         v ← vv[j]
8:         x, a vector of features
9:         for all integer ∈ v do
10:            x ← x + E[integer]
11:        end for
12:        x ← x/|v|
13:        X[j] ← x
14:    end for
15:    write(X)
16: end for
```

Algorithm 5 Training Phase 5: Train Classifier

```
1: X_files, the list of feature files.
2: y_files, the list of label files.
3: n_est, the number estimators per file.
4: rfc ← RFC(warm_start = True, n_est)
5: i ← 0
6: for i ← 1 to |X_files| do
7:     X ← X_files[i]
8:     if X has positive then
9:         if i ≠ 0 then
10:            rfc.n_est += n_est
11:        end if
12:        y ← y_files[i]
13:        rfc.fit(X, y)
14:    end if
15: end for
16: write(rfc)
```

The last phase of training is to train an actual classifier. After phase 4, we finally have the data in a format that can be ingested by a standard machine learning algorithm. We have a set of files that contain the feature vectors for each packet, and we have another corresponding set of files that have a binary label indicating a benign/malicious packet. Algorithm 5 outlines the iterative approach to learning. In particular we show pseudocode related to the Random Forest Classifier [5], but it can be easily generalized to other machine learning algorithms. An important point to note here is the *warm_start* parameter on line

4. Since we are training in batches over many files, we need to maintain what was learned from earlier files. The *warm_start* parameter of Scikit-learn [6] is used when multiple calls to the *fit* function are used. In the case of the Random Forest Classifier, a number of estimators (trees) are created per file. However, this doesn't work if a file does not contain any malicious examples. On line 8 we skip any files that do not have malicious packets. What *warm_start* means differs depending on the classifier used. For example, with neural networks we would initialize the model with the weights learned from training on previous files.

3 Results

In this section we discuss two aspects of performance: 1) the throughput achieved when applying a trained classifier, and 2) the classifier performance in detecting malicious network activity. The system we used for our experiments was a DGX [22], a supercomputer designed for accelerating deep learning applications with powerful GPUs. However, except for the Packet2Vec portion that creates embeddings, our code primarily uses the CPU. The CPU is a dual Intel Xeon 20-core E5-2698 v4 2.2 GHz processor with 512 GBs 2133 MHz DDR4 memory. There is some variability to the timing of runs as other users are also using the system concurrently.

We tested our implmentation on the DARPA-2009 data set [10]. DARPA-2009 is a generated data set covering a period of time from November 3-10, 2009. Traffic is simulated between a /16 local subnet that goes through a cisco router to the Internet. There are a variety of protocols (e.g. HTTP, SMTP, DNS) and malicious activities (e.g. DDoS, Phishing, port scans, spam bots). For this work, we treat all the malicious categories as single class so the problem is binary classification: malicious or benign. We evaluated our approach on the first day's worth of data (about 15.5 hours because the data starts around 8:30 am). In total for the first day there are 600 pcap files, each 1 billion bytes (954 MBs). Groundtruth labels are provided in the form a spreadsheet specifying the IPs, ports, and a bounding time window of when an attack occurred. For the portion we used, malicious activity accounted for 0.46% of the the total packets.

3.1 Processing Time

In this section we report on the processing time for applying a trained classifier on unseen data. It is important that our approach be able to keep pace with data creation. While application of a trained machine learning model is generally not a concern - testing is often orders of magnitude faster than training - our approach does have significant preprocessing steps. To classify unseen data, we need the following as input: 1) a pcap file, 2) the dictionary from n-grams to integers (created during Algorithm 1 and written to disk on line 19), 3) the n-gram embeddings (created from Algorithm 3 and written to disk on line 11),

and 4) the trained classifier (created during Algorithm 5 and written to disk on line 16).

The overall process of applying a trained classifier to unseen data is described below. We will make note of which portions are serial, serial but could be parallelized, and already parallelized.

1. Read pcap object: We read in a pcap object. Unless there is parallel I/O, this is largely a serial operation and cannot be parallelized.
