Interpreting Convolutional Neural Networks Applied to the Classification Malware Converted to Images

William Briguglio
School Of Computer Science, Windsor University, Canada
briguglw@uwindsor.ca

Keywords: Malware, Machine Learning, Detector Interpretation, Convolutional Neural Networks, Malware-Images

Abstract: Today, the fast-changing nature of malware provides a significant challenge to data security specialists. The increased ability for authors to bi-pass traditional detection methods has forced security specialist to look to machine learning and artificial intelligence for creating robust detection systems. One area of research has been the application of image classification for automatically classifying malicious binaries. In this scenario malware binaries are converted into grey-scale images before being used to train a convolutional neural network. This approach has been largely successful, however, malware analyst prefer interpretable detection systems. This is due to the need to fine tune these systems to decrease the amount of false positives and false negatives. In this paper we provide a brief overview of the application of image classification techniques to malware classification. We then propose our own interpretation friendly method for applying CNN's to malware images, and test it on a Microsoft data set containing the binaries of several different malware families. After, we provide an interpretation of the prediction results. Finally, we finish off with a discussion of our results and necessary future work.

1 INTRODUCTION

The use of machine learning has grown significantly in the last decade or so. This has largely been attributed to the increased processing power and the growing availability of large representative data sets since the early 2000's. The machine learning algorithms being used have also grown more complex as large data sets provide the opportunity to learn more complex patterns. This has lead to a significant increase in the ability of machine learning models in image classification tasks where architectures such as the Convolutional Neural Network (CNN) can now out preform humans in some scenarios. Machine learning has also grown more popular in other domains such as malware detection and analysis. This is because of the fast changing nature of malware, and the increased ability for malware authors to bypass traditional malware detection methods. Machine learning offers more generalized solutions allowing the detection of new malware binaries and making it more difficult for malware authors to bypass the detection system. The high performance of machine learning in the image classification domain has now lead to the adaptation of image classification algorithms, such as the CNN, in the malware detection domain. However, with these more complex classification algorithms, it becomes increasingly difficult to understand what exactly a model has learnt.

There are two categories of techniques used for analysing binaries to determine their functionality. The first is called static analysis, in which the binary is not executed and analysis is done by simply processing the raw binary or the disassembled code. One such method is the use of malware signatures. Malware signatures are typically cryptographic hashes of binary files, after some prepossessing has been done, which are unique to that malware binary. When a malware detector processes a new file it computes its signature and compares it to a list of signatures which belong to known malware. The problem is that very minor automatable modifications can change the signature of a malware making old signatures no longer match. Another issue is that signatures do not generalize to unseen examples. To remediate this issue, several approaches have been used such as fuzzy hashing which, rather then using a cryptographic hash, uses a similarity measure to compare binaries with old ones so that small changes in input do not cause false negatives.

The second technique is known as dynamic analysis where potentially malicious code is executed in

a controlled environment such as a Virtual machine (VM) in order to observe its behaviour. This method allows detectors to directly examine the functional characteristics of a binary and look for functional similarities between malware families. The down sides to such approaches are the computational overhead and the overwhelmingly large amount of data generated which can be full of noise and useless information. Further, it is hard to understand the full behaviour of the malware when it is not in the wild, since some of the functionality of the malware can only be detected when it is allowed to connect to the internet. Therefore, the malware may not execute some of its malicious behaviour, thus going undetected or not completely understood.

Machine learning techniques not used for image classification have also been applied to augment both of these approaches. This typically involves a complex feature engineering and extraction phase which has to be fine tuned for different systems and different malware families. This incentivized the use of deep learning in order to automate some portion of the feature extraction process. Further more, architectures which are designed to make use of the ordinal information contained within the input sample have been favoured. This is because the order in which instructions appear in a binary is massively significant in determining its functionality. Thus, recurrent neural networks (RNN) which have typically been used for natural language processing, are an obvious candidate. However, RNNs have trouble dealing with long term dependencies within the sample they are classifying. Further, (Nataraj et al., 2011) made the observation that malware converted to images belonging to the same family have visual similarities between them and have dissimilarities with malware belonging to different families which can be distinguished by the human eye. This caused some researchers to turn to CNNs as they also take advantage of ordinal information contained in the input data as well as use the spatial information contained in images.

The downside to such complex approaches is the resulting models are not easy to understand. Malware analyst prefer explainable solutions as they must fine tune their systems in order to limit the number of false positives and false negatives. However, if you do not know what the model has learnt, or why it is making a prediction, then it is difficult to make adjustments as you are essentially working with a black box. Further, the inherent risk involved with new technologies means that stake holders must be convinced the model is learning something relevant to the task at hand. This was much easier with traditional approaches where the classification was an easy to understand process, however now it is no longer evident how the model is making a classification. Additionally, a growing problem in malware analysis is the large amounts of data one must sift through to determine the functionality of a malware binary. Patterns the model learnt should be used to help with this issue.

