Learning to Embed Byte Sequences with Convolutional Autoencoders

Doug Sibley[1]

[1]*Cisco Talos*

Abstract
We propose a self-supervised approach to generating features for arbitrary byte sequences by training a convolutional autoencoder directly on raw bytes. The limited vocabulary of this task (256) makes it viable to train on sequences of at least 1MB in size. We evaluate this approach to byte-level feature engineering by first examining how accurate the autoencoder can be at reconstructing a variety of datasets, then testing this approach specifically on SOREL malware samples, extracting the learned features and comparing them against the EMBER V2 features for the task of malware tagging. Our results suggest that the learned features from the convolutional autoencoder rival those of the human-engineered set without requiring domain-specific preprocessing of Portable Executable files.

1. Introduction

Byte sequences possessing discernable structure and hierarchy, such as Portable Executable (PE) files, can be processed by domain experts to produce features amenable to machine learning tasks. The EMBER V2 features as presented in the EMBER[1] and SOREL[2] datasets are derived from processing such executable files and extracting features that researchers and data engineers in the information security space have identified as being salient for classification. However, this process is knowledge intensive, requiring domain expertise on the specifics of the data contained within a given format.

Earlier natural language processing (NLP) work[3] showed that it was possible to learn high-level concepts, such as sentiment, by simply predicting the next character within a text sequence. Krčál et al.[4] had success training a Convolutional Neural Network (CNN) to predict malware based on input samples comprised of a raw byte stream. Expanding on their work, we posit that it would be desirable to develop a method of extracting features from raw byte sequences without first requiring expert domain knowledge and a deep understanding of the format any potential input data may be stored in.

2. Methodology

We leverage the self-supervised task of autoencoding to train a convolutional network whose goal is to reconstruct an input sequence of bytes by predicting each byte value. Because a byte

CAMLIS'22: Conference on Applied Machine Learning in Information Security (CAMLIS), October 20–21, 2022, Arlington, VA
dosibley@cisco.com (D. Sibley)

can only be one of 256 values, the vocabulary for our prediction is small enough to directly calculate the cross-entropy loss without having to resort to approximate methods. This ceiling on our vocabulary size also provides a performance envelope on modern GPUs, whereby training models on sequences hundreds of thousands of bytes long becomes feasible. As we are reconstructing the raw input byte sequence, no preprocessing of the data is necessary, other than converting each byte to its representative integer value.

Our contribution to the field is the observation that the aforementioned byte reconstruction task is computationally viable. Furthermore, through this implementation we can design an autoencoder that has useful properties for downstream tasks. Given the potentially large length of input sequences, the priority for model design is one with good computational efficiency and throughput. We chose to focus on convolutional networks over recurrence as convolutions allow us to process an entire sequence in one training step and can be greatly accelerated by using GPUs. While various recurrent neural network designs could also be applied to this task, issues with long-term credit assignment and training speed suggests that these model architectures would be a poorer fit. Our autoencoder has three main components in its design:

1. A bi-directional temporal CNN which produces an output for every position in our input sequence.
2. A global CNN which produces a fixed length output based on the output of the temporal CNN.
3. A decoder network which accepts the temporal and global CNN as inputs and attempts to predict the value of each byte.

Temporal convolutions[5] are a type of one-dimensional convolution, where at sequence position T our convolution is only looking at information prior to T, contrasted with a regular convolution, which sees information from before, at, and following T. Stacking multiple layers of temporal convolutions increases the receptive field, but still constrains the network to only seeing information prior to T. A bi-directional temporal CNN applies this concept in both directions. At position T the network will have information from both before and after position T, but crucially not at T itself. Due to this design, when the model attempts to predict what byte is at position T, the network must use the information from the rest of the sequence to make a prediction, rather than simply learning to predict the value it sees at that location. Since every position in our input sequence has this constraint, we can attempt to predict every byte in a single training step, producing a high-quality training signal.

