Classifying Sequences of Extreme Length with Constant Memory Applied to Malware Detection

Edward Raff, [1,2,3] **William Fleshman,** [4] **Richard Zak,** [1,2,3]
Hyrum S. Anderson, [5] **Bobby Filar,** [6] **Mark McLean** [1]
[1] Laboratory for Physical Sciences, [2] Booz Allen Hamilton, [3] University of Maryland, Baltimore County
[4] U.S. Army, [5] Microsoft, [6] Elastic
{edraff, rzak, mrmclea}@lps.umd.edu, {raff_edward, zak_richard}@bah.com, {raff.edward, richard.zak}@umbc.edu,
william.c.fleshman.mil@mail.mil, hyruma@microsoft.com, robert.filar@elastic.co

Abstract

Recent works within machine learning have been tackling inputs of ever-increasing size, with cybersecurity presenting sequence classification problems of particularly extreme lengths. In the case of Windows executable malware detection, inputs may exceed 100 MB, which corresponds to a time series with $T = 100,000,000$ steps. To date, the closest approach to handling such a task is MalConv, a convolutional neural network capable of processing up to $T = 2,000,000$ steps. The $\mathcal{O}(T)$ memory of CNNs has prevented further application of CNNs to malware. In this work, we develop a new approach to temporal max pooling that makes the required memory invariant to the sequence length T. This makes MalConv $116\times$ more memory efficient, and up to $25.8\times$ faster to train on its original dataset, while removing the input length restrictions to MalConv. We re-invest these gains into improving the MalConv architecture by developing a new Global Channel Gating design, giving us an attention mechanism capable of learning feature interactions across 100 million time steps in an efficient manner, a capability lacked by the original MalConv CNN. Our implementation can be found at https://github.com/NeuromorphicComputationResearchProgram/MalConv2

1 Introduction

Cybersecurity has received increased attention from machine learning practitioners and researchers due to the number of challenges that exist within the space. Industry datasets are routinely measured in petabytes (Spafford 2014), have noisy labels, are both structured and unstructured, suffer from continuous concept drift (Kantchelian et al. 2013), and adversarial attacks have been well motivated as a daily occurrence for decades (Rajab et al. 2011). In this work we are interested in the task of *static malware detection*, where using the on-disk byte representation, one wishes to predict if a new executable program is benign or malicious. Current industry models rely heavily on domain knowledge feature extraction, which is time consuming and expensive, and requires intimate knowledge of Windows and low-level assembly and software design. Because malware authors adapt, this feature engineering is a continuous processes, which can require reverse engineering effort to determine what new features should be extracted. To quantify the cost of such efforts, a single program can take weeks for experts with decades of experience to reverse engineer (Votipka et al. 2019), so the ability to build models that perform their own feature extraction and construction can save an enormous amount of time if successful.

Toward this goal, we follow the approach of MalConv, which proposed to tackle the problem of malware detection as a time series classification problem (Raff et al. 2018). For an input file x of T bytes in length, a neural network must learn to produce an output label $y \in \{\text{Benign}, \text{Malicious}\}$. The MalConv architecture was relatively small (≈ 1 million parameters) but represented a meaningful malware detector by performing convolutions over raw byte inputs. Additionally, the work identified a number of challenges with this task. In particular, their approach would process up to 2 MB of a file—equivalent to a time series prediction problem with $T = 2,000,000$ steps. The next longest time series task we are aware of is only on the order of $\leq 16,000$ steps (van den Oord et al. 2016). Due to the extreme length of raw byte inputs, the MalConv solution required an NVIDIA DGX-1 with 128 GB of GPU memory to train over one month of compute time. This has made MalConv difficult to replicate, while simultaneously neglecting the fact that 2 MB is relatively small with respect to the distribution of observed executable file sizes, where the tails can reach in excess of 100 MB.

In this work, we produce a solution to the high memory cost to train MalConv, making the memory use *invariant* to the length of the input — allowing us to train on data points in excess of 200,000,000 time steps in length using a single GPU. This reduces the memory used to train MalConv by a factor of $116\times$ while simultaneously providing an up to $25.8\times$ speedup, reducing the compute requirements from a DGX-1 down to a free Google Colab instance. Our approach leverages the sparse gradients of temporal max pooling to cap the memory requirement during training on long inputs. By significantly reducing the runtime and memory constraints of MalConv, we are able to explore more advanced architectures for the task of time series classification. In particular, we develop a new *global channel gating* (GCG) that allows us to enhance MalConv to learn interactions of features across the entire input space. GCG can be implemented in only 7 lines of Python, making it easy to implement while improving the

accuracy of the end-to-end deep malware detection model. Although we explicitly address malware detection where long input sequences are dramatic, our contributions are relevant generally to deep neural networks with long input sequences, some of which are discussed in the following section. We also note the task of learning interactions over a sequence of unprecedented length of intrinsically interesting from a pure ML perspective and benefits a real-world task.

We have organized the paper as follows. In §2 we will review the work related to MalConv, which has received significant attention from the cybersecurity community that motivates our research, as well as other work in the domain of processing long input sequences. Next we will detail our approach to making the memory cost of MalConv style architectures invariant to feature length in §3. These improvements are necessary to make our global channel gating possible, which we detail in §4. The results detailing our speedups and memory improvements are presented in §5, followed by our conclusions in §6.

2 Related Work

The desire to perform malware classification from raw bytes, to alleviate expensive and constant feature engineering, has long existed. This was originally based on the Normalized Compression Distance (NCD) (Li et al. 2004), which has found extensive use for this task (Wehner 2007; Bailey et al. 2007; Hayes, Walenstein, and Lakhotia 2008; Bayer et al. 2009; Borbely 2015; Alshahwan et al. 2015; Faridi, Srinivasagopalan, and Verma 2019; Menéndez et al. 2019; S. Resende, Martins, and Antunes 2019; Walenstein and Lakhotia 2007). Recent works like LZJD (Raff and Nicholas 2017, 2018; Raff, Aurelio, and Nicholas 2019) and BWMD (Raff, Nicholas, and McLean 2020) are built from compression algorithms and useful in unsupervised settings, but are less effective in supervised ones. We will use these methods as baselines to compare against.

