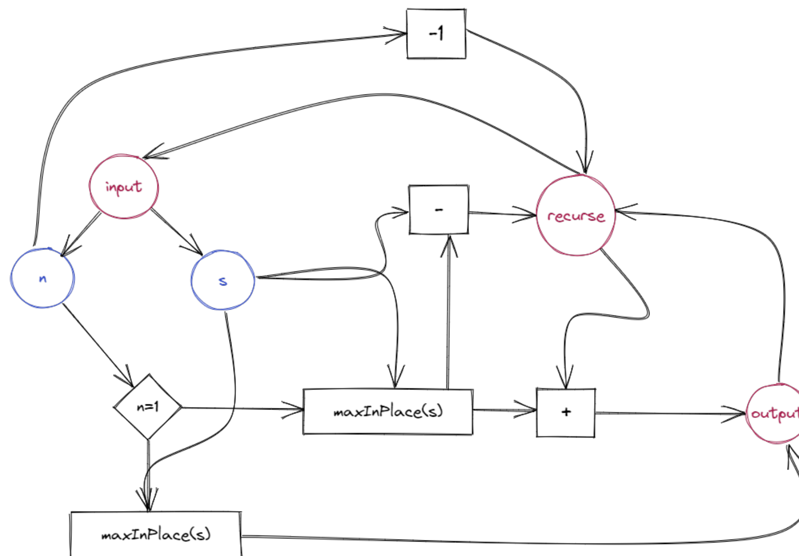
ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΗΣ ΕΛΛΑΔΟΣ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Avoiding attacks on machine learning models for malware detection



Του φοιτητή
Γεώργιου Μακρή
Αρ. Μητρώου: 164702

Επιβλέπων
Κωνσταντίνος Διαμαντάρας
Καθηγητής

22 Ιουνίου 2022

Τίτλος Δ.Ε. Evading Adversarial Attack by Design
20136
Γεώργιος Μακρής
Κωνσταντίνος Διαμαντάρας
31/03/2020
21/06/2022

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικώνς Συστημάτων του ΔΙ.ΠΑ.Ε.

Η παρούσα εργασία αποτελεί πνευματική ιδιοκτησία του φοιτητή Γεώργιου Μακρή που την εκπόνησε. Στο πλαίσιο της πολιτικής ανοικτής πρόσβασης, ο συγγραφέας/δημιουργός εκχωρεί στο Διεθνές Πανεπιστήμιο της Ελλάδος άδεια χρήσης του δικαιώματος αναπαραγωγής, δανεισμού, παρουσίασης στο κοινό και ψηφιακής διάχυσης της εργασίας διεθνώς, σε ηλεκτρονική μορφή και σε οποιοδήποτε μέσο, για διδακτικούς και ερευνητικούς σκοπούς, άνευ ανταλλάγματος. Η ανοικτή πρόσβαση στο πλήρες κείμενο της εργασίας, δεν σημαίνει καθ' οιονδήποτε τρόπο παραχώρηση δικαιωμάτων διανοητικής ιδιοκτησίας του συγγραφέα/δημιουργού, ούτε επιτρέπει την αναπαραγωγή, αναδημοσίευση, αντιγραφή, πώληση, εμπορική χρήση, διανομή, έκδοση, μεταφόρτωση (downloading), ανάρτηση (uploading), μετάφραση, τροποποίηση με οποιονδήποτε τρόπο, τμηματικά ή περιληπτικά της εργασίας, χωρίς τη ρητή προηγούμενη έγγραφη συναίνεση του συγγραφέα/δημιουργού.

Η έγκριση της διπλωματικής εργασίας από το Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του Διεθνούς Πανεπιστημίου της Ελλάδος, δεν υποδηλώνει απαραιτήτως και αποδοχή των απόψεων του συγγραφέα, εκ μέρους του Τμήματος.

«Στον Sango, και ας με ξυπνάει όταν μαλώνει με τις άλλες γάτες.»

Πρόλογος

Η συγκεκριμένη εργασία, θα έλεγε κανείς, πως ξεκίνησε σαν μια πρόκληση από τον κύριο Διαμαντάρα και την κυρία Πετροπούλου του Rutgers University. Η πρόκληση ήταν το κατά πόσο θα μπορούσε κάποιος να χρησιμοποιήσει μια μέθοδο, που οι ίδιοι ανέπτυξαν για την επίληση ενός προβλήματος στο πεδίο της επεξεργασίας σημάτων, ώστε να μπορέσει να κάνει το MalConv λιγότερο ευάλωτο σε adversarial attacks. Η εφαρμογή της μεθόδου τους πάνω στο MalConv τέθηκε σαν πρόταση από έναν διδακτορικό φοιτητή υπό την επίβλεψη της κυρίας Πετροπούλου, Yi Han, ο οποίος βοήθησε στα αρχικά στάδια της εργασίας τρέχοντας μερικά πειράματα για την επαλήθευση των αποτελέσμάτων του MalConv. Επίσης βοήθησε στο να επαληθευθεί η ορθότητα του κώδικα που είχα γράψει για την εκτέλεση των επιθέσεων στο MalConv. Το ότι θα είχα την ευκαιρία να κάνω κάτι διαφορετικό και πρωτότυπο ήταν οι κύριοι λόγοι που με ώθησαν στο να ασχοληθώ με το συγκεκριμένο θέμα ως την πτυχιακή μου εργασία.

Έχοντας φτάσει στο τέλος, μπορώ να πω ότι πλέον καταλαβαίνω το κομμάτι των νευρωνικών δικτύων σε ένα εντελώς διαφορετικό επίπεδο από ότι πριν. Απέκτησα γνώσεις και έχτισα απόψεις οι οποίες όχι μόνο θα με συνοδέψουν σε μελλοντικές ερευνητικές δράσεις, αλλά ανυπομονώ να τις εξελίξω.

Περίληψη

Η συγκεκριμένη εργασία εξετάζει την περίπτωση του MalConv, ενός νευρωνικού δικτύου που επιτυγχάνει αναγνώριση κακόβουλου λογισμικού μέσω μη-προεπεξεργασμένων εκτελέσιμων αρχείων, και προσπαθεί να το κάνει πιο ανθεκτικό σε αντίπαλες επιθέσεις (adversarial attacks). Αντίπαλες επιθέσεις έχουν καταφέρει να ρίξουν την επίδοση του μοντέλου σε τέτοιο βαθμό ώστε να καθίσταται μη-ασφαλές για οποιαδήποτε πρακτική χρήση. Προς αυτόν το σκοπό, προτίθεται μια νέα μέθοδος απώθησης τέτοιων επιθέσεων, βασισμένη σε επεξηγήσιμες αρχιτεκτονικές αλλαγές, η οποία εξετάζεται και αναλύεται.

Τμήμα αυτών των αρχιτεκτονικών αλλαγών είναι και η πρόταση ενός νέου, διαφορικού, αλγορίθμου, το Learn to Mask (L2M). Το L2M έχει τις ρίζες του στην "αυστηρή προσοχή" (hard attention) ενώ παραμένει εξαιρετικά ελφρύ σε σύγκριση με άλλες παραγωγίσημες μορφές των μεθόδων προσοχής. Το L2M εφαρμόστηκε με μεγάλη επιτυχία στον MalConv, δείχνοντας σημάδια ενός πολλά υποσχόμενου αλγορίθμου που ίσως να είναι σε θέση να συνεισφέρει και σε πολλά άλλα μοντέλα.

Τέλος, η χρήση αντιπάλων επιθέσεων αναλύεται ως ένα εργαλείο για την απόδειξη ορισμένων ιδιωτήτων του μοντέλου, όπως είναι η αντίσταση στον θόρυβο. Τα επιστημονικά εργαλεία που είναι διαθέσημα για την εξαγωγή συμπερασμάτων γύρω από τον βαθμό γενίκευσης ενός νευρωνικού δικτύου είναι ελάχιστα και η συγκεκριμένη εργασία υπενύσεται την ύπαρξη ενός νέου εργαλείου. Η πλήρης τυποποίηση αυτής της ιδεάς όμως παραμένει εκτός του εύρους αυτής της εργασίας.

Το μοντέλου που παράχθηκε μέσω αυτής της μεθόδου είναι εξαιρετικά πιο ανθεκτικό σε επιθέσεις, κατηγοριοποιώντας σωστά άνω του 90% των αντιπάλλων δειγμάτων. Επίσης το τελικό μοντέλο έχει χαμηλότερες απαιτήσεις μνήμης και αυξημένη ακρίβεια σε κανονικά δείγματα.

«Avoiding attacks on machine learning models for malware detection»

«Georgios Makris»

Abstract

This work considers the case of MalConv, a neural network that performs malware detection on raw binaries, and attempts to make it more robust against adversarial attacks. Adversarial attacks have significantly reduced the effectiveness of MalConv, making it insecure for any practical use. To mitigate this problem, a new method for thwarting such attacks, based on architectural changes and explainability, is introduced and analyzed.
Part of these architectural changes is the introduction of a novel, differentiable, algorithm, Learn To Mask (L2M). L2M has its roots in hard attention while remaining uncharacteristically lightweight when compared to other differentiable forms of attention that manage to accomplish the very same task. L2M is applied with much success to MalConv, showing signs of a promising algorithm that might be suited for a variety of other models.
Lastly, the use of adversarial attacks is examined as a tool to prove specific properties about the model such as "noise-resistance". The scientific tools available to draw conclusions about a neural network's degree of generalization are scarce and this work aims to hint at the direction of a new one. The complete formalization of this idea is far beyond the scope of this thesis project.
The model produced through this method is significantly more resilient to attacks, more than 90% of the adversarial samples are classified correctly. Moreover, the altered model exhibits a smaller memory footprint and increased accuracy on regular samples.

Ευχαριστίες

Η σκληρή δουλειά ήταν εξίσου απαραίτητη με την υποστήριξη που προσέφεραν τόσο η οικογένειά μου, οι καθηγητές μου, οι φίλου μου αλλά και η εταιρεία στην οποία δουλεύω.
Ανάμεσα σε αυτούς θα ήθελα να ξεχωρίσω τον κύριο Διαμαντάρα, επιβλέποντα της συγκεκιρμένης πτυχιακής εργασίας, για την αμέριστη συμπαράστασή του. Επίσης θα ήθελα να πω ένα μεγάλο ευχαριστώ στον Σταύρο Δωρόπουλο, της DataScouting, για την υπέρ του δέοντος κατανόηση που έχει δείξει σε σχέση με τον χρόνο που χρειάστηκα για συναντήσεις με τον επιβλέποντα, εν ώρα εργασίας.

Περιεχόμενα

Πρόλογος . . . iv
Περίληψη . . . v
Abstract . . . vi
Ευχαριστίες . . . vii
Περιεχόμενα . . . viii
Κατάλογος Σχημάτων . . . x
Κατάλογος Πινάκων . . . x

1 Introduction **1**
1.1 Related Work . . . 1
1.2 Contributions . . . 2
1.3 Structure . . . 3

2 Neural Networks **5**
2.1 Perceptron . . . 5
2.2 Multi-Layer Perceptron . . . 7
2.2.1 Compositionality . . . 7
2.2.2 Universal Approximator . . . 7
2.2.3 Dense layers . . . 8
2.2.4 Partially Connected Layers . . . 8
2.3 Convolutional Neural Networks . . . 9
2.3.1 Max Pooling . . . 10
2.4 Embedding Layer . . . 11
2.5 Activation Functions . . . 11
2.5.1 Rectified Linear Unit . . . 11
2.5.2 Sigmoid . . . 12
2.5.3 Gaussian Error Linear Unit . . . 12
2.5.4 Softmax . . . 13
2.6 Attention . . . 13
2.6.1 Soft Attention . . . 13
2.6.2 Hard Attention . . . 15
2.7 Back Propagation . . . 16
2.7.1 Gradient Descent . . . 17
2.8 Differential Dynamic Programming . . . 17

3 Adversarial Attacks **19**
3.1 Input Space or Attack Vector? . . . 19
3.2 Interpreting Adversarial Attacks . . . 20
3.3 Adversarial Attacks as Black Box Attacks . . . 20
3.3.1 Transfer Attacks . . . 20
3.3.2 Gradient Estimation Attacks . . . 21
3.4 Defense Strategies . . . 21
3.4.1 Preventive measures . . . 21
3.4.2 Adversarial Training . . . 23
3.4.3 Architectural Defences . . . 23

4 Learn to Mask **26**
4.1 Inspiration . . . 26
4.2 Definition . . . 26
4.3 Learn To Mask as a form of Attention . . . 28
4.4 Learn To Mask as a form of Dropout . . . 29
4.5 Hypotheses . . . 30

5 Malware Detection **31**
5.1 Traditional Malware Detection Methods . . . 31
5.1.1 Static analysis . . . 31
5.1.2 Dynamic & Hybrid Analysis . . . 32
5.2 Deep Learning as a static analysis tool . . . 33

6 MalConv **35**
6.1 Why MalConv? . . . 35
6.2 Architecture . . . 36
6.3 Attacking MalConv . . . 38
6.3.1 Attack Success Rate . . . 39
6.3.2 Random Byte Padding Attacks . . . 39
6.3.3 Fast Gradient Sign Method Attacks . . . 40
6.4 Redesigning MalConv . . . 41

7 Experiments & Results **44**
7.1 Experiments . . . 44
7.1.1 Dataset . . . 45
7.2 Results . . . 46
7.2.1 Testing . . . 47
7.2.2 Attacking . . . 47
7.3 Variations of Redesigned MalConv . . . 49
7.3.1 Variation: Gated MLP neurons . . . 49
7.4 Variation: Number of selections . . . 50
7.5 Variation: Group size . . . 51
7.5.1 Overview . . . 52

8 Conclusions **53**
8.1 Challenges . . . 53
8.2 Future work . . . 53
8.2.1 Unexplored areas of Learn to Mask . . . 54
8.2.2 Unexplored areas of Architectural Defense strategies . . . 54
8.2.3 Malware Detection as a Service . . . 55

bibliography **56**

Κατάλογος Σχημάτων

2.1 Visualization of a perceptron with 3 inputs. 5
2.2 Visualization of a neural network. There are 2 dense layers (input and hidden) along with a perceptron used for output. 7
2.3 Visualization of a neural network. There are 2 partially connected layers (input and hidden) along with a perceptron used for output. 9
2.4 Visualization of a convolutional neural network. The input is a 4x4 image and the convolution layer has a 3x3 kernel with a single filter to produce 1 output. 10
2.5 Visualization of max pooling using a 2x2 kernel and 2x2 strides. 11
2.6 Visualization of a single “head” of multi-head attention. “Concat” is the point where all “heads” converge. 14
3.1 Visualization of a gradient-based attack as a state-machine. 19
4.1 Visualization of Learn To Mask as a finite state machine. 29
5.1 Diagram of the contributions by neural networks to malware detection. 33
6.1 Visualization of MalConv’s architecture. 37
6.2 Visualization of MalConv’s redesigned architecture. 42

Κατάλογος Πινάκων

7.1 Overview of the original & the redesigned MalConv. 45
7.2 Redesigned MalConv’s hyper-parameters. 46
7.3 Train & Validation set results table, higher is better. 47
7.4 Attack Success Rate results table, lower is better. 48
7.5 Gated MLP neurons variation: Parameters of different variations of the redesigned model. 49
7.6 Gated MLP neurons variation: Train & Validation set results, higher is better. 50
7.7 Gated MLP neurons variation: Attack Success Rate results table, lower is better. 50
7.8 Number of selections variation: Parameters of different variations of the redesigned model. 50
7.9 Number of selections variation: Train & Validation set results table, higher is better. 51
7.10 Number of selections variation: Attack Success Rate results table, lower is better. 51
7.11 Group size variation: Parameters of different variations of the redesigned model. 51
7.12 Group size variation: Train & Validation set results table, higher is better. 52
7.13 Group size variation: Attack Success Rate results table, lower is better. 52

Chapter 1: Introduction

In recent years, there has been an increasing number of solutions utilizing deep learning, neural networks in particular, for critical tasks, where slight inaccuracies can lead to disaster. Autonomous driving [13], bio-metric authentication [67], medical diagnosis [86], and malware detection[76] are such critical tasks. Modern solutions to these problems have become increasingly reliant to the effectiveness of neural networks. Though they are effective, they provide little to no explanation as to what do they owe their success. One can test their effectiveness, given previously unseen by the model data points, but that is a mere fraction of the input space and often not indicative of the model's performance in the real world. This is especially true when taking into account malicious attempts that aim to exploit vulnerabilities inherent in neural networks, labeled adversarial attacks.

Adversarial attacks [36] are methods that camouflage the original label of some valid input x to produce some new input $\hat{x}$ in such a way that the labels produced for each of these inputs differ drastically. In the context of malware detection, which will be the main topic of this work, one can easily imagine the implications where x is a malware and is classified as such by the model while $\hat{x}$ is the same malware, with the same malicious functionality, though now the model cannot detect it as such. The immensity of this issue is only magnified by the fact that current methods for shielding models against such attacks suffer from multiple issues that cause them to be expensive and unreliable.

In this work, the case of MalConv [76] is examined as a malware detection model that has been shown to be very vulnerable, with the goal of shielding it from attacks. To that extend, two novel ideas are examined, implemented, and then compared with the original model to prove their effectiveness.

The first idea presented has to do with the approach towards improving adversarial robustness. Current approaches either focus on training against them or finding ways to prevent adversarial samples from being examined in the first place. The method introduced here shows that vulnerabilities to adversarial attacks often stem from bad architectural decisions. Therefore, it should be possible to increase a model's adversarial robustness simply by redesigning key sections of it.

Secondly, to aid with the previous goal mentioned, a novel algorithm is introduced, Learn To Mask, which is designed with adversarial attack prevention in mind. After exhaustively analyzing Learn To Mask, not only does it deliver in that regard but it also provides multiple other benefits that can make it the right tool for many other models. This algorithm also manages to stand out from others that produce similar results through being much lighter, differentiable, and easy to apply.

Lastly, a couple of new tools are introduced to help observe the difference in performance before and after the changes were applied to MalConv.

1.1 Related Work

Since the discovery of MalConv's vulnerability, many have rushed to design new attacks and execute them. Far less people though have rushed to its aid, in order to provide some form of defense. Attempts were indeed made but they are few and the results leave much to be desired.

In [33] some of MalConv's authors have attempted to defend their model. Their solution was to restrict MalConv's weights to be only positive. This method did manage to provide significant adversarial attack evasion capabilities but at the cost of accuracy. Their model dropped an entire 4% in accuracy. It is also mentioned that even though their results show an attack evasion rate of 95% it is emphasized that there are still ways of producing malicious payloads, through the same attacks, that would be much harder to defend against.

The authors of [20] have also done some research in providing adversarial defenses through, what they call, randomised chaining. Their approach is to essentially create a committee of models which focus on different characteristics of the binary to make their decision. The higher the number of randomised detectors in use, the harder it is to produce an adversarial sample capable of fooling them all. Unfortunately, this method comes at a great cost with regards to the hardware required in order to operate it. Also, such an approach, while it is academically viable, will not scale with respect to the model's input space. Lastly, This method is only viable when used against attacks that produce samples with a low transferability rate, attacks that do produce more transferable adversarial samples should end up fooling the majority of the detectors included in the committee, especially since they share the same architecture.

A slightly different approach was presented in [118]. Their approach revolves around adversarial training. A Generative Adversarial Network (GAN) is used to generate potent adversarial samples that then become part of the training process. The resulting model has a reduced accuracy for legitimate samples of 90% and an attack evasion rate of 90.5% against common attacks, such as the fast gradient sign method. Such a method only tackles attacks in which the malicious payload is appended in the binary and would require multiple GANs to be trained, one for each type of attack in order to defend against new attacks. The usage of this method implies that the method used to attack the model is known to the defender which will most likely not be the case for real-world applications. There are also concerns about how the model's baseline accuracy would change, if one were to extend this method to cover multiple attacks, since it has already dropped considerably.