The second component of our autoencoder is comprised of a global CNN. This part of the model starts with the output from the temporal CNN, passes the data through several layers of convolutions and pooling, and is completed by applying a global max pooling layer. This architecture produces a fixed-length vector, whereby its size represents a hyperparameter for the network and can be treated as an embedding of the entire input sequence. The authors of Malconv[6] found success with a large initial receptive field followed by a global pooling layer. We similarly view the global pooling layer as being critical to extracting high-level features that may be present at any position within our byte sequence.

Finally, the decoder component is a dense neural network. It takes the full sequence of the temporal CNN and the fixed vector of the global CNN, producing a prediction for each byte position of the input sequence. The loss for this overall network is the average cross-entropy loss for each predicted byte.

The desired goal for this design is that the temporal CNN learns features at a local context for predicting byte values, while the global CNN learns features representing the entirety of the sequence. We can then embed any variable-length input sequence to a fixed-length feature vector by extracting just the global CNN output from our model.

Model Architecture

The core requirement for the design of a model on this autoencoding task is to return a prediction sequence the same length as the input byte sequence. All experiments in this paper were conducted with the same model design, as described below. This design has approximately 700k parameters and was trained on a single NVIDIA V100 GPU at a speed of 2.4 MB/s. Unless otherwise noted, all convolutions in the network are implemented in the densely connected style[7].

The bi-directional temporal CNN consists of two identical networks, one in each temporal direction, where the final output of both networks are concatenated together. The input to the network is the input byte sequence, which is embedded with a length 8 vector. Each network consists of 6 layers having 16 features with a width of 15. Dilation rates in order are 1, 1, 5, 9, 13, 1. The output of the temporal CNN is a sequence the same length as the input.

The global CNN takes as input the results of the bi-directional temporal CNN concatenated with the embedding vectors from the input byte sequence. Its primary purpose is to produce a fixed length vector from the variable length input, using a global max pooling layer to accomplish this. It is comprised of 9 convolutional layers containing 32 features with a width of 7. Layers 1 and 3 have a stride of 3, with a width 3 average pooling layer after layer 6. Layers 5 and 8 have a dilation of 3, while layers 6 and 9 have a dilation of 5.

The decoder component takes as input the results of the bi-directional temporal CNN as well as the results of the global CNN, which is broadcast back to the shape of the temporal CNN. It additionally uses a single width 17 convolutional layer with 64 features, which is applied solely to the embedded input byte sequence. This layer is constrained during training such that the weights at position T are fixed to 0, effectively allowing the layer to see 8 bytes before and after position T. Failure to constrain the weights in this way, or stacking multiple of these layers together, would allow the decoder to observe the exact input byte it is attempting to predict and would short circuit the training objective. These three inputs are passed through a 4 layer fully connected network, each with 64 features, with the final output being a sequence with the identical length as the input for calculating the per-byte cross entropy prediction loss.

3. Autoencoder Evaluation

To evaluate the efficacy of this design on our autoencoding task, we trained a set of models with fixed hyperparameters on a variety of datasets to observe the accuracy of the autoencoder for predicting the correct byte values across multiple domains. The tested datasets were comprised of:

1. Wikipedia site dumps in XML format, containing natural language articles inside of XML structure, roughly 1GB of data for each selected language. This data tests the autoencoder's ability to represent single and multi-byte character sets.
2. MNIST images represented as either the pixel values in a numpy array (ideal format), or bytes from saved PNG/JPG images. This data tests the autoencoder's ability to capture information when the underlying bytes are not well modeled by a 1d CNN, or when the information is compressed.
3. SOREL 20M malware PE files. This data tests the ability of the autoencoder to represent complex real world inputs.
4. Byte values drawn from a uniform random distribution. This tests that the model is being forced to predict byte values from the surrounding context and not by passing through the input byte.

Below we report the average accuracy for predicting the byte value after the model has been trained:

Dataset	Accuracy
English Wiki	69%
German Wiki	68%
Greek Wiki	82%
Hebrew Wiki	74%
Japanese Wiki	71%
Russian Wiki	80%
Chinese Wiki	64%
MNIST Ideal	88%
MNIST PNG	32%
MNIST JPG	7%
SOREL Malware PE	17%
Random Uniform	0.39%

Accuracy for the Wikipedia data is observed to be the highest among the tested datasets, as natural language and XML both have strong context for a given byte in the closest surrounding bytes.

