Classification of Malware Using Deep Learning

Berkeley D. Willis

University of Bellevue

Abstract

Malware has been a thorn in the side of people for decades, from the average person to the largest of multinational corporations they are attacked with malware. The malware used has gotten more complex in the ways that it is hiding in plain sight, and the magnitude of these attacks have increased to millions of attacks a year. The types of malware that are being used are quite vast as well, because they have different goals and means to attain those goals. With so much variation in malware, it would be helpful to be able to easily categorize them based on how they function. Understanding what separates and what makes malware a certain class using it's raw features will give insights on how to identify malware in the wild. Using many features that were able to be attained by simple static analysis of binary Windows malware, allowed the creation of a Deep Learning model that is able to classify malware correctly at a rate of 94% and giving insights on how these classes differ.

Keywords: Deep Learning, Malware, Classification, Dense Neural Networks

Classification of Malware Using Deep Learning

Introduction & Background

Malware is a type of software that is created for malicious purposes that has been used for decades. Malware, also known as malicious code, refers to a program that is covertly inserted into another program with the intent to destroy data, run destructive or intrusive programs, or otherwise compromise the confidentiality, integrity, or availability of the victim's data, applications, or operating system (Souppaya and Scarfone). These programs have different goals such as giving access to an unauthorized user, spying on the computer's user, or just to cause general destruction. The malware many times is built differently depending on these goals, and today have even more possible representations since they are no longer limited to only binary malware. Today there are examples of not only binary malware, but several that are written in interpretative languages like Python, JavaScript, or even VisualBasic. This leads to even more variations in the exploitative and malicious software that is being built. Classification of this type of malicious software is important, because it gives insight to the goals of the software, how it was built, and can give more ways to identify potentially malicious software in the wild.

Malware in the wild, brand new or old, are being used at a large scale across the world. According to one report on cyber attacks in 2020, they recorded 9.9 billion different malware attacks and was actually a decrease from the year before (SonicWall). This likely isn't very targeted malware attacks and mostly are being used in phishing attacks, but many people and companies fall victim to these attacks nonetheless. This scale of attacks would take an inordinate amount of time to classify these by hand, even if many of them were the same exact malware being used. Thus it makes sense to try to find a better way to try to classify and do basic analysis on malware extractions.

To classify anything there needs to be some criteria that separates different types of malware. There isn't a fully defined framework to differentiate types of malware, but there are some basic colloquial ways to determine and label types of malware mostly relying on the goals of the malware itself. According to one cybersecurity firm in Washington D. C. most malware can fit into 9 categories: viruses, keyloggers, worms, trojans, ransomware, logic bombs, bots/botnets, adware/spyware, and rootkits (Firch). This isn't going to be able to catch or define everything so far, but it is a start and gives a good idea of how to classify malware based on some high level observations by humans.

- **Viruses:** A very common type of malware attack, infecting computers and attempting to spread themselves via any means they can and are programmed to. Almost like a catch all of many types.
- **Keyloggers:** A type of malware that captures a user's keystrokes or other inputs without the user's knowledge.
- **Worms:** Similar to viruses, a worm self-replicates and spreads copies of itself via network connections but typically doesn't need a host program to run.
- **Trojans:** Malware that is disguised as legitimate or helpful software, encouraging a user to install it for the helpful features. However, when installed it will spy on the user, steal their sensitive data, or even act as a back door for the bad actor.
- **Ransomware:** Malware that is designed to encrypt all files it can access and deny the victim access to their files until a ransom is paid.

- **Logic Bombs:** Malware that usually hides itself in viruses and worms, but are designed when triggered will cause damage to the infected computer.
- **Bots/Botnets:** Malware designed to infect and quietly participate in a larger network of other infected computers.
- **Adware/Spyware:** Types of malware that either by sending unwanted advertisements or even to possibly capture sensitive information that a user submits through web browsers.
- **Rootkits:** Malware that is designed to act as a backdoor program to infect computers to some bad actor, and many times give privileged access to the infected device.

Though it isn’t all the possible classes of malware, it is able to describe most types of malware that are commonly seen and will be seen. It would be interesting and very helpful if an authoritative source of information would be able to create a more stable and stronger typed malware definitions and descriptions.

To be able to build a classifier, a framework to understand these types of malware is being used to understand the software at a high level in order to identify features that can help. This framework is basically identifying raw attributes of Windows binary malware, meaning that this is fully compiled malware for the Windows environment and isn’t one of the previously mentioned interpreted languages. In one study on the classification of malware using dynamic and static analysis, it was using some of these low level attributes from malware because even with all the various obfuscation and anti-debugging techniques that they might use that the malware still has to use the hardware (Banin and Dyrkolbotn). Thus this similar type of lower level data can be collected from binary malware to build a different classifier, and an added benefit in the data being used is that nearly all of it can be collected without running the malware on a system.