2. N-gram the packets: For each of the packets in the pcap object, we n-gram them. This step has been parallelized.
3. Translate the n-grams into integers: Using the dictionary, we translate each vector of n-grams into a vector of integers. This step has been parallelized.
4. Create the feature matrix: This step takes the translated packet data of integers and converts them into embedding vectors, averages the embeddings, and then fills a matrix that has all the feature vectors. This step should be parallelizable but since we use a python object within C++ as the feature matrix, we run into issues with the Python global interpreter lock only allowing one thread. This should be surmountable, but will require a deeper dive into Boost python [2] and the NumPy C-API, which is C-based API for manipulating NumPy data structures (the feature matrix is a *NumPy.ndarray*).
5. C++ to python overhead: The function to create the feature matrix is written in C++ but we added a python interface. The python function reports on average 13.6 seconds more than the corresponding C++ implementation. We hypothesize this may be due to memory transfer costs. Regardless, this will be difficult to optimize without a deep exploration into Boost python.
6. Making predictions on the feature matrix: Here we apply the trained classifier to the now prepared feature matrix. We use the Scikit-learn library [6] for the machine learning models. This step could also be parallelized using one of the python libraries for parallel execution, but we have not taken that step yet.

To evaluate the parallel performance of the pipeline to apply a trained classifier to unseen data, we trained a Random Forest Classifier [5] on one pcap file and then tested it on another pcap, varying the number of threads. Figure 2 gives the overall time while Figure 3 provides the relative speedup as we increase the thread count. As expected, the parallel portion's total time decreases as we increase the number of threads, though the overall speedup plateaus around 10 threads.

Since we have good understanding of which portions of the program are parallel and which are serial, using Amdahl's law we can estimate the maximum achievable speedup: $Speedup(t) = \frac{1}{(1-p)+\frac{p}{t}}$, where we can think of t as the number of cores applied to the program and p is the proportion of the code that benefits from parallel execution. As $t \to \infty$, the equation becomes just $Speedup(t) = \frac{1}{1-p}$. Table 1 shows the maximum theoretical speedup based upon the times from using one thread. The *Current* row shows the times for the parallel and serial portions for our current implementation. Based on those numbers,

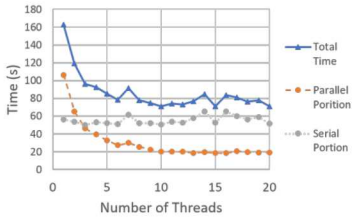

Fig. 2: Time for Testing One File: We apply a trained Random Forest Classifier to unseen data and report the times. The portion of the code that has been parallelized shows improvement up to ten threads.

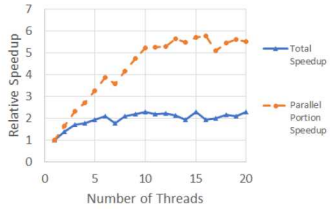

Fig. 3: Relative Speedup: Same data as Figure 2 but now showing relative speedup of the overall testing phase and the parallel portion.

our maximum speedup is about 2.9. Experimentally, we achieved a 2.3 speedup with ten threads. If we parallelized steps four and six, which certainly seems possible, then the maximum speedup is close to 9.2. Of course this is only single node speedup, and we can obtain greater aggregate throughput on a distributed system. If pcap data is ingested on multiple nodes, the task of classifying network traffic is embarrassingly parallel once the dictionary, embeddings, and trained classifier have been distributed.

Table 1: Theoretical Speedup

	Parallel Portion	Serial Portion	Max Speedup
Current	106.4	56.5	2.9
Future	145.2	17.7	9.2

Table 2: Testing Throughput

	Num Files	Time (hours)	Size (GBs)	Rate (MB/s)
Data creation	600	15.5	559	10.3
RFC 10	600	14.5	559	11.0
RFC 820	300	25.5	279	3.1
Naive Bayes	300	7.6	279	10.5

We also did some longer runs of applying a classifier to large sets of pcaps to gauge average throughput. Table 2 summarizes the results. For the simpler models, we can classify at about 10.5-11 MB/s while packet data is created at an average rate of 10.3 MB/s. The original data (the first day of DARPA-2009) comprises 15.5 hours and 600 files. We ran the Random Forest Classifier trained on one pcap on the entire data set. We also ran another Random Forest Classifier that was trained on 300 files and tested it on the other 300 files. Similarly, we trained a Naive Bayes classifier on 300 files and tested on the other half. For all the runs we utilized ten threads.