Accuracy on the ideal MNIST data is also very high, however most pixels in MNIST are represented by the value 0, excluding predictions for byte value 0 the autoencoder had roughly 36% accuracy on the ideal data format. The same adjustment to the PNG accuracy yields a value of 25%, while the accuracy on the JPG dataset is the same when 0 is included or excluded. As such, we can observe a loss of accuracy as the structure of the signal in the images becomes more complex in the PNG dataset and can observe accuracy further degrading in the JPG dataset by the inherent entropy of lossy-compression.

Accuracy with the SOREL malware samples shows that the reconstruction task is more challenging than natural language, but easier than the most challenging MNIST format. As PE files can contain a variety of types of information, such as headers, executable code, and arbitrary data such as strings, the model faces a varying challenge even in the same sample.

The random uniform data serves as an implementation correctness test, as we want the model to be learning to predict byte values from the surrounding context without using the byte's identity directly. If there were an error in the implementation, such as mis-aligned temporal convolutions, the model would be able to achieve near perfect accuracy by passing through the input. The results confirm that our model is properly constrained to the surrounding context and limited to an accuracy of 1/256.

4. Learned Feature Evaluation

To evaluate if the autoencoding task is forcing the network to learn interesting features, trained versions of the MNIST and SOREL models were used to produce fixed-length feature vectors for their respective training sets by extracting the output values from the global encoder portion of the autoencoder. These features were then used to train a Random Forest classifier to predict MNIST digit classes or SOREL Malware tags, allowing us to evaluate if the features learned from the purely self-supervised training task are salient to a known classification task for the data.