This is the extend of different attempts made to shield MalConv. Two out of them end up impacting the model's accuracy negatively and the fact that this sacrifice is acceptable in the face of adversarial robustness says a lot about how dire the situation is. Also, two thirds of these approaches cannot be considered commercially viable due to the huge maintenance costs they would incur. All these signs point at a field that is desperate to find some sort of reliable defense against adversarial attacks. This is true not only for the domain of malware detection, since the methods described in this section are pretty standard for most critical applications of neural networks.

1.2 Contributions

This work is a direct attempt at improving the methods through which defenses against adversarial attacks are provided. The goal is the introduction of a method that can help increase adversarial robustness while preserving asymmetric scaling, with regards to hardware requirements, towards the number of attacks. Such a method should, also, not compromise on the model's accuracy for legitimate samples. Along with this new approach, the right tools and methodologies are designed and presented to help achieve this goal.

A novel algorithm is introduced, Learn to Mask, as a way of selecting a tiny amount of important information to be used by the malware detection classifier. This allows the model to effectively discard any malicious payloads that are inserted into the model. Learn to Mask is a unique form of attention which is shown to not only aid in evading adversarial attacks but to also be directly responsible for raising the overall accuracy of the model. This work also shows how this method leads to more explainable architecture as well as the link between explainability and vulnerability to adversarial attacks.

A new defense strategy against adversarial attacks is also presented and explored. This strategy has its basis on the fact that a model's behavior stems off its architecture. This implies that vulnerability to specific attacks are directly linked to different part of the model's architecture. Therefore, changing a model's layout can be sufficient to considerably increase the model's defensive capabilities.

Lastly, it was noticed that part of the work mentioned in the Related Work section is only examined against a single attack. Changes in a model's training or input data in such a way as to cover the attack vector exploited by a single attack could just as well create a new attack vector. This work shows how one can be sure that no new vulnerabilities are created for known attacks through changes to defend against a new one. It also provides the basis of what can be considered as successful adversarial training, which does not agree with sacrificing accuracy for adversarial robustness.

To summarize, for ease of reference later on, this work's contributions are:

- a novel, attention based algorithm, Learn to Mask, which aids in both adversarial robustness and the overall model's performance through selective information filtering.
- a new defense strategy which does not increase the cost of training and operating the model produced through this method.
- processes through which one can keep track of vulnerability status throughout the model's architectural evolution.

1.3 Structure

This section presents an overview of the chapters and the topics discussed in each one of them.

Chapter 2: Neural Networks acts a short guide to the inner workings of neural networks. It starts with basic concepts, such as the perceptron, and builds on top of them in order to introduce more complicated layers which are the basis of modern neural networks.

Chapter 3: Adversarial Attacks explains how all neural networks are vulnerable to adversarial attacks. This relies on a lot of the concepts that were explained in the "Neural Networks" chapter. Some insight is also given with regards to how adversarial attacks are designed and how they operate. Here are also discussed the current defense strategies and their pitfalls along with the defense strategy proposed.

Chapter 4: Learn to Mask introduces Learn to Mask. A large part of this chapter focuses on drawing parallels between Learn to Mask and other algorithms that share common characteristics with it. Finally, some hypotheses are created about Learn to Mask, which will be examined at a later chapter.

Chapter 5: Malware Detection gives some background about malware detection as a field and how the methods used have evolved from primitive to utilizing neural networks. The criticality of the field is emphasized by examining its large history of measures and counter-measures implemented by attackers and defenders respectively. Also, comparisons are made between MalConv and other deep learning approaches for the sake of completeness.

Chapter 6: MalConv provides an in-depth explanation of the architecture and further expands on the reasons behind its selection for this case study. Its architecture is thoroughly examined and weak points are emphasized in order to tackle them in the redesigned version of the model which is demonstrated at the end of this chapter. Lastly, two popular attacks against MalConv are explained and analyzed. Those two attacks will later become the baseline against which the redesigned model will be tested.

Chapter 7: Experiments & Results is where the experiment parameters and dataset are explained. The second section of this chapter exhibits the results produced by training, testing, and attacking both the original model and the redesigned one. Hypotheses made in previous chapters are recalled in order to be validated or not.

Chapter 8: Conclusions summarizes the conclusions produced in the previous chapter. Here are also presented the challenges faced throughout this work and research avenues left unexplored along with promises of pursuing them.

Chapter 2: Neural Networks

Neural networks are at the heart of the subject of our work. Therefore, a good understanding of their internals and implementation is required, in order to see the reasoning behind the decisions and assumptions made later on. This chapter starts with the simplest form of a neuron and slowly builds from there. Knowledge of advanced calculus concepts is assumed, as covering them would be out of the scope of this work.

They fall under the broader label of Deep Learning [35], a subset of Machine Learning [46]. Neural networks are best described as a set of composable algorithms [55]. The way data flows through them is thought of as a directed acyclic graph, each node offering a layer of abstraction over the previous one.

2.1 Perceptron

Neural Networks are named after their resemblance to human neurons and synapses. A single neuron, in both cases, is defined by its input, weights or synapse strength, and activation. The strength of the activation signal is proportional to the strength of the inputs in conjunction with their respective synapse strength.

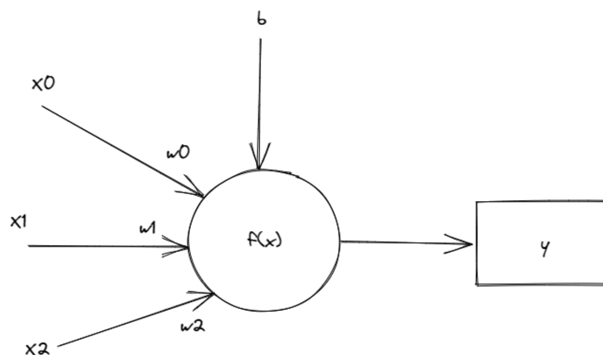


Figure 2.1: Visualization of a perceptron with 3 inputs.

The core idea is that, just like human neurons adjust their synapse strength to facilitate learning, so too artificial neurons have to make adjustments to their initial weights with respect to the distance between the output $\hat{y}$ and the desired output (target) y.

Typically, a single neuron is implemented as a 1-dimensional horizontal vector of its weights **w** and a separate value for bias b:

$$\mathbf{w} = \begin{bmatrix} w_1 w_2 \ldots w_n \end{bmatrix} \tag{2.1}$$

where n the number of input values to the neuron. Similarly inputs are represented by a 1-dimensional column vector **x**, with each value representing features. The output of the neuron $\hat{y}$ (prediction) is a linear transformation, calculated through matrix multiplication between the weights **w** and input plus the bias value b:

$$\hat{y} = \mathbf{w}\mathbf{x} + b \tag{2.2}$$

Having just defined a Perceptron [79], it is now time to understand how they are trained. Training is an iterative process, the termination of which is determined by a threshold, with regards to the error, or a maximum number of steps, commonly called epochs. As a form of supervised learning, perceptron requires multiple inputs along with their correct output, in order to learn how to predict unseen input. The weights and bias are updated after every input with the following rules, respectively:

$$\mathbf{w}_j(t+1) := \mathbf{w}_{i,j}(t) + \gamma|\mathbf{y}_i - \hat{\mathbf{y}}_i|\mathbf{x}_{i,j} \tag{2.3}$$

$$b(t+1) := b(t) + \gamma|\mathbf{y}_i - \hat{\mathbf{y}}_i| \tag{2.4}$$

where i the index of the current input **x** being trained on and j the current feature. γ represents the learning rate, a value in $[0, 1]$ which is used to control the granularity of the changes after each step. Higher learning rates can lead to over-correction of the weights resulting in slow, or none at all, convergence to a solution. On the other hand, corrections that are too small also cause training to take longer than it should. Learning rate is a hyper-parameter and is selected empirically.

For the sake of comparison with elements that will be later introduced, it is important to extract 2 key components from the weight/bias update rules. The first one is the error function. The error function is responsible for calculating the distance between current and desired outputs. Above, the absolute error function can be seen in the form of:

$$AbsoluteError(y, \hat{y}) = |y - \hat{y}| \tag{2.5}$$

The second part is the optimizer, responsible for how the weights get updated:

$$\delta\mathbf{w}_j := \mathbf{w}_j + \gamma AbsoluteError(\mathbf{y}_i, \hat{\mathbf{y}}_i)\mathbf{x}_{i,j} \tag{2.6}$$

It should be noted that learning rate γ is part of the optimizer and that the optimizer does require the error in order to calculate the updates.

While the perceptron is only able to solve problems that are linearly separable, it remains the building block for more complex arrangements such as multi-layer perceptrons (MLP), convolutional neural networks (CNN) and a lot more.

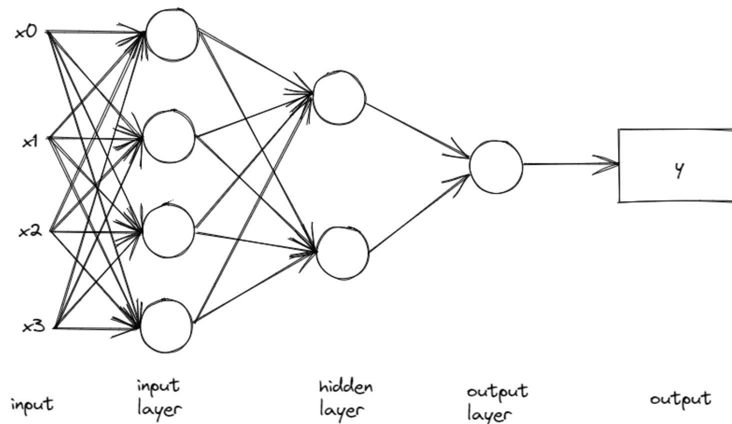


Figure 2.2: Visualization of a neural network. There are 2 dense layers (input and hidden) along with a perceptron used for output.

2.2 Multi-Layer Perceptron

The Multi-layer perceptron expands on the idea of a single perceptron and is considered the cornerstone of neural networks and deep learning in general. This is due to 2 main reasons: The first one is that MLPs are composable and the second is that stacking 2 layers of MLPs has been proven sufficient to approximate any contiuous function, through the universal approximation theorem [89, 48, 74].

2.2.1 Compositionality

Layers satisfy all the criteria needed to be considered pure mathematical functions and are treated as such both in theory and implementation. As a result, through composition, a model comprised of multiple layers can be expressed as a single function.

Given the MLP layers f:$\mathbb{R}^k \rightarrow \mathbb{R}^l$ and g:$\mathbb{R}^l \rightarrow \mathbb{R}^n$, appropriately constructed, a model m:$\mathbb{R}^k \rightarrow \mathbb{R}^n$ that feeds input $\mathbf{X}$ to f, f to g, and g produces $\hat{\mathbf{Y}}$ can be described by:

$$m = f \cdot g \tag{2.7}$$

This bridges the terms "layer" and "model", their only difference being context. Layers could represent the entire model and models can be used as layers in other models, for the sake of reusability and organization.

2.2.2 Universal Approximator

The universal approximation theorems test the capability of multi-layer perceptron models to approximate any "well-behaved" function, given appropriate weights. This matter has been studied for both bounded depth of layers with unbounded neurons [74] and unbounded depth with bounded neurons [48]. Similar theorems exist to prove the same case but for different activation functions or layers such as CNNs, Transformers and even graph neural networks [109, 10].

Such theorems serve as guarantees that a properly formulated problem, that satisfies the constraints of the previously mentioned theorems, can be solved by neural networks without the need to consider their viability on a theoretical level.

2.2.3 Dense layers

One of the most commonly seen layers made out of perceptrons is the Dense formation, meaning that all of its inputs are connected to every neuron in that layer. Part of its early adoption was its ease of implementation. Weights are represented as a matrix $\mathbf{W} \in \mathbb{R}^{(m,n)}$, each row corresponding to a single perceptron. Similarly, the bias value is now represented by a bias column vector $\mathbf{b} \in \mathbb{R}^{m}$.

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} \tag{2.8}$$

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{2.9}$$

The feed-forward calculation of a dense layer is a generalization of the one used by the perceptron, essentially a matrix multiplication between the weights **W** and the input vector **x** plus the bias vector **b**. The number of outputs from a dense layer is equivalent to the number of its neurons. The training process, which will be the same for layers explained in later sections, will be explained in the end of this chapter.

$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{2.10}$$

2.2.4 Partially Connected Layers

While dense layers have been proven to work, there is often an excess of weights created due to the fact that every input is connected to all the neurons. It is not uncommon, during training, that a lot of the weights will converge near 0. With modern models being exceedingly large, having a large amount of operations that do not contribute to the model's efficiency can be problematic. A common solution to that is pruning, a technique that will remove connections or even entire neurons that, after training, do not make significant contributions. In cases where the input data can be related to each other, through their position or other means, it is more efficient to force closely correlated data to the same neuron, restricting connections as an architectural choice, essentially creating partially connected layers.

Partially connected layers have not seen as wide adoption as dense layers, due to their circumstantial nature. Their use is also inhibited by the fact that, for computational efficiency reasons, all neurons

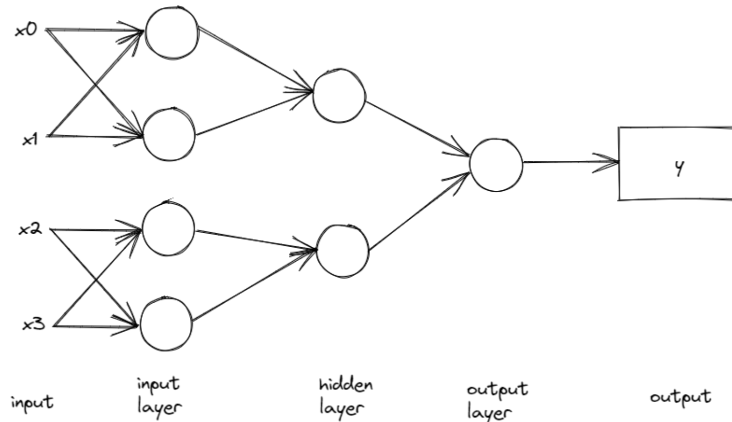


Figure 2.3: Visualization of a neural network. There are 2 partially connected layers (input and hidden) along with a perceptron used for output.

present in a partially connected layer, should have the same number of inputs. While technically possible for that to vary, the result of the feed-forward operation would be a sparse matrix, making it rather hard to use in conjunction with other standard layers. To succinctly represent their operations the Einstein notation [92] will be used.

Unlike the dense layer, a partially connected layer, expects its input to be 2-dimensional. Given an input vector $\mathbf{x} \in \mathbb{R}^n$, in order for it to be acceptable input to a partially connected layer, it would have to be transformed to a matrix $\mathbf{X} \in \mathbb{R}^{(m,k)}, m \times k = n$. This implies that $\mathbf{X}$ represents m groups of k values. The weights $\mathbf{W} \in \mathbb{R}^{(m,k)}$ and biases $\mathbf{b} \in \mathbb{R}^m$ for partially applied layers preserve the same structure as dense layers though the feed-forward calculation is described by:

$$\hat{\mathbf{y}} = (\mathbf{W}, \mathbf{X}, \mathbf{b})_k \tag{2.11}$$

In short, the feed-forward operation is the dot product between groups and neurons plus the bias. This configuration implies no overlap in the values being fed to each neuron. Altering the constraint $m \times k = n$ to $m \times k > n$ will allow for configurations that support such overlap, the feed-forward operation is not affected.

2.3 Convolutional Neural Networks

Convolutional Neural Networks[116] (CNNs), are a case of a model, commonly used as a single layer. They have been popularized after many successful applications in areas such as signal processing[97, 76] and computer vision[51]. They operate under the assumption that input values in close proximity to each other, such as pixel values in an image, have high correlation.

Convolution is a well defined as a mathematical operation and has its origins in signal processing. Given functions $f : \mathbb{R} \to \mathbb{R}, g : \mathbb{R} \to \mathbb{R}$ convolution produces a function $(f * g)(t)$ that expresses the amount of overlap of g as it is shifted over f. Application of the convolution operator $*$ is done as shown below:

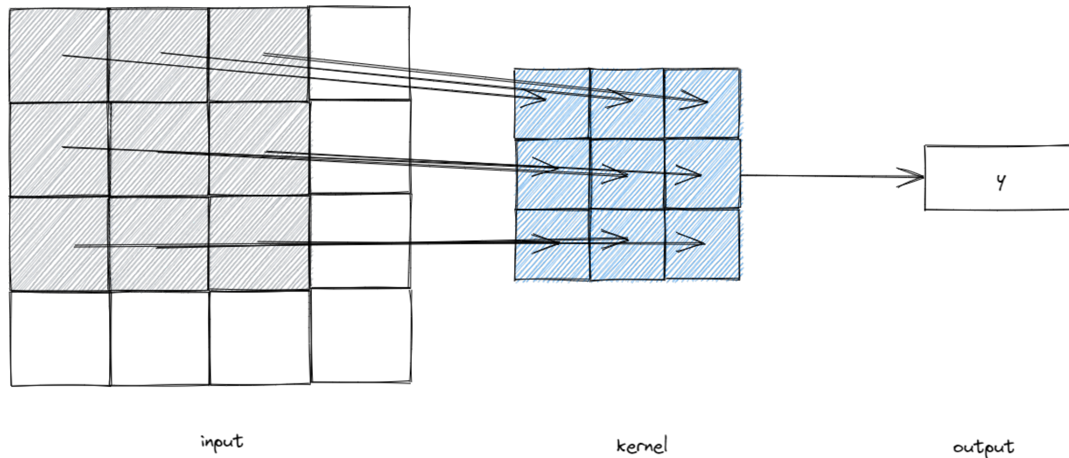


Figure 2.4: Visualization of a convolutional neural network. The input is a 4x4 image and the convolution layer has a 3x3 kernel with a single filter to produce 1 output.

$$f * g \equiv \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.12}$$

The same logic is applied to CNNs but the first function would represent the input and the second the weights. The idea is that, by optimizing the weights, the resulting function would be capable of describing features that will help following layers to make useful, to the current task, predictions.

1-dimensional CNNs are described by a function c:

$$c : \mathbb{R}^{(features,n)} \rightarrow \mathbb{R}^{(steps,filters)} \tag{2.13}$$

where features, also known as channels, imply the number of different features present in each input. Steps are the total number of convolution windows calculated, while filters define the total number of neurons utilized in the convolution process. Input is defined as a matrix $\mathbf{X} \in \mathbb{R}^{(channels,n)}$ while weights $\mathbf{W} \in \mathbb{R}^{(kernel,filters)}$, where kernel stands for the size of the convolution window. The existence of bias is present in CNNs as well, in the form of a vector $\mathbf{b} \in \mathbb{R}^{filters}$. One additional hyper-parameter is required in order to allow for easy tweaking of the distance between convolution windows, named stride, which directly affects the number of steps produced as the output.

2.3.1 Max Pooling

More often than not, the use of a CNN ends up increasing the dimensionality of the data provided. While that does indeed mean more information for the network to make predictions from, it also greatly increases the model's needs in processing power and memory. For that reason, layers such as global max pooling have been introduced. Max pooling is a non-trainable layer, it holds no weights of its own, and reduces dimensionality of data by compressing areas of predetermined sizes into their max value. Max pooling utilizes a window sliding technique to traverse the data, which has been divided into p groups and extracts the maximum values of each group. The output would be a tensor with p values.