The difference between the two Random Forest Classifiers is that the one trained on one pcap file has ten estimators while the one trained on 300 files has 820 estimators. The difference comes from the fact that in order to incorporate

knowledge from other files to an existing Random Forest Classifier, we had to increase the number of estimators, essentially creating additional trees for each file. Thus, the Random Forest with 820 estimators has a much lower throughput because the longer predictions times (about 6 seconds versus 214 seconds). In the future, we plan to parallelize the prediction for loop which will likely make the difference in throughput less drastic. The Random Forest Classifier with ten estimators and the Naive Bayes were able to keep pace with the data creation rate.

3.2 Classifier Performance

We tested out two classifiers, the Random Forest Classifier [5] and Gaussian Naive Bayes [7]. We split the first day of DARPA-2009 into two sets of 300 files, one for training and one for testing. We listed all 600 files and gave training the even files and testing the odd files. This gave both sets representative data throughout the day.

We report two metrics, the area under the curve (AUC) for both the Receiver Operating Characteristic (ROC) curve and the Precision/Recall curve. The ROC curve plots true positive rate against the false positive rate as the threshold is varied. A perfect score for the AUC is 1.0. The ROC is known to provide overly optimistic results when data skew is present, as is with DARPA-2009.

The Precision/Recall curve emphasizes how good the predictions are for the minority class (i.e. malicious traffic). Precision is defined as the true positives divided by the true positives and false positives. So it is the fraction of results that are correct returned by the model: $Precision = \frac{TP}{TP+FP}$. Recall is defined as the number of true positives divided by the true positives plus the false negatives: $Recall = \frac{TP}{TP+FN}$. This gives you the fraction of the entire target class that are being returned by the model.

Table 3 gives an overview of both classifiers and both metrics. The AUC ROC metric gives a somewhat optimistic impression of the classifier's skill, with values between 0.988 and 0.996, while the AUC or the precision/recall curve range between 0.604 and 0.667. The AUC of the precision/recall curve is probably more useful as it gives an idea of how good the classifier does at predicting the minority class. Figures 4 and 5 present the ROC and precision/recall curves for the Random Forest Classifier, respectively, while Figure 6 and 7 are for Gaussian Naive Bayes.

Table 3: Classifier performance

	AUC ROC	AUC Precision/Recall
Random Forest Classifier	0.996	0.604
Gaussian Naive Bayes	0.988	0.667

In both cases, there is a signficant change in the precision/recall curve when recall is about 0.94. For Gaussian Naive Bayes, the plot is a little deceptive as

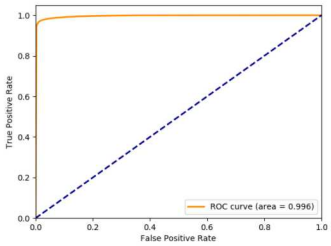

Fig. 4: Random Forest Classifier - Receiver Operating Characteristic

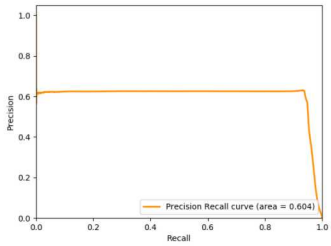

Fig. 5: Random Forest Classifier - Precision/Recall

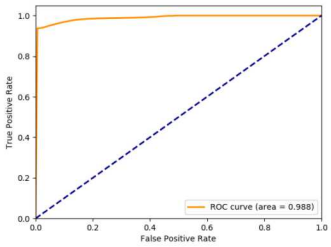

Fig. 6: Gaussian Naive Bayes - Receiver Operating Characteristic

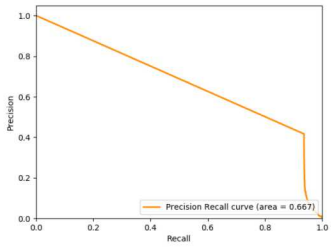

Fig. 7: Gaussian Naive Bayes - Precision/Recall

the first point from the data is $(0.937, 0.417)$ with a threshold of 1, meaning that any prediction less than one was considered benign. The point at $(0, 1)$ is by definition. We believe there is a large class of malicious behavior, likely the DDoS traffic, that both classifiers have a relatively easy time predicting. The transition at $recall = 0.94$ is likely for the other classes of malicious behavior.