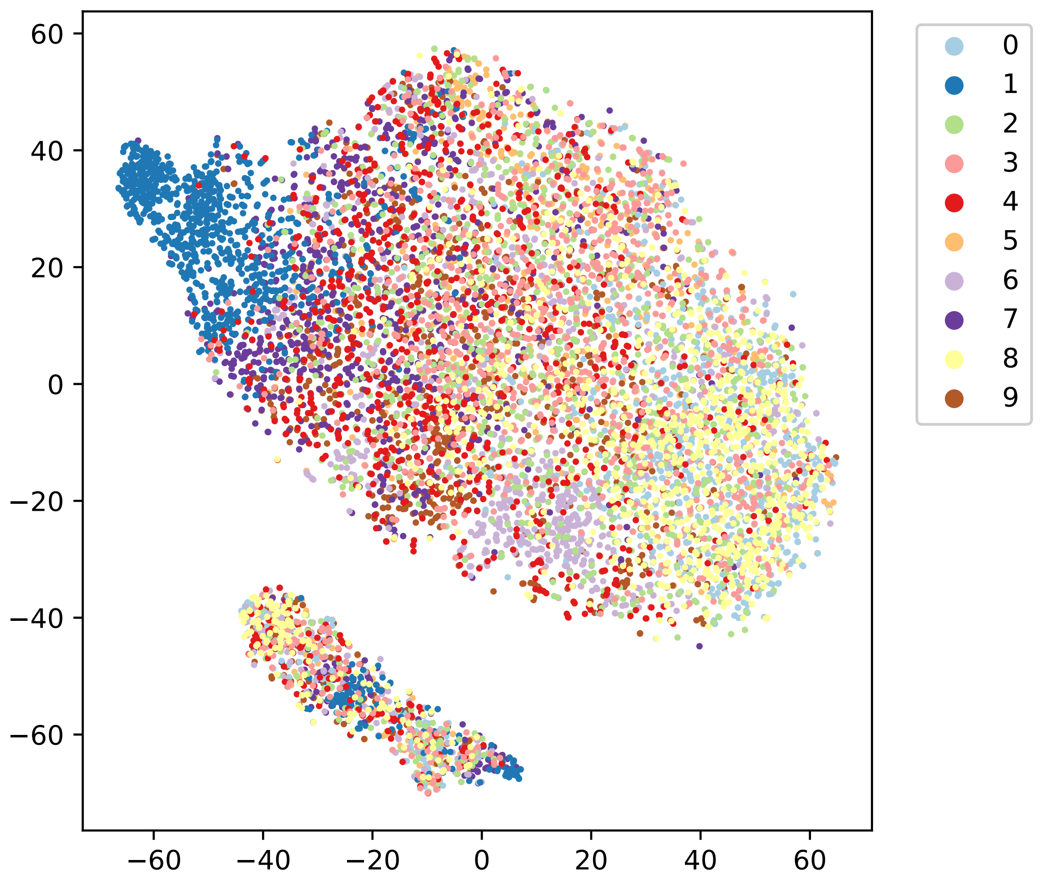
MNIST Evaluation

The MNIST model was evaluated on all three data formats in order to understand how increasing the complexity of the input data format impacts the quality of the learned representation. For each model, we embed the MNIST test set and split it into a new training and test set to evaluate the accuracy of predicting the associated digit. This approach is done before and after the autoencoder model is trained on the MNIST training set, so that we can observe if the self-supervised training causes the fixed feature representation to better represent the classes in the downstream classifier.

Data Format	Untrained Autoencoder Accuracy	Trained Autoencoder Accuracy
Ideal	71.6%	79.3%
PNG	26.9%	59%
JPG	26.3%	48.7%

For all three models, we observe the accuracy for predicting the associated digit improves when using representations from the trained version of the autoencoder, even though the autoencoding task is not using any information on the class digits. The untrained autoencoder on the ideal data format is still relatively skillful. In this case it is essentially functioning as an extreme learning machine, however it still improves its performance with the self-supervised training. For the PNG and JPG data the improvement is more pronounced, as the input data is more compressed and less directly representative of the underlying class compared to the ideal pixel format. It is also interesting to note that the relative performance of the classification models matches the order seen in the autoencoder training. However, the drop-off of reconstruction accuracy was more pronounced than the drop-off of classification accuracy, showing that even if the autoencoder is unable to achieve a high reconstruction accuracy it is still capable of learning features relevant in a downstream classification context. Figure 1 shows a tSNE embedding of the autoencoder feature vectors extracted with the PNG model, revealing that

Figure 1: tSNE embedding of 10000 MNIST PNG images using the Autoencoder feature vector.