MalConv was the first proposed approach to detect malware from raw bytes, processing inputs of up to 2 MB in length (Raff et al. 2018). Through a broad search across network architectures, the authors report that many classical "best practices" for neural network architectures did not apply. For example, they found that BatchNorm prevented convergence, and that a network with 1 layer of extremely wide convolutions performed better than deeply stacked narrow filters. Since (Raff et al. 2018), a number of others have replicated their approach or proposed alterations to better study it, but all have reduced the input size in order to reduce computational costs. Authors from the anti-virus company Avast restricted their study to files that where $\leq$ 512 KB (Krčál et al. 2018). Their work is notable for being the first to compare the approach with hand engineered domain knowledge features from their production malware classifier. They found that the CNN was close in performance, and combining the domain and CNN features improved accuracy by 4%, indicating the CNN was learning features or feature interactions not previously found by domain experts. FireEye did an in-depth reverse engineering of what a MalConv-like network learned showing it corresponded well to what an analyst would look for, but had to restrict their model to 100 KB(Coull and Gardner 2019). Anderson and Roth (2018) introduced the Ember dataset and found MalConv slightly worse than hand-crafted features, but needed 25 hours/epoch to train on up-to 1 MB. Recent work(Galinkin 2019) has even shown MalConv has an ability to generalize across x86 architectures, detecting x86 macOS and Linux malware when trained only on Windows data. Other works have used the same or similar architectures to perform malware detection on datasets other than Windows executables, including for Android APKs(Hasegawa and Iyatomi 2018), PDF files(Jeong, Woo, and Kang 2019), as well as malicious JavaScript and Visual Basic code detection(Stokes, Agrawal, and McDonald 2018).

These works have all demonstrated the value of the byte based approach to malware detection, but simultaneously show the computational limitations. These solutions all suffer from an artificial limit in the maximum file size imposed by memory constraints; these potentially degrade performance and enable easy evasion in an adversarial scenario. Many works have shown MalConv is susceptible to evasion (Demetrio et al. 2019; Kolosnjaji et al. 2018; Kreuk et al. 2018; Fleshman et al. 2018), but these attacks can be thwarted at a cost to accuracy (Fleshman et al. 2019). This defense is only moderately effective because MalConv can be thwarted by simply inserting the malicious payload after the 2 MB file limit. Because malware authors are real active adversaries attempting to evade detection, this is a serious limitation. After years of activity and development, our work finally removes this trivial limitation from this research area, which also makes (Fleshman et al. 2019) more effective.

While MalConv has received significant interest for its applications in malware detection, few other works within machine learning approach the same length of sequence processing. Recent work extending the Transformer approach to more efficiently handle long inputs has reached $T = 64,000$ time steps (Kitaev, Kaiser, and Levskaya 2020). While the Transformer is able to learn more robust representations than our current work, it is still orders of magnitude too short to be able to process most executable files. Work by Voelker, Kajić, and Eliasmith (2019) proposed an extension of Recurrent Neural Networks, showing them to be capable of learning on synthetic time series of $T = 1,000,000$ steps. Their approach requires over an hour to process a single time series of this length, making it computationally infeasible — where our approach enables MalConv to run on a similar length input in under 42 milliseconds. While our approach improves the representational power of MalConv and is faster to train, it has less representational power compared to these other works. We provide more details on the failed attempts with transformers, and other approaches in a "What Did Not Work" the appendix.

Our approach to fixing the memory cost of MalConv is similar to checkpoint (or "rematerialization") (Griewank and Walther 2000). This approach involves re-computing results during the backward pass to avoid saving results in the forward pass, trading more compute for less memory but guaranteeing identical results. All work in this domain has focused on ways to balance this trade off for different types of acyclic network graphs (Chen et al. 2016; Gruslys et al. 2016; Kumar

et al. 2019; Kusumoto et al. 2019; Beaumont et al. 2020). Our work instead performs recomputation in the forward pass, so that the backward pass produces an equivalent result, while using less compute time and less memory.

Although we focus exclusively on the application of malware detection from byte sequences, we note that other domains may similarly benefit from tools for classification over long time series. For example, Genome Wide Association Studies (GWAS) can exceed 500,000 time steps in length, and have long dealt with issues in discovering interactions across GWAS(Wu et al. 2010). When constrained to smaller sequences with $T \leq 5000$, architectures similar to MalConv have found use for GWAS based prediction tasks (Liu et al. 2019).

3 Fixed Memory Convolution Over Time

The original MalConv architecture is shown in Figure 1. It contains an embedding layer (of $\mathbb{R}^8$) that is used over an alphabet of 257 tokens, 256 bytes + an End of File marker. These are fed into two sets of 128 convolutional filters with a width of 512 and a stride of 512[1], which are then used in a gating approach proposed by (Dauphin et al. 2017). The gated result is then converted to a fixed length feature vector using temporal max pooling (i.e., global max pooling, or max pooling over time), after which it is fed into a simple fully connected layer for prediction of the benign/malicious label. Since there is only one layer, the receptive window size W is equal to the kernel width 512.

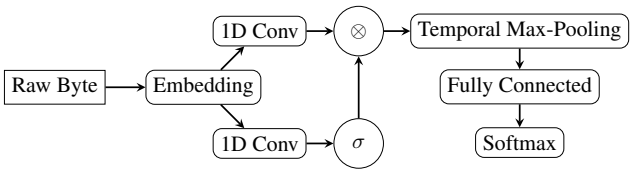

Figure 1: Original MalConv architecture (Raff et al. 2018) with ≈ 1M parameters, but required 128 GB of GPU RAM to train. $\otimes$ indicates element-wise product, and σ the sigmoid activation.