This is a critical next step to creating more dynamic and adjustable anti-virus solutions. Many of the basic anti-virus solutions wager signature based methods, which are limited to detecting known malware using things like hash calculations and comparing it but will fail to detect unknown malware or polymorphic malware (Tahir). This means that the current malware detection methods are not enough, but deep learning and other machine learning techniques can be part of the solution. A new solution that would be able to detect brand new malware and polymorphic ones, that are able to adjust and be rebuilt in case there are new methods that are able to subvert these models.

Method

Data Collection

The dataset was retrieved from Kaggle, but is originally from a Microsoft Malware Big Data challenge. This challenge had thousands of samples of malware that were used against Windows machines, and engineered features from thousands of static analyses. This is another reason that binary malware was being chosen by the data stewards. Static analysis is the process of analyzing the code and structure to determine its function, but dynamic analysis is an examination of the program while it is running to almost do some type of behavioral analysis of the running code (Sikorski and Honig). The issue with dynamic analysis is that it is rather hard to automate, but static analysis can be easily run and captures many useful features that might be able to understand the malware at a surface level. Some of these features that are able to be captured would be useful for understanding the purpose of a program. Another advantage to using this type of static analysis observations that can be automated is that this can be built into a program that will take new malware and be able to classify it without requiring human analysis.

Though there exists malware that wagers interpreted languages to run, static analysis doesn’t reveal as much about them. The reason is because they don’t get compiled, and therefore aren’t structured the same way fully compiled and don’t give the same information about them. This comes more apparent when digging into the features that are available in the data that has been collected.

Analysis

To confirm that these are realistic and helpful features in this dataset, an analysis of the features that were collected was required. As well a basic analysis would determine any data cleaning operations that would be required to create a classification model. Taking a look at the data was simple and the features that were engineered were very simple because they were all vectors of numbers. This means that it would be relatively easy to create a model for this data using various methods like deep learning neural networks. However, figuring out what each of the features are and what they represent is something entirely.

When digging into the features it takes a bit deeper of an understanding of the structure of a Windows binary, and some of the libraries that are available in the Windows system calls. There are some data points that are going to be very helpful right off the bat, like calculations of offsets for both the starting size of the program and the virtual size that will be allocated for this. This would help identify a strategy called packing, which is an obfuscation technique where a program is compressed to avoid being read and is commonly implemented with malware in order to avoid basic Anti-Virus. Another set of variables that are exposed are the number of imports for certain Windows System libraries. In the headers after offsets a program will declare what libraries are linked and the functions that are being imported in the program. This helps because it essentially must declare what it is importing, and that can easily give insights into what the malware is being used for. For example, a keylogger will have to import some Windows System functions and libraries that will give access to detect and record keyboard inputs in order to function correctly. There are other factors as well included like the number of variables in the binary, types of variables, and lots of other attributes that may help understand what it’s purpose is but not through as direct means as library and function imports.

The problem while trying to do the same type of analysis on a piece of malware that is written in an interpretive language is that the same information isn’t as easily collectible. One of the biggest differences is the packing and encoding of malware in these programs can get more complicated. In many of these programming languages it is even possible to dynamically build code and strings and then run them as code, which can hide the true intentions, and can’t be decoded easily in automated means. Much of the time this requires an analyst to reverse engineer what is happening, creating decoding programs, or even running the malware in a safe environment in a debugging like process to see what the true intentions are for. This means that it can hide the same type of information that is possible to gather using the binary file and make it inaccessible.

Data Cleaning

After a basic analysis of the data was conducted, figuring out any data cleaning operations was necessary to build a helpful model. Luckily there wasn’t much that required much cleaning at all, since all of that data was already in numeric form, didn’t have any gaps in the data, and no odd values were found. There was only one slight change that was required for the models to be built, and it is because of the nature of the “Class” column that contained values from 1-9 which wasn't a problem initially. However to build the model and have it evaluated for accuracy it was required to be one-hot-encoded to put it into matrix form. After the encoding, that is all that is required to be able to build and test a deep learning model using Tensorflow.

When trying to implement something that would be able to do all of this data collection and cleaning that would be required, it could be done via a simple static analysis of the malware’s binary file. It would just take the reading of the binary file, leading the head information into structs and interpreting it to get all the data points. Using the data from static analyses, even if it is packed via detection and automate the data collection. From there after a reliable model is built, the classification of malware can then be automated as well for new malware that has a binary form.