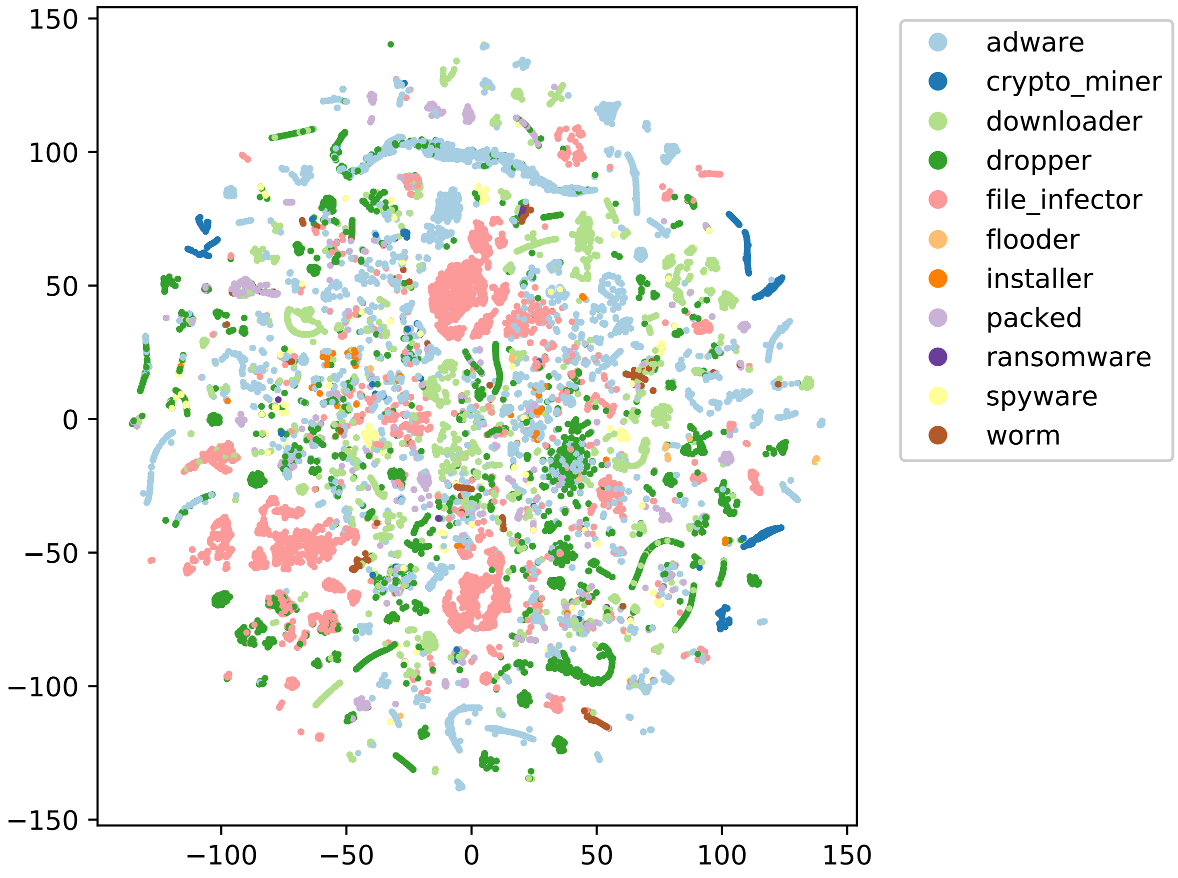


even in PNG format the autoencoder is able to learn features which are salient to the classes present in the data.

SOREL Evaluation

To evaluate how effective our autoencoder was on the SOREL Malware samples, a set of samples which were not used in the self-supervised task were held out and embedded with the trained SOREL autoencoder. Three Random Forest models were trained, one on the autoencoder feature vector, one using the provided EMBERv2 features, and one with both the autoencoder and EMBERv2 features. The training data for the classifier was comprised entirely of SOREL malware samples, with the models attempting to predict any of the malware tags associated with the given sample.

