Improving the Robustness of Convolution Neural Networks using N-Version Programming

Carl Hildebrandt[1], Advait Kulkarni[1] and Tyler Williams[1]

Abstract— **The goal of this work is to increase the robustness of convolutional neural networks. We approach this problem by using N-Version programming. N-version programming has been shown to increase the robustness of software applications. Traditionally N-version programming suffers from difficulty in creating and implementing independent programs. The creation of neural networks is stochastic by design and thus the creation of N independent programs is cheap. We use this to our advantage and apply N-version programming to the problem of creating robust networks. We take this one step further in attempting to distill our N-version program into a single network, with the goal of transferring the robustness properties from the N-version program into a final distilled network. This final network thus will have the benefits of being as robust as an N-version program but require very little processing power to run.**

I. Introduction

Machine learning has quickly become one of the most popular topics in computer science. Recent advancements in the performance and accuracy of machine learning techniques have to lead to many breakthroughs in self-driving cars, classification of live video feeds and natural language processing. The use of these networks, however, go far beyond self-driving cars and autonomous drones. These networks are now used in gaming [10], modeling of environments such as liquid[17], human pose estimation[16], and much more. These networks are achieving state-of-the-art results in terms of their accuracy. However, the robustness of the neural networks is often a second thought. The lack of interest in robustness leaves highly accurate networks sensitive to adversarial attacks.

Our work will attempt to improve the robustness of neural networks through N-version programming. N-version programming uses system redundancy to improve the accuracy of the overall system. We hypothesize that this system redundancy will also improve the robustness of the system, and thus improve its ability to handle adversarial cases. We will test this hypothesis using a series of robustness measures. We will use both formal verifications as well as a self-created adversarial dataset to measure the robustness of our networks.

This paper is aimed at addressing four fundamental questions which lie on the intersection of dependability and machine learning.

- How robust are neural networks?
- Which type of adversarial attacks are most successful?
- Are adversarial attacks defendable through N-Version programming?
- Can we use N-Version programming to make more robust neural networks?

II. APPROACH

The goal of our project was to generate a more robust CNN using techniques learnt in our dependable computing class. Machine learning has the distinct advantage of being able to create many version of the same program rapidly. Using this insight we decided to apply N-Version programming to CNN's to see if we could increase their robustness. We wanted to know if we could then take this knowledge and apply it to make a more robust CNN.

We broke this problem into five distinct parts namely: Network Generation, Adversarial Generation, N-Version Programming, Robustness Certification, and Network testing. Each of these sections is described in more detail below. An overview of our design can be found in Figure 1

A. Network Generation

This project required many versions of the same program. We define the program as a convolutional neural network which has learnt some function X. For this project, we set the goal function X to be the Mnist dataset[8]. Mnist is a dataset of 60000 single channel images depicting handwritten digits from 0 to 9. It should be noted that this our goal X could theoretically be any function.

Achieving variation in neural networks is not difficult. This is due to the stochastic initialization of the weights. However to achieve a larger degree of variation, and thus hopefully a better final robustness, we also varied the neural networks architectures. Our neural networks consisted of 4 basic layers. Dense layers, convolutional layers, maxpooling layers, and dropout layers. Each of these basic layers has different parameters; for instance, a dense layer can specify how many neurons that specific layer should have. We used these variable layer parameters as well as different orderings of layers to generate 100 different network architectures.

We designed 100 network architectures as text files using the layers and parameters described above. We wrote a custom python script to parse the text files and create a model in Keras[4] a machine learning framework. Once our python script had generated the model it would train the model using the Mnist training data. We did not vary any of the hyper-parameters between each of the CNN's. Using Keras's documentation we selected the training hyper-parameters recommended in the example code and applied that to all

This work is done in requirement of our Dependable Computing class

[1]Computer Science Department at the University of Virginia, USA. ch6wd@virginia.edu,mmb4vu@virginia.edu