Tables 4 and 5 provide some points along the precious/recall curve for the two classifiers, along with the corresponding F1 score. This is to give an idea of the tradeoff between finding malicious behavior and dealing with false positives. For instance, the first row of Table 4 shows that the Random Forest Classifier can find 98.8% of the malicious traffic, but you have to deal with about 94% of the returned results being false positives. If that is too many, one could use the threshold of the third line, where about half of the returned results are actually malicious and you still catch 95% of the total malicious behavior.

4 Related Work

A work similar to our own is that of Lotfollahi et al. [14] and their approach called *Deep Packet*. They focus on two problems, traffic characterization (e.g. identifying peer-to-peer traffic) and application identification (e.g. identifying traffic eminating from Skype or Tor), and use raw packet data as their data source. Like our approach, they avoid hand crafted features, but instead of a

Table 4: Random Forest Classifier - Precision Recall

Precision	Recall	Threshold	F1
0.060	0.988	0.006	0.113
0.108	0.981	0.009	0.195
0.504	0.951	0.029	0.659
0.630	0.930	0.285	0.751

Table 5: Gaussian Naive Bayes - Precision Recall

Precision	Recall	Threshold	F1
0.050	0.963	2.0e-134	0.095
0.010	0.947	1.31e-118	0.181
0.417	0.937	0.999	0.577

Word2Vec-based approach, they directly feed the packet bytes into a deep learning architecture. Packets are truncated or padded to be 1500 bytes long, and then fed into either a 1D convolutional neural network or a stacked autoencoder.

There are several papers that use deep learning, but they apply the network to already derived features. For the most part they test out deep learning strategies on either KDD or NSL-KDD [24, 26]. KDD is a challenge dataset from 1999 with artificially generated network data. The data was composed of benign and malicious connections, with each connection comprising 41 features. NSL-KDD is a modification of the original KDD data set to remove redundant records.

Javaid et al. [12] use Self-taught Learning [23] on NSL-KDD. Self-taught learning is an approach where you first use an unsupervised machine learning technique to create another representation of the data. For example, Javaid et al. use an autoencoder to translate the NSL-KDD feature set into a smaller representation. This new representation is then used as the basis for classification in a supervised training algorithm. Yin et al. [27] also employ deep learning, this time with recurrent neural networks, but they also test their approach on NSL-KDD. We agree with the conclusions of Malowidzki et al. [16], that many of the labeled public datasets are outdated, including NSL-KDD.

In terms of work that has examined the same data set, Moustafa and Slay [19] ran *tcptrace* on the first 30 files of DARPA-2009 to create flow-based features from which they filter down to 11 features in total. It is somewhat difficult to compare their work with ours as they are doing classification at the flow level, rather than at the packet level as we do. Also, they only examine 30 files, of which they report that 99.995% of the malicious activity is related to DDoS, while our 600 files covers a much broader range of categories of malicious activity. Also, they report that malicious flows account for 45.5% of their data set. It may be a difference between flows and packets, but we found malicious packets to account for far less: 0.46%. Their best recorded model was a decision tree, that missed 10 positive examples (there were 12 total non DDoS flows) and had no false positives.