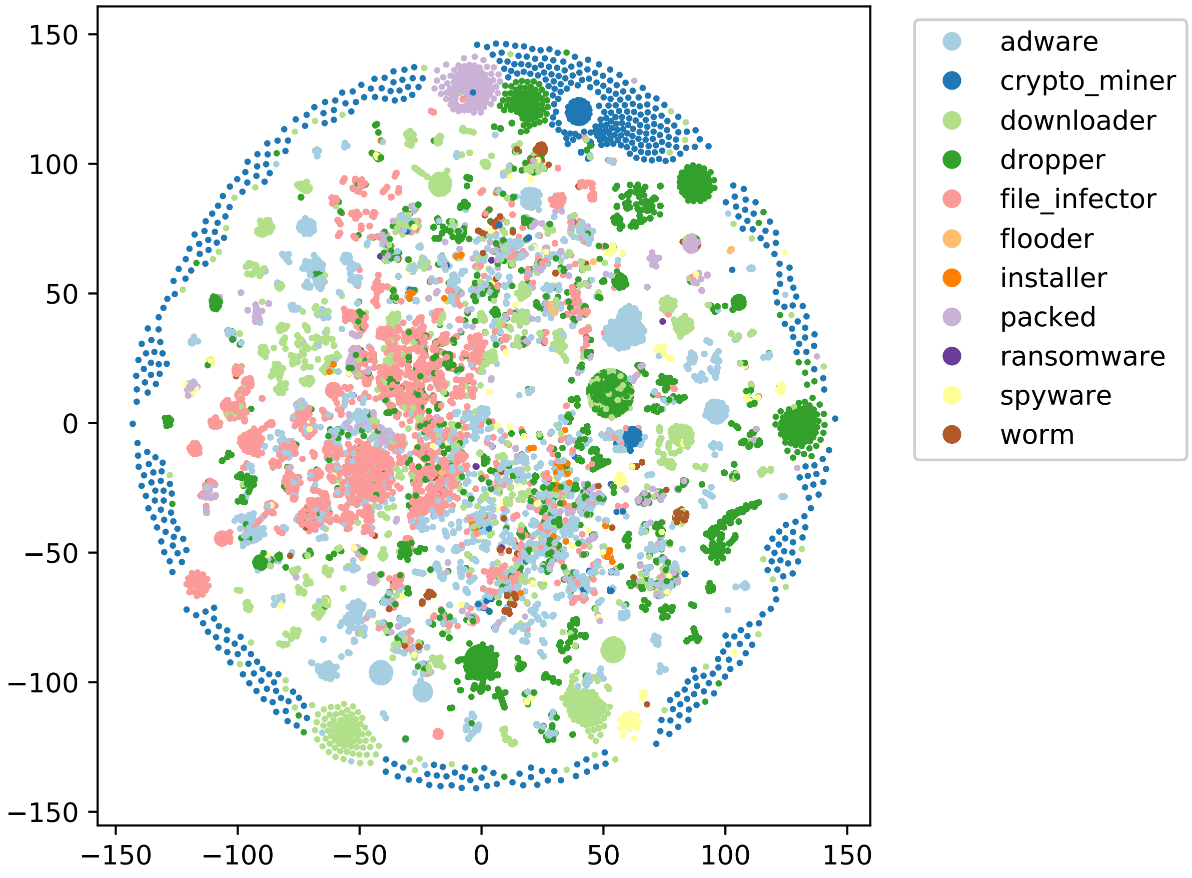
Figure 2: tSNE embedding of 50000 SOREL Malware samples using the Ember features.



	EMBERv2 Model	Autoencoder Model	Joint Model
Accuracy	83.97%	83.96%	83.97%
Precision	96.88%	96.88%	96.87%
Recall	92.92%	92.93%	92.94%

For the reported metrics, we observe that the model using only features from the autoencoder has performance comparable with the model using the human-engineered features from EMBERv2. Figures 2 and 3 visualizes a tSNE embedding of the same set of 50k samples using the Ember and Autoencoder features, showing that both have the ability to partition the data to a certain extent with respect to the malware tags. Performance between all three models is quite close. We attribute this to label noise in the malware tags, which are themselves derived from machine learning[8] and likely contain some level of error. With this in mind, we do not claim that the autoencoder features outperform the human-engineered set, but rather that the autoencoder can learn a representation with similar performance without requiring

Figure 3: tSNE embedding of 50000 SOREL Malware samples using the Autoencoder features.