Despite its simplicity, MalConv was the first architecture to demonstrate that neural networks could learn to perform malware detection from raw bytes and the first to show classification over time series/sequences of up to $T = 2,000,000$ steps. However, only the first 2 MB of the input was processed in training MalConv because it required 128 GB of GPU memory to train on a batch of 256 files up to the 2MB limit. This is owing to the large memory cost of performing an embedding and convolution over a time series of 2 million steps (1 for each byte), and the resulting activations alone require almost all of the GPU memory. Every subsequent work we are aware of has processed less than the original 2 MB cap.

[1]Originally a width and stride of 500 was used, but it has been noted in several works that using a power of two performs better due to assembly code being aligned on powers of two when written to an executable.

To overcome these issues, we developed a novel Temporal Max-Pooling strategy that makes memory costs invariant to the sequence length T. Importantly, we do this by noting that Temporal Max-Pooling causes the gradient with respect to the sequence to be sparse. For C channels, saving all $C \cdot T$ activations is unnecessary, as only C values will actually be used, one for each channel. Thus we are using many times more memory than needed to train, and also performing redundant GPU computations on the backward pass since the majority of gradient values are exactly 0. When working with normal images and standard applications of max-pooling, the sparsity ratio may be 1:2 or 1:4, which is generally not sparse enough to make exploitation of that fact useful. This is because every non-zero value requires storing its associated index, doubling the memory use of those values. Second, operations on dense vectors/matrices result in more efficient computation, delivering computational throughput closer to the theoretical limits of modern hardware when using modern BLAS libraries and software like CUDNN. As such, libraries like PyTorch and Tensorflow do not support sparse gradients through max-pooling.

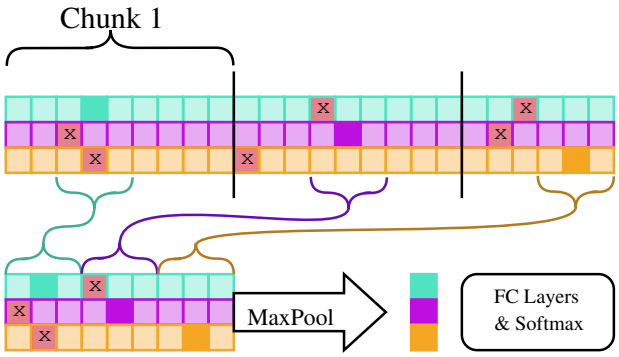

Figure 2: Diagram of Temporal Max Pooling with fixed memory. The original input (top) is a 1D sequence with 3 channels and is broken up into four chunks based on window size $W = 3$. Without gradient computation/tracking, the maximum activation index is found within each chunk. Solid colors show max values kept, "×" max in chunk but no maximal. Winning indices are copied to a new shorter sequence (bottom), which runs with gradient tracking. The result is the same output and gradient, but fixed memory cost.

Conversely, we obtain the benefits of sparse activations while also retaining the higher computational throughput of dense operations, without requiring any new code, as follows.

1. Turn off gradient computation (e.g., `with torch.no_grad():` if using PyTorch) and break the input sequence of length T into at most $T/(W \cdot 2)$ *overlapping* chunks of size $W \cdot 3$.
2. Perform max pooling over each chunk; for each channel track the maximum value and its absolute index.
3. Compare values within each chunk to a set of global winners. If a new chunk's maximal activation exceeds the global winner, it becomes the new global winner.

Once we have processed all chunks, we know the C locations, one for each channel, that will win the max-pooling over time.

The chunks overlap so that this computation is correct, and not impacted by windowing issues.

With these C locations, we may simply concatenate their values into a new sequence of length $T' = C \cdot W$. This new sequence is now small enough that the full set of embedding, convolutional layers, and temporal max pooling can be done in a dense fashion (retaining computational efficiency benefits), using memory proportional to what would be achieved with sparsity-exploiting code. The total memory use is now independent of the original sequence length T, and a diagram of the process is presented in Figure 2.

Details on windowing artifacts: We noted that in the concatenation of different chunks in Figure 2 into one new sequence, it is technically possible for a new index to become the maximal activation due to the receptive window length W crossing between two chunks that were previously not adjacent. This results in a pattern that has potentially not been seen previously, which thus creates new activation values. We have never observed this issue in practice, and so have not taken any special steps to avoid this situation (with more details in the appendix).

This hypothetical issue could be prevented by performing the convolution and a max-pool over the chunks independently. Then, the pooled results could be concatenated and a second round of pooling performed. We have not observed any issues warranting this extra complexity and overhead.

4 Global Channel Gating

With an efficient method for handling large input sequences, we can explore a broader set of neural network architectures. In particular, we note a weakness in the original design of MalConv: the use of temporal max-pooling after a single convolutional layer results in a somewhat myopic model: learned features are purely local in nature. That is, with the existing architecture, the model output does not consider interactions between features that are far apart in time within an input sequence/file.

To demonstrate why this is important in malware detection, consider that a common feature to learn/extract is the use of encryption libraries, which may indicate, for example, functionality common in ransomware. However, if the program does not access the file system, the use of encryption becomes less suspicious and less likely to indicate malware. In its current embodiment, it is impossible for MalConv to learn logic like this because the presence/absence of the associated information may be in disparate regions of the input, and the receptive window of the network (512 bytes) is far smaller than most inputs (2^{21} bytes).