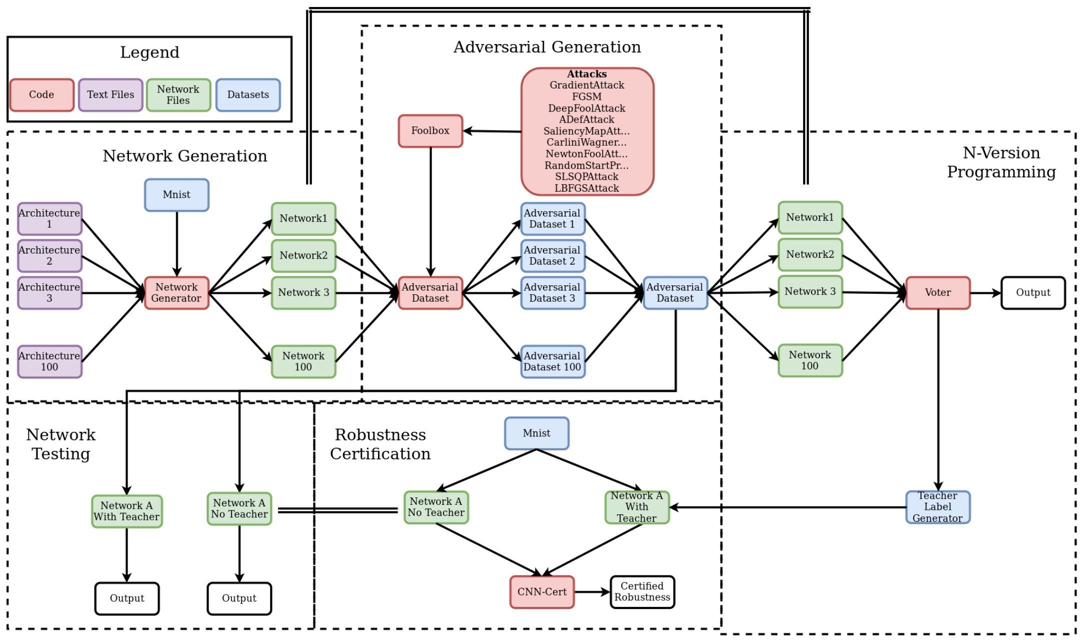

Fig. 1. An overview of the designed and implemented system.

network generation. We did this to avoid any networks not learning due to incorrect hyper-parameter selection.

Our custom python script then saved each of the networks as well as recorded the final training accuracy for later processing.

B. Adversarial Generation

To test how well N-version programming did compared to a single network, we needed to generate an adversarial dataset. Adversarial images can be defined as inputs that are almost indistinguishable from natural data and yet classified incorrectly by the network. The adversarial dataset would contain a set of adversarial images. We generated the adversarial dataset by generating adversarial images for each of the individual networks and combining them to form a final adversarial dataset. Recent research has shown that you can achieve robustness against one adversarial technique however this does not guarantee you are robust against all adversarial techniques [3].

Thus to reduce the chance of simply being robustness against a single adversarial technique we implemented 10 different adversarial techniques using Foolbox[14]. Foolbox is a Python toolbox to create adversarial examples. Each of the techniques we used is listed below in Table I. We should note that all our techniques are gradient-based techniques. This was an oversight by our group, as we should have added not only more adversarial techniques but also a wider variety of adversarial techniques such as score based adversarial techniques. This was left for future work.

We wanted to create and an adversarial dataset which had a similar size to that of the Mnist test dataset. Mnist has a 10000 image test dataset. We thus needed to generate 100 images for each of the 100 networks to get an adversarial dataset of 10000 images. This was not straight forward as the generation of adversarial images can fail for one of the attacks but not the other. We thus requested our script to generate 13000 images allowing for some adversarial generation failures.

C. N-Version Programming

1) N-Version Programming as Defense Against Adversarial: We wanted to answer two questions with N-version programming. Did N-version programming using convolutional neural networks have similar trends to N-version programming with traditional programs? Secondly, how accurate could we make the N-version program against our adversarial dataset?