Model

There has been some previous work or ongoing work with the identification and classification of malware, many companies have found this idea worth investing. Fire-Eye has already created a version of a model that tries to detect all various types of malware, though it seems to still be in testing rather than on the market, transforming many of its features into an embedding space into convolutional filters that eventually determine whether or not it is malware or not (Coull and Gardner). This shows how important this is becoming and experimental it is, with researchers trying to find reliable methods of automatic detection of previously non-existent malware instances. Mitigating the possible threats of millions of malware attacks before they even possibly get sent out in the wild.

Creating a model depends on what structure, and the data that is being collected is essentially a vector for a single record. Meaning that there are many deep learning neural networks that are not available, such as convolutional neural networks. However, what is possible is to create a neural network through one or multiple Dense layers. Dense layers are a type of neural network and deep learning layer that act like linear layers but use a non-linear activation function, which then can in turn be stacked on with several other layers to form a deep learning neural network (Moawad). This means that though the project might be limited in the variations of deep learning models that can be created, Dense layers will play an important role since it is able to use vector data if a deep learning model is to be created.

The idea is basically to create multiple models using a variation of Dense layers, seeing if the number of layers and how many neurons that they might have using a for each layer using the rectified linear unit calculation as the activation function. The main unifying factor is the final Dense unit that consolidates all the neurons into nine outputs that is then used to represent the classification. Otherwise they will be different in the number of layers and neurons throughout the model in hopes to find the most efficient model. The first model will go straight from the input layer to a dense layer that will output the predictions, where the second model will include another layer in between with a number of inputs for the layer in between the starting and the final, and same for the preceding three models adding more layers and neurons. The theory here is that either the simplest will just perform the best because it generalizes the best, or adding more layer and slowing having the dense layers decreasing the number of neurons at each layer will be able to best adjust for all input cases that are given.

There are two components to the training phases that are added both as a heuristic to speed up the training and another that will allow the best possible version of each model to act as the one in validation. The early stopping mechanism is a function that evaluates over the last number of epochs whether or not the validation accuracy has improved, and if it hasn't over that entire period of runs will stop early. The reasoning behind this is because the model isn't likely to improve over another epoch if it hasn't improved over the last 40 epochs. With similar logic there is a function that will essentially save the model as it stands whenever it makes an improvement in the validation accuracy. It does have the threat of saving a model that is overfitting, but that can be seen with what epoch it was saved at and looking at the validation accuracy trends. Though this is a threat that is present, it should help in order to find out of all the models that are developed, to use the best performing models for final testing.

Results

The results are quite impressive showing the model's abilities to correctly classify many of the malware examples without overfitting to the training examples. All of the models had an accuracy above 86% and the best model being just above 94% accurate. As well, with the intermediate saving methods upon improvements of the validation accuracy was successful. It allowed for many more epochs to be run without ruining final results and evaluations of each model, despite the certain dips in performance when running far too many epochs. For more information on the separate runs, refer to the Appendix.