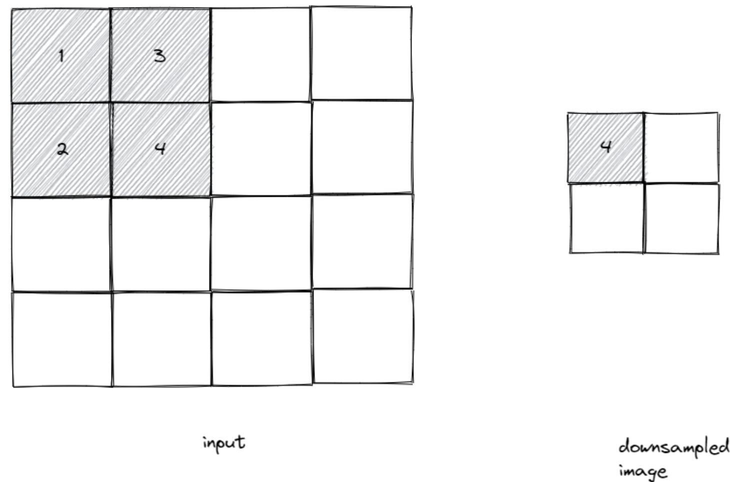


Figure 2.5: Visualization of max pooling using a 2x2 kernel and 2x2 strides.

2.4 Embedding Layer

Embedding layers are tasked with projecting single values into an n-dimensional space. The projection of the input values starts out random, sampled from a uniform distribution, and converges along with the model's training.

The mapping from values to projections is performed by a look-up table utilizing a matrix structure. Embedding layers can only be used as the first layer of each network and the range of the input space has to be finite and countable. The values mapped through the embedding layer are its *vocabulary* and each of them is assigned an index, each corresponding to a specific row in the look-up table, used for retrieval.

This layer is closely associated with natural language processing tasks, often used to embed words. By the end of training, words that are semantically close would also be close in the embedding space. While words do not apply to malware detection, it is mentioned as it is used by MalConv [76] to embed the input bytes.

2.5 Activation Functions

Activation functions are stacked on top of trainable layers such as the MLP or CNN layers as a transformation of their output. Activation functions have to be differentiable as their derivative directly affects the magnitude of the error distributed to each weight, monotonically increasing, and defined for all real numbers. They also play a big role in breaking up the linearity of the model and bounding the output space. Different transformations of the output space in each layer can cause drastic differences the model's performance. Below, a subset of those is briefly explained as they will be useful later on.

2.5.1 Rectified Linear Unit

More commonly referred to as ReLU, the Rectified Linear Unit is the go-to choice for any intermediate layer. The reasons for its wide adoption is that it is simple to compute both its output and gradient while preserving the magnitude of the error during back propagation. It is bounded in $[0, \infty]$ and the purpose

of it is to prohibit any negative values from moving on with the justification that they might inhibit the training of the model. Below are given the function and its derivative, as will be done for all subsequent activation functions.

$$ReLU(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{2.14}$$

$$ReLU'(x) = \frac{d\,ReLU(x)}{dx} = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{2.15}$$

2.5.2 Sigmoid

The sigmoid, or logistic, is bounded in $[0, 1]$ and is well suited for the last layer of a network tackling a binary classification task, with 0 denoting the first class and 1 the second. While its initial calculation is more complicated than ReLU's, it is important to note how that computation can be reused for the calculation of its derivative.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.16}$$

$$\sigma'(x) = \frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x)) \tag{2.17}$$

2.5.3 Gaussian Error Linear Unit

Abbreviated to GeLU [39] it is a more modern activation function than the previous ones and takes a stochastic approach. Where it differs from the other previous two functions is that it tries to scale its output in accordance with its difference from the rest of the input but using probabilistic methods. This is done by scaling the input x by the probability of any other input **X** is either lass than or equal to it, $P(\mathbf{X} \leq x)$.

$$P(X) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \tag{2.18}$$

$$GeLU(x) = xP(\mathbf{X} \leq x) = x\frac{1}{2}[1 + erf(x/\sqrt{2})] \tag{2.19}$$

$$GeLU'(x) = \frac{d\,GeLU(x)}{dx} = \frac{1 + erf(\frac{x}{\sqrt{2}})}{2} + xP(\mathbf{X} = x) \tag{2.20}$$

2.5.4 Softmax

Softmax is primarily used to fit the input values into a probability distribution. It is described as $S : \mathbb{R}^k \to (0,1)^k$ and its output always sums up to 1. Most commonly, it is used at the last layer of a neural network to normalize the output values when those are more than one. Besides its use in multi-class classification problems and reinforcement learning it has also seen wide adoption with the rise of attention, to be discussed shortly.

$$S(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_i}} \tag{2.21}$$

where $i \in \{1, 2, \ldots, k\}$, and its derivative:

$$\frac{\partial S(x_i)}{\partial x_j} = \begin{cases} S(x_i)(1 - S(x_i)), & \text{if } i = j \\ -S(x_i)S(x_j), & \text{if } i \neq j \end{cases} \tag{2.22}$$

2.6 Attention

Attention [60] is one of the newest additions to the neural network toolkit. There has been a lot of excitement around its performance in tasks that are notoriously difficult, such as natural language processing and computer vision. Even though attention takes many different forms, it has lately become synonymous with transformers [98] due to their heavy use of it and wide adoption.

Attention mechanisms have also provided significant improvements in malware detection or classification models.[99, 3, 14] This is especially true when the classification is based on features extracted from binaries, such as System API calls and file signatures. Another way Attention has proven useful in this domain was by aiding in feature extraction and attribution [111, 18, 61, 112] through attention maps.

There are two different categories of attention, soft attention and hard attention [63]. While soft attention is more popular, and encompasses transformers, hard attention is generally considered to provide better results [34]. Each of them will be explained here but more emphasis will be placed on hard attention as it has been explored much less.

2.6.1 Soft Attention

Soft attention introduces concepts such as Queries $\mathbf{Q} \in \mathbb{R}^k$, Keys $\mathbf{K} \in \mathbb{R}^k$ and values $\mathbf{V} \in \mathbb{R}^v$ in order to produce the relative importance of the input values. The way attention works is by using the queries and keys to produce, on-the-fly, weights, called soft-weights that can help calculate the importance of the values.

In general, soft attention can be considered a sort of memory module given that they manage to compress their input into a key-value store consisting of merely features.

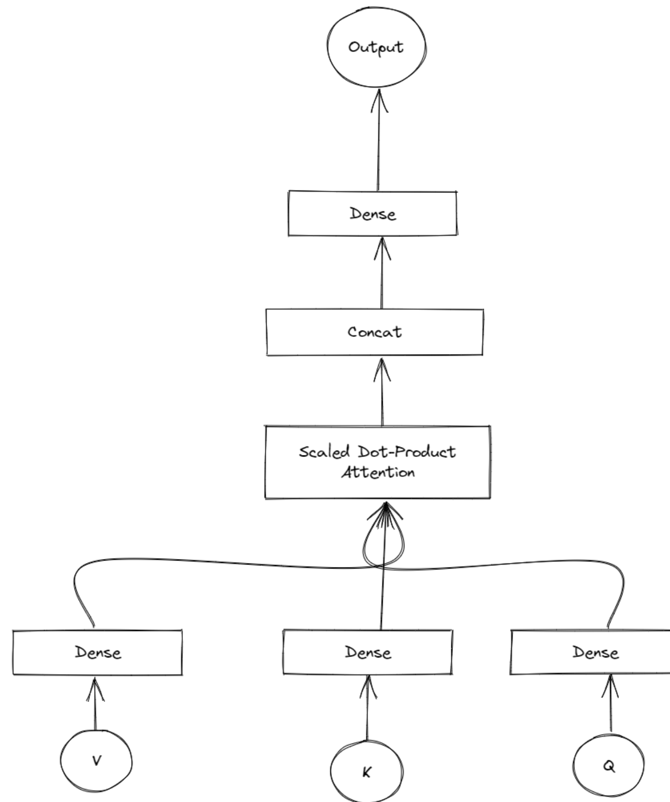


Figure 2.6: Visualization of a single "head" of multi-head attention. "Concat" is the point where all "heads" converge.

$$SoftWeights = S(\mathbf{QK^T}) \tag{2.23}$$

where S is the softmax function described earlier. The soft-weights are then multiplied with the values to produce the output of the attention layer:

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = SoftWeights(\mathbf{Q}, \mathbf{K})\mathbf{V} \tag{2.24}$$

These equations define dot-product attention. Dot-product attention can be thought of as a look-up operation, where the query is mapped to a key-value pair.

More frequently used is the scaled dot-product attention [98] which differs from dot-product attention only in the calculation of the soft-weights:

$$SoftWeights = S(\frac{\mathbf{QK^T}}{k}) \tag{2.25}$$

where k is the dimensionality of the queries and keys.

Scaled dot-product attention relies on the fact that keys **K** and queries **Q** to be first passed through separate

dense layers. The evolution of this concept can be seen in the form of multi-head attention [98].

There are other forms of soft attention, such as additive attention, they all have their own characteristics and use cases though they will not be analyzed here as soft attention is not directly related to the work presented in later chapters. The same cannot be said about hard attention which is much more relevant. Multi-head attention also introduces multiple parallel branches of scaled dot-product attention which utilize different dense layers to produce the input required by the scaled-dot product attention.

2.6.2 Hard Attention

Having seen how soft attention essentially multiplies values with their importance to produce valuable features, it should be easy to imagine what hard attention tries to accomplish. Hard attention takes a more strict, binary approach towards the importance of values, essentially marking them as important or not important, unlike soft attention that produces an entire range of values with regards to importance.

Despite the fact that, as mentioned earlier, that hard attention is considered to provide better results than soft attention it has not seen the same range of adoption. This fact can be easily attributed to the fact that current implementations of hard attention are cumbersome.

The use of hard attention does come with a few challenges that stem off its implementation. Hard attention appears in a couple of forms, all of which introduce a new problem that has to be dealt with. The ideal form of hard attention is one that is:

- easy to train,
- differentiable,
- and only alters non-important values.

Common implementations of hard attention work either in approximation, essentially producing masks that are as close as possible to binary masks, or with the help of reinforcement learning, where a reinforcement learning agent is trained to produce binary masks which are then multiplied with the input values to produce the output of hard attention.

In work such as [54] it is claimed by the authors that they have produced a form of differentiable hard attention. Upon closer examination, though, this is not accurate. Their method relies on a sigmoid function with a really high slope value. What this does is produce vectors that are as close to a binary mask as possible. While technically correct, this does not satisfy the theoretical definition for hard attention which required that values are either selected or discarded. This method is not "wrong" because it simply does not satisfy the definition, it is wrong because it produces wrong gradients. Were one to produce a function that fully complies with the definition, then it would produce a gradient of 1 for selected values and 0 for those discarded. The sigmoid function was examined earlier in this chapter and its gradients are far from what would be expected.

Another case for hard attention is presented in [4] where the authors use a neural network to estimate the best selections. The resulting method is not end-to-end differentiable. This has very direct implications in

the training process as now there is a need to first train the "hard attention" model first, before being able to train the model that actually performs the required task. Those 2 models cannot be trained simultaneously because the selection network's output is processed through non-differentiable means. Since the gradients have no way of flowing from one model to the other then 2 separate training phases must be used.

Similarly to the last work analyzed, the authors of [29] introduce a location module in their model. The point of this module is to produce binarized importance vectors of each region in an image. Unfortunately, those binarized vectors are a result of non-differentiable processing. This essentially splits the model into two, requiring that the location modules are trained ahead of the actual model. These methods are very resource intensive and rather cumbersome, both to train and use.

An evolution of these techniques appears in [34] where the authors have found a way to unify the training of the stochastic hard attention model and the actual model. To put it simply, their model branches out to incorporate the stochastic hard attention model and its output, which is a binary mask, is then multiplied with the data used to produce the binary mask. The actual model used to perform the hard selection is called REINFORCE [5], and has since become the standard way of performing end-to-end differentiable hard attention [83, 29, 114, 108, 105]. This suffers from the same issue as [54], the gradients produced are not what one would expect. It is an approach that produces correct results but could affect the model's ability to learn a smooth function, without which it might be vulnerable to adversarial attacks.

After this deep dive into how different implementations of hard attention work under the hood, it is safe to say that there is room for improvement. At a later chapter Learn to Mask will be introduced as a solution to many all of the issues presented here.

2.7 Back Propagation

Back propagation is the mechanism that distributes the value of the error function to all the weights so that adjustments can be made with the goal of reducing the model's overall error. The process remains the same as the one described in the perceptron section for all output layers since their relation to the error is direct. For all intermediate, hidden, layers the error would have to be calculated with respect to the errors of the following layer.

The calculation of the error of the output layer is defined as the gradient of the cost function J with respect to the jth output activation a, multiplied by the gradient of the activation function σ at z_j^L.

$$\delta_j^L = \frac{\partial J}{\partial \alpha_j^L} \sigma'(z_j^L) \tag{2.26}$$

Propagating the error backwards to layer l is done in terms of the error in the next layer:

$$\delta_j^l = \frac{\partial \delta_j^{l+1}}{\partial w_j^{l+1}} \sigma'(z_j^l) \tag{2.27}$$

Similarly, for bias, the error for any layer:

$$\frac{\partial J}{\partial b_j^l} = \delta_j^l \tag{2.28}$$

Given the consecutive distribution of the error to all of the model's weights, earlier layers receive much smaller gradients than the ones position closer to the end. Deep models often suffer from the vanishing gradient effect, in essence, the earlier layers end up receiving no gradients at all making training nearly impossible. This effect is often combated with skip connections between layers and the use of activation functions whose gradients do not drastically diminish their input values.

2.7.1 Gradient Descent

Despite being able to calculate the error with respect to each weight and bias, there is still the need for the appropriate update rules. Gradient descent [81] is one such method, previously called optimizer, that cares for those updates. Effectively, it tries to minimize the output of the cost function J with regards to the inputs of the model.

$$w_{t+1} := w_t - \eta \nabla_x J \tag{2.29}$$

$$b_{t+1} := b_t - \eta \nabla_x J \tag{2.30}$$

where η the learning rate. Gradient descent is a generic optimization method, and will later be used, by adversarial attacks, as a tool to optimize the malicious payload in order to maximized the misclassification probability of the adversarial sample.

More complex optimizers are normally used for weight updates, such as Adam [9], that have proven to offer faster convergence. The underlying principle remains the same, to minimize the error function, so other optimizers will not be analyzed.

2.8 Differential Dynamic Programming

Besides the standard layers mentioned in this chapter, and a whole lot more that have been introduced over the years, the need for more complex modules and explainable models has led to the rise of custom layers being implemented as needed through differential dynamic programming [113, 56].

Differential dynamic programming is the practise of writing code that can be automatically differentiated by modern frameworks [1]. Constructs like conditionals and loops can be interpreted in a way that allows gradients to flow through them providing the benefit of complex logic without the need to sacrifice end-to-end training. End-to-end training refers to training models which receives raw data and provides the final output without the values having to ever be taken out of the model.

This is not done just for convenience but also for efficiency. Programs written in that way can be trivially parallelized by auto-differentiation frameworks. This allows for Single Input Multiple Data (SIMD)

execution on the GPU without the need for the programmer to handle a wide variety of tensor shapes explicitly. The other benefit is that since code written in that way has to be differentiable, it can help guide and form the gradients that will be passed down to previous layers.

In case one wanted to introduce a non-differentiable layer as part of the model, it would soon become very apparent that this decisions has led to the creation of 2 separate models that have to be trained separately. End-to-end differentiable solutions enjoy many optimizations that are provided by auto-differentiation frameworks, and are much easier to train and operate than if that was not the case.

Chapter 3: Adversarial Attacks

Adversarial attacks [36] are a well-known vulnerability inherent to all neural networks. To one degree or another, all neural networks are susceptible to such attacks and the damages caused by them, in critical applications, can be quite catastrophic. The goal of adversarial attacks is to leverage the very fact that neural networks represent a continuous and differentiable function against them.

Executing such an attack requires that the attacker has access to a copy of the trained model's weight and architecture. Having that, it is now possible, by blending legitimate input with pseudo-random noise sampled from a normal distribution and feed it to the model. This is very unlikely to change the model's output but, were one to measure the loss J between the model's output and the desired output then all it takes is to differentiate the loss function with respect to the noise added to the original input. These gradients can now be used to optimize the initial noise and blend it with the original image to produce a new, stronger, adversarial sample.This process is reiterated as necessary until the adversarial sample is classified into the desired class.

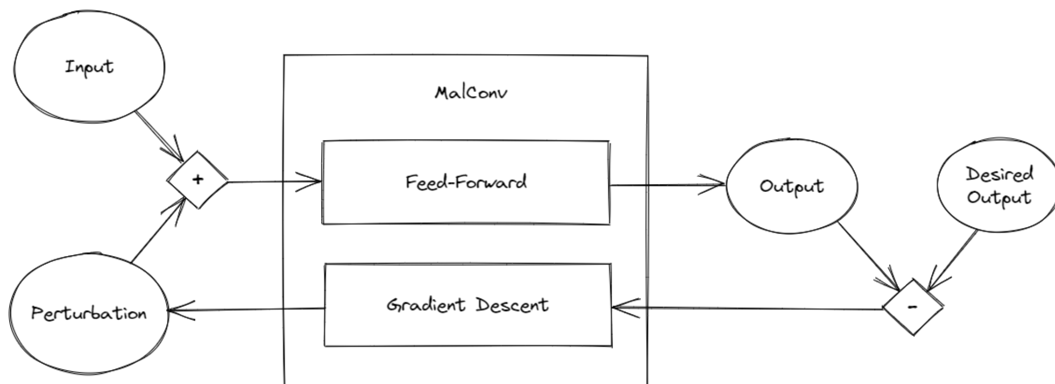


Figure 3.1: Visualization of a gradient-based attack as a state-machine.

It is important to keep in mind that those perturbations are incredibly imperceptible to humans. In tasks such as face or speech recognition it is impossible to differentiate adversarial samples from legitimate ones. Despite the small size of those changes, the output is drastically different.

Adversarial Attacks are the product of the lack of explainability in neural networks. Those two subjects are fundamentally linked together and progress on one of them means progress in both. One of the hypotheses tested in this work is whether a more explainable model would be more robust against such attacks.

3.1 Input Space or Attack Vector?

By definition, inputs to a neural network $\mathbf{X} \in \mathbb{R}^n$ where n the number of features. By today's standards, the input space of any model is chaotically large, too large to sufficiently explore with any dataset. Therefore, by the time a model has been trained and tested, conclusions have been drawn about its behavior by observing the results of a fraction fraction of the possible input. Though this is the golden standard for training models it does little towards validating the correct behavior of the model being

examined, regardless of the amount of test samples used.

The existence of adversarial attacks very clearly state that there are such sub-spaces around most samples that should belong to the same class as the sample but the model is completely incapable of assigning them to it. This issue is in many ways connected with our inability to explain what a model has learned throughout the entire training process. Standard validation and testing procedures, despite being the golden standard in the deep learning community, pull us into a false sense of security about the model's ability to generalize.

Specific attacks will be analyzed at a later chapter and applied on MalConv [76] as well as the proposed model.

3.2 Interpreting Adversarial Attacks

To a large extend, we have accepted that models are black boxes, incapable of providing any information, whatsoever, with regards to how they organize knowledge gained from training. Yet, adversarial attacks are excellent tools at finding loopholes in a model's knowledge. Therefore, by induction, adversarial attacks can be considered a form of proof, a proof with respect to a very specific attribute about the model at hand. The introduction of adversarial attacks in [36] was with a technique that blends pseudo-random noise in pictures and optimizing it through gradient descent until the model produces wrong output. This should serve as valid proof that this model is not resistant to noise, which implies a faulty architecture. The architecture has forced the weights to converge in a way that the resulting function lacks certain traits. Most commonly, this is seen as an issue with the dataset itself, which causes wrong and expensive steps to be taken, as remedies to that situation.

