THE PENNSYLVANIA STATE UNIVERSITY

SCHREYER HONORS COLLEGE

DIVISION OF ENGINEERING, BUSINESS, AND COMPUTING

DEEP ENSEMBLE LEARNING FOR ROBUST DEFENSE AGAINST ADVERSARIES

ETHAN ADAMS

SPRING 2020

A thesis submitted in partial fulfillment of requirements for baccalaureate degrees in Information Sciences and Technology and Security Risk Analysis with honors in Information Sciences and Technology

Reviewed and approved* by the following:

Mahdi Nasereddin

Associate Professor of Information Sciences and Technology

Thesis Supervisor

Sandy Feinstein

Professor of English

Honors Adviser

*Electronic signature available

ABSTRACT

Several measures have been taken to increase the security of AI models against adversarial inputs in response to growing reliance on the technology. While many data-driven solutions have been created, they do little to address the structural problems of the model. A novel model is tested with an ensemble structure, which subsamples its dataset, assigning portions to each sub-model in a nested configuration. The model is then tested against a control model trained on the overall dataset. The ensemble solution showed significant improvements in accuracy across multiple testing procedures when compared to the control model. Ensemble structuring is shown to be an effective method for improving model robustness without extending training time or processing workload required.

TABLE OF CONTENTS

ABSTRACT ..... **i**
TABLE OF CONTENTS ..... **ii**
LIST OF FIGURES ..... **iii**
LIST OF TABLES ..... **iv**
ACKNOWLEDGMENTS ..... **v**
INTRODUCTION **1**
AI Security 1
Structural Experimentation 2
Research Objective 3
LITERATURE REVIEW **4**
Neural Networks 4
Deep Neural Networks 5
Adversarial Examples 6
Ensemble Learning 7
MNIST 8
Generating Adversarial Examples 9
METHODOLOGY **10**
Objectives 10
Control Model Implementation 10
Generating Adversarial Examples 10
Ensemble Model Implementation 12
Testing Adversarial Effectiveness 13
Development 13
RESULTS & DISCUSSION **18**
Recording Procedures 18
Training Data 19
Result Data 23
CONCLUSION **27**
APPENDIX **29**
Source Code 29
BIBLIOGRAPHY **46**

LIST OF FIGURES

Figure 1: Hidden layers of NNs and DNNs 5
Figure 2: Horse with one-pixel perturbation 6
Figure 3: standard and adversarial MNIST images 8
Figure 4: Base MNIST image 11
Figure 5: Non-targeted misclassifier 11
Figure 6: Targeted misclassifier 11
Figure 7: Algorithm for generating targeted misclassifier 12
Figure 8: MNIST image with color mask 14
Figure 9: Perturbations added to original MNIST image 15
Figure 10: Control model accuracy 20
Figure 11: Control model loss 21
Figure 12: Ensemble model accuracy 22
Figure 13: Ensemble model loss 23

LIST OF TABLES

Table 1: Results gathering procedures 19
Table 2: Standard testing results 24
Table 3: Initial adversarial testing 24
Table 4: Targeted adversarial results 25
Table 5: Composite adversarial testing 26

ACKNOWLEDGMENTS

The creation of this thesis took place during a turbulent time in my life. While writing the following chapters, I was in the process of moving past a long-term relationship, securing a full-time offer, and of course, attempting to graduate from university in the midst of a global pandemic. Without the support of those around me, the only thing I know for certain is that this thesis would not exist. In particular, I would like to thank my advisors, Dr. Sandy Feinstein and Dr. Mahdi Nasereddin. Dr. Feinstein spent an absolutely unreasonable amount of time helping me refine my writing. She assisted with small things like sentence fragments, but she also helped me develop an entire philosophy behind my writing. I joined Dr. Feinstein on a three-year quest, beginning as a terrified novice and ending as a person who can effectively wield writing in the world. Dr. Nasereddin provided a deep technical foundation and understanding of AI that I would have been completely lost without. Our countless hours breaking down and debugging the code for this thesis shaped it into something that I can be legitimately proud of.

Next, I would like to thank Bridget Baksa and Philip Ciunkiewicz. Having forged an immaculate thesis herself, Bridget served as a shining example to emulate. Whether she realizes it or not, she shaped my writing for the better. Philip was a great resource for me as I was learning TensorFlow and Keras. He provided insights on my data parsing sequence that only an experienced data scientist could communicate. I'd also like to thank the Erickson Discovery Grant committee and the Boscov family for their generosity. I would not have had the financial freedom to produce this thesis without them. Finally, I want to show appreciation to my professors, friends, and family, as they have kept me sane (or insane) enough to focus on this intellectual endeavor for an extended period of time.