To endow our network with the ability to learn such relationships while retaining computational tractability, we develop a new attention inspired gating approach we call *global channel gating* (GCG). The idea is that given a long time sequence with C channels, $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T\}$ where $\boldsymbol{x}_t \in \mathbb{R}^C$, we want to globally suppress certain time steps based on the content of all channels. We approach this in a style similar to the gated linear unit and the additive attention (Dzmitry Bahdana et al. 2015), using a learned context $\bar{\boldsymbol{g}} \in \mathbb{R}^C$, as shown in Eq. 1.

$$GCG_W(\boldsymbol{x}_t, \bar{\boldsymbol{g}}) = \boldsymbol{x}_t \cdot \sigma\left(\boldsymbol{x}_t^\intercal \tanh\left(W^\intercal \bar{\boldsymbol{g}}\right)\right) \quad (1)$$

The entries of the vector $\boldsymbol{x}_t \in \mathbb{R}^C$ at time t may be suppressed by the scalar quantity on the right hand side of the GCG equation. Due to the sigmoid operation $\sigma(\cdot)$, $\boldsymbol{x}_t$ will be scaled by a value in the range of $[0, 1]$, resulting in a context sensitive suppression of each entry in the vector.

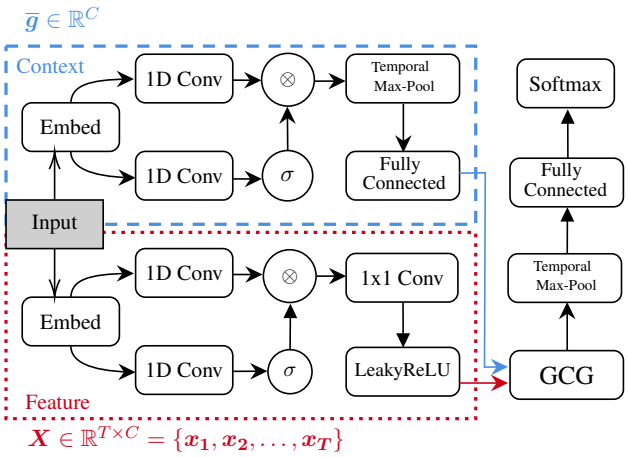

Figure 3: Our new proposed architecture with global channel gating (GCG). The blue thick dashed sub-network shows the context extractor, which is used to suppress information found from the Feature extraction sub-network (red, thinly dashed sub-network).

We detail a new malware classification architecture which we term *MalConv with GCG* in Figure 3 that leverages GCG. The top half of the network serves to learn a global context $\bar{\boldsymbol{g}}$, which is used as input to the GCG. The bottom half of the architecture shows the feature sub-network, which uses a different embedding layer to perform initial feature extraction and uses GCG to selectively suppress regions of the input, allowing for potential feature interactions over time. The inputs to GCG are a state vector from the top half context network, and a sequence over time generated from the bottom half, which has Eq. 1 applied point-wise over time. This is followed by temporal max pooling, where we apply the fixed memory approach from §3 to make the training feasible with fixed memory costs.

4.1 Gating via Convolution

Care must be taken to implement the GCG approach effectively. The naive strategy to implement GCG requires reshaping the input array, and either running over every time step with a for loop to extract a slice $\boldsymbol{x}_t$ and perform a dot product, or alternatively, duplicating the context $\bar{\boldsymbol{g}}$ into a larger matrix and performing a larger BLAS operation against the input X. The first approach suffers from excessive Python and auto-grad overhead in our testing. The latter approach is more efficient in terms of FLOPs, but still cumbersome and slow due to the duplication of $\bar{\boldsymbol{g}}$.

Instead, we exploit the nature of grouped convolutions (Krizhevsky, Sutskever, and Hinton 2012) to efficiently im-

```
def gcg(self, X, g):
  # X.shape = (B, T, C)
  B, T, C = X.size(0), X.size(1), X.size(2)
  # g.shape = (B, C)
  # create context vector z = tanh(W^T g)
  # self.w references a nn.Linear(C, C)
  ↪  layer
  z = torch.tanh(self.w(g))

  # Size is (B, C), but we need (B, C, 1) to
  ↪  use as a 1d conv filter
  z = torch.unsqueeze(z, dim=2)
  # roll the batches into the channels
  x_tmp = X.view(1,B*C,-1)
  # apply a conv with B groups; each batch
  ↪  gets its own context applied
  # This computes x_t^T z forall t = 1...T
  x_tmp = F.conv1d(x_tmp, z, groups=B)
  # x_tmp has a shape of (1, B, T); re-order
  ↪  as (B, 1, T)
  gates = x_tmp.view(B, 1, -1)

  # effectively apply x_t · σ(x_t^T tanh(W^T g))
  return X * torch.sigmoid( gates )
```

Figure 4: PyTorch code demonstrating how to implement global channel gating in a computationally efficient manner. The input context g is projected and re-shaped, such that it can be used as the filter weights in a 1D convolution grouped by the batch size. This results in computing the dot product over time.

plement the GCG over time. Given a batch of B time series, we reinterpret the input activation/context $\bar{g}$ as a set of 1D convolution weights/filters in a $B \times C \times 1$ matrix, and perform a grouped convolution with B groups. Thus we convolve the context with the input X where the window size is 1 (considering only one time-step at a time), the B different contexts become the number of output "channels", and by grouping each context is applied only to its appropriate input. The grouped convolution allows us to apply the different filters to each batch in one operation. We find this easiest to demonstrate with code, and present a working PyTorch implementation of GCG in Figure 4. With this additional insight, the GCG operation is no more expensive than a 1×1 convolution, allowing us to leverage it for inputs with hundreds of millions of time-steps without issue.