The process of explaining a models classifications is called interpretation. If a fine grained interpretation of a malware classification model can be obtained, one which isolates specific lines of code as significant for a classification of a single sample (i.e. a local interpretation), then this interpretation can be used to aid in down stream tasks such as highlighting code snippets which significantly contributed to a classification decision. This would give malware analysts a starting point and help limit the amount of time and effort needed to determine the functionality of a malware sample. Further, isolated lines of code which are deemed significant can be used to detect when a model is learning irrelevant relationships. These can then be corrected for to decrease false positive and false negative rates. Lastly, if these significant lines of code can be shown to corroborate industry knowledge then this can show the model has learnt something which is relevant and help improve confidence from stakeholders. This would not only put stakeholders minds at ease but would increase industry adoption for this emergent technology.

Thus, in this paper we focus on augmenting the approach of using a CNN trained on the image representation of malware binaries for static analysis. We do this with the goal of providing a fine grained local interpretation of prediction results while maintaining positive classification performance relative to similar models in the literature as well as keeping the simple automated feature extraction from raw data which CNNs provide. We start with a brief review of the application of CNNs to malware classification, we then detail the specifics of our method for generating and classifying malware images and interpreting our classification results. Next we have a discussion of our results and end with conclusions and future work. To the best of our knowledge, the interpretation approach used in this work has not been done before.

2 LITERATURE REVIEW

Recently there has been some interest in applying CNNs to malware classification. In (Zhang et al., 2016), they were able to achieve a 96.7% accuracy classifying malicious binaries against benign binaries. This was accomplished by first mapping op code se-

quences of length 2 from a sample to a 2 dimensional feature map where the value of each "pixel" in the feature map was determined by multiplying the information gain of the corresponding op code sequence in the sample by the probability of said op code given said sample. Next the resulting "images" where enhanced to create a larger contrast between the malicious and benign samples before applying a CNN to the final image set for classification.

In (Wei Wang et al., 2017) they converted the first 784 bytes of various network traffic representations into 28×28 grey scale images to train a CNN to detect malicious network traffic and different families of malicious network traffic. With their best preforming representation, their CNN achieved a 100% accuracy when distinguishing between malicious and benign network traffic and a 98.65% accuracy when distinguishing between families of malicious network traffic.

In (Kolosnjaji et al., 2017), they were able to classify a data set containing both benign and malicious binaries belonging to 12 different malware families by using a hybrid feed forward-CNN classifier. The feed forward portion of the classifier used features extracted from the PE meta data and imports while the CNN used opcode sequence data where each row of the input volume corresponded to the one hot encoding of an opcode vector. Their architecture was able to achieve a 0.92 f1-score, however the feed forward and CNN alone were able to achieve a 0.90 and 0.91 f1-score respectively while an SVM trained on the same features achieved a 0.92 f1-score, so these results serve more as a proof of concept rather then indicating a superior solution.

As you can see there are various methods used to convert malware samples to input for CNN classifiers. However one popular method put forward in (Nataraj et al., 2011) and used in the following works is to convert the binaries to grey scale images by interpreting the raw binary data as a sequence of pixels, where each byte represents the grey scale value of its corresponding pixel in the range [0,255]. The problem with this process is that the resulting images are not of uniform length, thus they must be reshaped in order to match the input dimensions of the CNN classifier.

In (Cui et al., 2018), the authors used a CNN with alternating convolutional then subsampling layers and several fully connected layers to classify a data set of grey scale images from 25 malware families. Here the input was rescaled to uniform dimensions, losing some information. They also preform image augmentation such as rotation and shifting to reduce overfitting. The resulting model managed to obtained a 94.5% accuracy when classifying malicious vs. benign samples.

The authors in (Su et al., 2018) were able to classify malicious Internet of Things (IoT) malware by converting the binaries in a data set containing 365 samples to grey scale images and then rescaling the images to uniform dimensions to train a CNN with. Half of the samples were IoT malware belonging to two major IoT malware families, and the other half were benign Ubuntu system files. The resulting classifier achieved a 94% accuracy.

Transfer learning was used in (Rezende et al., 2017) by taking the first 49 layers of the ResNet-50 architecture and swapping the last layer for a 25-node softmax layer to make classifications on a data set that contained grey scale images of 25 different families of malware. The images were first converted to RGB since ResNet-50 is designed for 3-channel image input. During training all but the final layer weights were frozen and the classifier was able to obtain an accuracy of 98.62%. Here, the varying size of the malware binaries created varying sized images that were rescaled, still offering good results despite the loss of information.

(Chen, 2018) also used transfer learning on the same data set as the above, except they used the Inception-V1 architecture and froze all but the last fully connected layer and the softmax layer which was replaced with a 25-node softmax layer. Similarly they converted the grey scale images to RGB images by duplicating the grey scale channel three times. Their approach obtained a very impressive 99.25% accuracy.