To answer these questions we first passed the adversarial dataset to each of the individual networks. This would give us a baseline of how accurate an individual network would be against these attacks. It should be noted, that in the best case, a network should still fail on at least 130 of the images from the adversarial dataset. That is because 130 of the images were specifically targeted at that network during generation. We assumed however that due to all networks trying to learn the same function, adversarial images for a specific network would overlap with other networks.

We then took each of the networks and configured them in an N-version programming setup. To create the N-version program voter we passed the input to each of the individual networks. The networks each gave us a probability distribution (softmax output). The distributions gives you the probability the image was from each of these classes. When you are trying to get the prediction from a neural network you would simply find the highest probability and output that as your class. We, however, took each of the probability distributions and calculated the average distribution. Thus

TABLE I
ALL ATTACKS USED AND A DESCRIPTION DESCRIBING HOW THEY WORK.

Adversarial Attack	**Description**
Gradient Attack[5]	Perturbs the image with the gradient of the loss w.r.t. the image,gradually increasing the magnitude until the image is misclassified.
Gradient Sign Attack[5]	Adds the sign of the gradient to the image, gradually increasing the magnitude until the image is misclassified. This attack is often referred to as Fast Gradient Sign Method.
Deep Fool Attack[11]	Simple and close to optimal gradient-based adversarial attack.
ADef Attack[1]	Adversarial attack that distorts the image by changing the locations of pixels.
Saliency Map Attack[12]	Create a saliency map to map the input to output and use that to craft adversarial images.
Carlini Wagner Attack[3]	Attack that maximized the L2 distance of the input and adversarial image.
Newton Fool Attack[7]	Adds small, often imperceptible, perturbations by analysing the gradient of the weights.
Random Start Projected Gradient Descent Attack[9]	Gradient attack with random start and projection of the gradient descent.
SLSQP Attack	Uses SLSQP to minimize the distance between the image and the adversarial under the constraint that the image is adversarial.
LBFGS Attack[15]	Uses L-BFGS-B to minimize the distance between the image and the adversarial as well as the cross-entropy between the predictions for the adversarial and the the one-hot encoded target class.

our voter took the mean of all the networks output and then found the class with the highest probability.

2) N-Version Programming as a Teacher: Network distillation is starting to become more popular in neural network verification and performance boosting [6][13]. Network distillation is done by assigning a larger network the role of a teacher network. One takes a smaller network and assign it the roll of a student network. The goal is for the student network to minimize the difference between the teacher and student output given the same input.

We assumed that our larger N-version program would be more robust against adversarial attacks. We thus tried to transfer this robustness from the N-version program to a student network using distillation. We distilled the networks by creating a new set of labels called the teacher labels. These labels were the mean softmax output of the N-version programs voter as described above in Section II-C.1.

To achieve a fair comparison we trained another network with the same architecture and same hyper-parameters as our student network. However, the baseline network was trained using the standard Mnist images and labels while the student had access to the Mnist images and the teacher labels.

D. Network Testing and Robustness Certification

The final step of our project was to compare the two networks to see if were able to achieve our goal of training more robust CNNs. To do this we took the baseline network and the student network and generated three robustness metrics. The first metric was generated using formal verification. This would give us the formally verified region of robustness for our networks. One might argue that this is enough as if you can formally verify how robust a network is you do not need any other metrics. However formal verification of CNN's is in its infancy and thus these verification techniques only return formal verification of regions it can verify. There might be large regions which are unknown due to the tools not being able to handle them.

Thus this gives you a good metric, however we decided that more metrics should be included. The second metric we used was how well the networks performed on the adversarial dataset we generated. This would give us a sense of whether or not you are able to distill the robustness properties from the N-version programming. The final metric was using our original 10 adversarial techniques to find what the adversarial frequency of both networks was. That is how often were the adversarial techniques able to generate adversarial examples for each of the networks. The adversarial frequency would give us a sense of how vulnerable the networks are to future attacks.