5 Results

The Ember2018 corpus (Anderson and Roth 2018) has 600,000 training samples and 200,000 test samples. At ≈1 TB it is our primary test set due to size and difficulty. Both training and testing sets are evenly split between benign and malicious, and all testing samples were first observed after all training samples. The predecessor 2017 corpus was explicitly noted to be "easy" due to the way it was created, and MalConv obtained an AUC of 99.8%, close to that of a domain knowledge approach which achieved 99.9% AUC. We prefer the 2018 corpus because it was designed to be more challenging, and MalConv obtains an accuracy of only

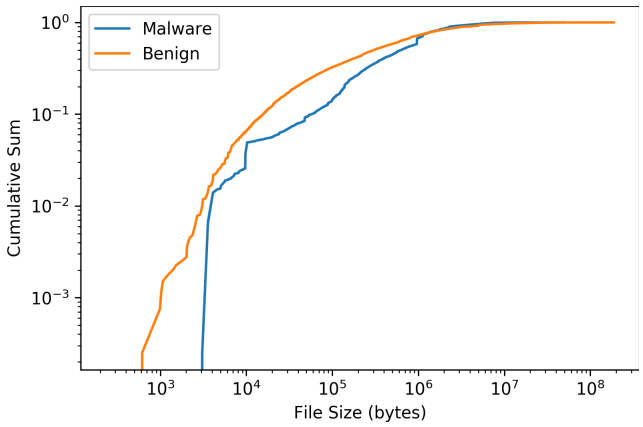

Figure 5: Distribution of file lengths (x-axis, log-scale) and percentage of files of an equal or lesser size (y-axis, log-scale) for all files in the Ember2018 corpus. The largest file is 271.1 MB.

91% on the newer corpus. The domain knowledge features were less impacted, dropping to only 99.6% AUC. This better demonstrates the gap between current deep learning and domain knowledge based approaches for classifying malware. We also use the Common Crawl to collect 676,843 benign PDF files and VirusShare (Roberts 2011) 158,765 malicious ones. This gives 464 GB of data total, with 10% used as a test set. Malicious PDF files are easier to detect than malicious executables, so the effect size of our improvements are expected to be smaller. We include this test to show that our methods still work on other types of data.

The distribution of file lengths in bytes is shown in Figure 5, with the longest file corresponding to a time series with 271,082,368 time steps. This is $135.5\times$ longer than the original MalConv work, and thus two orders of magnitude longer than any previous time-series classification task we are aware. We were able to train Malconv with and without GCG on these data without any truncation. This removes the trivial adversarial attack of moving malicious code past the 2 MB limit.

For all networks, we trained using the Adam optimizer (Kingma and Ba 2015) with the recently proposed decoupling of weight-decay (Loshchilov and Hutter 2019), using the recommended default parameters. A batch size of 128 was used in each experiment. All experiments were performed on a DGX-1. We note that our improved training procedure no longer requires this level of compute power, however, we do this to appropriately compare training time in our experiments with previous work. We denote MalConv trained with the original approach, truncating to the first 2MB of the input file, as "MalConv (2MB, Orig)". In what follows, we use "MalConv" to denote the original architecture from Figure 1 trained with our new fixed-memory approach specified in §3. Finally, our new MalConv with GCG from §4 will be the last model we train for comparison. Both MalConv and MalConv with GCG are trained to processes the entirety of the input files, up to 271 MB. We train all models for 20 epochs, using the result from the last epoch.

For MalConv we use a filter size of 512, a stride of 512, 128 channels for each 1D Conv block, and an embedding dimension of 8. For MalConv with GCG we use a filter size of 256, a stride of 64, 256 channels for each convolution, and an embedding dimension of 8. For all models we incorporate the suggestion of (Fleshman et al. 2019) of including a special token after the EOF that maps to an embedding vector of all zeros. Details on the hyper-parameter selection, including attempts at improving the standard MalConv, can be found in the appendix. Below we will show the results indicating how our methods have improved MalConv, and we provide a discussion of other attempts to improve upon the MalConv approach that were unsuccessful, and how they impacted our approaches' final design in the appendix.

5.1 Training MalConv with Fixed-Memory Max-Pooling

Model	Time Per Epoch	GPU RAM
MalConv (2MB, Orig)	21 hr 29 min	128 GB
MalConv	1 hr 10 min	1.1 GB
MalConv w/ GCG	4 hr 5 min	2.2 GB

Table 1: Results on training time and computational efficiency.

We first evaluate the impact of our fixed-memory approach to training over long sequences. The original MalConv required 128 GB of GPU memory, and 21.5 hours per epoch on the Ember2018 dataset. In Table 1 we can see the timing information and memory use compared to our new approaches.

Our fixed-memory approach to temporal max pooling results in significant benefits, with a $116\times$ improvement in memory use and a $18.4\times$ reduction in training time. This takes MalConv training down from the order of a month to just a day. We note that the results are further improved when we consider that fixed-memory pooling is faster while processing more data, since it considers the entirety of each file. Since 14.2% of files are greater than 2MB, we are actually processing a total $1.4\times$ more data than the original MalConv, making our speedup effectively $25.8\times$ per byte processed. Our new approach makes it possible now for anyone with a GPU and data to train MalConv.

Without these speed and memory improvements, our new MalConv with GCG architecture would not have been possible to train. Naive scaling of the results indicates we would have needed 256 GB of GPU RAM (which would have only been possible with a DGX-2), and approximately 1 month of training time.

5.2 Improved Accuracy

In Table 2 we show the classification performance of all three models, and two state of the art compression based methods LZJD and BWMD using 9-nearest neighbor classification. We see that training MalConv with the prior approach but on the entire sequence length has no appreciable difference in accuracy (fluctuations of 0.1 percentage points). This shows

Model	Accuracy	AUC
MalConv (2MB, Orig)	91.27	97.19
MalConv	91.14	97.29
MalConv w/ GCG	**93.29**	**98.04**
LZJD	73.43	84.98
BWMD	81.97	91.12

Table 2: Ember 2018 results on accuracy and AUC for each model.