They also claim to provide an interpretation of the predictions by using LIME (Ribeiro et al., 2016) to highlight important areas of an input sample. However, the proposed method can only highlight important regions of the input image called "super-pixels" which encompass very many pixels which each mapping to a byte of code. The regions are of varying size but there are 200 total, meaning that even a modestly sized binary of 200,000 Bytes would have super pixels highlighting on average 1000 bytes of code each. We would like to improve on this approach in order to provide more granular interpretations. Further, (Chen, 2018) also used their approach on the same data set used in this paper and obtained a 98.13% accuracy. We will return later to these results for comparison between our methods.

Additionally, (Gibert et al., 2019) converts the same data set used in this paper into grey scale images and trains a CNN classifier without the use of transfer learning after down sampling the images to unifrom size. The classifier has 3 convolutional layers followed by a fully connected layer and a softmax clas-

sification layer and was able to achieve a 97.5% accuracy. We will return also to these results for comparison between our method with a trained from scratch method.

There has been a plethora of papers published with differing techniques used to interpret or visualize what machine learning models and neural networks have learnt. However, for the sake of brevity, we will discuss some of the techniques used for convolutional neural networks only.

Layer-wise Relevance Propagation (LRP), described in (Bach et al., 2015) as a set of constraints, is used to visualize where the model is placing its emphasis when making classifications by back propagating a models prediction using its weights and some decomposition function which returns the relevance of previous nodes to that prediction. The constraints ensure that the total relevance is preserved from one layer to the next as well as that the relevance of each node is equal to the sum of relevance contributions from its input nodes which in turn is equal to the sum of relevance contributions to its output nodes. Any decomposition function following these constraints is considered a type of LRP.

In (Shrikumar et al., 2017), they propose DeepLIFT which, in contrast to LRP, attributes to each node a contribution to the difference in prediction from a reference prediction. DeepLIFT back propagates just this relative difference in prediction scaled by the difference in intermediate and initial inputs.

The authors in (Ribeiro et al., 2016) put forward Local Interpretable Model-agnostic Explanations (LIME), which was used in (Chen, 2018) above, to explain predictions using an approach which trains an interpretable classifier by heavily weighing samples nearer to a sample of interest in order to locally approximate the non-interpretable or black-box model. This work was extended by Tomi Peltola in (Peltola, 2018) to generated local interpretable probabilistic models by minimizing the Kullback-Leibler divergence of the predictive model and the interpretable model in order to provide explanations that account for model uncertainty.

There have also been several implementations of the above methods. The creators of LIME developed a python library available at `https://github.com/marcotcr/lime`. Additionaly, there is the iNNvestigate (Alber et al., 2018) python library available at `https://github.com/albermax/innvestigate` which was used in this paper and provides implementations of LRP as well as many other useful interpretation methods met for convolutional neural networks, although many of them can be applied to other types of neural networks not working with image data.

3 METHOD

Training and classification were done on a subset from a data set of 10,896 malware files belonging to 9 different malware families.[1] The data set is discussed in (Ronen et al., 2018). Each sample consists of the hexadecimal representation of the malware's binary content in a .bytes file as well as its corresponding assembly code in an .asm file. The hexadecimal representations were preprocessed as followed. First, we determined the total length of each binary. Since our goal is to provide a fine grain interpretation, we wanted to avoid resizing images to fit the input of the CNN if the resizing caused information loss. Thus, we deiced to only scale up images by padding them with zeros, rather then scaling them down. This is because when we scale down an image the resulting image's pixels will actually map to more then one pixel in the original, and therefore more then one byte of code. Therefore, even if we obtain the importance of a single pixel of the rescaled input image, we still do not have a fine grained approach, since the mapping from rescaled input image to full sized image and then to the bytes and finally the assembly code will be a one-to-many mapping, which is increasingly so with large binaries that require more drastic rescaling. So, we picked a size range where the majority of the binaries resided, that is the range of 101,400 to 200,934 bytes. The files which were less then 200,934 bytes in length were padded with bytes of all 0's before and after the binary so that all the binaries were the same size. The padding before and after the binaries were equal so that the actual binary was centred verticallyy in the image, although it took up the entire width of the image. The resulting data set contained 2,114 binaries with 6 classes total. The class details are summed up in table 1.