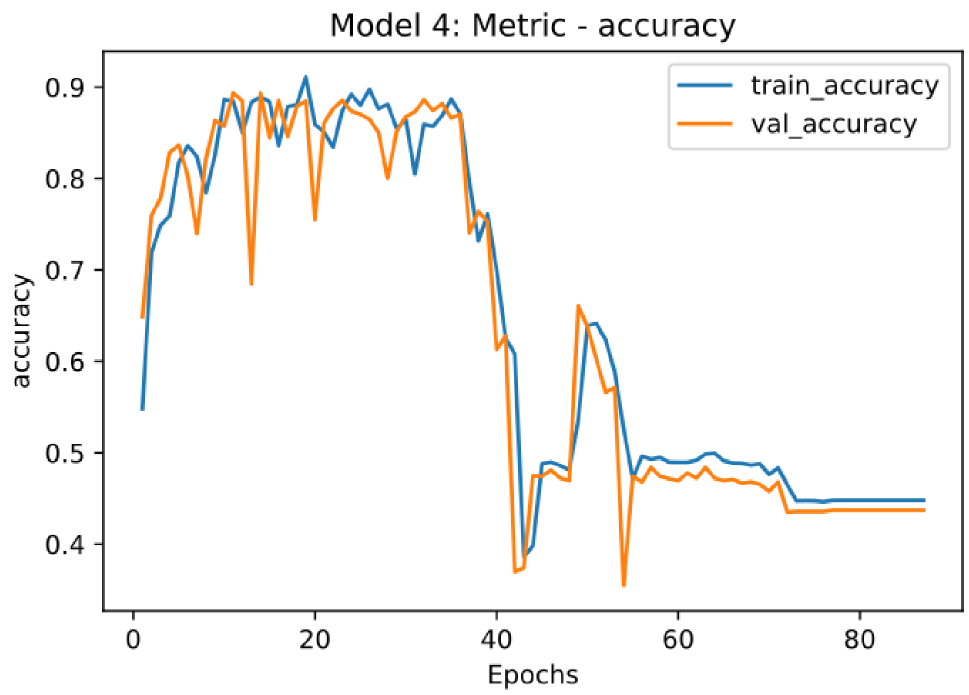

Figure 1: Model 4 Accuracy through training for over 80 epochs

Using Figure 1, it can be seen for the fourth model, that it varied a lot in both training and validation accuracy during training. After the fortieth epoch run, for both measures of accuracy there was a very drastic drop in accuracy that it never really recovered from. However, because again the monitoring functions would have saved the model that performed the best with validation which would be around 90% accurate for the validation data.

Model	Loss	Accuracy
Dense 1 (1 layer)	9989.216797	94.25%
Dense 2 (3 layers)	14895.929688	88.79%
Dense 3 (4 layers)	9930.941406	86.48%
Dense 4 (7 layers)	28.104568	90.49%
Dense 5 (8 layers)	3.472593	89.21%

Table 1: Final Evaluation of Loss and Accuracy of each model

Using Table 1 for the final results from evaluating the models with test data split from the beginning, there are lots of interesting observations to be made. First, the accuracies recorded found that the simplest of all the models was the most accurate but the second most was one with many more layers. Interesting because it does confirm that the best model was the simplest here but there was an uptick for certain models with many more layers later on. Next the loss calculations were fascinating, because the loss was relatively high for the first three models, but drastically went down for the last two models. For the last two models it does show that there is likely some overfitting in between some of the layers, but that doesn't exactly coincide with a much lower accuracy.

Discussion and Future Works

The final results were helpful, because it does show with a good amount of certainty that creating a classification model for malware is possible with a high amount of accuracy. This of course can be improved on further via several methods. One of the improvements could be made by further manipulating the structure of the data, making each one of the records a different shape. This could allow different types of operations and models to train on the data, for example a convolutional neural network could be created if each record were presented more of the shape of a matrix rather than a singular vector. These new models would still have to use Dense layers, but it would give more variation in the models that could be created. The only other major improvement would have to be in the increase of available data, because the larger the dataset the better the problem could be analyzed and trained in a deep learning model.

This project does show that there are some good features and indications of how to classify the purpose of a program to an extent. It could also mean that it would be able to be changed, or at least help in the creation of better models that can differentiate between malware and non-malware. These are future works that would depend on some more flexible data sources that are planned, but will have to take place a bit into the future. Another major future work that would be very promising is to do some basic automated dynamic analysis of some of these binaries being considered, that would result in many more features that would be even more revealing in the purpose and possibly intent of any developed software.

References

Coull, S. & Gardner C. 13 December 2018. *What are Deep Neural Networks Learning About Malware?* Retrieved from https://www.fireeye.com/blog/threat-research/2018/12/what-are-deep-neural-networks-learning-about-malware.html.

Firch, J. (2021). *9 Common Types Of Malware (And How To Prevent Them)*. Retrieved from https://purplesec.us/common-malware-types/#:~:text=The%20most%20common%20types%20of,adware%20%26%20spyware%2C%20and%20rootkits.

Moawad, A. 8 October 2019. *Dense layers explained in a simple way.* Retrieved from https://medium.com/datathings/dense-layers-explained-in-a-simple-way-62fe1db0ed75.

Saha, S. 15 December 2018. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way.* Retrieved from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

Sergii Banin, Geir Olav Dyrkolbotn, *Multinomial malware classification via low-level features, Digital Investigation, Volume 26, Supplement, 2018*, Pages S107-S117, ISSN 1742-2876, https://doi.org/10.1016/j.diin.2018.04.019.

Sikorski, M., & Honig, A. (2012). *Practical malware analysis: The hands-on guide to dissecting malicious software.* San Francisco: No Starch Press.

SonicWall (2020). *2020 SONICWALL CYBER THREAT REPORT: THREAT ACTORS PIVOT TOWARD MORE TARGETED ATTACKS, EVASIVE EXPLOITS.* Retrieved from https://www.sonicwall.com/news/2020-sonicwall-cyber-threat-report/.

Souppaya, M & Scarfone, K. (2013). *Guide to Malware Incident Prevention and Handling for Desktops and Laptops.*Retrieved from https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-83r1.pdf.

Tahir, R. (2018). *A Study on Malware and Malware Detection Techniques.* Retrieved from http://www.mecs-press.org/ijeme/ijeme-v8-n2/IJEME-V8-N2-3.pdf.

Z. Xu, S. Ray, P. Subramanyan and S. Malik, "*Malware detection using machine learning based analysis of virtual memory access patterns,*" Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, 2017, pp. 169-174, doi: 10.23919/DATE.2017.7926977.