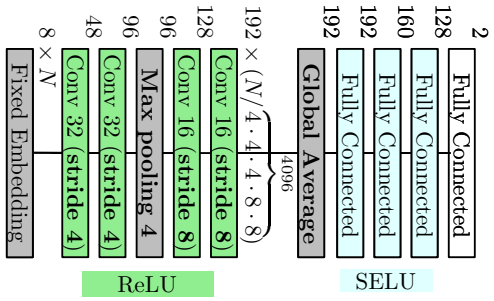

Figure 6: "AvastConv" architecture from (Krčál et al. 2018). Their approach was originally trained on entire executables, but each executable was $\leq$ 512 KB.

1) that we are able to still learn effectively while processing more information, and 2) that our approach does not hinder training in any way. As noted previously, parsing all of the input file is also beneficial for thwarting the trivial attack of moving all malicious code to the end of an executable. We also see that our MalConv with GCG improves upon the accuracy by 2.2% and AUC by 0.87% of the original MalConv architecture.

We prefer evaluation on the Ember 2018 corpus because it is both large and challenging. Our evaluations on the PDF corpus are done to show that our improvements transfer to other file types as well. On our PDF corpus we obtain an Accuracy of 99.16% and an AUC of 99.76%. MalConv with GCG improves this to 99.42% and 99.80%. Because PDF file are easier to processes, the baseline MalConv is already nearing maximal performance, so the gain is smaller — but shows our GCG approach is still an improvement.

5.3 Ablation Testing of Avast Architecture

A difficulty of research in this space is that large testing of over millions of executables can only be done in partnership with commercial anti-virus companies, who have large corpora to test against. Of the prior works we discussed in §2, the work by Krčál et al. (2018) is of particular interest for two reasons, 1) they found that global max pooling produced a 20% relative drop in performance compared to their use of global average pooling, and 2) it is the only extension to MalConv we are aware that is easy to adjust with our fixed memory max pooling form §3. The architecture they use,

which we will call *AvastConv*, is given in Figure 6. Its primary differences are the use of more layers of smaller filter widths (32 followed by 16), a hard coded embedding rather than a learned embedding, and the aforementioned use of global average pooling instead of global max pooling.

The pooling difference was the largest factor according to the ablation testing by (Krčál et al. 2018) at 20%. They found that fixed vs learned had no performance impact, and other differences between their and our current architectures accounted for no more than a 4% difference. The biggest untested factor between these works is that their study was constrained to smaller executables $\leq 512KB$ in size, where in our work we consider unbounded size with inputs over 200 MB in size.

As such, we choose to perform a small ablation test against this architecture, replacing the global average pooling with our new fixed memory temporal max pooling. Training their architecture for 20 epochs, we obtain an accuracy of 85.8% and an AUC of 94.6%. These results are significantly lower than MalConv and our improved version shown in Table 2.

While it may be possible that global averaging would restore performance to their approach, there is not enough remaining accuracy for a 20% relative improvement to occur. This would seem to indicate their initial results on the strength of global pooling are not as strong when factoring in larger file sizes. This is beneficial from the perspective that we can use max pooling to achieve fixed memory cost, which is not possible with average pooling.

These results also give credence to a relatively shallower architecture with wider convolutional filters, which is maintained in our current design. This runs in contrast to normal applications of CNNs in the vision, signal, and natural language processing domains, where the community has more firmly rested on smaller filters with more layers being the canonical design approach.

5.4 Example of Interactions Over Time With GCG

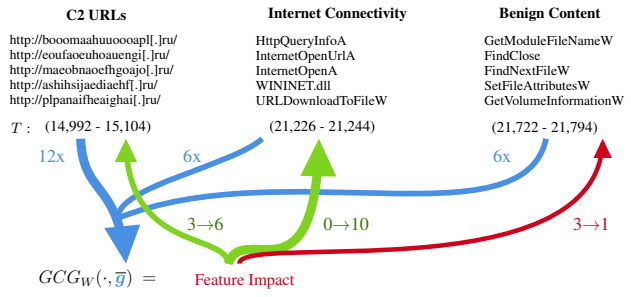

Figure 7: The time steps T that feature content is found is shown in parenthesis. $\bar{g}$ suppressed increased focus on internet functionality. This shows GCG can related disconnected portions with no overlapping receptive field, learning complex interactions.

As our last result, we demonstrate an example of how our new GCG has effectively learned non-linear interactions across large time ranges. In Figure 7 we show a diagram of the relevant content from a malicious sample, and how the context impacts the selected features once the temporal max-pooling is performed. In particular, three types of content found: Command-and-Control ("C2") URLs used for remote control of the malware, system calls used for internet connectivity, and other innocuous benign content. The blue lines into the context vector $\bar{g}$ denote the number of filters that had their maximum activation occur in the byte range of that content, with 12 filters selecting the C2 URLs, and 6 each for the Internet connectivity and other benign content. The values in parenthesis indicate the time-step T (i.e., byte location) of the content within the file. We can see that the C2 URLs are $\geq$6,122 steps/bytes away from the rest of the content, far larger than the receptive field of the convolutions.

After the GCG gating is applied to the activations of the feature sub-network, we see a large change in what is selected by the final max-pooling operation. Without GCG, none of the internet connectivity features were selected. In this case, the GCG suppresses the activations of other regions in the binary, but opens the gate (i.e., $GCG_W(x_{21,226}, \bar{g}) \approx 1.0$) for the internet content. As such, 10 of the filters now activate for this region. Similarly, the number of filters activating for the C2 URLs increases from 3 to 6. We also see that the innocuous content for working with the file system is suppressed by the gate, reducing activations from 3 down to 1. Combined, the C2 URLs and the use of APIs to connect to them over the Internet are significant indicators for confirming this file as malicious, which the network successfully performs. This is helpful information for malware analysts, and others who wish to know how the malware performs.