1) Formal Verification: We used a formal verification tool created by IBM called CNN-Cert[2]. CNN-cert can certify lower bounds on the minimum adversarial distortion. That is to say, given input, we can change it by **any** amount less than the minimum adversarial distortion and we can guarantee that the input class will not change. Using CNN-Cert we generated two metrics for the minimum adversarial distortion. The first metric was the L1 norm which in effect calculates the Manhattan distance between the input and the adversarial image. The second metric was the L2 norm which in effect calculated the Euclidean distance between the input and adversarial image.

2) Adversarial Dataset: The next test we performed was to see how well the networks performed on the adversarial dataset. This was done by taking the adversarial dataset we created in Section II-B and comparing the outputs. This was the simplest test but would give us insight into how well we could transfer the robustness (never done before, as generally, the goal is the transfer of accuracy/performance) properties between N-version network and our student network.

3) Adversarial Frequency: The final test was to see how susceptible both networks were against future attack. This was done by re-running the adversarial generation techniques on both networks and counting the number of times each technique was successful.

III. RESULTS

To accurately measure how well our technique did we tested each of the systems independently. Testing independent systems gave us a better understanding of how the system worked as a whole, thus allowing better intemperate our final results. The results are broken up into the five sections listed in the approach. They are also represented by the dotted lines in Figure 1.

A. *Network Generation*

The first step of our project was to generate N versions of different architectures. We decided that we were going to generate 100 networks, that is $N = 100$. Having a large N meant we could accurately see the trends of N-version programming in a machine learning context. We trained 100 networks on Mnist data and plotted the testing accuracy. This is the accuracy of the trained networks on the testing data provided by Mnist. The results can be seen in Figure 2.

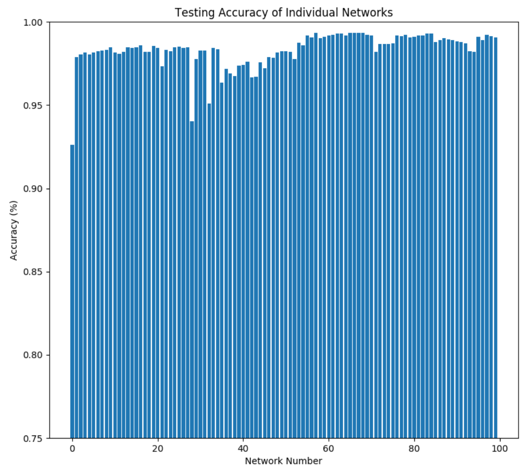

Fig. 2. The testing accuracy of all 100 trained networks. All networks obtained between a 92% and 99%.

Figure 2 shows that all networks achieved accuracies between 92% and 99%. These high accuracies indicated that all the networks were trained and able to identify images with high precision successfully. It should be noted that all networks were generated from a single python file which dynamically created the model from text files describing the architecture. This again proves that machine learning is well suited for N-version programming.

B. *Adversarial Generation*

The next step was to generate an adversarial dataset for these networks. The ideal dataset would contain roughly the same number of images as the testing dataset. Due to its adversarial nature, all networks should also perform worse on this dataset as compared to the testing dataset. To a human however, there should be no perceivable difference between the two datasets. We first ran and displayed examples of the adversarial techniques to get a better understanding of whether or not humans could perceive the difference between the examples generated and original images. The resultant images are shown in Figure 3.

Using the output from Figure 3 allowed us to accept 9 of the 10 attacks. The SLSQP attack was rejected as the final output image was identifiable as adversarial by a human. Thus the next step was to generate the adversarial images for the networks. We generated 13 images for each of the networks and appended them to create an adversarial dataset. We then passed the adversarial dataset to each of the networks to obtain an accuracy and plotted that accuracy against the testing accuracy to see if the accuracy is lower for the adversarial dataset. This is shown in Figure 4.

It can be seen that for all networks the accuracy on the adversarial data dropped. This meant that the data we had generated was indeed adversarial. We also noted that there was not a strong correlation between the accuracy of the Mnist test dataset and our adversarial dataset. This indicates that network performance and robustness are not correlated and thus should be treated as two separate measures. Therefore, although networks are designed to minimize the loss and thus increase the network performance, they are not specifically designed to increase the robustness. This indicates that as a developer, it is our job to make sure the networks we develop are robust.