Appendix A

Model 1 Structure and Training Results

This first model was the simplest in nature because it only really had two layers, but it performed the best overall. What this one will do is essentially use the input layer and bring it down to nine neurons, where the value of the neurons will output and represent which class that it is being predicted as. Of course, doing an evaluation in keras will result in a simple answer whether or not the prediction was correct.

Layer (type)	Output Shape	Param #
Input Layer	(1804, None)	NA
Dense	(None, 9)	16245

Table 2: Model Summary of Dense Model 1

The above structure describes the entire model for Dense Model 1.

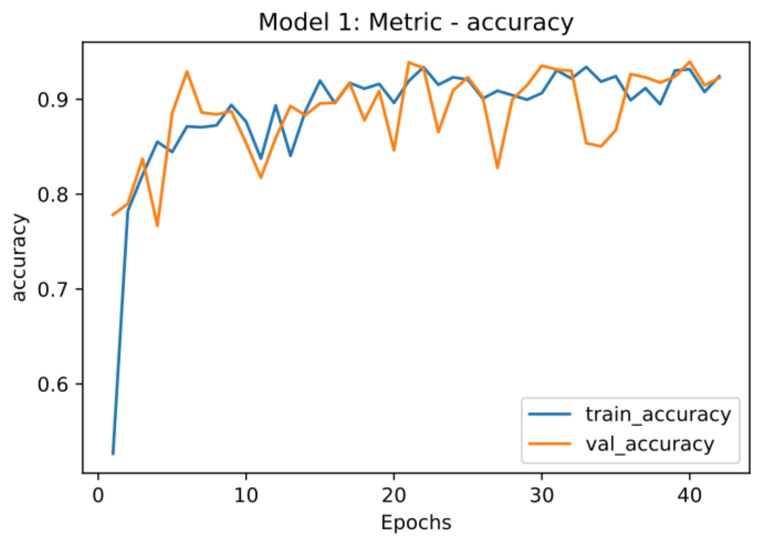

Figure 2: Accuracy trend for training of Dense Model 1

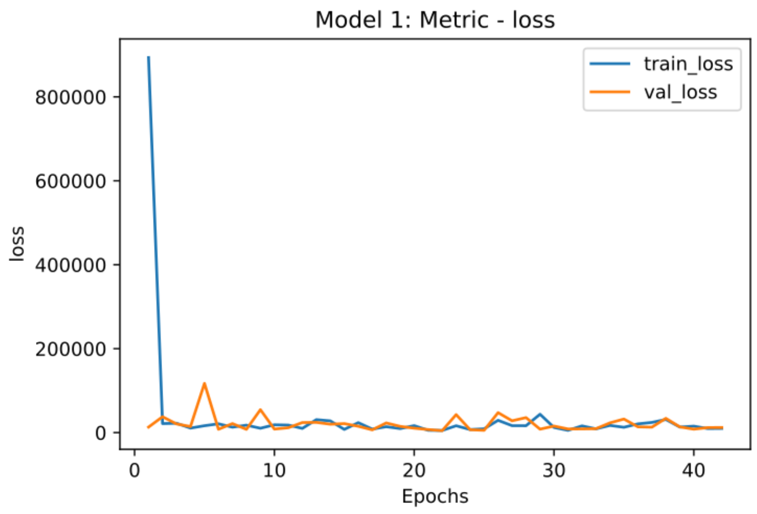

Figure 3: Loss trend for training of Dense Model 1

Using Figures 2 and 3, the overall training and performance of this model was exceptional, very quickly dropping the loss and steadying out. The accuracy does a steady climb and remains fairly steady throughout.

Appendix B

Model 2 Structure and Training Results

The second model is where the models begint to get a little more complicated with slower changes to the shape slowing whittling it down to the last dense layer that results in the 9 classes. This one performed rather well but had dipped in performance in comparison to the first model.

Layer (type)	Output Shape	Param #
Input Layer	(1804, None)	NA
Dense 1	(None, 64)	115520
Dense 2	(None, 32)	2080
Dense 3	(None, 9)	297

Table 3:: Model Summary of Dense Model 2

The above structure describes the entire model for Dense Model 2, and the number of parameters are starting to increase dramatically with each layer added.