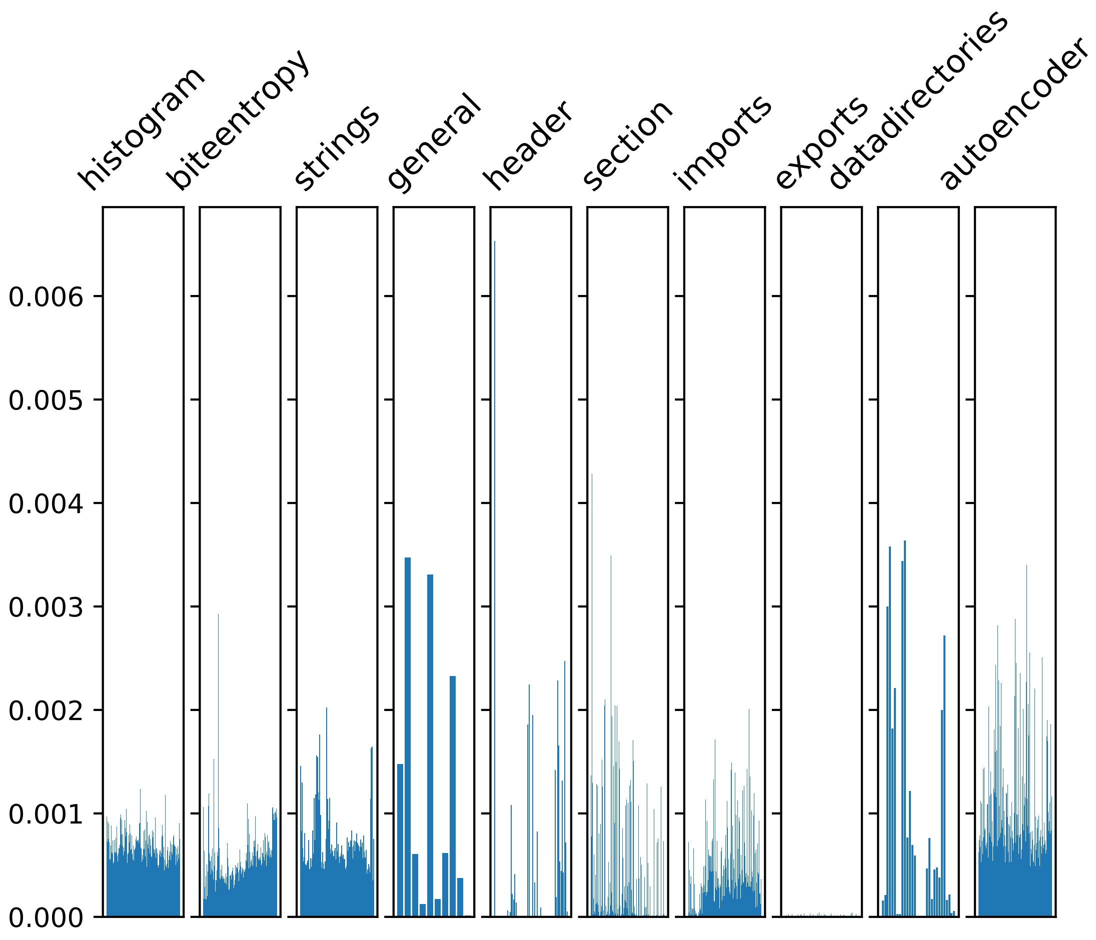


human domain expertise. We further validate this assumption by examining the joint Random Forest model, where we find 42 of the top 100 most important features were produced from the autoencoder. Figure 4 shows all feature importances from the model, divided into the 10 subsections that comprise the Ember feature set as well as the autoencoder features.

5. Conclusion

Our experiment suggests that learning representations of bytes with a convolutional autoencoder can be effective for identifying salient features present in a set of data. Evaluating this approach on the SOREL dataset shows that this method rivals the efficacy of human-engineered features, with the added advantages of our approach not requiring any domain knowledge and can be applied to raw input sequences. Additionally, having the majority of the autoencoder being implemented via convolutions, our model architecture benefits from the prior work completed in the industry to optimize the processing speed of convolutions on GPUs.

Figure 4: Feature importance graph from the joint Random Forest model, comparing the Autoencoder features to the 10 sections of Ember features. Individual features for each subsection are plotted along the X axis, the order of which is not relavant. The Y axis is the feature importance reported by the model, higher is more important



References

[1] H. S. Anderson, P. Roth, EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models, 2018.

[2] R. Harang, E. M. Rudd, SOREL-20M: A Large Scale Benchmark Dataset for Malicious PE Detection, 2020.

[3] A. Radford, R. Jozefowicz, I. Sutskever, Learning to Generate Reviews and Discovering Sentiment, 2017.

[4] M. Krčál, O. Švec, O. Jašek, M. Bálek, Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only, 2017.

[5] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, K. Kavukcuogluo, WaveNet: A Generative Model for Raw Audios, 2016.

[6] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, C. Nicholas, Malware Detection by Eating a Whole EXE, 2017.

[7] G. Huang, Z. Liu, L. van der Maaten, K. Q. Weinberger, Densely Connected Convolutional Networks, 2016.

[8] F. N. Ducau, E. M. Rudd, T. M. Heppner, A. Long, K. Berlinr, Berlin. Automatic Malware Description via Attribute Tagging and Similarity Embedding, 2019.