INTRODUCTION

AI Security

As AI simplifies the lives of billions, an ever-increasing incentive brings those with malicious intent or curious minds to exploit the technology. Some may want to leverage the power and reach of important AI models for personal gain, while others may just want to prove a point. Thus, researchers have been exploring many areas of AI security, from data modification to attempts to strengthen AI models themselves. While AI models can be extremely accurate, they are subject to exploitation as they are partially unknowable by nature. This is because each model changes continuously during training based on the data being used. Continuous change on its own can be predicted, but the changes to the model are being made inside of a black box, or a sequence of hidden layers within the model that change in unpredictable ways. In this way, the biggest weakness of an AI model is also its greatest strength, as the hidden layers within a model allow it to identify features and trends across a given dataset.

Most exploitations of AI models thus far have been benign in nature. In some cases, responsible researchers in a controlled environment exploit models to better understand them. In other circumstances, unexpected inputs have been fed unintentionally to models in use. Regardless, researchers quickly became aware that AI models were not infallible, and the subfield of AI security emerged. AI security researchers attempt to better understand what makes AI models fail against adversarial examples, or inputs that can trick models into producing incorrect outputs.

Research in AI security takes two key forms, which will be referred to as red and blue research respectively. Red research aims to fool models, causing them to produce undesired

outputs. For example, red researchers have previously shown that small changes to street signs and road paint can cause autonomous vehicles to behave erratically and even crash (Sitawarin, 2018; Tencent, 2019). The goal of blue research, on the other hand, is to mitigate the potential for the model to succumb to red research. Blue researchers have shown ways in which AI models can be altered structurally to lower the probability of being fooled by adversarial examples (He, 2017). Both sides of this research field work together to increase the overall knowledge of AI.

Structural Experimentation

Much is still unknown about AI security, and there are several reasons why. First, having two competing roles (red and blue) causes unexpected growth, as researchers are constantly attempting to test each other's work. When blue researchers create a new model resilient to malicious alteration, red researchers are incentivized to present a new attack vector capable of defeating the new resilient model. Second, the field of AI security is still less than a decade old, meaning that innovation is almost as common as iteration. Finally, as a subfield, AI security is subject to the changes and advancements in AI research more broadly. Given this dynamism, there are innumerable facets of the field waiting to be explored. One underrepresented aspect of the field as a whole is the role of meta-structured models in blue research.

Meta structures stem from another AI subfield known as Meta Learning, which focuses on creating compound models with nested subcomponents (Frans et. al., 2017). The cross-section of AI security and Meta Learning hosts a broad range of specific topics, ranging from secure Generative Temporal Models (GTMs) to ensembled structures (Gemici et. al., 2017). Despite the potential for progress in both blue and red research, this overlap of the two sub-fields has remained relatively unexplored. A focus on ensemble learning addresses this gap in research. An

ensemble is a type of meta-structure that seems best fit to address the weaknesses of neural networks, which are a standard use of AI.

Adversarial examples can be generated to fool neural networks, which are a standard type of AI model in use today (Goodfellow, 2015). By creating a grouping (or ensemble) of slightly weaker neural networks, the adversarial input is less likely to fool the overall model. Even if one nested neural network in the group is fooled by the adversarial example, it is unlikely that it will fool more than one, with that probability decreasing as the ensemble gains more members.

Research Objective

The following chapters intend to add to the blue side of the field of AI security and address the weaknesses of ordinary neural networks. It develops a new model structure which reduces the effectiveness of adversarial examples. The model developed includes an ensemble of nested models and uses the same volume of data and training iterations as a control in order to determine is relative effectiveness. The objective of this model is to determine if an ensemble structure can improve resilience, or the ability of the model to produce a correct output, even when adversarial data is input.

LITERATURE REVIEW

Neural Networks

A neural network is a computational model derived from the connection of physical neurons and their connective structure in the brain. While biological neurons are extremely complicated due to their many subcomponents, the artificial neurons used in a Neural Network only have two properties. The first property is known as weight, which is represented by a coefficient from 0 to 1. Weight denotes how powerful of an effect the artificial neuron will have on the neurons to which it connects. The second property is known as a bias, or a value that a neuron will have to exceed during calculation in order to determine whether or not it should "fire," or produce an output. In a neural network, unlike in a biological system, neurons can produce a gradient of output of 0 to 1 that denotes the strength of its firing.