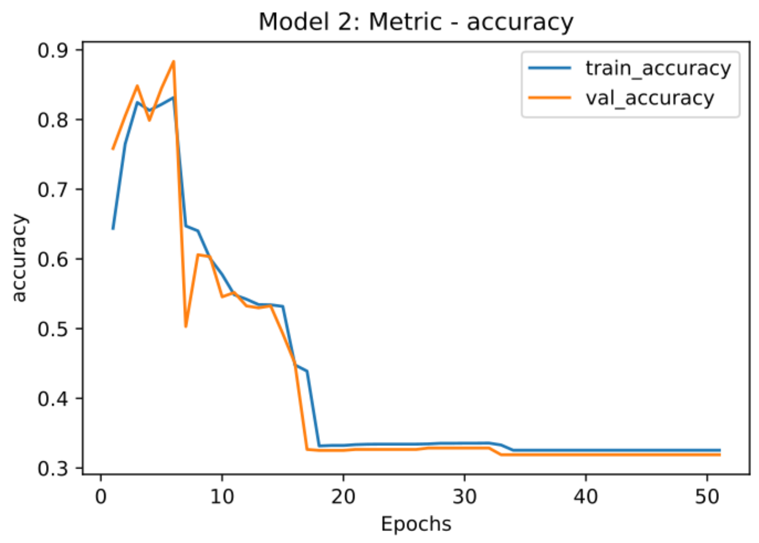

Figure 4: Accuracy trend for training of Dense Model 2

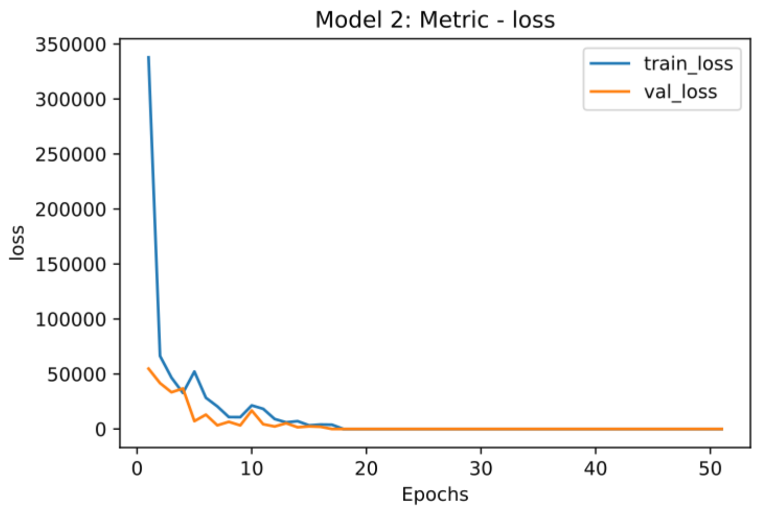

Figure 5: Loss trend for training of Dense Model 2

Using Figures 4 and 5, it can be seen that the accuracy was pretty high initially, but the model obviously wanted to decrease the loss metric more than improving accuracy. So the loss had dropped very hard, but the accuracy dipped in performance very quickly. The only thing that saved the performance of the model in the end was the intermediate saving functions on accuracy improvement.

Appendix C

Model 3 Structure and Training Results

The third model here, is another layer being added and had similar performance issues that were seen in Model 2.

Layer (type)	Output Shape	Param #
Input Layer	(1804,)	NA
Dense 1	(None, 128)	231040
Dense 2	(None, 64)	8256
Dense 3	(None, 32)	2080
Dense 4	(None, 9)	297

Table 4: Model Summary of Dense Model 3

The above structure describes the entire model for Dense Model 2, and again the number of parameters are increasing but the output shapes changes are becoming slower.