The final measure we wanted to take was how effective were our adversarial attacks on the networks. To do this we measured how often an attack was able to generate an adversarial image for a network. Each attack was run on 13 images for all 100 networks. Thus each attack was called 1300 times. We thus divided the number of successful adversarial images from a specific attack by 1300 to get the effective percentage. This is summarized in Table II:

TABLE II
ATTACK EFFECTIVENESS ON 100 RANDOMLY GENERATED CNNS.

Adversarial Attack	**Success Count**	**Effective Percentage**
Gradient Attack	939	73.23%
Gradient Sign Attack	1272	97.84%
Deep Fool Attack	1281	98.53%
ADef Attack	1240	95.34%
Saliency Map Attack	1254	96.46%
Carlini Wagner Attack	1300	100%
Newton Fool Attack	1297	99.77%
Random Start Projected Gradient Descent Attack	1300	100%
LBFGS Attack	1300	100%

Table II again clearly shows the shortcomings of CNN's in their robustness. The mean effective percentage of these attacks was 95.5%. That is an incredible number given that these networks have an average accuracy of more than 90%. This further shows that robustness needs to be considered as a separate metric.

C. *N-Version Programming*

We now wanted to see how well the networks would perform if we configured them in an N-version setup. We were also interested in understanding how the addition of networks to our N-version setup would affect the performance of the setup. Specifically, we wanted to understand if the stochastic nature of neural networks caused an additive growth in the

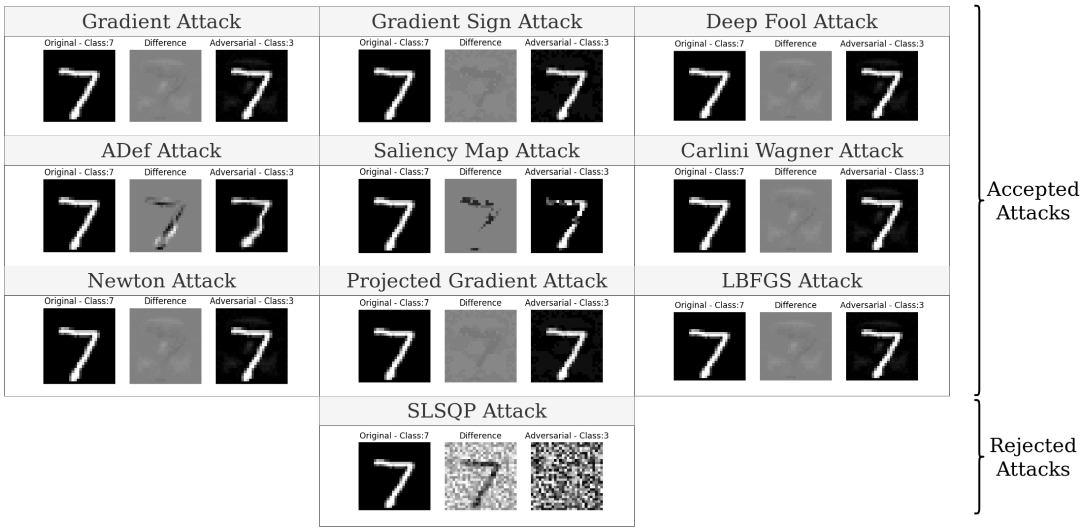

Fig. 3. Different attacks final attacks displayed. The attacks also show the difference between original and final image.