Table 1: Class distribution in Data Set

Class No.	Family	Samples	Type
1	Ramnit	635	Worm
2	Lollipop	68	Adware
4	Vundo	188	Trojan
6	Tracur	145	TrojanDownloader
8	Obfuscator.ACY	810	obfuscated malware
9	Gatak	268	Backdoor

[1]The data set was downloaded from `www.kaggle.com/c/malware-classification/data`

Next, the binaries where converted into image tensors of the shape $(183, 183, 6)$. This was done by placing the first 6 bytes of the binary in the input tensor positions at location (0,0,0) through (0,0,5) the next six bytes in the tensor positions at location (0,1,0) through (0,1,5), and so on, until all the bytes were processed. The reason 6 was chosen as the number of channels was because (Raff et al., 2016), which examined the effectiveness of different lengths of byte-grams for malware classification, found that 6-byte-grams were the most effective compared to other lengths of byte-grams. A byte-gram is as a sequence of bytes which appear consecutively in a binary. These are analogous to *n*-grams which are a sequence of *n* words or characters which appear in text. The intuition here was that each pixel, which contains the 6-byte-gram in the pixel's 6 channels, would contain what could be thought of as a "byte word" leaving the filters to learn to detect significant byte words and in later layers significant sequences of byte words.

To the best of our knowledge, this the first time someone has applied a CNN to a malware binary classification task where the binaries were converted to "images" with 6 channels without the use of downsizing. The 6-channel input has an added benefit of fitting more information into a smaller volume, this means we can have a compact input volume which can still contain all the information of a 200,934 byte binary, therefore helping in our task of creating fine grain interpretations. The other two dimensions were set to $\sqrt{200,934 \div 6} = 183$, since this created a square image, which is what the models we are using for comparison (discussed in section 2) also used. Fig 1 shows the resulting images of two samples for each of the 6 classes. The images in the column labeled F are the first three channels of the samples interpreted as RGB channels while the images in the columns labeled L are the last three channels of the samples interpreted as RGB channels. As you can see there is some similarity between images of the same classes and greater difference between images of different classes. Additionally, there is a lot of similarity between the last and first three channels. However, the CNN model is indifferent to the number of colour channels or human detectable features and can find structure, imperceptible to humans, in an arbitrary number of channels, therefore we must wait until the classification results before we can draw any conclusions.

The neural network starts with 2 blocks of the classic convolution, ReLU, MaxPool architecture. This architecture was chosen as it has been shown to work well in the literature and is also similar to what was used by the models used for comparison.

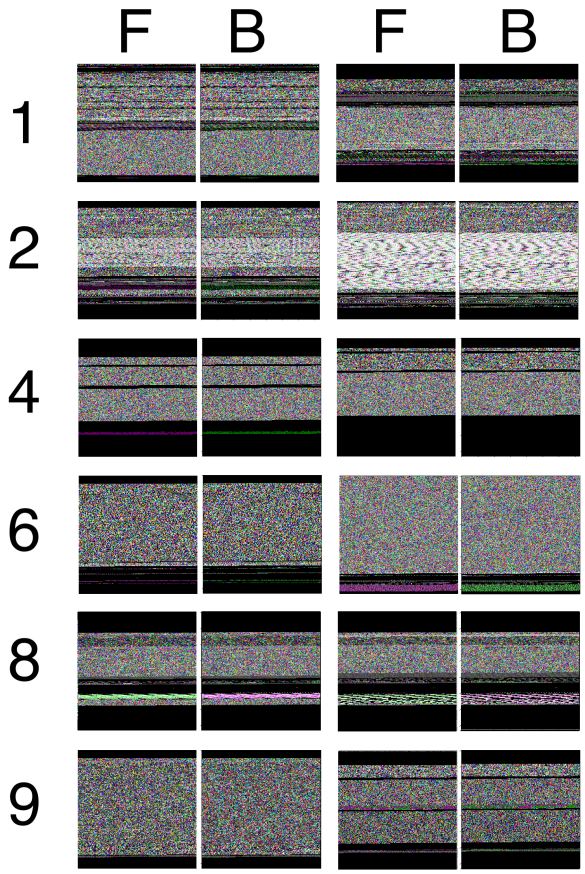

Figure 1: Our Model Architecture

We went with just 2 blocks as we wanted to limit the number of parameters given the small size of our data set. These 2 blocks were followed by a single fully connected layer with 512 neurons with the ReLU activation function then a softmax classification layer with 6 neurons. 512 neurons were used as experimentation showed this number to work best on the validation set despite adding a large number of parameters and the possibility of over fitting. The two convolutional layers used 128, then 256, 5x5 filters with strides of 2 and the MaxPool layers used 2x2 pool size with strides of 2. This was done mainly because the the small data set size meant we had to shrink the volume as fast as possible in order not have too many layers or too many neurons in the last volume before the first dense layer thus keeping the number of parameters low. To reduce over fitting dropout with a rate of 0.4 was also used on the connections between the last MaxPool layer and the first fully connected layer since these connections accounted for 13,107,712 of the 13,949,575 trainable parameters. Further, L2 regularization was used with a 0.1 penalty on both convolutional layer weights

and the weights between the last MaxPool layer and the first fully connected layer. Figure 2 shows a summary of the architecture of the CNN used. The figure was created using software available online at `http://alexlenail.me/NN-SVG/LeNet.html` which is described in (LeNail, 2019).