These artificial neurons are organized in a network structure with three key layers, an example of which can be seen in figure 1.1 below. The input layer is a direct representation of the desired input mapped onto artificial neurons. How this mapping occurs varies widely depending on the type of data being tested (text, numerical values, images, etc.). The hidden layer is a section of artificial neurons that cannot be directly manipulated by the researcher. The artificial neurons are updated as the model improves in a process known as backpropagation, to be described shortly. Finally, there is the output layer. This layer consists of a simplified representation of the input, often in the form of a generalization. For example, classifications might include handwritten digits, where the input layer takes an image and the output layer returns a value from 0 to 9.

All of these properties operate within a loss function. The loss function minimizes the error of the model when compared to an optimum by altering the aforementioned weights and biases with each iteration. This loss function can operate indefinitely, but it inevitably reaches a minimum, at which point the model is considered "trained," (Nielsen, 2019).

Deep Neural Networks

The Machine Learning model focused on here is a Deep Neural Network (DNN). A DNN differs from a traditional neural network in a few ways. While it relies on the same base components (weights, biases, etc.) of a neural network, the model includes a hidden layer comprised of more intermediary steps, as shown below.

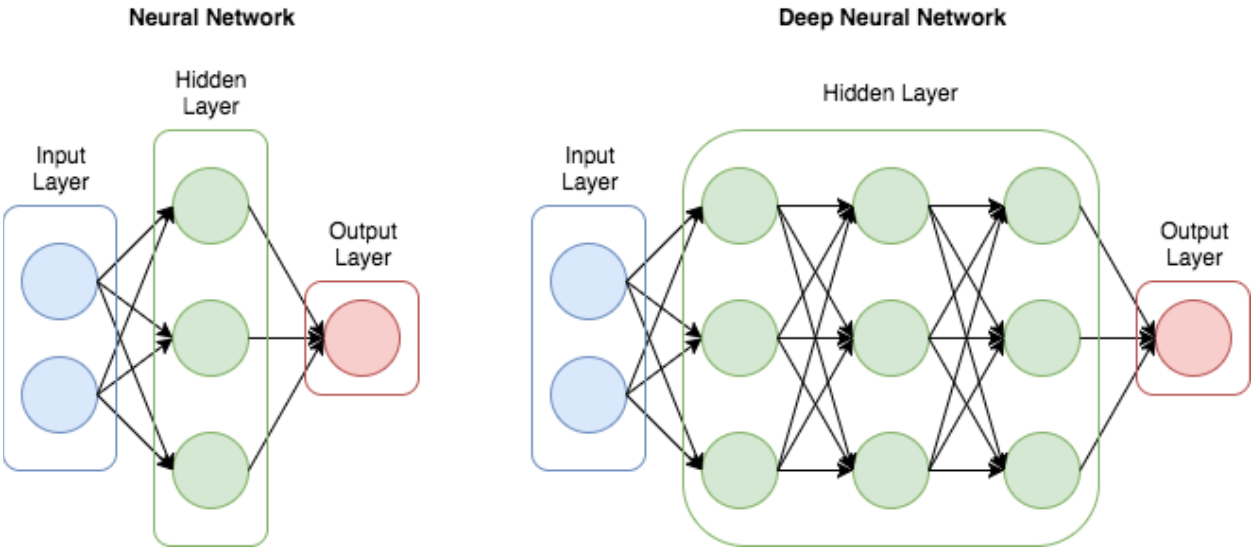

Figure 1: Hidden layers of NNs and DNNs

Image courtesy of Shuhei Kishi

In order to train a DNN, an additional step known as backreaction is used. The weights and biases of one artificial neuron rely on those of others in the network. When training the model, the weights and biases of each artificial neuron are updated in forward order. After each training example, backpropagation occurs, modifying the weights and biases of each artificial

neuron from the end to the start of the model (Nielsen, 2019). Backpropagation addresses the incompleteness of a one-way update by operating in reverse sequential order.

Adversarial Examples

Since Deep Learning is involved in many aspects of digital personalization today through things like recommended ads and news stories, attackers could benefit greatly from disrupting a DNN. The primary vector for targeting a DNN is known as an adversarial attack, in which adversarial examples are used on the model. Adversarial examples refer to artificially crafted inputs that cause the target model to produce an undesired output.

Figure 2: Horse with one-pixel perturbation (Su, 2019)

In response to the threat of adversarial attacks, red researchers have been focusing on potential attack vectors and blue researchers have uncovered potential defensive strategies (Madry, 2019). In a recent example, researchers were able to fool a DNN with an input image of a horse perturbed by a single pixel change. As seen in the image at right, the model is 99.9% confident that the image is of a frog (Su, 2019). From the defensive angle, attempts to obscure DNN structure have been unsuccessful in preventing attacks. However, researchers have laid out the process for generating reliable adversarial examples to aid in further study (Goodfellow, 2015).