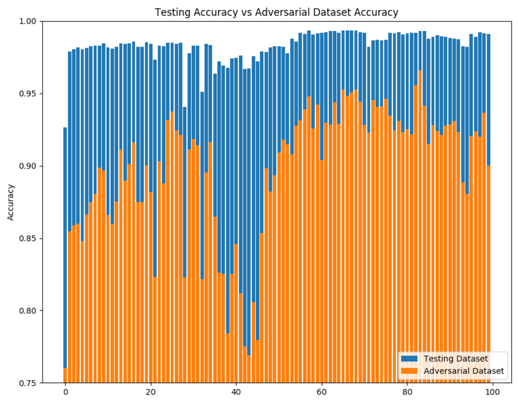

Fig. 4. Accuracy per network on the Mnist testing data (blue) compared accuracy per network on our adversarial dataset (orange).

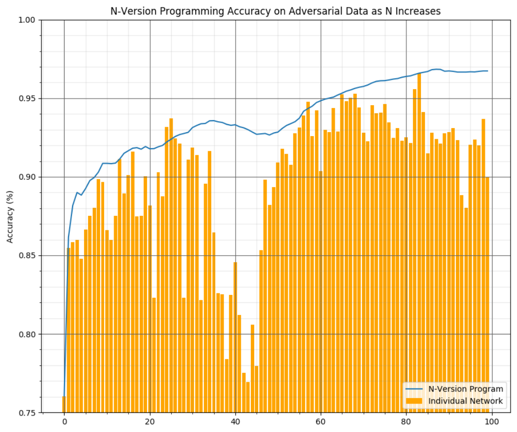

Fig. 5. N-version system performance on adversarial data as N is increased (blue line) compared to individual network performance on adversarial data (orange bars).

noise of the system due to adding networks. This noise would, in theory, cause a decrease in N-version performance. This is opposite to traditional N-version programming in which additions of networks of similar performance cause continued system performance increases. To see how well the networks would perform configured as an N-version setup we plotted the system accuracy of the setup on the adversarial dataset. This is shown as the blue line in Figure 5. Here we increase N from 1 to 100 while measuring the accuracy.

We noticed that during the addition of networks 35 to 50 there was a decrease in system accuracy. To understand this each of the performance of the individual network on the adversarial data was superimposed onto Figure 5. We noticed that networks 35 to 50 were not very robust against the adversarial data and thus caused a drop in system performance as they were added to the network. This is analogous to adding components with a low component reliability.

From this data, we can deduce that N-version programming does indeed improve the robustness of neural networks. With N set to 100 the system had a 96.86% accuracy against the adversarial data. On average the individual networks only had a 89.77% accuracy against the adversarial data, with the best individual network getting a 96.58% on the adversarial data. Thus N-version programming outperformed even our best network.

D. Network Testing and Robustness Certification

The final part of our project was the distillation of the N-version setup into a single student network. We would use two measures of robustness to confirm if our technique was able to generate networks which are more robust. We took the N-version setup and distilled it into a small network consisting of two convolutional layers and 2 dense layers. At the same time, we trained a new network with the same architecture using only Mnist data. We then ran IBM's CNN-Cert on the data and formally verified L1 and L2 minimum distance bounds. Another metric we measured on both networks was the adversarial frequency. That is how often adversarial images could be generated for both networks. Finally, we also gave the adversarial dataset to both networks and measured the accuracy against it. The results for all these tests can be found in Table III.

TABLE III
FINAL MEASURES BETWEEN BASELINE NETWORK AND STUDENT NETWORK.

Metric	**Baseline Network**	**Student Network**
Testing Accuracy	99.16%	98.68%
L1 Norm (Formal Verification)	0.1526	0.1609
L2 Norm (Formal Verification)	0.0743	0.0809
Adversarial Dataset Accuracy	96.04%	95.35%
Adversarial Frequency	94.33%	98.00%

The testing accuracy, adversarial accuracy and adversarial frequency all became worse in the student network. This suggested a less robust network. For formal verification however both the L1 and L2 norms increased suggesting a more robust network. These results were conflicting and thus it was thus clear we needed to do more analysis to determine if distillation was able to transfer robustness properties.

To do these we compared the student network to all 100 networks accuracy against the adversarial dataset. To do this we plotted the 100 networks accuracy against the adversarial dataset as a distribution using a box and whisker diagram. We then drew a blue line for the accuracy achieved by our student network. This is seen in Figure 6.