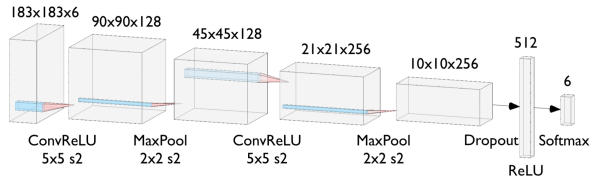

Figure 2: A sample represented as an image

The model was trained as follows. First the data set of 2,114 samples was randomly split into 3 disjoint sets, the training set, validation set, and the test set, each with approximately the same class distribution. The Training set had 1,514 samples, the validation set 100 samples, and the test set had 500 samples. The validation set was used to tune the model's hyper parameters and needed to be small to ensure the test set was large enough to be significant for evaluation while the training set was large enough for the model to learn generalize patterns despite the small size of the total data set. The test set was not used except at the end of training in order to evaluate the final model. The training was done using the Adam optimizer with 256 batch size over 80 epochs. Classes were weighted inversely proportionately to the class size in order to account for class imbalances. The model was implemented and trained using the Keras (Chollet et al., 2015). python library.

4 RESULTS

4.1 Evaluation of Our Model

After training the model and using the weights with the lowest validation loss over the 80 epochs the model obtained a 98.1% balanced categorical accuracy and a 0.237595 categorical crossentropy loss on the left out test set. Balanced accuracy was used since there was a class imbalance in the data set. Further, the model does not seem to suffer from over fitted as indicated in figure 3 which shows the test and validation loss history plotted against the number of epochs. This is also evident from figure 4 which shows the test and validation categorical accuracy plotted against the number of epochs.

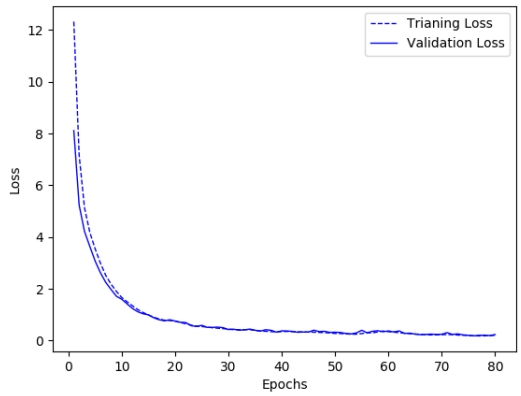

Figure 3: Test and Validation Loss History

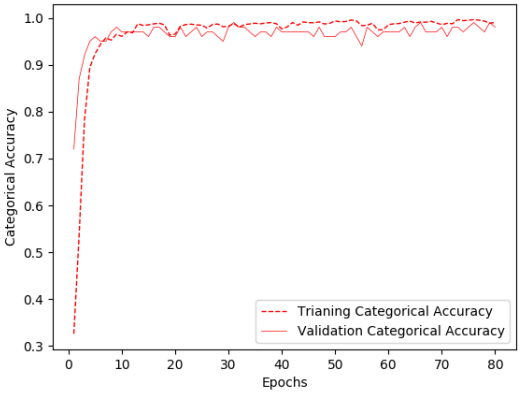

Figure 4: Test and Validation Balanced Categorical Accuracy History

4.2 Interpretation

After training, a process called Layer-wise Relevance Propagation (LRP) (Bach et al., 2015), which returns the relevances of all input nodes to a sample's prediction, was used to find important input pixels. In our experiment we used the iNNvestigate (Alber et al., 2018) python library's LRP implementation. Once we had the relevances of each input node we averaged them across the 6 channels to get a 2D relevance map. For visualization, we used the seismic colour map from matplotlib (Hunter, 2007) to plot the relevance map. Figure 5 shows the image resulting from taking the first 3 channels of the sample associated with the binary with ID 0AnoOZDNbPXIr2MRBSCJ, and the image resulting from taking the last 3 channels of the same sample, as well as its corresponding relevance map. The pixels highlighted with red contributed positively to the models prediction while the pixels

highlighted in blue contributed negatively. Sample 0AnoOZDNbPXIr2MRBSCJ was correctly classified by our model with a 99.99% chance of belonging to class 1.