This malicious examples demonstrates that our GCG mechanism successfully learns the kinds of nuanced interactions we desire over large time ranges. The use of file system and internet connectivity is intrinsically non-suspicious on their own. Correctly focusing on the right content requires observing the suspicious URLs contained withing the file. Simultaneously, this shows MalConv learning to perform sub-tasks, like determining if a URL looks suspicious or not, to make these informed contextual gating decisions. Because this kind of analysis is expensive, we include more results from the PDF corpus in the appendix.

6 Conclusion

Prior approaches to classifying time series of extreme length where limited by the memory required to train the models. We have developed a new approach that exploits the sparsity of temporal max pooling to make this memory cost invariant to the time series length, while simultaneously being faster to compute. Using this improvement we designed a new approach to malware detection from raw bytes using a global channel gating mechanism that gives us the capability of learning feature interactions across time, despite extreme input lengths in excess of 100 million time steps. This contribution further improves the accuracy of our malware detection model by up to 2.2%.

References

Alshahwan, N.; Barr, E. T.; Clark, D.; and Danezis, G. 2015. Detecting Malware with Information Complexity. *arXiv* URL http://arxiv.org/abs/1502.07661.

Anderson, H. S.; and Roth, P. 2018. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints* URL http://arxiv.org/abs/1804.04637.

Bailey, M.; Oberheide, J.; Andersen, J.; Mao, Z. M.; Jahanian, F.; and Nazario, J. 2007. Automated Classification and Analysis of Internet Malware. In *RAID*, 178–197. URL http://dl.acm.org/citation.cfm?id=1776434.1776449.

Bayer, U.; Comparetti, P. M.; Hlauschek, C.; Kruegel, C.; and Kirda, E. 2009. Scalable , Behavior-Based Malware Clustering. *NDSS* 9.

Beaumont, O.; Herrmann, J.; Pallez (Aupy), G.; and Shilova, A. 2020. Optimal memory-aware backpropagation of deep join networks. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378(2166): 20190049.

Borbely, R. S. 2015. On normalized compression distance and large malware. *Journal of Computer Virology and Hacking Techniques* 1–8.

Chen, T.; Xu, B.; Zhang, C.; and Guestrin, C. 2016. Training Deep Nets with Sublinear Memory Cost. *arXiv* 1–12.

Coull, S. E.; and Gardner, C. 2019. Activation Analysis of a Byte-Based Deep Neural Network for Malware Classification. In *In Proceedings of the 2nd Deep Learning and Security Workshop (DLS)*. San Francisco, CA: IEEE, URL https://arxiv.org/pdf/1903.04717.pdf.

Dauphin, Y. N.; Fan, A.; Auli, M.; and Grangier, D. 2017. Language Modeling with Gated Convolutional Networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, 933–941. International Convention Centre, Sydney, Australia: PMLR, URL http://proceedings.mlr.press/v70/dauphin17a.html.

Demetrio, L.; Biggio, B.; Lagorio, G.; Roli, F.; and Armando, A. 2019. Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries. In *3rd Italian Conference on Cyber Security, ITASEC*. URL http://arxiv.org/abs/1901.03583.

Dzmitry Bahdana; Bahdanau, D.; Cho, K.; and Bengio, Y. 2015. Neural Machine Translation By Jointly Learning To Align and Translate. In *ICLR*.

Faridi, H.; Srinivasagopalan, S.; and Verma, R. 2019. Parameter Tuning and Confidence Limits of Malware Clustering. In *CODASPY*, 169–171.

Fleshman, W.; Raff, E.; Sylvester, J.; Forsyth, S.; and McLean, M. 2019. Non-Negative Networks Against Adversarial Attacks. *AAAI-2019 Workshop on Artificial Intelligence for Cyber Security* URL http://arxiv.org/abs/1806.06108.

Fleshman, W.; Raff, E.; Zak, R.; McLean, M.; and Nicholas, C. 2018. Static Malware Detection & Subterfuge: Quantifying the Robustness of Machine Learning and Current Anti-Virus. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, 1–10. IEEE.

Galinkin, E. 2019. What is the Shape of an Executable? In *Conference on Applied Machine Learning for Information Security*. URL https://www.camlis.org/2019/talks/galinkin.

Griewank, A.; and Walther, A. 2000. Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Trans. Math. Softw.* 26(1): 19–45.

Gruslys, A.; Munos, R.; Danihelka, I.; Lanctot, M.; and Graves, A. 2016. Memory-Efficient Backpropagation Through Time. In *NeurIPS*, 4125–4133. URL http://papers.nips.cc/paper/6221-memory-efficient-backpropagation-through-time.pdf.

Hasegawa, C.; and Iyatomi, H. 2018. One-dimensional convolutional neural networks for Android malware detection. In *2018 IEEE 14th International Colloquium on Signal Processing & Its Applications (CSPA)*, 99–102. IEEE.

Hayes, M.; Walenstein, A.; and Lakhotia, A. 2008. Evaluation of malware phylogeny modelling systems using automated variant generation. *Journal in Computer Virology* 5(4): 335–343.

Jeong, Y.-S.; Woo, J.; and Kang, A. R. 2019. Malware Detection on Byte Streams of PDF Files Using Convolutional Neural Networks. *Security and Communication Networks* .

Kantchelian, A.; Afroz, S.; Huang, L.; Islam, A. C.; Miller, B.; Tschantz, M. C.; Greenstadt, R.; Joseph, A. D.; and Tygar, J. D. 2013. Approaches to Adversarial Drift. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, 99–110. New York, NY, USA: ACM.

Kingma, D. P.; and Ba, J. L. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.