Why adversarial examples are even possible to generate is still not entirely clear. As was recently shown, adversarial examples tend to exploit non-robust features, which are those that are indistinguishable to humans yet still provide desirable outputs (Ilyas, 2019). This distinction is an effect of the "black-box" nature of DNNs, which has led some researchers to address the

issue by elucidating the hidden layers of such a model. While research by A. Karpathy has led to a better understanding of the DNN model in its various forms through visualizing activations of individual neurons, this approach has yet to produce actionable defenses against adversarial examples.

Ensemble Learning

A promising method for addressing fragility in Deep Learning is the application of Ensemble Learning. Outlined by T.G. Deitterich in 2002, Ensemble Learning refers to running one learning algorithm multiple times with differentiation built into the design and having all iterations perform a voting sequence. Deitterich describes voting sequences as a way to determine the final output of the ensemble model, and the sequence itself can take any form so long as it narrows down the outputs of each nested model into a final output. An ensemble output can take the form of a classification or generative example produced by the model, depending on the model (Polikar, 2012). Built into this design is the stability in numbers. As more iterations run, the output is less susceptible to volatility. These iterations are forced to produce slightly different outputs through various means. The most common method is by taking several subsets of the initial dataset so that each have some overlapping and some unique data. While much of the data overlaps during each iteration, the differences formed lead to a slightly different output or hypothesis for every run (Deittereich, 2002).

While Deitterich's initial work was based on learning algorithms from decades prior which were used for a different purpose, research has since shown that ensemble learning can be applied to many different modern models including DNNs (Qui, 2014; Deng, 2014; Shaham, 2016; Yin, 2017). One important new application of ensemble learning is its use in defending against adversarial examples. Previous research implemented ensemble learning through several

different learning algorithms. An approach using an ensemble of varied weak classifiers against adversarial examples was implemented, but the results were ineffective against adversarial attacks. Nevertheless, this varied approach laid the groundwork for standardized voting across models trained on subsampled data (He, 2017).

MNIST

The data used will come from the MNIST dataset. The MNIST dataset consists of handwritten digits and was created by LeCun, Cortes, and Burges in 1999. It contains 70,000 images (60,000 for training and 10,000 for testing), each with a 28x28 pixel dimension. All of the images in this dataset fall into one of 10 categories, specifically, a digit from 0 to 9 (LeCun et. al. 1999).

The MNIST dataset lends itself to adversarial testing due to its visual nature. Namely, adversarial perturbations can be seen visually as shown in the images below. The first row of images represents originals from the MNIST dataset, and the second row represent their adversarial counterparts.

Figure 3: standard and adversarial MNIST images (He, 2017)

This visual representation assists in testing and demonstration because of the effectiveness of adversarial examples at "fooling" a DNN without a noticeable change in human perception. Another important note about these datasets is their age, 10 and 20 years respectively at the time of writing. These datasets have been used extensively in research over this period,

therefore contributing to a very firm understanding of both the images themselves and the expected results when using these datasets with various models (Goodfellow, 2015; He, 2017; Ilyas, 2019; Carlini and Wagner, 2017).

Generating Adversarial Examples

In order to generate adversarial examples to test against an experimental model, a combination of the original dataset and a specialized function is needed. While many different algorithms have been developed for this purpose, research has shown Carlini and Wagner's algorithm known as L_2 to be the most effective published attack at this time. L_2 achieves this goal by minimizing distortion of an image while maximizing potential for misclassification of the generated example, a work based on the original DeepFool algorithm created by Sayed-Mohsen in 2016. The results of this work are images that, to the human eye, are nearly indistinguishable from the original, and fool a DNN nearly 100% of the time (Carlini and Wagner, 2017).

L_2 works by exploiting edge cases, or local maxima, which are located just outside the barrier of classification. Edge cases are the result of the geometric nature of model training and are referred to as specific sets of labelled inputs which are placed by the model in a category of the wrong label. Through the function DeepFool, these edge cases of model classification can be found and then exploited to produce an output entirely contrary to human perception. Using DeepFool, adversarial examples will be generated for the MNIST dataset to be used during the testing phase of the DNN implementation (Seyed-Mohsen, 2016).

METHODOLOGY

Objectives

The ensemble approach aims to make the target model more robust by implementing a voting system in which subsidiary models trained on overlapping but unique subsets of the overall dataset function cooperatively to determine the highest confidence output. In order to test the effectiveness of this approach, several components are needed. In this section, each of these components and their relation to the overall objective are discussed.