Ackerman et al. [3] also examines DARPA-2009. They divide up the data into temporal chunks of one minute each, resulting in 13,835 chunks over the ten days, with 1,848 being malicious (if any malicious activity occured during the time period) and 11,987 benign. They then selected 25 features that were aggregate computations over the time intervals. They used diffusion maps [8] for dimensionality reduction. Then from a single initial point in the new feature

space, they expand to find all similar points by recursively adding ones that are within a certain distance of an existing point. They do not report precision/recall numbers, but from what they do state we calculated an average precision of 0.03 and and average recall of 0.08, both of which are considerably lower than our results. However, they obtain their results from a single example. for finding other instances of malicious behavior in unlabeled data.

Part of the allure of deep learning is the ability to extract relevant features. Other work that focuses on feature extraction include Ngyuen et al. [21], where they use sketches [20] to approximate values in the stream of network data and Field-programmable gate arrays (FPGA's) to increase throughput, achieving a rate of 21.25 Gbps. Das et al. [9] also use an FPGA-based approach and a Feature Extraction Module (FEM) based on sketches.

5 Conclusions

We have a presented a novel application of Word2Vec, called Packet2Vec, that translates packets into vectorized representations. We have demonstrated promising results, with classifiers achieving an AUC of the ROC between 0.988-0.996 and an AUC of the Precision/Recall curve between 0.604-0.667. The method can be used on raw packet data and does not require any domain expertise to extract relevant features.

There are many possible avenues for future work: **Temporal phenomenon**: We completely ignored temporal information. Many detection strategies utilize temporal information to distinguish between human actors and bots. How to incorporate temporal information within a deep learning strategy for cyber data is unexplored to our knowledge. **Aggregating predictions**: We made classification at the packet level. However, to a human analyst, it is likely more useful to roll up predictions to the level of a flow, or an IP, or a domain. **Existing features**: While we rely upon the deep learning model to extract relevant features, augmenting with existing approaches could be a fecund avenue to explore.

We believe that deep learning has much to offer cyber analysis, and that this work is just an intial step into discovering solutions for pressing security problems.

Acknowledgment

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA0003525.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A.,

Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), https://www.tensorflow.org/, software available from tensorflow.org

2. Abrahams, D., Grosse-Kunstleve, R.W.: Building hybrid systems with boost.python (2003)
3. Ackerman, D.A., Averbuch, A., Silberschatz, A., Salhov, M.: Similarity detection via random subsets for cyber war protection in big data using hadoop framework (2015)
4. Bay, H., Ess, A., Tuytelaars, T., Gool, L.V.: Speeded-up robust features (surf). Computer Vision and Image Understanding **110**(3), 346 – 359 (2008). https://doi.org/https://doi.org/10.1016/j.cviu.2007.09.014, http://www.sciencedirect.com/science/article/pii/S1077314207001555, similarity Matching in Computer Vision and Multimedia
5. Breiman, L.: Random forests. Machine Learning **45**(1), 5–32 (Oct 2001). https://doi.org/10.1023/A:1010933404324, https://doi.org/10.1023/A:1010933404324
6. Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., Varoquaux, G.: API design for machine learning software: experiences from the scikit-learn project. In: ECML PKDD Workshop: Languages for Data Mining and Machine Learning. pp. 108–122 (2013)
7. Chan, T.F., Golub, G.H., LeVeque, R.J.: Updating formulae and a pairwise algorithm for computing sample variances. In: Caussinus, H., Ettinger, P., Tomassone, R. (eds.) COMPSTAT 1982 5th Symposium held at Toulouse 1982. pp. 30–41. Physica-Verlag HD, Heidelberg (1982)