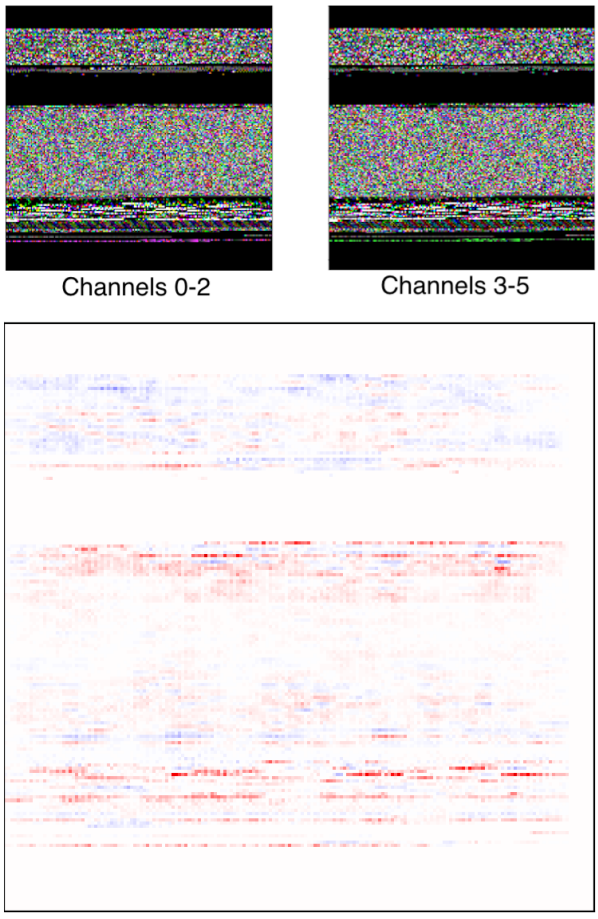

Figure 5: 0AnoOZDNbPXIr2MRBSCJ Relevance Map

Although we cannot obtain a lot of specific information by looking at these images and the LRP visualization, we can still obtain some broad insight into the models prediction. As you can see from figure 5, the classifier recognized a large set of pixels as an indication that this sample belonged to class 1. Further, there was still some pixels which contributed negatively to this prediction, mostly in the upper portion of the image above the band of white which corresponds with bytes of 0's. This suggest that the portion of the code containing functionality related to class 1 in located in the lower portion of the binary and the upper portion is mostly related to other classes or is benign. However, these statements are always dependent on how well the classifier was trained to learn relevant information. Further, if the relevance was intensely focused on a few small areas we could be worried that the model is relying on a small set of features to make classifications, meaning small changes could lead to misclassifications, and this can be taken advantage of by malware authors.

To obtain a fine grained interpretation such as highlighting specific lines of code, we obtain a list of indices of the input pixels sorted by their relevance. We then move down the list in descending order of relevance and obtain the most relevant 6-grams for interpretation first. The relevant 6-gram's position in the sample's input tensor, the input tensor itself, and the corresponding byte file are needed to determine the 6-gram's address in the assembly code. These are needed to determine the amount of padding to account for, as well as the starting address in the byte file, since they don't all start at the same number. However, in a application scenario all of this information would be available. Once we have the exact address in the assembly code it is a trivial task to find the lines of Assembly code which contain the relevant 6-gram.

It should be noted that finding the exact lines of code which contributed to a prediction, as well as their exact ordering relative to the size of their contribution, is very difficult, if not impossible with most other model architectures. For example, if n-gram analysis was used, where the presence or frequency of an n-gram is used as a feature, then even though we may have the contribution of each n-gram feature, we do not know which occurrence of said n-gram in the binary contributed the most. This is true in some way for most statistics based techniques. Further, for other CNN based techniques, we have the problem of rescaling causing the contribution of one input node being distributed across many pixels in the original image. It is only when the information of position of each 6-gram is maintained from raw binary to input tensor, as we have done so here, where we are able to so easily make such precise interpretations of the model.

Figure 6 shows the terminal output when working backwards from the relevant 6-grams to the code snippets in the assembly code. Note that some of the assembly code has been truncated so we could not find the corresponding code snippets to many of the most significant 6-grams but we have done so for the 100^{th} and 122^{nd} most significant. This is not a problem however as typically you would have the full assembly code.

The significance of finding these code snippets which contributed heavily to a prediction is large. For example, there is the case where the code snippets are completely irrelevant to classifying binaries according to functionality despite the model achieving good classification results. In this situation, there is the likely culprit of an incomplete or non-representative