Kitaev, N.; Kaiser, L.; and Levskaya, A. 2020. Reformer: The Efficient Transformer. In *ICLR*.

Kolosnjaji, B.; Demontis, A.; Biggio, B.; Maiorca, D.; Giacinto, G.; Eckert, C.; and Roli, F. 2018. Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables. In *26th European Signal Processing Conference (EUSIPCO '18)*. URL https://arxiv.org/pdf/1803.04173.pdf.

Krčál, M.; Švec, O.; Bálek, M.; and Jašek, O. 2018. Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only. In *ICLR Workshop*.

Kreuk, F.; Barak, A.; Aviv-Reuven, S.; Baruch, M.; Pinkas, B.; and Keshet, J. 2018. Adversarial Examples on Discrete Sequences for Beating Whole-Binary Malware Detection. *arXiv preprint* URL http://arxiv.org/abs/1802.04528.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NeurIPS*, 1097–1105. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

Kumar, R.; Purohit, M.; Svitkina, Z.; Vee, E.; and Wang, J. 2019. Efficient Rematerialization for Deep Networks. In *NeurIPS*, 15172–15181. URL http://papers.nips.cc/paper/9653-efficient-rematerialization-for-deep-networks.pdf.

Kusumoto, M.; Inoue, T.; Watanabe, G.; Akiba, T.; and Koyama, M. 2019. A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation. In *NeurIPS*, 1163–1172. URL http://papers.nips.cc/paper/8400-a-graph-theoretic-framework-of-recomputation-algorithms-for-memory-efficient-backpropagation.pdf.

Li, M.; Chen, X.; Li, X.; Ma, B.; and Vitanyi, P. M. 2004. The Similarity Metric. *IEEE Transactions on Information Theory* 50(12): 3250–3264.

Liu, Y.; Wang, D.; He, F.; Wang, J.; Joshi, T.; and Xu, D. 2019. Phenotype Prediction and Genome-Wide Association Study Using Deep Convolutional Neural Network of Soybean. *Frontiers in Genetics* 10.

Loshchilov, I.; and Hutter, F. 2019. Decoupled Weight Decay Regularization. In *ICLR*. URL https://github.com/loshchil/AdamW-and-SGDW.

Menéndez, H. D.; Bhattacharya, S.; Clark, D.; and Barr, E. T. 2019. The arms race: Adversarial search defeats entropy used to detect malware. *Expert Systems with Applications* 118: 246–260.

Raff, E.; Aurelio, J.; and Nicholas, C. 2019. PyLZJD: An Easy to Use Tool for Machine Learning. In *Proceedings of the 18th Python in Science Conference*, 97–102.

Raff, E.; Barker, J.; Sylvester, J.; Brandon, R.; Catanzaro, B.; and Nicholas, C. 2018. Malware Detection by Eating a Whole EXE. In *AAAI Workshop on Artificial Intelligence for Cyber Security*. URL http://arxiv.org/abs/1710.09435.

Raff, E.; and Nicholas, C. 2017. An Alternative to NCD for Large Sequences, Lempel-Ziv Jaccard Distance. In *KDD*, 1007–1015.

Raff, E.; Nicholas, C.; and McLean, M. 2020. A New Burrows Wheeler Transform Markov Distance. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence*, 5444–5453.

Raff, E.; and Nicholas, C. K. 2018. Lempel-Ziv Jaccard Distance, an effective alternative to ssdeep and sdhash. *Digital Investigation* .

Rajab, M. A.; Ballard, L.; Jagpal, N.; Mavrommatis, P.; Nojiri, D.; Provos, N.; and Schmidt, L. 2011. Trends in CircumventingWeb-Malware Detection. Technical Report July, Google, URL https://security.googleblog.com/2011/08/four-years-of-web-malware.html.

Roberts, J.-M. 2011. Virus Share. URL https://virusshare.com/. Last accessed 2020-01-07.

S. Resende, J.; Martins, R.; and Antunes, L. 2019. A Survey on Using Kolmogorov Complexity in Cybersecurity. *Entropy* 21(12): 1196.

Spafford, E. C. 2014. Is Anti-virus Really Dead? *Computers & Security* 44: iv.

Stokes, J. W.; Agrawal, R.; and McDonald, G. 2018. Neural Classification of Malicious Scripts: A study with JavaScript and VBScript. *arXiv* URL http://arxiv.org/abs/1805.05603.

van den Oord, A.; Dieleman, S.; Zen, H.; Simonyan, K.; Vinyals, O.; Graves, A.; Kalchbrenner, N.; Senior, A.; and Kavukcuoglu, K. 2016. WaveNet: A Generative Model for Raw Audio URL http://arxiv.org/abs/1609.03499.

Voelker, A.; Kajić, I.; and Eliasmith, C. 2019. Legendre Memory Units: Continuous-Time Representation in Recurrent Neural Networks. In *NeurIPS*, 15544–15553.

Votipka, D.; Rabin, S. M.; Micinski, K.; Foster, J. S.; and Mazurek, M. M. 2019. An Observational Investigation of Reverse Engineers ' Processes. In *USENIX Security Symposium*.

Walenstein, A.; and Lakhotia, A. 2007. The Software Similarity Problem in Malware Analysis. *Duplication, Redundancy, and Similarity in Software* URL http://drops.dagstuhl.de/opus/volltexte/2007/964.

Wehner, S. 2007. Analyzing Worms and Network Traffic Using Compression. *Journal of Computer Security* 15(3): 303–320, URL http://dl.acm.org/citation.cfm?id=1370628.1370630.

Wu, M. C.; Kraft, P.; Epstein, M. P.; Taylor, D. M.; Chanock, S. J.; Hunter, D. J.; and Lin, X. 2010. Powerful SNP-set analysis for case-control genome-wide association studies. *American journal of human genetics* 86(6): 929–942.