8. Coifman, R.R., Lafon, S., Lee, A.B., Maggioni, M., Nadler, B., Warner, F., Zucker, S.W.: Geometric diffusions as a tool for harmonic analysis and structure definition of data: Diffusion maps. Proceedings of the National Academy of Sciences **102**(21), 7426–7431 (2005). https://doi.org/10.1073/pnas.0500334102, https://www.pnas.org/content/102/21/7426
9. Das, A., Nguyen, D., Zambreno, J., Memik, G., Choudhary, A.: An fpga-based network intrusion detection architecture. IEEE Transactions on Information Forensics and Security **3**(1), 118–132 (March 2008). https://doi.org/10.1109/TIFS.2007.916288
10. Gharaibeh, M., Papadopoulos, C.: Darpa-2009 intrusion detection dataset report. Tech. rep., Colorado State University (2014)
11. Gutmann, M.U., Hyvärinen, A.: Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. J. Mach. Learn. Res. **13**(1), 307–361 (Feb 2012), http://dl.acm.org/citation.cfm?id=2503308.2188396
12. Javaid, A., Niyaz, Q., Sun, W., Alam, M.: A deep learning approach for network intrusion detection system. In: Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (Formerly BIONETICS). pp. 21–26. BICT'15, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium (2016). https://doi.org/10.4108/eai.3-12-2015.2262516, http://dx.doi.org/10.4108/eai.3-12-2015.2262516

13. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 25, pp. 1097–1105. Curran Associates, Inc. (2012), http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
14. Lotfollahi, M., Zade, R.S.H., Siavoshani, M.J., Saberian, M.: Deep packet: A novel approach for encrypted traffic classification using deep learning. CoRR **abs/1709.02656** (2017), http://arxiv.org/abs/1709.02656
15. Lowe, D.G.: Object recognition from local scale-invariant features. In: Proceedings of the Seventh IEEE International Conference on Computer Vision. vol. 2, pp. 1150–1157 vol.2 (Sept 1999). https://doi.org/10.1109/ICCV.1999.790410
16. Maowidzki, M., Berezinski, P., Mazur, M.: Network intrusion detection: Half a kingdom for a good dataset (04 2015)
17. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. CoRR **abs/1301.3781** (2013), http://arxiv.org/abs/1301.3781
18. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Burges, C.J.C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 26, pp. 3111–3119. Curran Associates, Inc. (2013), http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf
19. Moustafa, N., Slay, J.: Creating novel features to anomaly network detection using darpa-2009 data set. In: 14th European Conference on Cyber Warfare and Security (2015)
20. Muthukrishnan, S.: Data streams: Algorithms and applications. Found. Trends Theor. Comput. Sci. **1**(2), 117–236 (Aug 2005). https://doi.org/10.1561/0400000002, http://dx.doi.org/10.1561/0400000002
21. Nguyen, D., Memik, G., Memik, S.O., Choudhary, A.: Real-time feature extraction for high speed networks. In: International Conference on Field Programmable Logic and Applications, 2005. pp. 438–443 (Aug 2005). https://doi.org/10.1109/FPL.2005.1515761
22. Nvidia dgx-1 datasheet (2017), http://images.nvidia.com/content/technologies/deep-learning/pdf/Datasheet-DGX1.pdf, accessed: 2017-08-18
23. Raina, R., Battle, A., Lee, H., Packer, B., Ng, A.Y.: Self-taught learning: Transfer learning from unlabeled data. In: Proceedings of the 24th International Conference on Machine Learning. pp. 759–766. ICML '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1273496.1273592, http://doi.acm.org/10.1145/1273496.1273592
24. Revathi, S., Malathi, A.: A detailed analysis on nsl-kdd dataset using various machine learning techniques for intrusion detection. International Journal of Engineering Research & Technology (IJERT) **2**, 1848–1853 (01 2013)
25. Symantec: Internet security threat report (2018)
26. Tavallaee, M., Bagheri, E., Lu, W., Ghorbani, A.A.: A detailed analysis of the kdd cup 99 data set. In: 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications. pp. 1–6 (July 2009). https://doi.org/10.1109/CISDA.2009.5356528
27. Yin, C., Zhu, Y., Fei, J., He, X.: A deep learning approach for intrusion detection using recurrent neural networks. IEEE Access **5**, 21954–21961 (2017). https://doi.org/10.1109/ACCESS.2017.2762418