3.3 Adversarial Attacks as Black Box Attacks

Earlier in this chapter it was mentioned that in order for an adversarial attack to be executed, the attacker would need access to the trained model and its architecture. This is not necessarily true. While adversarial attacks were initially considered white box attacks, meaning that the attacker would need access to the system, ways to perform adversarial attacks as black box attacks have emerged.

3.3.1 Transfer Attacks

One way of performing black box adversarial attacks is achieved through a very interesting property of them, transferability [27, 101, 103]. Perturbations that are generated by attacking model A, often called surrogate, in domain X and have cause the model to misclassify a sample, are more than likely to work equally well on a model B, called target, but in the same domain X. Essentially, adversarial samples produced on sufficiently complex surrogate models can also fool models that solve the same problem while there is no knowledge of their architecture or weights.

Transferability, as a metric, was more formally defined in [27] as the loss attained on the target classifier. What is also discussed in this paper is that there appears to be a connection between a model's complexity and its vulnerability to attacks. Simpler classifiers tend to learn smoother functions and this directly

correlates to their adversarial robustness. As it stands, models that have no regularization applied to them, during the training process, are much less robust.

3.3.2 Gradient Estimation Attacks

The second way of performing black box adversarial attacks is through the gradient estimation method [17, 45, 80, 6]. This method has involved from perturbation transfer attacks and uses the gradients computed on the surrogate model to estimate the gradients on the target model in order to produce stronger adversarial samples than if one were to just use simple perturbation transfer attacks. This works with various degrees of success and, as mentioned in the explanation of transfer attacks, the white box model and the black box model would need to be of similar complexity.

These attacks are harder to execute but have been shown to yield results. They do require a significantly larger amount of iterations than white box attacks but it is still a widely unexplored field with new methods appearing regularly. A great example of gradient estimation attacks is presented in [45]. The authors' idea was to attack a video-based recognition model through the gradient estimations produced by an image-based recognition model. Their results show a 93% attack success rate after a couple million iterations. This proves that even though videos have a couple of orders of magnitude larger input space, attacks on images can still transfer to them.

3.4 Defense Strategies

Wherever adversarial attacks threaten the viability of deep learning models as a solution to critical tasks there have been developed methods to try and thwart such attacks [68, 57, 88, 23, 52, 53]. Lots of research has gone into creating methods which will secure neural networks from adversarial attacks but this is primarily done in a case by case basis. This field is still in its infancy and lacks concrete methodologies to provide defences regardless of the domain. Despite that, published work on defending against attacks falls into one of two categories. Current defense strategies either focus on detecting the adversarial input before it is processed by the model or make use of adversarial samples as part of the training.

At the end of this section, a new method is proposed to defend against adversarial attacks that is based on property testing and architectural changes. This method in no way alters the input, during training or inference, but gravitates towards building models that carry some form of proof about their properties, and have a more explainable architecture that constrains the spaces between layers in such a way that reduces the overall attack vectors of the model.

3.4.1 Preventive measures

As a defense mechanism, it is not uncommon to screen all input that will query the model and filter out any that are judged to be adversarial samples [68, 57, 88, 23]. In cases where the data is simple and can be validated algorithmically, this method presents a good way to protect a weak model. Algorithmic validation provides little overhead, when compared to querying a neural network, with great results. The issue with this form of validation is that the data given to neural networks are often rather complex, in

fact the reason someone would use deep learning as a solution to an issue is because it is very hard to analyze the data with traditional means.

There have been a few tools developed to aid in that regard with the most common being monitoring the input's entropy levels. Every input space for a given domain is characterized by an entropy distribution, adversarial samples, due to the perturbations, often have entropy levels that do not comply with that distribution and may be identified in that way. Of course, as always with security issues, this is an arms race, an astute attacker would optimize the perturbations with 2 goals in mind, high misclasfficiation capabilities and low entropy change. Another tool for such preventive measures is extra preprocessing. Preprocessing such as denoising can help mitigate the effect that the perturbations have on the models decision process. Those too are measures than can be counteracted simplify by identifying them and making them part of the perturbation optimization process. While not an easy task, it is easy to see that the amount of security provided through such means is rather temporary. More advanced tools have been developed which are based on probabilistic methods [95] but their effectiveness is inversely proportional to the size of the input space.

Given the complex nature of identifying adversarial samples many sought a solution in neural networks. Neural networks should be sufficiently capable of examining input of any complexity and classifying it as legitimate or adversarial. This method really blurs the lines between preventive measures and adversarial training, which will be analyzed later on, it combines the benefits of both while also inheriting their pitfalls. As a preventive measure it will work but partially, the adversarial sample identifier is only as good as the training data. This implies that training the identifier model will be done so with multiple adversarial samples produced from legitimate samples. At this point it should be reminded that there are a multitude of ways to generate adversarial samples while more unique attacks can be created depending on the domain. Therefore, an identifier model would have to be trained against as many attacks as possible.

This is simply not viable, this method cannot scale well for production-grade solutions as this significantly increases the hardware requirements of the solution. The need to have all input validated in this way greatly increases the response time of the solution as well. The security benefits are hard to measure and the success rates vary, and even when they are good, one must very carefully consider the cost of this solution, in both time and money required for the development and operation of this. Lastly, the model produced for this purpose is vulnerable to attacks itself. Granted, the attack will now have to fool two models instead of one but it could be argued that one needs only attacks the identifier model since the domain model lacks any concrete defences.

In conclusion, acting preventively essentially aims to tighten the input space in a way that only legitimate input are considered valid. The reasoning is valid but the execution often leaves much to be desired. Acting preventively can be a valid defense strategy when one can reason about their input in an algorithmic way. I do not believe that producing an identifier model is worth the effort and cost, this method simply relocates the problem to the identifier model instead of solving its root cause, that the domain model lacks the capability to produce correct output irrespective of minor alterations to otherwise legitimate input.

3.4.2 Adversarial Training

Another common defense mechanism employed against adversarial attacks is adversarial [52, 53]. This method aims to fix the vulnerability during training, by including adversarial examples in the process. This allows the model to explore a much larger section of the input space with the hope that this provides the required regularization effect on the weights to produce a robust model that can withstand most attacks.

This method also has some validity and effectiveness but comes with two very serious pitfalls, the cost of implementation and the cost of maintenance. Adversarial training requires that all known attacks are implemented so that one can produce adversarial samples. Adversarial training works much like dataset augmentation methods with some added complexity due to the need to re-calibrate the perturbations while training. All this complexity takes its toll on the time it would take to train such a model and the overall resources required. Should the model's complexity be insufficient to generalize over the adversarial samples then a more complex model would have to be introduced, further increasing the cost of this method.

The cost of sustaining this method becomes really apparent when the attacker is capable of figuring out new attack vectors that were not covered in training. The model would have to be retrained, either from scratch or from its current state. Though, in order to do that one would have to reverse engineer the adversarial samples in order to extract the attack vector utilized. This is an incredibly difficult task that would require an incredible amount of expertise and knowledge on the field, not to mention the amount of time it would take, which would leave the model vulnerable.

Lastly, this method often comes at a cost of accuracy for the original model. There are cases [53] where to achieve some adversarial robustness it made sense to sacrifice the model's accuracy. Altering the training process in that way is more significant than it may seem and may actually introduce new attack vectors. Research has shown [62] that even the order in which input samples are introduced can drastically alter the model's behavior. Since there are no guarantees about the attributes of model which implies that methods which directly alter the training process in such a drastic way may introduce new attack vectors while trying to patch existing ones.

3.4.3 Architectural Defences

During the analysis of the already existing defense mechanisms, it was made clear that they create more problems than they solve. The main concern is that they are ad hoc solutions that make no attempt to tackle the root of the problem, that deep learning, as a field, is producing models that do well in benchmarks but we do not necessarily understand. Adversarial attacks exploit the lack of knowledge there is about any model's inner workings and manage to cripple model performance. So far the answer to that was to throw more resources at the problem, more models or larger models. This way of operating is not sustainable in the long term, and security is a long term commitment. Any change to the model's input will cause a completely different model to be produced from training, and, in the end, they are compared using metrics that do not aid at all in understanding how the model has changed its behavior. It all comes down to the fact that we are trying to achieve behavioral changes in models but try and measure these changes with performance metrics.

Attackers use the mathematical nature of neural networks in their favor, but so can the defending side. Neural networks are mathematical functions, and even though it might be incredibly tough to provide explanations for every single weight in them, there are better ways of learning things about them. Conclusions can be drawn about a model's properties. Given a test set $\mathbf{X}$ of legitimate input, we can produce an altered test set $\mathbf{X}^*$ in which every sample from the original test set has been altered in one specific way. For example, $\mathbf{X}^* = \{x, x + c|\mathbf{X}\}$ where c is pseudo-random noise sampled from a normal distribution. Using those 2 test sets we can now measure the model's resistance by comparing the changes in the cost function's output. This can be generalized to comparing performance metrics before and after the attack for both models.

In that way, it is possible to understand the behavioural differences between two models in a more fundamental way than if performance metrics were blindly used. This further explains why I would consider identifier networks as a really bad idea, since they are a different network from the classifier, there is no sensible way to compare their behavior with the method described above. This would be possible though with adversarial training, by calculating the above correlation coefficients with for both the original model's weights θ and the weights after the adversarial training has completed θ^*. If the adversarial training was successful then it should hold that:

$$[\mu J(\theta, \mathbf{X}) \geq \mu J(\theta^*, \mathbf{X})] \wedge [\mu J(\theta, \mathbf{X}^*) > \mu J(\theta^*, \mathbf{X}^*)] \tag{3.1}$$

The retrained model should have a decreased cost on the test set, or at the very least equal, when compared with the original. For the adversarial training to have been a success the resulting model should also have produced a lower cost when being tested against the altered set. An equally valid, more practical, comparison between the two models can also be made by comparing their accuracy metrics with the standard test set and the altered test set.

Another issue taken with the current approaches is that they end up greatly increasing the training time and resources required for the model's operation. Instead of making changes to the training data or trying to build layers of protection around the model it is proposed that adversarial attacks can be evaded through purely architectural changes. The model's behavior is a function of its weights and it logically follows that if the behavior is undesirable then function must change. Adversarial training does technically change the model's behavior but not fundamentally, it is essentially a very costly weight regularization method [40, 102].

The proposed way of defending against such attacks is through changes to the model's architecture, so that the model possesses all the required attributes upon completion of the training process. To create a robust model one would have to map out all the attributes that the model is required to have for that specific task and design the architecture around that. This way of thinking is akin to property testing [71] in software engineering while traditional testing, in machine learning, would be more in line with unit testing [21].

In order to make educated architectural changes one would have to really understand the original architecture, and this directly links back to the relation between explainability and adversarial attacks. Making such

changes in most models is by no means an easy task, since most models lack some form of explanation as to why the architecture is one way or another, or what was accomplished by the use of a given layer at the point it was placed. Making an effort to create more explainable architectures would go a long way towards making informed decisions with respect to architectural changes.

To that extend, and as a case study, MalConv [76], a malware detection model is studied. In the following chapter, MalConv will be analyzed and then redesigned to achieve adversarial robustness with this method. The results before, and after, the redesign will be showcased in their dedicated chapter. Lastly, the use of hard attention, in the form of Lean To Mask, will be analyzed as a way of boosting MalConv's adversarial robustness.

Chapter 4: Learn to Mask

This chapter goes through a thorough explanation of the novel algorithm introduced, Learn to Mask (L2M), where it draws inspiration from, and how it impacts a model's architecture positively with regards to robustness against adversarial attacks.

Differential dynamic programming is the field that concerns itself with producing code which describes differentiable functions. L2M had to be implemented in such a way so that it can become an integral part of the network and allow gradients to flow through it. L2M's differentiability allows it to become part of training and correctly dictate the importance of each value that it receives. This chapter will also show how the fact that L2M manages to perform hard selections while remaining differentiable sets it apart from other constructs that manage to accomplish the same thing but fall short.

4.1 Inspiration

Learn To Mask draws inspiration from Learn to Select (L2S) [28] which has the same principle but a different execution. In fact, L2S was used in early experiments but some of its properties made it a bad fit for the task.

L2S also selects M out of N values but does so in a sparse matrix, in essence, each selection is represented by a different row in the output matrix, which has $(M \times N)$ dimensions. L2M came to be through 2 core changes to L2S. First of all the softmax function was removed and, secondly, all the selections were merged into a single vector, to conserve memory and reduce sparsity. After those changes, the model had a reduced memory footprint and increased accuracy, even when compared to the original model.

4.2 Definition

The input of the Learn2Mask sub-network is a sequence **s** of $(N \cdot K)$ values. It is important to state that **s** must satisfy $\mathbf{s} \in \mathbb{R}^{+}$, meaning that only positive values can be used as input for L2M. This sequence is split into N groups of K values per group. L2M recursively extracts the M most important values into a new sequence **y** while preserving their location and value. Effectively, the sub-network clears the values that are deemed less important for the classification task.

L2M's output is calculated through a recursive function:

$$y(n, s) = \begin{cases} maxInPlace(\mathbf{s}), & \text{if } n = 1 \\ y(n-1, \mathbf{s} - maxInPlace(\mathbf{s})) + maxInPlace(\mathbf{s}), & otherwise \end{cases} \tag{4.1}$$

where n is the number of iterations that should be performed and s is the vector from which the selection should be made. $maxInPlace$ is a helper function that produces a vector will all elements equal to 0 except for the one that is equal to the max value of the vector. For $\mathbf{s} = [\mathbf{s}_1, \ldots, \mathbf{s}_L]$:

$$maxInPlace(\mathbf{s}) = \big[sign(\mathbf{s} - max(\mathbf{s}) \cdot \mathbf{u}) + \mathbf{u}\big] \cdot max(\mathbf{s}) \tag{4.2}$$

$$\mathbf{u} = [1, \ldots, 1] \tag{4.3}$$

Then the output gradients are readily computed:

$$\frac{\partial \mathbf{y}_n}{\partial \mathbf{s}_n} = \begin{cases} 1, & \text{if } \mathbf{y}_n \neq 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

In order to aid in the understanding of the above function, its expansion for 2 selections and $s = [1, 2, 3, 4]$ is shown below:

$$\begin{aligned}
&y(2, [1, 2, 3, 4]) \Leftrightarrow \\
&y(2 - 1, [1, 2, 3, 4] - maxInPlace([1, 2, 3, 4])) + maxInPlace([1, 2, 3, 4]) \Leftrightarrow \\
&y(1, [1, 2, 3, 4] - [0, 0, 0, 4]) + [0, 0, 0, 4] \Leftrightarrow \\
&y(1, [1, 2, 3, 0]) + [0, 0, 0, 4] \Leftrightarrow \\
&maxInPlace([1, 2, 3, 0]) + [0, 0, 0, 4] \Leftrightarrow \\
&[0, 0, 3, 0] + [0, 0, 0, 4] \Leftrightarrow \\
&[0, 0, 3, 4]
\end{aligned} \tag{4.5}$$

In a similar fashion, follows the expansion of the $maxInPlace$ for $s = [1, 2, 3, 4]$:

$$\begin{aligned}
&maxInPlace([1, 2, 3, 4]) \Leftrightarrow \\
&(sign([1, 2, 3, 4] - max([1, 2, 3, 4])) + 1) \cdot max([1, 2, 3, 4]) \Leftrightarrow \\
&(sign([1, 2, 3, 4] - 4) + 1) \cdot 4 \Leftrightarrow \\
&(sign([-3, -2, -1, 0]) + 1) \cdot 4 \Leftrightarrow \\
&([-1, -1, -1, 0] + 1) \cdot 4 \Leftrightarrow \\
&[0, 0, 0, 1] \cdot 4 \Leftrightarrow \\
&[0, 0, 0, 4]
\end{aligned} \tag{4.6}$$

The astute reader would notice a very rare edge case in this algorithm. If the input vector s has more than 1 values tied as the maximum elements then this method will select all of them. Therefore, it has to be stressed that that n does not represent values selected but iterations of selections performed. On a practical level this case is so rare that it may never occur when using L2M. Even if it does happen, this

should, by no means, negatively impact the layer's performance or disrupt the differentiation process.

Finally, it should be noted that the process remains the same for multi-dimensional input data. The only difference is that the user must now decided the level at which selection are made. Consider the application of L2M to images as input with size $(Channels, Width, Height)$. By configuring the axis in which L2M is applied it can be tweaked to select single values from any dimension, rows, columns or all channel values of a pixel. In order to select regions a form of partially connected layer would be required.

During training, L2M learns to pay attention to specific binary parts of the input while masking out the rest. This acts as a defense mechanism against adversarial attacks, causing the model to prioritize bytes that were deemed important to making the distinction between malicious and benign software. Such a masking mechanism helps in producing a model that is much more resistant to adversarial attacks, without compromising the model's accuracy. The primary benefit presented with the use of L2M is the ability to evade adversarial attacks without explicitly training against them.

A second, equally important benefit, is that it is the only end-to-end differentiable hard attention algorithm that does not require training. Instead, it forces prior layers to adapt in such a way that they reinforce their activations for important information while pausing training for those not important.

Interestingly enough, it is possible to utilize a different number of selections for inference than it was used in training. Early experiments showed the reducing it during inference could have some positive benefits for the model's performance but this cannot be said with certainty as this feature remains largely unexplored.

Lastly, when compared with dropout, it can be seen that L2M should provide a similar regularizing effect that should help the model learn smoother functions and combat overfitting. This will become more apparent by the end of this chapter, through the parallels that will be drawn with attention mechanisms and dropout layers.

4.3 Learn To Mask as a form of Attention

Given that L2M selects M values out of N inputs it belongs in the family of attention mechanisms and it best fits the definition of hard attention [63] as the values selected values are never altered. L2M manages to stand out in this category as a solution that remains differentiable, does not inhibit training, and does not alter the input values that are important enough to be allowed through. To my knowledge, this is a first as other techniques that can be categorized as hard attention, discussed in the chapter about neural networks, do not share those characteristics.

L2M is unique in the sense that it is the very definition of hard attention, not only does it preserve differentiability but it also provides the expected gradients. It allows the error to be propagated to previous layers unaltered, for important values, and completely prevents all training for values that are deemed not important.

Since L2M belongs in the same family as hard attention, it would make sense to expect similar benefits.

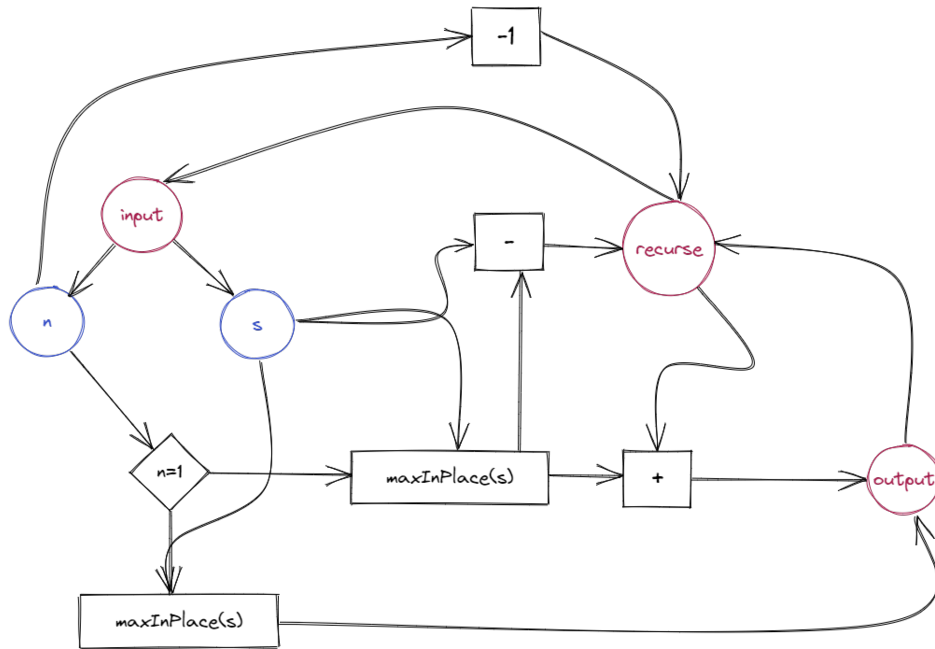


Figure 4.1: Visualization of Learn To Mask as a finite state machine.