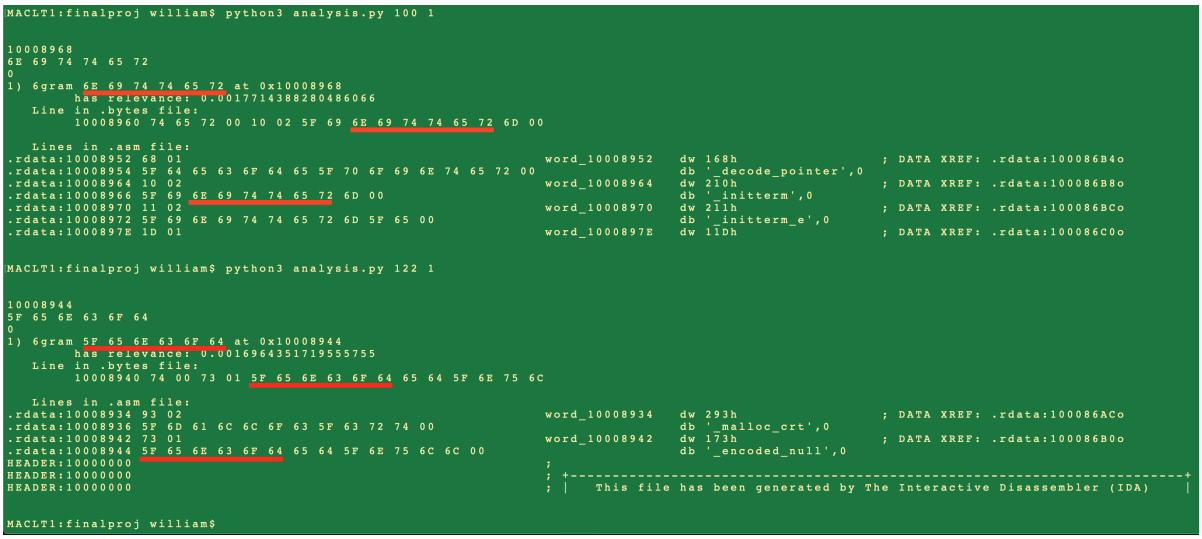

Figure 6

data set. It could be that one class has irrelevant but frequently occurring code-snippets that by chance do not appear in the other classes. Here, gathering a larger data set, or even augmenting the current data set so that other classes also include this irrelevant code snippet, can force the model to learn different patterns that exclude this irrelevant feature, which should also improve generalization performance. If the classification metrics drop after this, then this could hint at poor feature engineering, since the remaining representative features no longer help the model make predictions. In the case of CNNs applied to images of binaries, this could mean a deeper model architecture that can create more abstract hidden features might be needed, or that the representation of binaries as images themselves is unhelpful.

In the case where code snippets with high relevances to the model's predictions are known to be relevant to classifying binaries based off their functionality, then the results of the model are in a way, validated. As we said earlier, this can help malware analysts as they can be shown where to start their static analysis, as well as help stakeholders feel confident in the black-box CNN model.

4.3 Comparison With Other Models

Table 2 shows the confusion matrix of our model on the left out test set where we can see the model preformed well for all classes. Table 3 and 4 show the confusion matrix of the models used in (Chen, 2018) and (Gibert et al., 2019) respectively (both are discussed in section 2). The columns and rows for classes 3, 5, and 7, which were not used in our experiment, have been omitted.

Table 2: Confusion Matrix for Our Model on the Left Out Test Set

Class No.	1	2	4	6	8	9
1	**149**	0	0	**1**	0	0
2	0	**16**	0	0	0	0
4	**1**	0	**44**	0	0	0
6	0	0	0	**32**	**1**	**1**
8	**2**	0	0	0	**190**	0
9	**1**	0	0	0	0	**62**

Table 3: Confusion Matrix for Model used in (Chen, 2018)*

Class No.	1	2	4	6	8	9
1	**154**	0	0	0	**3**	0
2	0	**238**	0	0	**3**	**1**
4	**1**	0	**33**	**1**	0	0
6	**1**	0	0	**63**	**1**	0
8	**2**	0	0	0	**119**	0
9	0	**4**	0	0	0	**102**

*columns and rows of classes classes 3,5, and 7 have been omitted

Using these confusion matrices to calculate the balanced accuracy score of each model, we can get a decent comparison. However, the reader should note that the other models were trained on more classes and with much more training data so these are not perfect direct comparisons of our approaches. Table 5 sums up the comparisons between the three models. As you can see, we score competitive balanced accuracy score with a very light weight model. Further, we are able to give a fine grained analysis of our

Table 4: Confusion Matrix for Model used in (Gibert et al., 2019)*

Class No.	1	2	4	6	8	9
1	**1490**	**4**	**2**	**9**	**28**	**3**
2	**6**	**2440**	0	**7**	**8**	**16**
4	**3**	0	**461**	**1**	**3**	**2**
6	**8**	**6**	**2**	**713**	**10**	**9**
8	**44**	**4**	**8**	**17**	**1138**	**8**
9	**2**	**2**	0	**6**	**5**	**996**

*columns and rows of classes classes 3,5, and 7 have been omitted

predictions using the method detailed in section 4.3 and this method cannot be easily applied to the other models without added processing and sacrificing the preciseness of our method. This is due the decision to not rescale the images, meaning we can map one relevant input node to exactly one 6-gram in the binary and then to the corresponding assembly code.

Our model does have draw backs however. Unlike the other models ours is only designed to work with samples in a specific size range and therefore one would need multiple models in order to achieve the same effect across different size ranges. One possible solution would be to train on a data set where the samples in the appropriate size range are not rescaled but padded like we did here, and the images which are too large are rescaled to fit. This however would mean the interpretation method would only maintain its fine granularity when classifying samples which were not resized.