Control Model Implementation

To test the effectiveness of an ensemble structure against adversarial attacks, a frame of reference for these attacks must first be established. To this end, a relatively standard (Madry, 2019; Su, 2019) model is used with the MNIST image set to classify the images with ten distinct labels ranging from 0 to 9. This model uses a sequential approach with 11 stacked layers. The first 3 layers consist of 2-dimensional convolutions followed by a 2-dimensional max pooling layer. Next, a dropout later is added before the input is flattened into a 1-dimensional vector for the final stage of the input processing. The final 5 layers consist of alternating dense and dropout functions. When tested with unmodified data, the control model correctly classifies the input image with an average of 98.38% accuracy.

Generating Adversarial Examples

The form of adversarial attack being used with this model is often referred to as targeted misclassification. This type of adversarial differs from its counterpart, non-targeted misclassification. Where non-targeted misclassification produces random image noise in the

input to reduce the confidence of the model on any one classification, targeted misclassification uses carefully generated noise to trick the model into classifying the input incorrectly. For comparison, figure 4 below shows an expected input for the model, a handwritten digit without any additional noise. In the MNIST dataset used in this research, a non-targeted misclassification would look like static to the human eye, as seen in figure 5. A targeted misclassification on the other hand (as seen in figure 6 below) looks like an eight to the human eye, yet the noise in the background produces an output classification of "2" in with extremely high confidence in a standard MNIST model.

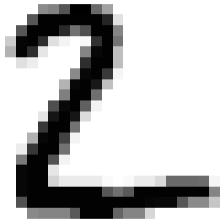

Figure 4: Base MNIST image

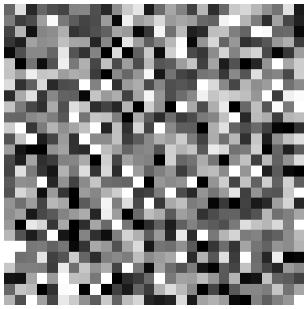

Figure 5: Non-targeted misclassifier

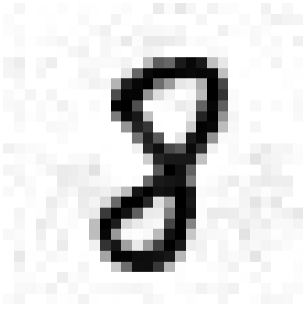

Figure 6: Targeted misclassifier

Both examples are unexpected for a model given its training data of standard handwritten digits. Mathematically, targeted misclassifiers can be produces using two terms, which this model implements using the Keras library for TensorFlow. Just like the model it is attacking, adversarial examples are generated using a loss function shown in figure 7 below. This function consists of two main terms. The term on the left is sufficient to produce a non-targeted misclassification, but the term on the right is necessary to produce a targeted misclassification. In this function, we aim to minimize both terms on the right as much as possible. The value of C represents the total loss of the algorithm, or the distance we are from the optimum configuration of the model. The $\vec{x}$ represents our target image in the form of a 784-dimensional vector.

Although our original input image is 28x28 pixels, a 784-dimensional vector is used by flattening the image for easier processing by the model.

$$C = \frac{1}{2}\|y_{goal} - \hat{y}(\vec{x})\|_2^2 + \lambda\|\vec{x} - x_{target}\|_2^2$$

Figure 7: Algorithm for generating targeted misclassifier

y_{goal} is the desired classification for our adversarial example, and $\hat{y}$ is the neural network model that we have trained. $\vec{x}$ is fed into this function and then subtracted from y_{goal}. The closer this result is to 0, the more accurate the model is. An additional penalty is created for both main terms outlined by squaring the output. This regularizes the optimum input by penalizing large coefficients which would look visually jarring to humans.

The second main term in figure 7, on the right and outlined in blue, is another minimization function. This term however focuses on the visual component of the output. While the first term discussed is enough to generate an example like in figure 5, the second term allows for an example like in figure 6 to be generated. The only new notation in this term is the x_{target} which represents what the desired output should look like (again this is being subtracted from the target image of $\vec{x}$ from before) and a λ in front of the term, which represents a coefficient we can adjust to change the relative importance of the term in relation to the first term.

Ensemble Model Implementation

The ensemble-based model acts as a meta-model which contains five nested models. Each nested model shares the same structural design to the control model discussed previously in the *Control Model Implementation* section. The difference between these new models lies in the

data that they are trained on. Every nested model is trained on an overlapping portion of the original dataset rather than on the entire dataset which the control model is trained on. This design creates a partial redundancy for each of the models as some of the training inputs are shared between them while others are not.

A benefit to this structure is that it requires the same amount of data as the control model. However, due to the increase in total training being done for the ensemble structure with its five nested models, this method is more computationally demanding. This increase in demand becomes less of an issue during use of the model, as running a single input through the ensemble model is only negligibly slower (0.00004 seconds) than running the same input through the control model.