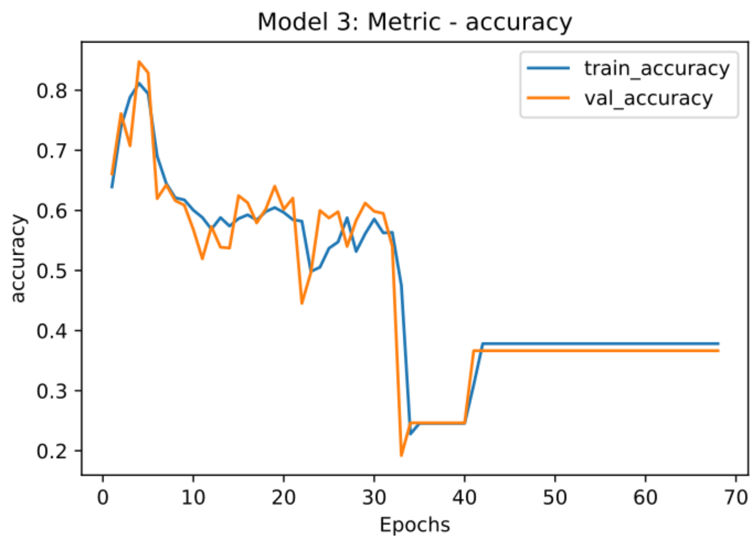

Figure 6: Accuracy trend for training of Dense Model 3

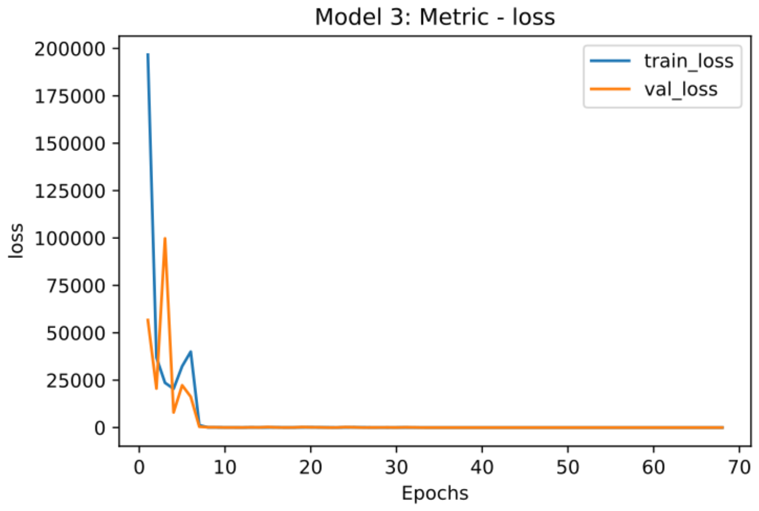

Figure 7: Loss trend for training of Dense Model 3

Using Figures 6 and 7, the performance never got very high and still had major issues on a large number of epochs. The loss does seem to be lower, but the performance continues to decrease in accuracy.

Appendix D

Model 4 Structure and Training Results

By the fourth model there are enough layers to slowly go down from the original output to the 9 final neurons on the last layer. The performance was a bit higher than the Models 2 and 3 but not exceeding Model 1.

Layer (type)	Output Shape	Param #
Input Layer	(1804,)	NA
Dense 1	(None, 1024)	1848320
Dense 2	(None, 512)	524800
Dense 3	(None, 256)	131328
Dense 4	(None, 128)	32896
Dense 5	(None, 64)	8256
Dense 6	(None, 32)	2080
Dense 7	(None, 9)	297

Table 5: Model Summary of Dense Model 4

The above structure describes the entire model for Dense Model 4, this is one of the first time when there are millions of parameters that are being considered.