The authors of MalConv did mention that they had no success when applying soft attention to their model thought they never mention any attempts with hard attention. It makes sense that they never actually considered hard attention as a viable solution since the large input space of MalConv would require that a quite heavy reinforcement learning agent be introduced to produce the binary masks. The benefits of its use are simply outweighed by its cost and complexity as a solution. The same is not the case for L2M. L2M bears no weights and, thus, its impact to the model's complexity and memory consumption is minimal.

4.4 Learn To Mask as a form of Dropout

Learn To Mask also shares a lot of similarities with the Dropout layer [90]. Dropout was designed to tackle a completely different problem, it aims to reduce overfitting during training. The way it achieves that is by randomly deactivating a predefined number of neurons from the previous layer. Deactivating a subset of the neurons in a layer, along with their input and output, is akin to experimenting with multiple different architectures in parallel in one training session. This process adds noise to the training process forcing neurons to take various degrees of responsibility of their input. In [90] it is mentioned that, in a standard neural network, neurons may be forced to adapt their weights so that they make up for the mistakes of other neurons. The authors call that these produce complex co-adaptations that inhibit the model's ability to generalize. Dropout breaks up those co-adaptations, producing better performing models.

Given the above description, it is not hard to conceptualize dropout as a binary mask generator, which hints at the relation with L2M. They both cause layers to produce sparse output. Where they differ though is that dropout is only active during training, after which it is discarded. Obviously, it would not make sense to randomly drop values that could be important to the model, but L2M can afford to drop values

during inference since it is able to extract the important ones. With all this in mind, it the regularizing effect of dropout is something that should also be present in L2M.

4.5 Hypotheses

After explaining how L2M works, and studying it in comparison with other similar algorithms, it is time to summarize the list of hypotheses. These hypotheses are to be tested through the application of L2M in MalConv. The use of L2M should:

1. increase the model's accuracy, based on its relation with hard attention
2. be able to help reduce overfitting while improving the model's generalization capabilities due to its commonalities with dropout [90]
3. should considerably boost MalConv's adversarial robustness, given that it should not allow the malicious payload to pass through it.

Chapter 5: Malware Detection

In this chapter there will be given some background on the domain of malware detection, traditional methods and their limitations. Newer methods, primarily based on deep learning will also be discussed, what do they improve upon, along with the shortcomings that they carry. All of that context is provided in order to build up to the explanation of MalConv in the following chapter.

Malware Detection is an inherently complex task. There is very little room for error as mistakes on both sides of the spectrum can produce damages. It is easy to imagine the damages when a malicious file is not detected but it should be noted that it is an equally costly mistake to produce an anti-virus software that will flag down perfectly legitimate software. False positives are equally bad as false negatives. Malware detection tools have to be reliable, trusted and many steps ahead of the attackers. It is, by all sense of the phrase, a critical application.

5.1 Traditional Malware Detection Methods

Due to the varied nature of malware it is increasingly hard to differentiate it from benign software. There are a couple of different approaches, all used in conjunction, for the best possible result. Malware detection is split into three schools of thought [22]:

- static analysis,
- dynamic analysis,
- and hybrid analysis.

5.1.1 Static analysis

Static analysis refers to the examination of the binary itself, without it ever being executed. This is the main form of analysis performed by standard anti-virus software for personal computers. Method that apply to this range from very simplistic, such as checking the validity of the binary's signature to reverse engineering methods that aim to recreate the original source code in order to pinpoint the malicious code.

A binary's signature is a unique number that characterizes the binary. The way it is created is through cryptographic hash functions such as SHA1 or MD5. Once a binary has been identified as malicious its signature will be added to various malware repositories. Malware repositories are routinely consulted by anti-virus software as a means of relying on prior analysis to produce fast and accurate results for already discovered malware. The same thing happens for software that contain no malicious code. This is a very reliable method but one that cannot offer any protection against new threats.

Things get a bit more involved when tasked to identify a binary which has never been seen before [32, 82, 72, 69]. This immediately increases the complexity of the methods. One of the simplest is the extraction and analysis of all system API calls. In order for the malware to cause any damage, it does have to reach out some form of resource, whether that is the network, the file system or some other resource it does not

matter. What matters is the access pattern, and what it is trying to access. If a binary is seen to attempt to access kernel files that no reasonable application should have to, or sending data to suspicious URLs, it would be a good indicator that it is up to no good. Notice the word "indicator", most of these tools do not provide concrete proof that a file is or is not malicious, all they do is go through a set of predetermined rules which help score its level of maliciousness. In order to have concrete proof that a file is indeed malicious, an expert would need to pinpoint the malicious code.

Experts that do this short of analysis rely on a variety of tools that relate to the field of reverse engineering [11]. Depending on the type of the executable this would involve de-compiling the binary into source code, looking at the assembly instructions and data or any other method that could aid in the process. This is where tools like [37, 106, 107, 70, 11] have emerged to aid the process using heuristic or probabilistic methods.

It is important to keep in mind that for all the methods described, and all those that will be described, there have been attempts to develop counter-measures from the point of view of the attacker. It is common practise to obfuscate the source code before compilation in order to made it illegible by anyone that tries to extract it from the binary, though obfuscation on its own is not enough to flag a binary as a malware since software companies do follow the same practise to protect the business logic of their valued applications. Another method is to encrypt any data present in the binary and decrypt them on run-time after recovering the decryption key from a remote server.

The amount of expertise and resources required to provide security just through static analysis has led to other approaches. These approaches constitute the field of dynamic and hybrid analysis.

5.1.2 Dynamic & Hybrid Analysis

Dynamic analysis revolves around the examination and probing of the malware during its execution [87, 7, 94, 2]. Specialized tools have been developed for that purpose, that monitor any changes to the system along with the ability to recognize malicious behavioural patterns in the way the executable interacts with the system.

Virtual machines and sand-boxed environments allow for contained execution of malware in such a way that they can be monitored with safety from tools placed on the host machine. Dynamic analysis has evolved to be as challenging of a field, if not more, as static analysis. With its arise, attackers sought to reduce the possibility of their malware being flagged by dynamic analysis tools. During this fields infancy it has been known of malware that were able to escape the sand-boxed environment or even alter their behavior if they could determine they were executed inside one. Specific malware can lie dormant for a long time before they actually "detonate" and start disrupting the system's normal functions.

To provide the most amount of certainty that a file is indeed malicious a combination of tools from both static and dynamic analysis are utilized, forming hybrid analysis. Hybrid analysis joins the two fields not only to identify malware but to also provide an accurate description of the malicious behavior and implementation details. This information is then formalized and utilized by analysis tools to further improve their results.

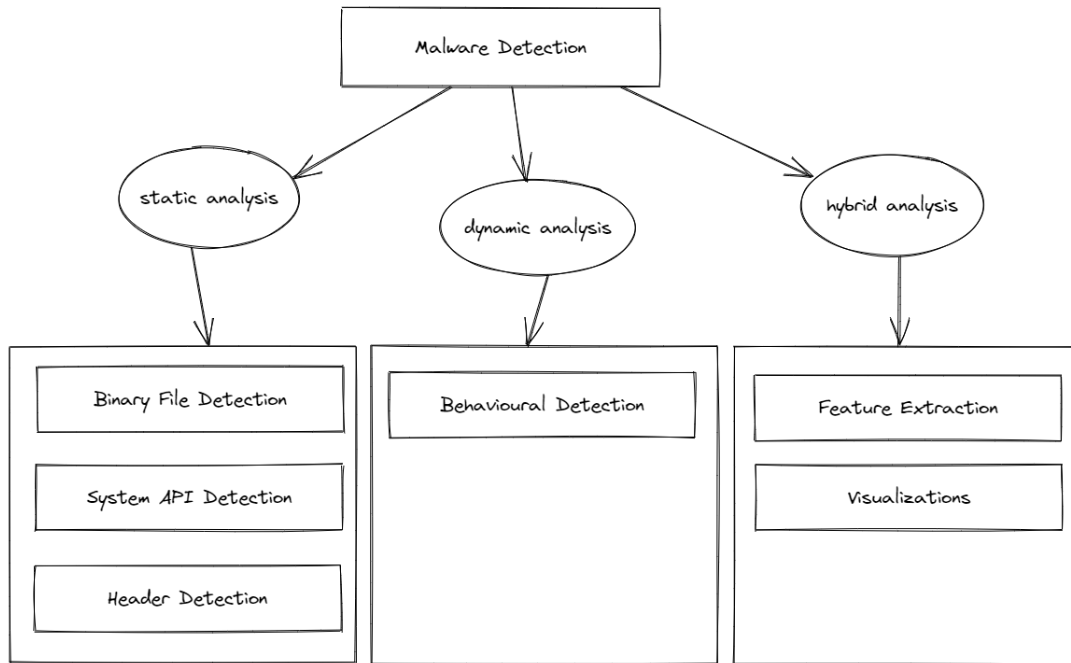


Figure 5.1: Diagram of the contributions by neural networks to malware detection.

The continuous arms race between malware developers and security researchers has eventually led to the employment of artificial intelligence as a defense mechanism.

5.2 Deep Learning as a static analysis tool

With the advent of deep learning as a tool to make complex decisions with super-human accuracy, it makes sense that it would eventually be used to add to the tool belt of both static [76, 25, 15, 38, 77] and dynamic [43, 91, 64, 117, 75] analysis tools.

Malware detection does present a challenge even for neural networks though. The size of binaries itself is often prohibitive for use in conjunction with deep learning models due to the memory requirements of the operations performed on such a large input space. A single byte is actually meaningless, unless placed in the proper context. Regardless, there have been workarounds to those issues that work remarkably well.

To make matters worse, different parts of that input space have to be interpreted in different ways in order for them to make sense. The headers, data and code sections all follow their own layout that cannot be encapsulated in the scope of a single byte. Interestingly enough, even though binaries are usually thought of as an endless stream of bytes, malware detection often utilizes the same algorithms and sub-networks present in natural language processing models. These would include the use of soft attention, such as transformers [15] and the use of embedding layers [76].

As a method of circumventing these issues, models were developed that perform static analysis on parts of the binary. Great results [25, 15, 38, 77] have been obtained through solely examining the header section of the binary. One of the issues present in static analysis using neural networks is that the attackers now have the ability to exploit weaknesses in neural networks themselves to fool the classifier. This is where adversarial attacks come into play. With slight perturbations in malware binaries, they may now be

misclassified as benign. Lots of research has been put towards applying adversarial attacks to malware detection models and cataloging their vulnerabilities [49, 12, 91, 58, 65, 84, 31].

On the other hand, the rise of adversarial attacks has led to the rise of research about adversarial attack evasion with regards to malware detection [41, 78, 44]. It has also driven research into the development of dynamic analysis models that would have far fewer chances of being exploited in that way as any perturbation present in the binary would not be fed to the model directly. The use of neural networks in the domain of dynamic analysis has been as a tool to analyze system calls and the behavioural patterns of binaries [43, 91, 64, 117, 75].

The last common way that neural networks have contributed in this domain is through feature extraction [75, 96, 73, 100]. Feature extraction is mainly concerned with compressing information from the binary into vectorized representations that can more easily be used by simpler, less hardware intensive algorithms and models. Neural networks could, for example, be used to create representations about different system API calls with respect to their maliciousness, expressed in a multi-dimensional space. This information alone can be used to train a second, more shallow neural network that will have improved accuracy for malware detection or help an expert uncover the malicious code inside a binary. Extracted features may also aid in the process by helping to create visualizations to be used by human experts.

Chapter 6: MalConv

This chapters goal it so explain MalConv's approach to Malware Detection, why was it even chosen as a case study for this work, and finally where does it come short. Along with that, an introduction to its structure is made, along with the attacks that will constitute the benchmark between the MalConv as provided by the authors its redesigned version that will be exhibited at the end of this chapter.

MalConv was introduced in [76] as a deep convolutional neural network that is able to classify Windows PE files as either malware or benign. It manages to score an accuracy of 93% using as input the entirety of the binary, as a one-dimensional signal of bytes. MalConv stands out because it has managed to solve a rather complex problem, with great accuracy and utilizing a very shallow neural network by today's standards.

It may not be considered a state-of-the-art model in the general category of malware detection, other models do outperform it [25, 15, 38, 77] with some like SLAM [15] boasting a detection rate of up to 98%. The difference is that those models work very differently. Their approach to malware detection relies on the examination of the binaries headers. Security is about being one step ahead, therefore ignoring the entirety of the binary except for a rather small section of it may be problematic. Consider the case where a well-known benign executable is modified to

The case of MalConv stands out as one the only one that accurately performs malware detection, and therefore static analysis, on the entire binary. Ingesting entire binaries for malware detection purposes is not entirely new though. It has been explored much earlier with the use of n-grams [42, 47, 66]. The use of n-grams does require a lot of memory available to build the distribution at the byte level. N-grams seem to be under-delivering though, the authors of [66] mention that n-grams do not generalize well and require multiple micro-optimizations in order to conserve memory as not all n-grams produced are useful. Another issue briefly discussed is that n-grams are not capable of retaining location information, which directly links back to the fact that different sections of a binary require different interpretation. Therefore a sequence of bytes that may be perfectly normal in the headers section, might actually be a hint towards malicious behavior in the data section.

6.1 Why MalConv?

MalConv was not chosen randomly for this case study. By this point, it has been established that it is a critical application, which, in turn, has proven to be reason enough for a multitude of attacks to be launched against it [49, 12, 31]. Many of the have managed to cripple its accuracy through adversarial samples.

What is interesting is that MalConv's input differ from most neural networks, in the sense that they are not real numbers. MalConv was designed for Windows Portable Executable (PE) files of up to 2Mb, therefore the input space is limited to $S^{2^{21}}, S = \{x | 0 \leq x \leq 255, x \in \mathbb{N}\}$. While the input space is rather wide, in the sense that the number of input values is over 2 million, each of those values is highly restricted in comparison to most models. While this would suggest that an attack vector would be tough to find, it has been proven time and again that, this is not the case.

Another reason why it should be harder to attack a model that has such strictly defined input is that binary executables have a very particular structure, altering a single byte could destroy that structure, rendering it unusable. The attack vector described in [49] revolves around appending random bytes at the end of the executable and optimizing them so that the model misclassifies malware for benign software. The opposite case, causing benign software to be mistaken for malware, will not be examined as it was deemed to hold no real-world value as an attack.

Finally, the shallowness of the model did play some role. It was initially thought that it would be much easier to redesign a small model where the impact of each change can be more easily observed. Smaller models also do have a shorter training time and it is less likely to face issues like the vanishing gradient problem.

6.2 Architecture

This section will analyze MalConv's Architecture as shown in Figure 6.1. Part of this analysis will be to identify architectural weaknesses that can be leveraged later on to design a more explainable model. The analysis is done with adversarial robustness in mind, given the fact that this is a very vulnerable model.To provide an overview, MalConv can be split into 2 parts, the feature extractor, and the classifier.

The feature extractor extends itself from the very first layer down to the global max-pooling layer. It is responsible for condensing all the information that is useful in malware detection into vectors than can be used by the classifier to make the final decision.

The very first step is to turn the binary into 8-dimensional embeddings, the point of the embeddings is to create semantic representations of each byte. By project each byte to the embedding space, the model can have a better understanding how bytes relate to each other semantically through their distance. The embedding layer drastically increases memory requirements for the model but in return provides very valuable information.

The embeddings are then fed to the gated convolution section. Gated convolution is characterized by the 2 convolutional layers which each get a copy of the embeddings as the input and are then fused by the multiplication layer. One of the convolution layers has its output normalized by a sigmoid activation function. The output of the activation function, during the multiplication, allows the model to further shrink non-prominent features produced by the sibling convolution layer. Gated convolution is often associated with a way of representing memory modules due to their ability to better capture context than older methods could, such as recurring layers. They have found great use in both computer vision [115] and natural language processing [110, 24].

Gated convolution at this stage was a great idea by the original authors. It does seem, by the model's results, to capture the context and produce great features for the classifier. As mentioned in the original paper [76], they used a window size of 500 with a kernel and stride of equal size along with 128 filters for both convolution layers. What they authors also did mention is that they would have liked the chance to use a larger window size but the hardware at their disposal could not support that memory-wise. Despite the great accuracy, which can be attributed to the gated convolution, here lie the biggest architectural flaws.

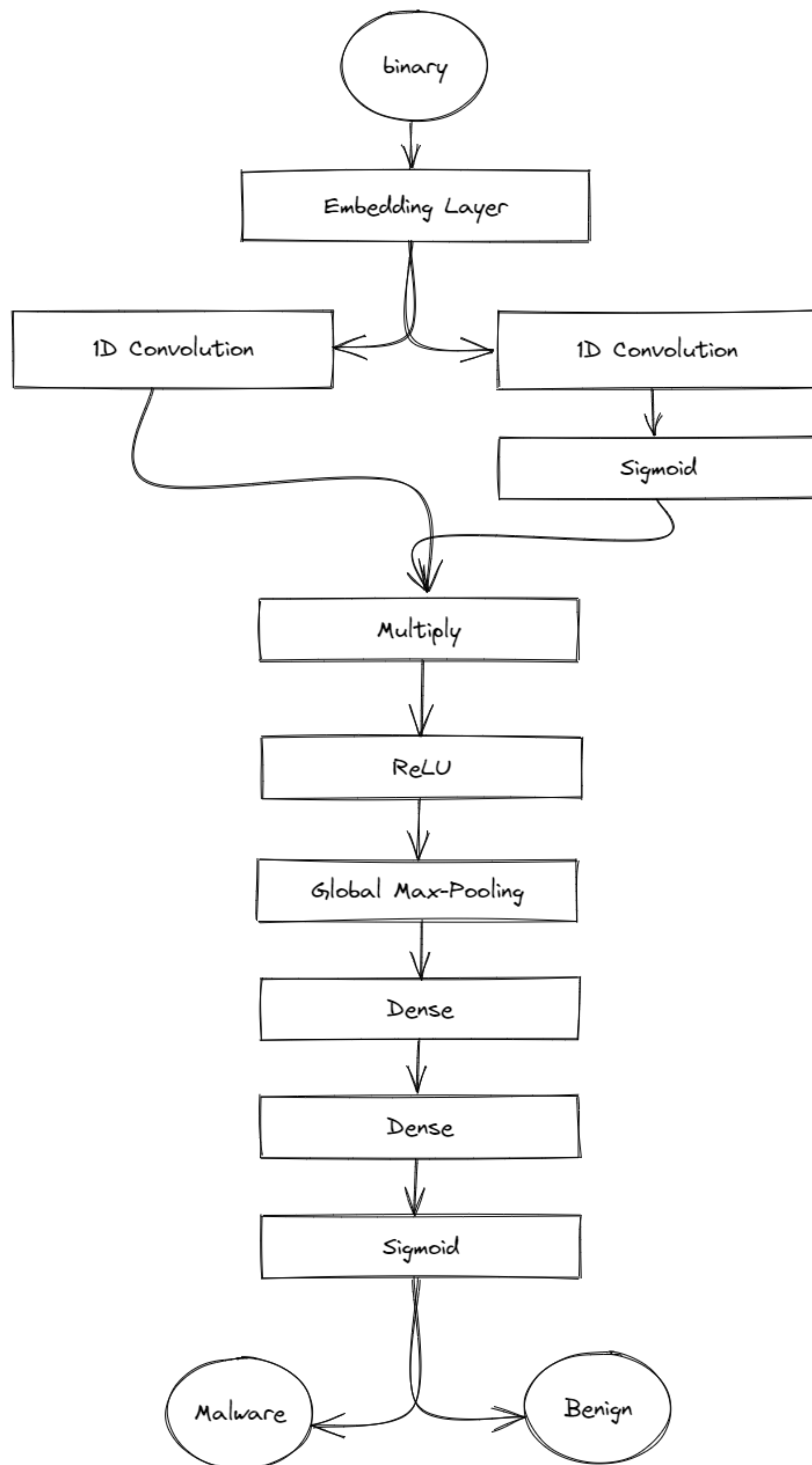


Figure 6.1: Visualization of MalConv's architecture.