Testing Adversarial Effectiveness

Once both the control and ensemble model have been implemented and tested on standard data, the final component is to test the effectiveness of generated adversarial examples against each of them. This is done by taking an identical batch of 10,000 generated examples and calculating the percentage of examples that cause a misclassification when processed through the ensemble model. The resulting percentage is used to compare against the percentage of misclassifications of the control model.

Development

The first step in development of this ensemble model was gaining experience and expertise in the tools used for this area of research, namely in Jupyter Notebooks using TensorFlow and Keras. This expertise was gained by implementing the control model outlined above in the *Control Model Implementation* section. Creation of the control model can be broken

down into three stages, with the first being the initial configuration of the model and the integration of the MNIST image set. Once completed, the model was trained on the dataset. Finally, one portion of the dataset was preemptively set aside by the Keras library in order to validate the model. To ensure overfitting had not occurred, another portion was set aside by Keras to test against so a measurement of accuracy could be obtained for the model.

With the control model was completed, the next step was visually representing original data by displaying a given image from the MNIST set using code. This was done by instantiating and utilizing matplotlib, a graphical library for Python which generates the image in its original 28 by 28-pixel grid. A mask was applied to the image, which is normally represented in black and white, in order to help better display the adversarial perturbations which would be developed later on. However, to increase visibility of the perturbations represented in targeted misclassifiers, this mask was changed to better display pixels which would normally be more difficult to see to the naked eye. An example of such an image with the mask can be seen below in figure 8.

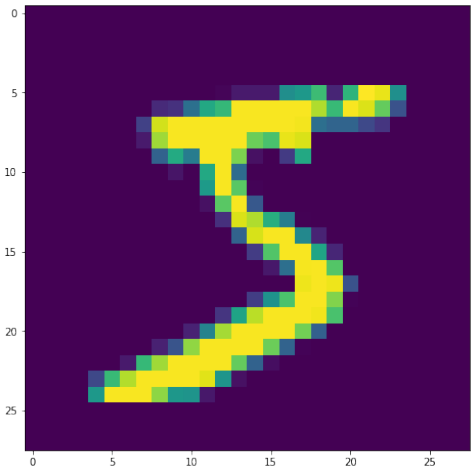

Figure 8: MNIST image with color mask

Once the original image was represented, work began on creating the adversarial perturbations which, ideally, create a targeted misclassification. This type of misclassification disrupts the prediction of a model while remaining visually quite similar to the originally image. This component of the development took the longest amount of time to complete as it was not a standard function of the TensorFlow library and required a deeper conceptual understanding of the calculations at work.

After research on generating adversarial misclassifications was completed, the equation in figure 7 in the section *Generating Adversarial Examples* was implemented in code. This code generates perturbations using a given input image like the one in figure 8 above. These perturbations are calculated using a gradient descent method alongside the already trained control model to determine what changes should be made to maximize the likelihood that the model will misclassify the perturbed image. Finally, an additional parameter is multiplied by these changes in order to regulate the effects so that the image does not become to distorted for humans to understand. One resulting image from this code can be seen below in figure 9.

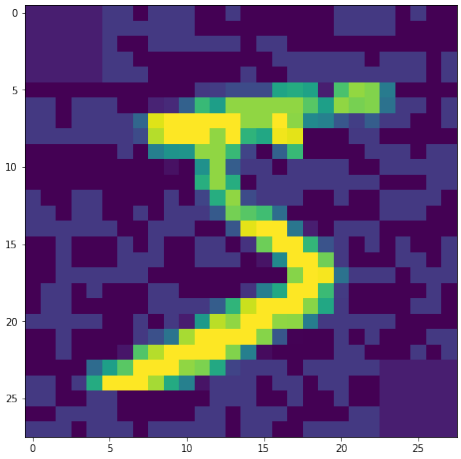

Figure 9: Perturbations added to original MNIST image

When compared against figure 8, this image shows how the perturbations affect the image as seen by human eyes.

While generating a single adversarial example helps readers gain an intuitive sense for how perturbations affect an input image, many thousands are needed in order to reliably test the effectiveness of the adversarial algorithm against a given model. For this reason, the next step in development was generating a new segment of data, similar to the test or validation segments mentioned earlier, which were tested against the model for a new measure of accuracy. Creating and testing a batch of these adversarial examples was done using object-oriented programming principles by first converting the adversarial algorithm to a function. Next, the function was called 10,000 times, with each call's output assigned to an index in a list. Each of these outputs represents an adversarial image. Finally, the accuracy of the model when tested against these adversarial images was generated using the same method used to generate the base accuracy of the model.