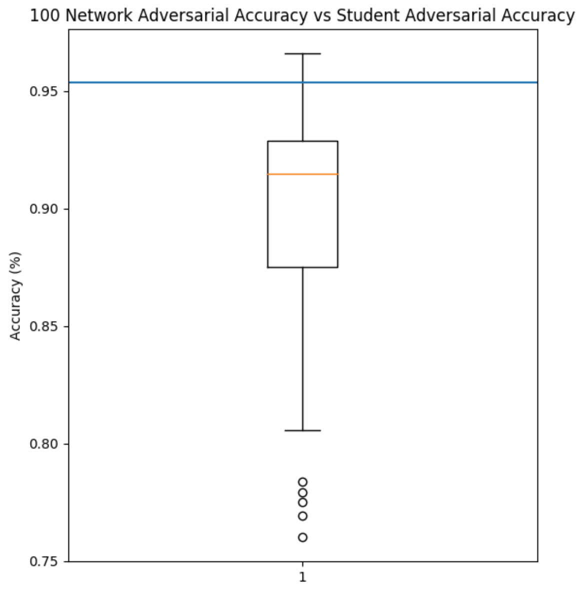

Fig. 6. All 100 networks accuracy against the adversarial dataset as a box and whisker plot. The student networks accuracy against the adversarial data as a blue line

We can clearly see that the student network is more robust than a large portion ($\geq 75\%$) of networks trained randomly. This suggests that our distillation of networks does indeed transfer the robustness properties from the N-version programming setup. In fact, the student network would have been the fourth most robust network against the adversarial data as compared to all 100 individually trained networks. To confirm this however, this experiment would need to be repeated numerous times which was out of the scope of this final project.

IV. CONCLUSION

In this projected, we successfully trained 100 individual neural networks. We created an adversarial dataset. We used techniques learnt in our dependable computing class, more specifically N-version programming, to increase the robustness of our system against the adversarial dataset. We also finally used neural network distillation to transfer these robustness properties from an N-version program into a smaller individual student network.

We were able to answer all our original questions, showing that neural networks are not inherently robust. They are susceptible to a large range of attacks. N-version programming does indeed increase the systems ability to handle adversarial images and thus increases robustness, and finally we there is a good reason to believe we are able to distill these properties into a smaller network. It should be noted that these results are only preliminary and more tests would need to be run in order to determine if the distillation in fact was successful.

V. APPENDIX

Our code is available on Github:
`https://github.com/hildebrandt-carl/ImprovingNeuralNetworks`

A video of our project can be found at:
`https://youtu.be/u_tLKoU_lro`

REFERENCES

[1] Rima Alaifari, Giovanni S Alberti, and Tandri Gauksson. Adef: An iterative algorithm to construct adversarial deformations. *arXiv preprint arXiv:1804.07729*, 2018.

[2] Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. *arXiv preprint arXiv:1811.12395*, 2018.

[3] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

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

[5] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[6] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[7] Uyeong Jang, Xi Wu, and Somesh Jha. Objective metrics and gradient descent algorithms for adversarial examples in machine learning. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 262–277. ACM, 2017.

[8] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[9] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[10] Deep Mind. Mastering starcraft 2. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

[11] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.

[12] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy*, pages 372–387. IEEE, 2016.

[13] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597. IEEE, 2016.

[14] Jonas Rauber, Wieland Brendel, and Matthias Bethge. Foolbox: A python toolbox to benchmark the robustness of machine learning models. *arXiv preprint arXiv:1707.04131*, 2017.

[15] P. Tabacof and E. Valle. Exploring the space of adversarial images. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 426–433, July 2016.

[16] Alexander Toshev and Christian Szegedy. Deeppose: Human pose estimation via deep neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[17] Kiwon Um, Xiangyu Hu, and Nils Thuerey. Liquid splash modeling with neural networks. *Computer Graphics Forum*, 37(8):171–182, 2018.