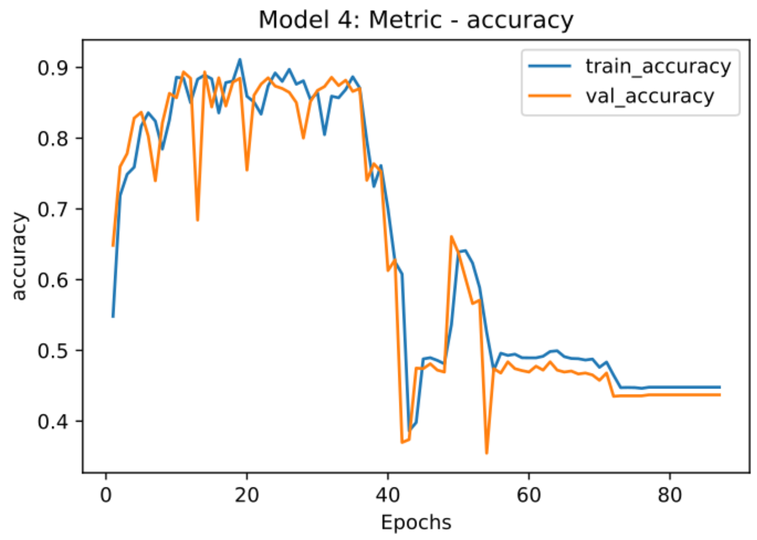

Figure 8: Accuracy trend for training of Dense Model 4

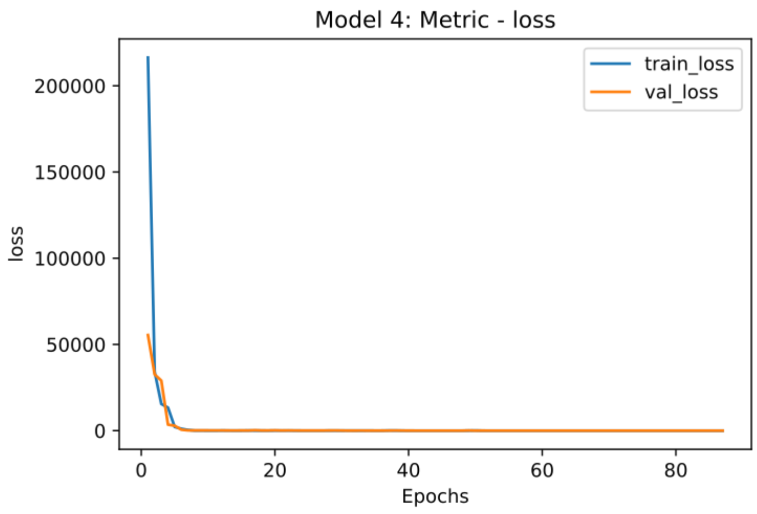

Figure 9: Loss trend for training of Dense Model 4

Using Figures 8 and 9, the performance does have a fairly high start for many epochs and only decreases around the 40th. The loss here shows that it is trying to overfit in some layers because of the very sharp decrease in loss calculations but with little improvement in accuracy.

Appendix E

Model 5 Structure and Training Results

The last model had the most layers, not only slowly decreasing the number of neurons at each layer but expanding them initially. The performance was surprisingly steady and high for an extended period of time. It might show that not only the layers but perhaps changes in the layers themselves might help in improving the accuracy since it is beginning to increase at this point in performance.

Layer (type)	Output Shape	Param #
Input Layer	(1804,)	NA
Dense 1	(None, 2048)	3693030
Dense 2	(None, 1024)	2096128
Dense 3	(None, 512)	524800
Dense 4	(None, 256)	131328
Dense 5	(None, 128)	32896
Dense 6	(None, 64)	8256
Dense 7	(None, 32)	2080
Dense 8	(None, 9)	297

Table 6: Model Summary of Dense Model 5

The above structure describes the entire model for Dense Model 5, and with many extra layers and neurons that are included it does slow down training and evaluations.

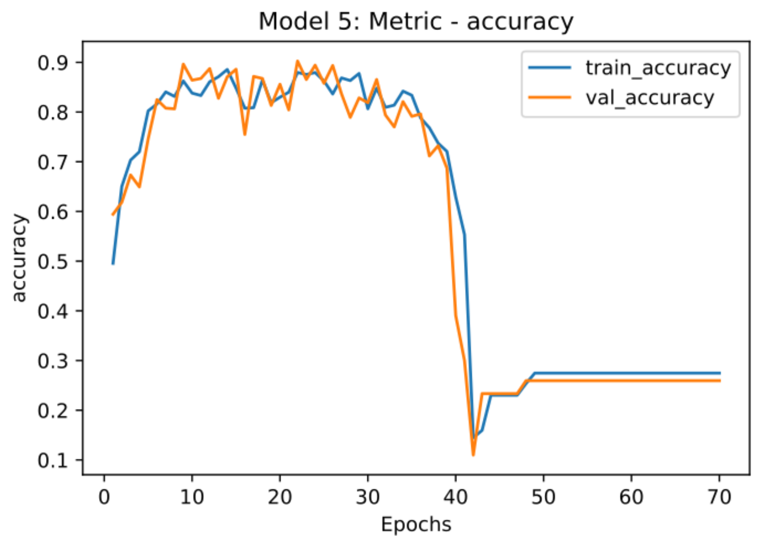

Figure 10: Accuracy trend for training of Dense Model 5

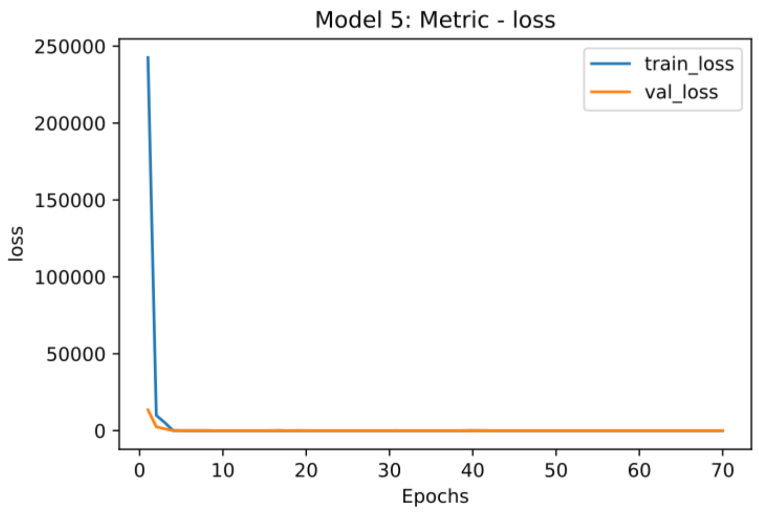

Figure 11: Loss trend for training of Dense Model 5

Using Figures 10 and 11, the accuracy is impressively high and stable for a very long time, but has a massive performance issue just after the 40th epoch. The loss calculations do seem to be improving vastly, just not improving accuracy at the same rate.