The last portion of development was the creation and implementation of the ensemble model. While the overall structure of this model introduced new challenges, the nested models it consisted of were each identical in all but training data to the control model. For this reason, the same object-oriented methods were used to create a function out of the configuration code of the control model which expects a single parameter of input data to be passed to the function upon calling it. This function was then called for each of the five nested models. Each model was passed a different subset of the overall training section of the MNIST dataset. The first model contained images 1-36,000, the second contained images 6,001-42,000, the third contained 12,001-48,000, the fourth contained 18,001-54,000, and the fifth contained 24,001-60,000. Through this subsampling process, all five models share at least 20% of the same images, but

through the process of training each nested model, the output for each nested model is unique from each other.

Once each of the nested models were trained, validated, and tested using the same methods as the control model, the final portion of code was implemented in the form of voting to create the full ensemble model. While changing the structure of nested models would require additional complexity for the voting section, a simple averaging of output predictions for each of the 5 models is sufficient for the purposes of this research. Finally, the ensemble model was tested against the previously generated batch of adversarial examples, and the results were recorded.

RESULTS & DISCUSSION

Recording Procedures

After the models were configured and implemented, they were trained on specific sections of the training set. The control model was trained using the entire initial dataset of 60,000 images of handwritten digits cycled over 20 total epochs. The ensemble model was trained on several subsets of the data as described in the *Development* section of the previous chapter. In summary, each of the nested models were trained on a total of 36,000 images from the original dataset in overlapping subsets each containing 60% of the original training set. To compensate for the complex structure of 5 nested models when compared to the single control model, each nested model was only trained for a total of 4 epochs, $1/5^{th}$ of the total epochs used to train the control model.

The control and ensemble models were then tested against several test sets. First, each model was tested against a standard test set of 10,000 images set aside from the original MNIST image set. Next, the models were tested against various forms of adversarial image sets. The first set of images were generated to target the control model, which proved to be biased in favor of the ensemble model. To address this imbalance, two corresponding adversarial test sets were generated. The control model was tested against adversarial examples generated to target said model. The ensemble model was tested against an image set comprised of 5 equal portions of adversarial examples, with each portion generated to target one of the nested models that make up the ensemble.

While this new testing procedure addresses the initial testing bias, another final testing procedure was conducted to ensure the validity of the results that were collected. In this

procedure, a comprehensive set of adversarial examples was generated for the control and the ensemble model to be tested on. The total set consisted of 10,000 adversarial images, 5,000 of which were targeted towards the control model. The other 5,000 images were composed of 1,000 images targeted at each of the 5 nested models. In this way, the total adversarial image set was intended to provide a balanced testing procedure that both models could be tested against.

Table 1: Data gathering procedures

Procedure	Test Set Size	Target Model	Description
Base Procedure	10,000 images	Both	Standard non-adversarial images to test model accuracy for both models
Adversarial Procedure 1	10,000 images	Control	Initial test for model robustness with a bias against the control model
Adversarial Procedure 2	2 sets of 5,000 images	Both separately	5,000 images generated to target the control model. Another 5,000 images generated to target the ensemble model, with 1,000 of the total image set targeting each individual nested model
Adversarial Procedure 3	10,000 images	Both combined	The two image sets from the previous procedure were combined, and that entire dataset was tested against both the control and the ensemble models

Training Data

While training both the control and ensemble models, data was collected to ensure they were properly configured for the experiment. This was done by setting up each model to be tracked. As training occurred, data on the loss and accuracy of a given model was collected in a standard interval known as an epoch. An epoch in our testing is one full cycle through of the initial set of training images. During data gathering, the accuracy for a given epoch is recorded continuously throughout the epoch. However, in the figures below, the data points represent the accuracy and loss of a model at the end of a given epoch. Additionally, the first epoch in this testing procedure is labelled as epoch 0. In figure 10, the accuracy of the control model during

both training and testing phases is shown. The relative plateau at the end of the graph validates the decision to configure the model training for 20 epochs.