It was discussed earlier that different sections of the binary need to be interpreted in different ways. The issue here is that the convolution operation will apply the same weights to the entire binary. What this means, is that, since the convolution layers are operating on byte embeddings, the model has no concrete way of placing different importance on different areas of the binary. They are all treated equally due to the re-use of the kernel weights on the entirety of the embedded binary. This hints at the reason behind MalConv's susceptibility to adversarial attacks where the payload is appended. Without a proper sense of location, the headers could be considered as important as the 0 byes appended at the binary While the model does learn to recognize helpful patterns, it never learns where those patterns are valid. It is now very clear that this is the main attack vector utilized by adversarial attacks to fool MalConv. Replicating a very positive pattern at the end of the binary has a very good chance of fooling the model, despite what other patterns identified might signify.

Another issue here is that examining a context of 500 bytes is rather small, given that machine instruction would range from 4 to 8 bytes, depending on then processor's architecture. This is simply too little context for the model to understand what is presented to it. The larger the window size, the harder it is to find a payload that will cause the model to misclassify a binary.

At this point, ReLU is used to cut-off any negative values, which are essentially values which the model has deemed as not useful to the classification task. The largest of the remaining values is extracted by the global max-pooling layer and this is the only value used by the classifier to detect malware.

The classifier is a pretty standard part of the model, it consists of the last two dense layers as well as the sigmoid used to normalize the output. The first dense layer has 64 neurons and the last one has 1 single neuron. It is a textbook classifier that, honestly, has no issue. Similar classifiers can be found in all classification models, robust or not. The classifier is as good as the data it is being given.

Before moving on to examine how different attacks were executed against MalConv, it is important that the assumptions made in this section about MalConv's architecture are summarized, for ease of reference later on. MalConv:

1. has no concept of location, due to the use of convolution.
2. can be fooled by a mere 500 bytes, so long that these contain high-activation benign patterns.
3. operates on a rather small context, caused by the convolution window.

6.3 Attacking MalConv

This section briefly analyzes some of the most well-known attacks against MalConv, how they work and their results. An attempt is made to provide some confirmation over the assumptions made in the previous section through the findings produced by the researchers that executed these attacks on MalConv.

MalConv is trained directly on the raw input bytes to discriminate between malicious and benign PE files. However, [12, 49, 31] have showed that MalConv can easily be fooled by adding some carefully-optimized padding bytes at the end of the file. To better understand the vulnerability of this deep network,

[26] extended feature attribution to aggregate information on the most relevant input features at a higher semantic level, in order to highlight the most important instructions and sections present in each classified file. Their analysis showed that MalConv learns to discriminate between benign and malware samples mostly based on the characteristics of the file header, in essence, almost ignoring the data and text sections, where malicious content may be hiding. This finding aligns with the assumption that MalConv is vulnerable due to the fact that its convolution output is irrespective of location. To put it simply, given a legitimate sample, the model will produce higher activation from the headers. The fact that it can be fooled by appending a payload implies that these payload, through optimization, approximate the patterns exhibited in the header section.

In [49], a gradient-based attack was introduced to generate adversarial malware binaries. Some bytes in each malware were manipulated to maximally increase the probability that the input sample is classified as benign. That work considered padding bytes appended at the end of the file, to guarantee that the intrusive functionality of the malware binary is preserved. The authors of [49] reported results according to which the accuracy of MalConv is decreased by over 50% after injecting only less than 1% of the bytes passed as input to the deep network. The size of the payload is consistent with the fact that in order to fool MalConv it is enough to produce “benign” patterns anywhere on the binary.

In [12] it was noted that most attack methods against MalConv appended at the end of malicious files perturbations, which were initialized by random noise and modified through an iterative process by gradient-based algorithms. However, the initialization step is very important and can lead to low attack success rates. To address this issue, [12] proposed white-box attack methods that use saliency vectors to select perturbations from benign files, and a black-box attack method for situations when the attackers don’t know the exact structure and internal parameters of the victim model. In layman’s terms, [12] produces saliency vectors by carefully selecting benign patterns that are more likely to cause a misclassification.

Out of all these attacks introduced here, the Fast Gradient Sign method and the Random Byte Padding method have been selected to be re-executed on MalConv, as a means of validation, and to benchmark the redesigned model. Those attacks will be explained in detail, shortly.

6.3.1 Attack Success Rate

Before moving on to the actual attacks, its time to introduce the metric through which they are evaluated, *Attack Success Rate*. Attack success rate measures the amount of samples that were initially correctly classified x over the amount of their equivalent adversarial samples that caused the network to misclassify them $\bar{x}$:

$$AttackSuccessRate = \frac{|\bar{x}|}{|x|} \tag{6.1}$$

6.3.2 Random Byte Padding Attacks

This attack, while unsuccessful against MalConv, is included for multiple reasons. To begin with, it is very simple to understand and execute, showing how a simpler attack works should allow the reader to

better comprehend the next attack which is much more involved.

Secondly, it aims to prove that different attacks exploit the absence of different traits. The trait tested here is whether MalConv is resistant to noise. Unlike images, in binaries noise cannot be blended throughout the binary so, instead, noise is appended at the end of the binary. The attack is performed by appending a predetermined number of bytes sampled from a random distribution to produce the adversarial sample and then feeding it to the network. The success rate of this attacks is incredibly low, less than 2%, which proves MalConv is resistant to noise.

Thirdly, this attack will later be used to verify that the altered model has not introduced new vulnerabilities. In essence, despite the architectural changes resistance to noise should be preserved. More attacks could be included to further reinforce this point but this is out of the scope of this work.

Lastly, random byte padding attacks are the starting point of fast gradient sign method attacks (FGSM), analyzed in the following sub-section. FGSM attacks start by blending, or appending in our case, a malicious payload to create the first iteration of adversarial samples and then work towards optimizing that payload to ensure maximum probability of misclassification for the adversarial sample.

6.3.3 Fast Gradient Sign Method Attacks

The Fast Gradient Sign Method [36] is a well known and highly effective method for producing adversarial examples with the intend of causing slightly altered input to be misclassified. This is done by taking legitimate input, that is already correctly classified by the model and making small alterations to it, called perturbations, by some constant ϵ in the direction of the sign of the the gradient with respect to the malicious payload.

In order for the attack to be properly executed, the original binary has to remain unchanged, otherwise the resulting adversarial example would, more than likely, not be considered a valid executable file. To that extend, a payload is appended to the original file and any perturbations are only applied to that payload.

It was earlier mentioned that the input given to MalConv are not real numbers but natural positive numbers, between 0 and 255. Were we to perform the attack on the input space, the perturbations would be too large, causing the gradient-algorithm to fluctuate between a range of values, unable to find a local, or maximum, minimum. In order to be able to perform small enough alterations to the payload it has to be created and modified in the embedding space. The entire attack is, in fact, executed in the embedding space. After causing the model to misclassify the embedded input, it is the reverted back to the closest possible binary. To reconstruct the adversarial binary a K-Nearest Neighbors algorithm [119], with K equal to 1, is used to map each byte embedding present in the payload to its closest byte value. The newly-produced adversarial binary is fed one last time to the model in order to validate that it is still classified in the wrong class. If that is not the case, then it will not be contained in the set of successful adversarial attacks.

6.4 Redesigning MalConv

MalConv has other shortcomings, besides its vulnerabilities, which have to be carefully considered before redesigning the model. Understanding of the model, its limitations and its issues in general is a very important step towards building a better model. A visualization of the resulting architecture is also provided by figure 6.2.

The main issue the authors of MalConv faced was processing a sequence of over 2 million time-steps. Not only does this pose a technical challenge but also greatly limits the number of algorithms capable of coping with such a vast number of inputs while producing accurate results. Many of the architectural decisions made in MalConv, such as the gated convolution, were made to keep the memory footprint and total number of parameters within reason.

As explained in earlier sections, MalConv performs gated convolution over byte embeddings. The issue with that is convolution assumes that values in the same channel/embedding have the same meaning. Nothing could be further from the truth in the case of binaries. Byte values do have different interpretations according to their context. PE files have different sections, all of which are differently structured. Moreover, a single byte is rarely a standalone piece of information but rather part of a whole. These facts should be reflected in the architecture, gated convolution treats every window as if they are structured in the same way.

The solution implemented for this issue directly affects the architecture and explainability of the model. Input bytes, after being embedded, are grouped into N non-overlapping groups of size K with E embeddings per value. Those groups are then fed to a partially connected layer, with weight sharing among embeddings, that produces a, per plane/embedding, feature for each group.

$$partial_1 : \mathbb{R}^{(E,M,K)} \to \mathbb{R}^{(E,M)} \tag{6.2}$$

Grouping the values in this manner, is a way of providing context for each value. All of that context is compressed into a single feature for each group, for each plane. This sort of information compression in such early layers is the main reason that the resulting model has a far smaller memory footprint despite having three times the number of parameters. To further reduce the dimensionality, and simplify the selection process for L2M later on, another partially connected layer is added which acts as a plane compressor. This layer has M neurons, each with E weights.

$$partial_2 : \mathbb{R}^{(E,M)} \to \mathbb{R}^{M} \tag{6.3}$$

It is important to note that these last 2 layers mentioned, much liked L2M, are not provided by the auto-differentiation framework used and had to be implemented from scratch. A non-trivial process as it requires to extend the framework itself in an efficient manner due to the number of calculations involved in those early layers.

Despite all these consecutive transformations the groups initially created never mix. At this point the

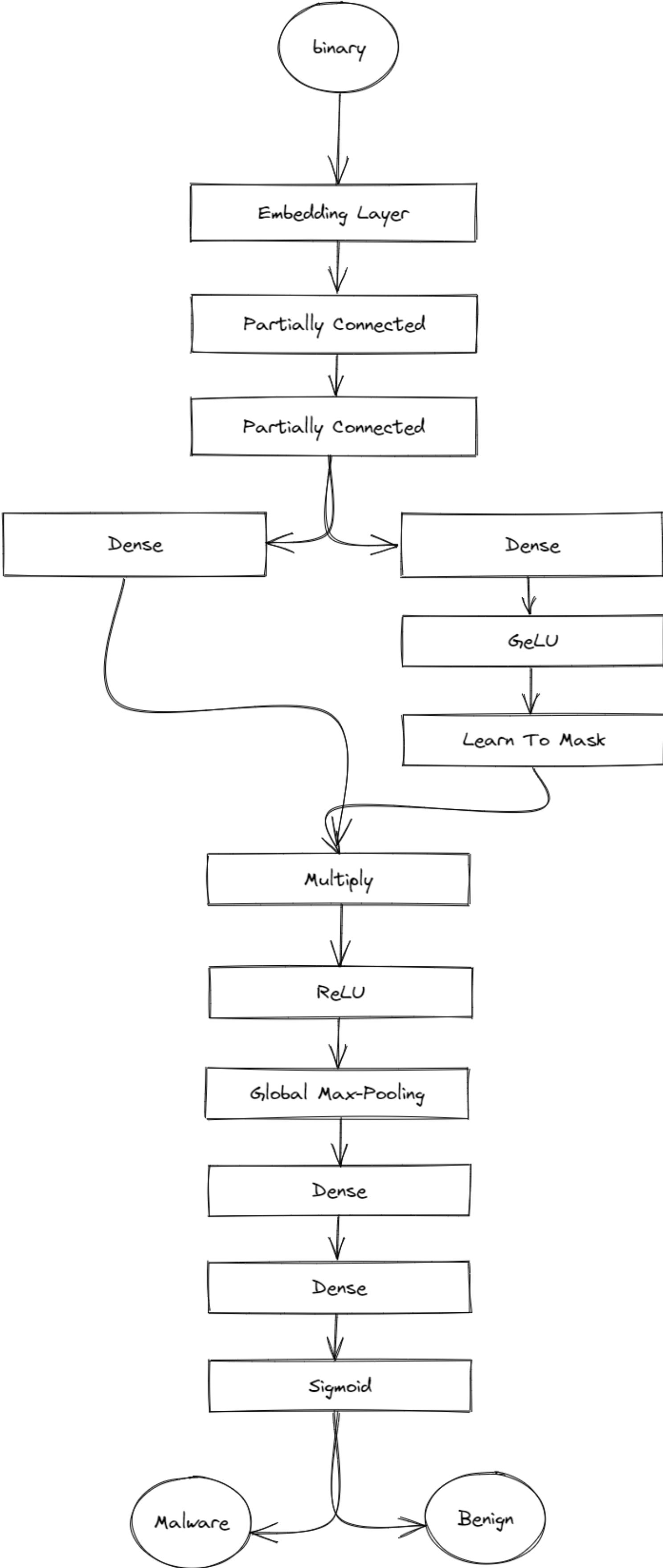


Figure 6.2: Visualization of MalConv's redesigned architecture.

there are M values produced for each inserted binary, a number that is far smaller than the size of the initial input. Therefore convolution is not the right tool anymore. Given how much the input has been compacted by this point it is now safe, memory-wise, to replace the gated convolution with a gated MLP [59] structure. Essentially, the use of a dense layer here is equal to a 1-dimensional convolution operation where the window size is equal to the number of inputs. This substitution is the cure to the convolution window problem. With the use of dense layers, instead of convolutional ones, each group representation has its own weight. This is a direct way of ensuring that the model treats each region differently, as it should. Should this model, during training, learn to produce higher activations for header patterns then, should it see a similar pattern in other regions of the binary, it does not guarantee that it will produce an equally high activation. In fact, since it has probably never been exposed to a similar pattern in that region before it is very likely that it will produce a rather small activation. This, theoretically, should help negate adversarial attacks. The use of gated MLP has also dictated GeLU [39] as the activation function that one of the siblings will utilize. Another benefit of using GeLU here is that its output is always positive, thus satisfying Learn To Mask's requirement of only positive inputs.

For the purposes of adversarial robustness, the gated MLP structure is augmented to also include L2M. L2M is placed directly between GeLU and the multiplication layer. This causes the model to exclude most values prior to the gating, so that any following layers will only make decisions based on what the model has deemed important up to that point. The idea is that L2M, with a small enough selection size, should be able to block out any medium activations that could end up getting augmented by the sibling dense layer, resulting in them being passed on to the classifier. By zero-ing them out at this point, the architecture places really strict limitations on which activations may pass through to the classifier.

The rest of the model remains unchanged. The initial architectural analysis did point out that convolution was not the correct tool for the job, despite its ability to produce great results. Above all benefits that resulted of this transformation, one stands out, explainability. Every single layer now not only has a purpose, but it came to be part of this new architecture through common logic. Besides an explainable architecture, this process has also produced a model that can be adapted to accommodate larger input, something that would require a considerable increase in resources with the original model.

At this point, it should be said that the same architectural changes may not provide any benefits to models that are built around different domains. The takeaway from this chapter is not the architecture itself but rather the thought process of pinpointing architectural issues and providing solutions for them. This thought process is the execution of the defense strategy proposed in the Adversarial Attacks chapter. All that remains is to validate that it can deliver what was promised.

Chapter 7: Experiments & Results

The purpose of this chapter is to describe the experiment environment, including the hardware utilized to run the experiments, the software used to develop the code as well as dataset acquisition. Finally, the results of the experiments are presented, for both the original model and the redesigned one. Separate sub-sections account for their performance with legitimate input and adversarial input. The results are analyzed and explained in both cases.

7.1 Experiments

Latest experiments were performed on a personal computer with:

- Processor: Intel i7-7700
- RAM: 32GB DDR4
- GPU: Nvidia RTX 3090 24GB

While initial experiments made use of a GTX 1070 8GB as the GPU. The memory was sufficient for training the altered model but it appeared to be not enough when training the original MalConv. To anyone looking forward to replicate the experiments a GPU of 11GB would be sufficient for both models. Another issue encountered is the size of the dataset after each executable was padded to the required size of 2MB, initial experiments were attempted with 16GB of RAM but that was problematic. Another hardware difficulty faced while carrying out the experiments that warranted the need for an upgrade was the storage medium. The dataset was originally stored on an HDD, this meant that loading the dataset took roughly 15 minutes per experiment. This was reduced down to 1 minute with the addition of an SSD.

Memory Usage While MalConv is not a deep model by today's standards it has to handle an enormous input space, far larger than is commonly used for demanding tasks such as computer vision. A neural network's memory footprint is determined by 2 factors, the number of weights, and the size of the intermediate tensors produced. Even though MalConv has very few parameters, about 1 million, the real issue is all the tensors produced between layers that also need to be allocated and kept in memory. MalConv's input tensor is 2MB and the one produced by the embedding layer is $2MB \times 4 \times 8 = 64MB$. The 4-fold increase comes from the change of value types, MalConv's inputs are byte values but the embedding layer's output is actually FP32, 8 is simply the size of each embedding vector. In order for gradients to be calculated during training this vector has to remain allocated for the duration that the entire batch is being processed. To make matters worse, one of these tensors is allocated per item in the batch.

All code for the experiments was written in Python 3.6 and the auto-differentiation framework used was TensorFlow 2.4.0 - 2.6.0 [1]. To reduce the development time, Keras [19] was used as a TensorFlow wrapper that really helped streamline the training and testing procedures. Near the end, a service, "Weights and Biases" [8] was introduced to help organize the experiment results. Source code had been uploaded

Attribute	Original MalConv	Redesigned MalConv
Trainable Parameters	1 million	2.2 million
Total Layers	10	13
Epochs	5	3
Batch Size	8	20
Average Training Time	1 minute	3 minutes
Average Validation Time	$<$ 1 minute	1 minute

Table 7.1: Overview of the original & the redesigned MalConv.

to GitHub but remains private, to be made public at a later date, after achieving some of the goals that did not make it into this project. Those goals will be analyzed at the final chapter.

7.1.1 Dataset

The binaries included in the dataset were collected from various sources. Malware executables were provided by VirusShare [30], a malware repository service. As for benign executables, they were sampled from a standard installation of Windows 10 and commonly used, freely distributed, software. It has to be mentioned that this is the same method followed by MalConv's authors to create a dataset. All files present in the dataset have a size smaller or equal to 2MB and belong to the Portable Executable (PE) format.

In total $4,274$ malware and $3,666$ benign executable files were used for training while $1,011$ malware and $1,000$ benign executable files were included in the test set.