Table 5: Model Comparisons

Model	**Balanced Acc.**	**Size**
Our Model	98.1%	2 conv, 1 Dense
Model from (Chen, 2018)	97.04%*	20+ layers see (Szegedy et al., 2014)
Model from (Gibert et al., 2019)	96.8%*	3 conv, 1 Dense
Model	**Interpretation**	**Sample Size**
Our Model	Fine grained & precise	104k-200k Bytes
Model from (Chen, 2018)	Broad & imprecise	any size
Model from (Gibert et al., 2019)	none given	any size

*calculated by omitting columns and rows of confusion matrix for classes 3,5, and 7

5 CONCLUSION

In summary, we were able to obtain competitive classification results on a subset of a classic benchmark data set. Compared to other methods we made appropriate trade offs in terms of broad applicability of our model (in that it only works for malware in a specific size range) in return for large gains in interpretability. We thus have provided a proof of concept for 6-channel image based malware classification using a simple convolutional neural network that did not suffer from excessive overfitting despite a small data set. To the best of our knowledge, this the first time someone has been able to interpret a CNN based malware detector with the granularity which has been achieved here, by applying CNNs to a malware converted to images with more then a single channel in order to avoid rescaling of the image and information loss.

For future work, there is much work to be done in order to better handle binaries of different sizes. If more sophisticated approaches for dealing with binaries of different sizes are implemented, which do not result in information loss, then fine grained interpretations, in the manner we have done so here, can be possible for any malware file. Further, the data set will not shrink as a result of not considering files which are too large. This means more advanced models can be deployed without over fitting the data set, potentially increasing the models performance.

Another interesting possibility is to explore is the application of our approach to graph convolutional neural networks (GCNNs), which are CNNs applied to graph representations, trained on malware classification. It is a popular approach in malware analysis to represent the behaviour or other features of a malware binary in a graph which in many cases will contain direct and explicit functional information. If a GCNN is trained on a dataset where each sample is one of these graph representations of a malware binary, then less time can be spent worrying about weather the model learnt to use features indicative of functionality and translating those features to functionality afterward for interpretation. Instead interpretability can be used to directly make statements about the relevant functionality which the model is perceiving within the sample to make its prediction.

REFERENCES

Alber, M., Lapuschkin, S., Seegerer, P., Hägele, M., Schütt, K. T., Montavon, G., Samek, W., Müller,

K.-R., Dähne, S., and Kindermans, P.-J. (2018). innvestigate neural networks!

Bach, S., Binder, A., Montavon, G., Klauschen, F., Müller, K.-R., Samek, W., and Suárez, O. D. (2015). On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. In *PloS one*.

Chen, L. (2018). Deep transfer learning for static malware classification.

Chollet, F. et al. (2015). Keras. https://keras.io.

Cui, Z., Xue, F., Cai, X., Cao, Y., Wang, G., and Chen, J. (2018). Detection of malicious code variants based on deep learning. *IEEE Transactions on Industrial Informatics*, 14(7):3187–3196.

Gibert, D., Mateu, C., Planes, J., and Vicens, R. (2019). Using convolutional neural networks for classification of malware represented as images.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

Kolosnjaji, B., Eraisha, G., Webster, G., Zarras, A., and Eckert, C. (2017). Empowering convolutional networks for malware classification and analysis. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 3838–3845.

LeNail, A. (2019). Nn-svg: Publication-ready neural network architecture schematics. *Journal of Open Source Software*, 4:747.

Nataraj, L., Yegneswaran, V., Porras, P., and Zhang, J. (2011). A comparative assessment of malware classification using binary texture analysis and dynamic analysis.

Peltola, T. (2018). Local interpretable model-agnostic explanations of bayesian predictive models via kullback-leibler projections. *ArXiv*, abs/1810.02678.

Raff, E., Zak, R., Cox, R. J., Sylvester, J., Yacci, P., Ward, R., Tracy, A., McLean, M., and Nicholas, C. (2016). An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, 14:1–20.

Rezende, E., Ruppert, G., Carvalho, T., Ramos, F., and De Geus, P. (2017). Malicious software classification using transfer learning of resnet-50 deep neural network.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "why should I trust you?": Explaining the predictions of any classifier. *CoRR*, abs/1602.04938.

Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., and Ahmadi, M. (2018). Microsoft malware classification challenge.

Shrikumar, A., Greenside, P., and Kundaje, A. (2017). Learning important features through propagating activation differences. *ArXiv*, abs/1704.02685.

Su, J., Vasconcellos, D. V., Prasad, S., Sgandurra, D., Feng, Y., and Sakurai, K. (2018). Lightweight classification of iot malware based on image recognition. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, pages 664–669.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions.

Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng (2017). Malware traffic classification using convolutional neural network for representation learning. In *2017 International Conference on Information Networking (ICOIN)*, pages 712–717.

Zhang, J., Qin, Z., Yin, H., Ou, L., and Hu, Y. (2016). Irmd: Malware variant detection using opcode image recognition. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1175–1180.