Windows Portable Executable Files Windows Portable Executable (PE) files have a specific structure, including a header, a section table, and data. The latter part contains the actual data related to each section, such as:

- *.text*, which contains code instructions,
- *.data*, which contains the initialized global and static variables,
- *.rdata*, which contains constants and additional directories that include meta-information, often used for debugging,
- *.idata*, which contains information about the used imports in the file.

Valid Windows PE files can be identified through the magic-number method. This method relies on the binary's metadata headers to identify a file's type. For a binary to be considered as a Windows PE file then the first 2 bytes must be 0x4D5A, these bytes translate into the ASCII characters *MZ*, the initials of Mark Zbikowski, the person that developed the format.

Manipulating PE files with the goal of preserving their functionality is in general a non-trivial task, as a single byte change would compromise them. There are a couple of method where a malicious payload

Parameter	Redesigned MalConv
Groups	512
Group Size	4096
Selections	12
Gated Dense Neurons	96

Table 7.2: Redesigned MalConv's hyper-parameters.

could be injected, some require more expertise than others. Methods range from injecting malicious payloads in the binary's headers, in the debug section or even in the code section. The last one works by placing the payload between assembly functions which are normally considered dead areas and code execution would never reach that point. The simplest method, and the one used in this work for the sake of comparison with other people's work is adding the payload at the end of the binary, essentially in the dead area after the very last assembly function.

7.2 Results

Procedure for producing a validated model dictates that the dataset is split into 3 non-overlapping segments:

- training data ($\sim$ 80%)
- validation data ($\sim$ 10%)
- test data ($\sim$ 10%)

where the training data are used to tune the model's parameters, the validation data are use to tune the model's hyper-parameters and architecture while the test data provide the final validation of the model's performance. Due to the random initialization of the weights, the numbers presented in the result tables to follow are an average of 10 experiments for each case.

The nature of this work is to test adversarial robustness and this does affect the standard procedure. This is an important part of the experiments and has to be explained well in order for the results to make sense. Training and validation data play the same role, the only difference lies in the test data. The test data in this case are essentially adversarial samples produced from the validation data.

In order to test the model's adversarial robustness, adversarial samples need to be created from the validation data. Not all validation data are candidates to be adversarial samples, there are 2 conditions that have to be met. Any validation sample that belongs to the malware class and can be successfully identified, by the model, as malware is used to create adversarial samples and test the model's robustness.

Finally, table 7.2 shows the hyper-parameters with which the redesigned model was tested and trained.

Much has been said about L2M in the previous chapters, but it is now time to put all those assumptions to the test. Besides the redesigned model that was introduced experiments will also be carried out on a version of the redesigned model that differs in one simple aspect: it does not have the L2M layer. This is done to better observe how it contributes to the performance of the redesigned model.

Accuracy	Original MalConv	Redesigned MalConv	Redesigned (No L2M)
Train Set	0.983	1.000	0.981
Validation Set	0.931	0.946	0.887

Table 7.3: Train & Validation set results table, higher is better.

7.2.1 Testing

Before testing the models against any attacks, it is important to measure their baseline performance. In previous chapters it was mentioned that adversarial robustness often comes at the cost of accuracy. The results prove that this is not the case with our method. Not only did the redesigned version not deteriorate in accuracy but it has, in fact, improved it. Training the redesigned version does take longer, due to the increase in operations from the added layers, and so does inference, even though marginally. But that is nowhere near the price that one would pay when following any of the other defense strategies.

Another thing to note is that the new model is capable of learning the training set to the point where it correctly classifies every single binary. This did not seem possible to do with the original model even if trained for more than 100 epochs. This directly links back to the hypotheses made in the chapter where L2M was introduced, that it should provide many of the benefits that we would expect dropout to provide in a similar application.

By interpreting the results produced by the redesigned model that lacks the L2M layer, it is safe to say that L2M plays an important role to the model's performance. Without it, accuracy drops by as much as 6%, making it even worse than the original MalConv.

Having validated the baseline performance, it is time to examine the difference that the architectural changes caused in terms of adversarial robustness.

7.2.2 Attacking

When explaining the proposed defense strategy in the Adversarial Attacks chapter, it was mentioned that there needs to be some certainty that the changes applied to the model did not create new vulnerabilities. Mapping out the entirety of the attributes that MalConv should possess to be secure is out of the scope of this work, therefore, to showcase this point, the attribute of resistance to noise will be used as an independent attribute that should not have regressed after the model's redesign. To that extend, two separate attacks are performed in each model, both of which where described in the chapter about MalConv.

The first attack is the random byte padding attack. It is known that MalConv is not vulnerable to this attack [93], and appending random bytes as an attack is equivalent to injecting noise. Therefore this attack is used to examine how an independent property of the model was affected by the changes. The size of the payload appended in each malware was $10,000$ bytes. Any malware whose size would exceed the 2MB limit after appending the payload was not used in this attack.

The second attack[50] that was implemented and executed was the Fast Gradient Sign Method. This is the main attack that should be prevented, it is quite more potent than simply appending random bytes as this method focuses on optimizing the payload for maximum misclassification probability. Some

Attack Success Rate	**Original MalConv**	**Redesigned MalConv**	**Redesigned (No L2M)**
Random Byte Padding	0.040	0.000	0.030
Fast Gradient Sign Method	0.359	0.056	0.393

Table 7.4: Attack Success Rate results table, lower is better.

experimentation was required to get the best results for this method as it relies on a hyper-parameter ϵ. This parameter is responsible for tuning the size of the adjustment made to the payload while the sign of the gradient signifies the direction of the adjustment. After experimenting, the results presented were produced with $\epsilon = 0.7$ as it yielded the highest misclassification rate in both models. The amount of bytes that make up the payload is defined as a function of their length L.

$$p(L) = c + (c - L \mod c) \tag{7.1}$$

where c is MalConv's convolution window size.

Out of the $1,011$ malware that are part of the validation set only those correctly classified were used in either attack. Using malware that the model has failed to detect during validation would only produce incorrect attack success rate results and were therefore omitted.

After the attack, MalConv was able to correctly classify only 64% of the malware. In comparison, the redesigned model was barely affected by the attack, being able to classify correctly 94% of the malware. The reason for this vast difference is that the gradients with respect to the malicious payload were observed to be, more often than not, 0, or quickly became 0 after a few iterations, restricting further optimizations from being made. This is directly related to the importance value assigned to each byte group by the end of training, meaning that they were not even selected by L2M to be used in classification.

The redesigned MalConv is definitely more robust against adversarial attacks, without altering the training procedure or sacrificing any performance. Moreover, with the use of the benchmark for random byte padding attacks, we can be sure that the produced model did not regress on a significant property, resistance to noise. On the contrary, the results show that this attribute was in fact strengthened by the changes. This goes a long way towards showing that a model's architecture is directly linked to its adversarial robustness.

One of the claims made early on, was that hard attention, in the form of L2M, is capable of providing significant benefits to evade adversarial attacks. As is shown in the results table above, L2M's role is integral. Without L2M, the attack success rate does show some improvement over the original model but it is still far higher than when L2M is included.

Malware detection models that utilize attention have shown results that are comparable, or sometimes even better than MalConv. Despite their success in malware detection tasks when they are done through manually extracted features, they have failed when faced with the entirety of a binary. The authors of MalConv [76] do mention that the use of attention did not produce a model that performs better than the one presented in the paper. One of the possible reasons that attention was not able to provide similar

Parameter	Redesigned MalConv	GDN64	GDN128
Groups	512	512	512
Group Size	4096	4096	4096
Selections	12	12	12
Gated Dense Neurons	96	64	128

Table 7.5: Gated MLP neurons variation: Parameters of different variations of the redesigned model.

improvements to MalConv was due to the fact that soft-attention would inevitably alter the byte values. Unlike pixels in images, where small alterations are not problematic, bytes in binaries are very specific and even the slightest change could cause significant semantic changes. The fact that L2M has managed to succeed where transformers have failed is a testament that it brings something new to the table.

7.3 Variations of Redesigned MalConv

In order to provide a more complete perspective of the model produced, this section will showcase results of the redesigned model but with different hyper-parameters.

To simplify the observations made, all variations presented differ, from the once discussed above, in one, and only one, parameter.

The variations examined below differ from the original architecture in:

- the number of neurons present in the Gated MLP dense layers,
- the number of selections made by L2M,
- and the group size used to extract features.

7.3.1 Variation: Gated MLP neurons

This variation examines the model's behavior with respect to changes in the number of neurons placed in each gated MLP dense layer. *GDN64* utilizes 64 neurons in each of the two dense layers present in the gated MLP structure instead of 96 neurons. The case of increasing the number of neurons is represented by *GDN256* which has 128 neurons.

As it turns out, increasing the number of neurons to 128 per dense layer shows a slight decrease in validation accuracy though the results are, still, better than those produced by the original model. While not an overfit model on its own, it is when compared to the baseline model.

The use of 64 neurons seems to be a much better decision that 128 when considering the accuracy of the resulting model. Still though, this confirms that 96 is the absolute best choice, so far, for this hyper-parameter.

GDN128 is definitely a worthy model, its attack prevention rate is marginally higher than the redesigned model presented earlier. The only reason that this was not chosen to as the final redesign was due to their

Accuracy	Original MalConv	Redesigned MalConv	GDN64	GDN128
Train Set	0.983	1.000	1.000	1.000
Validation Set	0.931	0.946	0.942	0.937

Table 7.6: Gated MLP neurons variation: Train & Validation set results, higher is better.

Attack Success Rate	Original MalConv	Redesigned MalConv	GDN64	GDN128
Random Byte Padding	0.040	0.000	0.000	0.000
Fast Gradient Sign Method	0.359	0.056	0.076	0.052

Table 7.7: Gated MLP neurons variation: Attack Success Rate results table, lower is better.

difference in accuracy. 0.009 higher accuracy was deemed to be a much better quality, for the model, than 0.004 attack evasion rate.

Unfortunately, the *GDN64*'s high accuracy comes at the cost of adversarial attack evasion. It is hard to judge which of the 2 models is better, what can be drawn from this is that narrower models might be easier to fool, while wider models may be prone to issues due to their own complexity.

7.4 Variation: Number of selections

This variation examines the model's behavior with respect to changes in the number of selections made by L2M. *SELECT6* selects the 6 most important values instead of 12 which is presented as the standard. The case of increasing the number of selections is represented by *SELECT24* which makes 24 selections.

Even with just 6 out of 128 values being selected *SELECT6* manages to outperform MalConv in basic malware detection. The loss of accuracy, though, when compared to the redesigned MalConv, is noticeable. Similar results can be observed when using a much higher number of selections, as *SELECT24* does. This increase in selections does not show any performance improvements. Only a small part of any input are useful to a model, this result can be attributed to the fact that a higher number of selections has a much higher chance of introducing less useful data to the classifier, therefore inhibiting the learning procedure. Neural networks are really about the right amount of information, no more, no less.

SELECT6 would be expected to perform better during attacks. The results do show the opposite and there are ways to explain this. It could be that the function learned from such a restrictive number of selections is not as smooth as the baseline model, resulting in poorer adversarial attack evasion rates. Essentially, given that most of the neurons are not trained due to their output being discarded, it might just create a small attack vector due to the largely untrained weights present in the model after training is complete. It could be possible that this particular variation would benefit greatly from slower training, so there is

Parameter	Redesigned MalConv	SELECT6	SELECT24
Groups	512	512	512
Group Size	4096	4096	4096
Selections	12	6	24
Gated Dense Neurons	96	96	96

Table 7.8: Number of selections variation: Parameters of different variations of the redesigned model.

Accuracy	Original MalConv	Redesigned MalConv	SELECT6	SELECT24
Train Set	0.983	1.000	1.000	1.000
Validation Set	0.931	0.946	0.939	0.935

Table 7.9: Number of selections variation: Train & Validation set results table, higher is better.

Attack Success Rate	Original MalConv	Redesigned MalConv	SELECT6	SELECT24
Random Byte Padding	0.040	0.000	0.000	0.000
Fast Gradient Sign Method	0.359	0.056	0.069	0.052

Table 7.10: Number of selections variation: Attack Success Rate results table, lower is better.

more time for all the possible sub-architectures to be explored.

Increasing the number of selections to 24 does bode well for the model's defenses, as it has managed to surpass the baseline model's attack evasion rate. With more values being selected it is easier, during such short training, to test the efficacy of the sub-architectures created through the sparsity of L2M's output. Even if more values are selected than were actually required, they will still converge to a rather low number, making them contribute far less to the output of the model. It is important to keep in mind that a higher number of selections also imply a higher number of co-dependencies being created between the neurons, this was explained in more detail when comparing L2M with dropout.

7.5 Variation: Group size

This variation examines the model's behavior with respect to changes in the group size of the embedded values. *GS2048* reduces the group size, from 4096 to 2048 while *GS8192* increases the group size to 8192.

Using a smaller group size has affected the model's accuracy negatively. This is mostly due to the increased number of connections that this change creates in the neurons of the dense layers present in the gated MLP structure. Utilizing a group size of 2048 implies that 1024 groups are created instead of 512. All those values are connected to every neuron in the above mentioned dense layer, therefore, more information loss occurs in this layer, resulting in a drop in accuracy.

Increasing the group size produces similar results to reducing it. The same information loss effects, as produced when reducing the group size, occur here, since every neuron in the first partially connected layer has a higher information loss ratio, since double the amount of values are compressed into a single feature.

Parameter	Redesigned MalConv	GS2048	GS8192
Groups	512	1024	256
Group Size	4096	2048	8192
Selections	12	12	12
Gated Dense Neurons	96	96	96

Table 7.11: Group size variation: Parameters of different variations of the redesigned model.

Accuracy	Original MalConv	Redesigned MalConv	GS2048	GS8192
Train Set	0.983	1.000	1.000	1.000
Validation Set	0.931	0.946	0.938	0.937

Table 7.12: Group size variation: Train & Validation set results table, higher is better.

Attack Success Rate	Original MalConv	Redesigned MalConv	GS2048	GS8192
Random Byte Padding	0.040	0.000	0.000	0.002
Fast Gradient Sign Method	0.359	0.056	0.075	0.076

Table 7.13: Group size variation: Attack Success Rate results table, lower is better.

The decrease in group size is also accompanied by a decrease in adversarial attack evasion rates. This result directly links to the information loss produced in early layers of the model, not allowing it to learn a smooth function and generalize properly.

GS8192 shows that an increase in group size is the wrong direction to move towards as now the model is not completely impervious to noise. The difference from a complete 0 to even just a value that is so close to 0 as 0.002 is still significant. Especially since all other architectures presented in this section do not exhibit similar symptoms.

Both of those models did not manage to outperform the redesigned version of MalConv. This could be a great sign that a link exists between information loss and the existence of an attack vector that can be exploited through the FGSM attack. In that regard, it makes sense that a group size of 4096 would work best, as it minimizes information loss in all layers of the model.

7.5.1 Overview

A few general conclusions can be drawn through observing the different variations. They all share the same characteristics, the same attributes. All of them perform better in training than MalConv does, something that can be said with great conviction since the results from all the different variations confirm those findings.

All of the above model are completely impervious to noise, except one, and their respective attack success rates far surpass those observed in MalConv. Exploring through the variations was very interesting, the intuition this process provided with respect to how L2M does affect the layers prior to it is remarkable.

It would make sense to attempt changes in all possible combinations, often referred to as a grid search, in hope of stumbling upon a slightly superior architecture but that was not the point of this. The point is to better understand the model and how tuning its hyper-parameters can affect it in small, but very insightful, ways.

Chapter 8: Conclusions

After seeing the results, and having confirmed all initial hypotheses that were made, it is safe to say that L2M is indeed something special. It manages to do a lot of things right, especially when compared with other hard attention mechanisms. Not only has it helped boost adversarial robustness but it has also improved the accuracy for legitimate binaries. It is new and exciting, definitely worthy of looking into and how it can help shaped well known models.

Analyzing MalConv's architecture and making educated assumptions about the reasons behind its vulnerability to the FGSM attack has also showed to yield results. It validated that just because the results of the initial model were correct, it does not mean that they were also produced in the correct way. The resulting model is much more scalable, both from a memory efficiency perspective and an input space perspective. It has also become aware with respect to the location of its input and can definitely see through a couple of standard attacks.

8.1 Challenges

Designing an algorithm that is completely new was not something that could be done overnight. Learn To Select had to go through a lot of changes and tests in order for Learn To Mask to emerge. A lot of brainstorming sessions, the employment of a large piece of melamine to act as a whiteboard, and long evening walks were required in order to reach the final design.

As it happens with all lengthy projects, a lot of personal issues came in the way of completion. Personal projects and a full-time job definitely slowed down progress, making the entire thesis last over 2 years.

Hardware challenges were mentioned in the previous chapter so here I would like to talk about how they were made even worse by the dataset. Any form of free online computing became unavailable to me since day 1. The reason is that all of them scan their cloud storage for malware. It would suffice to say that they were not very pleased when they would detect that a couple thousand had been uploaded to their system. This led to some brief mail exchanges in which providers, Google among them, would threaten with account termination if those malware were not deleted.

This is directly linked to the fact that the dataset used in this work will not be distributed, due to possible legal implications this might carry in case someone actually executes them.

8.2 Future work

As one would expect, when introducing so many new concepts it is impossible to explore them in their entirety. Both L2M and the adversarial attack evasion strategy introduced here have a long way to go before we are able to chart the full extend of what they offer.

8.2.1 Unexplored areas of Learn to Mask

Despite its good results, this was a single test case and by no means enough to be able to say, with conviction that L2M will be as beneficial in other cases.

Result validation through multiple test cases As of now, I am currently working on showing that is provides similar benefits in critical computer vision tasks, such as human iris recognition. So far I can only say that it did not require any changes in order to be used with images and that similar benefits to dropout were observed there as well but it is too early to say anything more than that.

Learn to Mask specific attack There was also work done towards an L2M specific attack. The way it would work is by optimizing the payload against two goals, maximizing the misclassification probability, in the same way that FGSM does, and maximizing the selection probability. While such an attack was designed and implemented for experimental versions of MalConv, it was deemed far too complex for this one and was left out. The, at the time, literature review had showed that no other multi-goal attack has been executed though this might not be true anymore. It is still an avenue that I would be very interested in pursuing since such an attack would be directly applicable to other attention mechanisms. For example, in the case of soft attention, the goal would be to maximize the importance of the values that belong to the payload. Such an attack could also pave the way for attacks that exploit multiple neural network vulnerabilities, simultaneously, for devastating results.

Variable selection It was mentioned earlier that L2M can vary its number of selections between training and inference. This is something that could help squeeze out some extra performance from the model but more experiments are required to validate this assumption. The way this change affects the model's adversarial robustness should also be thoroughly examined. Ideally the number of selections would be a trainable parameter of the model but I am not aware of the difficulties that may arise in the process. It is still worth exploring for the sake of completeness.

Pruning It is quite common that, during training, many of the weights converge to 0 or effectively 0. These weights do not make any meaningful contributions to the model's output and are just a hindrance. An area worth exploring is the levels at which the weights of layers prior to Learn To Mask converge to. Taking the case of MalConv where some areas are vastly more important than others, such as the binary headers, it could be that some synapses are completely unimportant and could thus be removed allowing for more efficient execution.

8.2.2 Unexplored areas of Architectural Defense strategies

While the basis of the method was introduced and applied here, I believe it has great potential. This work has only scratched the surface with regards to what attributes a neural network could or should posses. Formalizing this theory and expanding on it is a daunting task but one I find extremely interesting both

due to the challenge it presents and the value it could have for the field of deep learning from both a scientific and a commercial standpoint.

The scientific benefits are self-explanatory but the commercial ones are equally interesting. There are many cases in which deep learning models are called upon to make decisions that impact our lives in a very direct way, such as autonomous driving. Since neural networks are shamelessly described as a black box that "just works" it raises many concerns with regards to safety. Explainable models directly translate not only to more reliable models, but also models that people can trust. It is perfectly valid to show distrust towards something that has so much power and yet its inner workings are somewhat of a mystery. Such research would help alleviate people's concerns about artificial intelligence and build trust through transparency.

8.2.3 Malware Detection as a Service

Another project that I am currently pursuing is making the final model publicly available in the form of a service. This service will allow users to submit PE files to the model in order to produce a maliciousness score along with other information, such as file metadata, that could help the user decide for themselves if the file is dangerous or not.

As for testing the robustness of the model, an open challenge will be presented with some form of reward for anyone that can create a reproducible method of fooling the detector. This is both an incentive for people to take an interest in this work and to discover any present vulnerabilities. In turn this will lead to fixing the vulnerability, resulting in a new model and a tougher challenge.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. Malware dynamic analysis evasion techniques: A survey. *ACM Comput. Surv.*, 52(6), nov 2019.

[3] R. Agrawal, J. W. Stokes, K. Selvaraj, and M. Marinescu. Attention in recurrent neural networks for ransomware detection. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3222–3226, 2019.

[4] Roee Aharoni and Yoav Goldberg. Morphological inflection generation with hard monotonic attention, 2016.

[5] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

[6] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. Practical black-box attacks on deep neural networks using efficient query mechanisms. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[7] Taniya Bhatia and Rishabh Kaushal. Malware detection in android based on dynamic analysis. In *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, pages 1–6, 2017.

[8] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[9] Sebastian Bock and Martin Weiß. A proof of local convergence for the adam optimizer. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2019.

[10] Rickard Brüel Gabrielsson. Universal function approximation on graphs. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 19762–19772. Curran Associates, Inc., 2020.

[11] Bram C.M. Cappers, Paulus N. Meessen, Sandro Etalle, and Jarke J. van Wijk. Eventpad: Rapid malware analysis and reverse engineering using visual analytics. In *2018 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–8, 2018.

[12] Bingcai Chen, Zhongru Ren, Chao Yu, Iftikhar Hussain, and Jintao Liu. Adversarial examples for cnn-based malware detectors. *IEEE Access*, 7:54360–54371, 2019.

[13] Dian Chen, Vladlen Koltun, and Philipp Krähenbühl. Learning to drive from a world on rails. *CoRR*, abs/2105.00636, 2021.

[14] Jun Chen, Shize Guo, Xin Ma, Haiying Li, Jinhong Guo, Ming Chen, and Zhisong Pan. Slam: A malware detection method based on sliding local attention mechanism. *Security and Communication Networks*, 2020:1–11, 09 2020.

[15] Jun Chen, Shize Guo, Xin Ma, Haiying Li, Jinhong Guo, Ming Chen, and Zhisong Pan. Slam: A malware detection method based on sliding local attention mechanism. *Security and Communication Networks*, 2020:6724513, Sep 2020.

[16] Sizhe Chen, Zhengbao He, Chengjin Sun, Jie Yang, and Xiaolin Huang. Universal adversarial attack on attention and the resulting dataset damagenet. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2020.

[17] Shuyu Cheng, Yinpeng Dong, Tianyu Pang, Hang Su, and Jun Zhu. Improving black-box adversarial attacks with a transfer-based prior. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[18] Sunoh Choi, Jangseong Bae, Changki Lee, Youngsoo Kim, and Jonghyun Kim. Attention-based automated feature extraction for malware analysis. *Sensors*, 20(10), 2020.

[19] François Chollet et al. Keras. `https://keras.io`, 2015.

[20] Matthew Crawford, Wei Wang, Ruoxi Sun, and Minhui Xue. Statically detecting adversarial malware through randomised chaining. In *Australasian Computer Science Week 2022*, ACSW 2022, page 91–95, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, 2014.

[22] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, Feb 2017.

[23] Dipankar Dasgupta and Kishor Datta Gupta. Dual-filtering (df) schemes for learning systems to prevent adversarial attacks. *Complex & Intelligent Systems*, Jan 2022.

[24] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 933–941. PMLR, 06–11 Aug 2017.

[25] Baptiste David, Eric Filiol, and Kévin Gallienne. Structural analysis of binary executable headers for malware detection optimization. *Journal of Computer Virology and Hacking Techniques*, 13(2):87–93, May 2017.

[26] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv preprint arXiv:1901.03583*, 2019.

[27] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks, 2018.

[28] Konstantinos Diamantaras, Zhaoyi Xu, and Athina Petropulu. Sparse antenna array design for mimo radar using softmax selection, 2021.

[29] Gamaleldin F. Elsayed, Simon Kornblith, and Quoc V. Le. *Saccader: Improving Accuracy of Hard Attention Models for Vision*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[30] External Data Source. Virusshare dataset, 2018.

[31] Kreuk Felix, Barak Assi, Aviv-Reuven Shir, Baruch Moran, y Pinkas Benn, and Keshet Joseph. Adversarial examples on discrete sequences for beating whole-binary malware detection. *arXiv preprint arXiv:1802.04528*, 2018.

[32] Hossein Fereidooni, Mauro Conti, Danfeng Yao, and Alessandro Sperduti. Anastasia: Android malware detection using static analysis of applications. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2016.

[33] William Fleshman, Edward Raff, Jared Sylvester, Steven Forsyth, and Mark McLean. Non-negative networks against adversarial attacks, 2018.

[34] Hongyang Gao and Shuiwang Ji. Graph representation learning via hard and channel-wise attention networks, 2019.

[35] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[36] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015.

[37] Heena and B. M. Mehtre. Advances in malware detection- an overview, 2021.

[38] Yiming Hei, Renyu Yang, Hao Peng, Lihong Wang, Xiaolin Xu, Jianwei Liu, Hong Liu, Jie Xu, and Lichao Sun. Hawk: Rapid android malware detection through heterogeneous graph attention networks. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–15, 2021.

[39] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.

[40] Alex Hernández-García and Peter König. Data augmentation instead of explicit regularization, 2018.

[41] Shifu Hou, Yujie Fan, Yiming Zhang, Yanfang Ye, Jingwei Lei, Wenqiang Wan, Jiabin Wang, Qi Xiong, and Fudong Shao. <i>αcyber</i>: Enhancing robustness of android malware detection system against adversarial attacks on heterogeneous graph based model. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, page 609–618, New York, NY, USA, 2019. Association for Computing Machinery.

[42] Sachin Jain and Yogesh Kumar Meena. Byte level n–gram analysis for malware detection. In K. R. Venugopal and L. M. Patnaik, editors, *Computer Networks and Intelligent Computing*, pages 51–59, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[43] Jueun Jeon, Jong Hyuk Park, and Young-Sik Jeong. Dynamic analysis for iot malware detection with convolution neural network model. *IEEE Access*, 8:96899–96911, 2020.

[44] Yuede Ji, Benjamin Bowman, and H. Howie Huang. Securing malware cognitive systems against adversarial attacks. In *2019 IEEE International Conference on Cognitive Computing (ICCC)*, pages 1–9, 2019.

[45] Linxi Jiang, Xingjun Ma, Shaoxiang Chen, James Bailey, and Yu-Gang Jiang. Black-box adversarial attacks on video recognition models. In *Proceedings of the 27th ACM International Conference on Multimedia*, MM '19, page 864–872, New York, NY, USA, 2019. Association for Computing Machinery.

[46] M. I. Jordan and T. M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[47] BooJoong Kang, Suleiman Y. Yerima, Sakir Sezer, and Kieran McLaughlin. N-gram opcode analysis for android malware detection. 2016.

[48] Patrick Kidger and Terry Lyons. Universal approximation with deep narrow networks. In Jacob Abernethy and Shivani Agarwal, editors, *Proceedings of Thirty Third Conference on Learning Theory*, volume 125 of *Proceedings of Machine Learning Research*, pages 2306–2327. PMLR, 09–12 Jul 2020.

[49] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[50] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples, 2019.

[51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[52] Hyun Kwon and Jun Lee. Diversity adversarial training against adversarial attack on deep neural networks. *Symmetry*, 13(3), 2021.

[53] Alex Lamb, Vikas Verma, Juho Kannala, and Yoshua Bengio. Interpolated adversarial training: Achieving robust neural networks without sacrificing too much accuracy. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, AISec'19, page 95–103, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Sarah Leclerc, Erik Smistad, Thomas Grenier, Carole Lartizien, Andreas Ostvik, Frederic Cervenansky, Florian Espinosa, Torvald Espeland, Erik Andreas Rye Berg, Pierre-Marc Jodoin, Lasse Lovstakken, and Olivier Bernard. Ru-net: A refining segmentation network for 2d echocardiography. In *2019 IEEE International Ultrasonics Symposium (IUS)*, pages 1160–1163, 2019.

[55] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[56] Ruey-Hsun Liang and Yuan-Yih Hsu. A hybrid artificial neural network—differential dynamic programming approach for short-term hydro scheduling. *Electric Power Systems Research*, 33(2):77–86, 1995.

[57] Fangzhou Liao, Ming Liang, Yinpeng Dong, Tianyu Pang, Xiaolin Hu, and Jun Zhu. Defense against adversarial attacks using high-level representation guided denoiser. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[58] Xiang Ling, Lingfei Wu, Jiangyu Zhang, Zhenqing Qu, Wei Deng, Xiang Chen, Chunming Wu, Shouling Ji, Tianyue Luo, Jingzheng Wu, and Yanjun Wu. Adversarial attacks against windows pe malware detection: A survey of the state-of-the-art, 2021.

[59] Hanxiao Liu, Zihang Dai, David R. So, and Quoc V. Le. Pay attention to mlps, 2021.

[60] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[61] X. Ma, S. Guo, H. Li, Z. Pan, J. Qiu, Y. Ding, and F. Chen. How to make attention mechanisms more practical in malware classification. *IEEE Access*, 7:155270–155280, 2019.

[62] Yuxin Ma, Tiankai Xie, Jundong Li, and Ross Maciejewski. Explaining vulnerabilities to adversarial machine learning through visual analytics. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1075–1085, jan 2020.

[63] Mateusz Malinowski, Carl Doersch, Adam Santoro, and Peter W. Battaglia. Learning visual question answering by bootstrapping hard attention. *CoRR*, abs/1808.00300, 2018.

[64] Fabio Martinelli, Fiammetta Marulli, and Francesco Mercaldo. Evaluating convolutional neural network for effective mobile malware detection. *Procedia Computer Science*, 112:2372–2381, 2017. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 21st International Conference, KES-20176-8 September 2017, Marseille, France.

[65] Nuno Martins, José Magalhães Cruz, Tiago Cruz, and Pedro Henriques Abreu. Adversarial machine learning applied to intrusion and malware scenarios: A systematic review. *IEEE Access*, 8:35403–35419, 2020.

[66] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, and Gail Joon Ahn. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, page 301–308, New York, NY, USA, 2017. Association for Computing Machinery.

[67] Mostafa Mehdipour Ghazi and Hazim Kemal Ekenel. A comprehensive analysis of deep learning based representation for face recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2016.

[68] Seyed-Mohsen Moosavi-Dezfooli, Ashish Shrivastava, and Oncel Tuzel. Divide, denoise, and defend against adversarial attacks, 2018.

[69] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, 2007.

[70] Smita Naval, Vijay Laxmi, Muttukrishnan Rajarajan, Manoj Singh Gaur, and Mauro Conti. Employing program semantics for malware detection. *IEEE Transactions on Information Forensics and Security*, 10(12):2591–2604, 2015.

[71] Aditya V. Nori, Sriram K. Rajamani, SaiDeep Tetali, and Aditya V. Thakur. The yogi project: Software property checking via static analysis and testing. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 178–181, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[72] Ya Pan, Xiuting Ge, Chunrong Fang, and Yong Fan. A systematic literature review of android malware detection using static analysis. *IEEE Access*, 8:116363–116379, 2020.

[73] Anson Pinhero, Anupama M L, Vinod P, C.A. Visaggio, Aneesh N, Abhijith S, and AnanthaKrishnan S. Malware detection employed by visualization and deep neural network. *Computers & Security*, 105:102247, 2021.

[74] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143–195, 1999.

[75] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. A survey of android malware detection with deep neural models. *ACM Comput. Surv.*, 53(6), dec 2020.

[76] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware detection by eating a whole exe, 2017.

[77] Edward Raff, Jared Sylvester, and Charles Nicholas. *Learning the PE Header, Malware Detection with Minimal Domain Knowledge*, page 121–132. Association for Computing Machinery, New York, NY, USA, 2017.

[78] Hemant Rathore, Sanjay K. Sahay, Piyush Nikam, and Mohit Sewak. Robust android malware detection system against adversarial attacks using q-learning. *Information Systems Frontiers*, 23(4):867–882, Aug 2021.

[79] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[80] Binxin Ru, Adam Cobb, Arno Blaas, and Yarin Gal. Bayesopt adversarial attack. In *International Conference on Learning Representations*, 2020.

[81] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[82] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak. Static analysis of executables for collaborative malware detection on android. In *2009 IEEE International Conference on Communications*, pages 1–5, 2009.

[83] Paul Hongsuck Seo, Zhe Lin, Scott Cohen, Xiaohui Shen, and Bohyung Han. Progressive attention networks for visual attribute prediction, 2016.

[84] Maryam Shahpasand, Len Hamey, Dinusha Vatsalan, and Minhui Xue. Adversarial attacks on mobile malware detection. In *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*, pages 17–20, 2019.

[85] Vasu Sharma, Ankita Kalra, Vaibhav, Simral Chaudhary, Labhesh Patel, and LP Morency. Attend and attack : Attention guided adversarial attacks on visual question answering models. 2018.

[86] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep learning in medical image analysis. *Annual Review of Biomedical Engineering*, 19(1):221–248, 2017. PMID: 28301734.

[87] P.V. Shijo and A. Salim. Integrated static and dynamic analysis for malware detection. *Procedia Computer Science*, 46:804–811, 2015. Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace & Island Resort, Kochi, India.

[88] Yang Song, Taesup Kim, Sebastian Nowozin, Stefano Ermon, and Nate Kushman. Pixeldefend: Leveraging generative models to understand and defend against adversarial examples, 2017.

[89] Sho Sonoda and Noboru Murata. Neural network with unbounded activation functions is universal approximator. *Applied and Computational Harmonic Analysis*, 43(2):233–268, 2017.

[90] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[91] Jack W. Stokes, De Wang, Mady Marinescu, Marc Marino, and Brian Bussone. Attack and defense of dynamic analysis-based, adversarial neural malware detection models. In *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, pages 1–8, 2018.

[92] Christopher Stover and Eric W Weisstein. Einstein summation., 2014. Accessed: 2022-01-24.

[93] Octavian Suciu, Scott E. Coull, and Jeffrey Johns. Exploring adversarial examples in malware detection. *CoRR*, abs/1810.08280, 2018.

[94] Krishna Sugunan, T. Gireesh Kumar, and K. A. Dhanya. Static and dynamic analysis for android malware detection. In Elijah Blessing Rajsingh, Jey Veerasamy, Amir H. Alavi, and J. Dinesh Peter, editors, *Advances in Big Data and Cloud Computing*, pages 147–155, Singapore, 2018. Springer Singapore.

[95] Rajkumar Theagarajan, Ming Chen, Bir Bhanu, and Jing Zhang. Shieldnets: Defending against adversarial attacks using probabilistic adversarial robustness. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[96] Shun Tobiyama, Yukiko Yamaguchi, Hajime Shimada, Tomonori Ikuse, and Takeshi Yagi. Malware detection with deep neural network using process behavior. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 577–582, 2016.

[97] Ääron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.

[98] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[99] Lele Wang, Binqiang Wang, Peipei Zhao, Ruyi Liu, Jiangang Liu, and Qiguang Miao. Malware detection algorithm based on the attention mechanism and resnet. *Chinese Journal of Electronics*, 29:1054–1060(6), November 2020.

[100] Wei Wang, Mengxue Zhao, and Jigang Wang. Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. *Journal of Ambient Intelligence and Humanized Computing*, 10(8):3035–3043, Aug 2019.

[101] Xiaosen Wang and Kun He. Enhancing the transferability of adversarial attacks through variance tuning, 2021.

[102] Yulin Wang, Gao Huang, Shiji Song, Xuran Pan, Yitong Xia, and Cheng Wu. Regularizing deep networks with semantic data augmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(7):3733–3748, 2022.

[103] Zhibo Wang, Hengchang Guo, Zhifei Zhang, Wenxin Liu, Zhan Qin, and Kui Ren. Feature importance-aware transferable adversarial attacks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 7639–7648, October 2021.

[104] Xingxing Wei, Siyuan Liang, Ning Chen, and Xiaochun Cao. Transferable adversarial attacks for image and video object detection, 2018.

[105] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992.

[106] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69, 2012.

[107] Liang Xie, Xinwen Zhang, Jean-Pierre Seifert, and Sencun Zhu. Pbmds: A behavior-based malware detection system for cellphone devices. In *Proceedings of the Third ACM Conference on Wireless Network Security*, WiSec '10, page 37–48, New York, NY, USA, 2010. Association for Computing Machinery.

[108] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2048–2057, Lille, France, 07–09 Jul 2015. PMLR.

[109] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[110] Wei Xue and Tao Li. Aspect based sentiment analysis with gated convolutional networks, 2018.

[111] Hiromu Yakura, Shinnosuke Shinozaki, Reon Nishimura, Yoshihiro Oyama, and Jun Sakuma. Malware analysis of imaged binary samples by convolutional neural network with attention mechanism. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, page 127–134, New York, NY, USA, 2018. Association for Computing Machinery.

[112] Hiromu Yakura, Shinnosuke Shinozaki, Reon Nishimura, Yoshihiro Oyama, and Jun Sakuma. Neural malware analysis with attention mechanism. *Computers & Security*, 87:101592, 2019.

[113] Akihiko Yamaguchi and Christopher G. Atkeson. Neural networks and differential dynamic programming for reinforcement learning problems. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5434–5441, 2016.

[114] Shiyang Yan, Jeremy S. Smith, Wenjin Lu, and Bailing Zhang. Hierarchical multi-scale attention networks for action recognition, 2017.

[115] Chuanguang Yang, Zhulin An, Hui Zhu, Xiaolong Hu, Kun Zhang, Kaiqiang Xu, Chao Li, and Yongjun Xu. Gated convolutional networks with hybrid connectivity for image classification. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(07):12581–12588, Apr. 2020.

[116] Wei Zhang, Kazuyoshi Itoh, Jun Tanida, and Yoshiki Ichioka. Parallel distributed processing model with local space-invariant interconnections and its optical architecture. *Appl. Opt.*, 29(32):4790–4797, Nov 1990.

[117] Yi Zhang, Yuexiang Yang, and Xiaolei Wang. A novel android malware detection approach based on convolutional neural network. In *Proceedings of the 2nd International Conference on Cryptography, Security and Privacy*, ICCSP 2018, page 144–149, New York, NY, USA, 2018. Association for Computing Machinery.

[118] Yunchun Zhang, Haorui Li, Yang Zheng, Shaowen Yao, and Jiaqi Jiang. Enhanced dnns for malware classification with gan-based adversarial training. *Journal of Computer Virology and Hacking Techniques*, 17(2):153–163, Jun 2021.

[119] Zhongheng Zhang. Introduction to machine learning: k-nearest neighbors. *Annals of translational medicine*, 4(11):218–218, Jun 2016. 27386492[pmid], PMC4916348[pmcid], atm-04-11-218[PII].