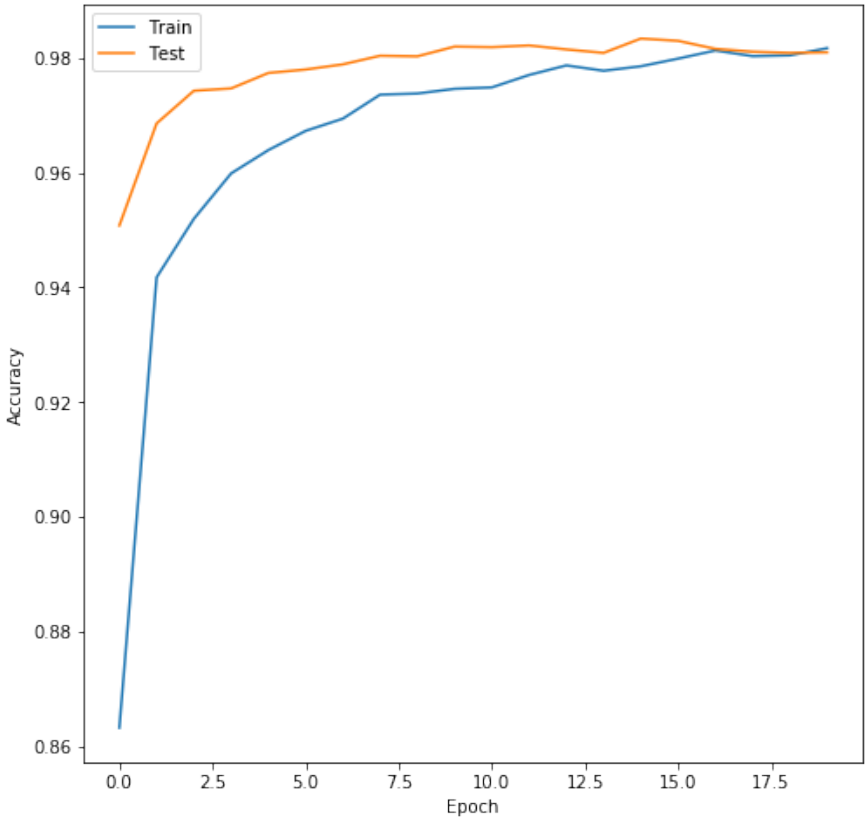

Figure 10: Control model accuracy

Figure 11, shown below, displays the loss for the control model, also during both phases of preparation.

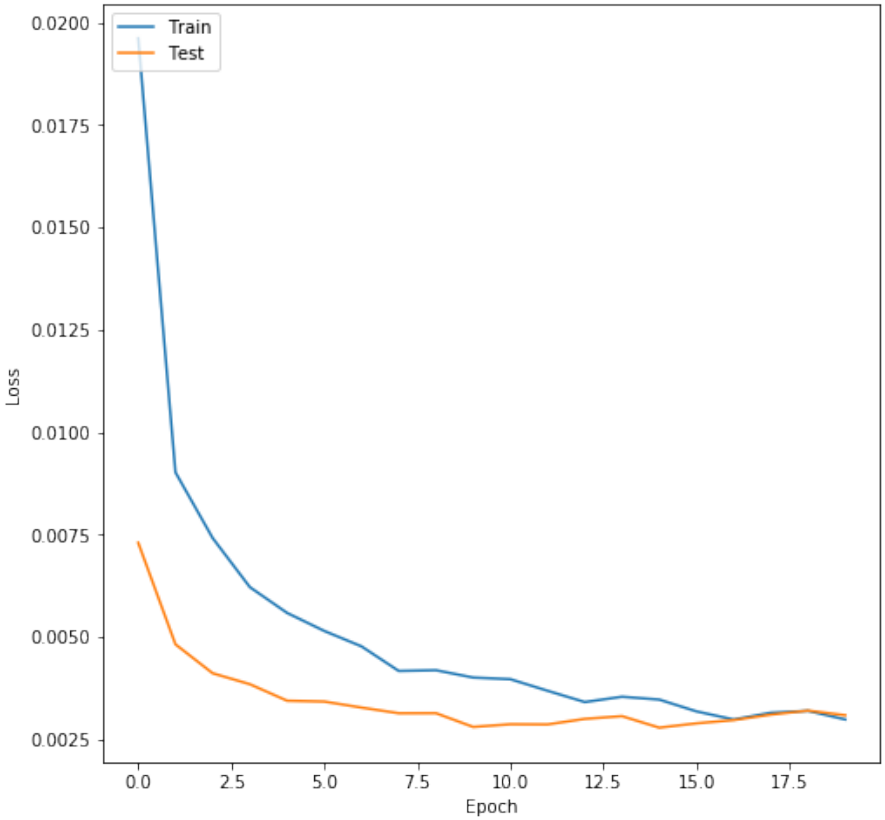

Figure 11: Control model loss

The training procedure for the ensemble model was notably different, as each of the nested models were trained on a significantly reduced 4 epochs out of the 20 epochs used for the control. This was done to constrain the ensemble model so that it took no longer than the control model to train. Due to this training process, only 4 data points were taken for the accuracy and loss of each nested model. This sequential data of loss and accuracy for the nested models was averaged, point by point, into an overall measure of the same attributes. Shown in figure 12 below is the resulting averaged data for accuracy in both training and testing phases.

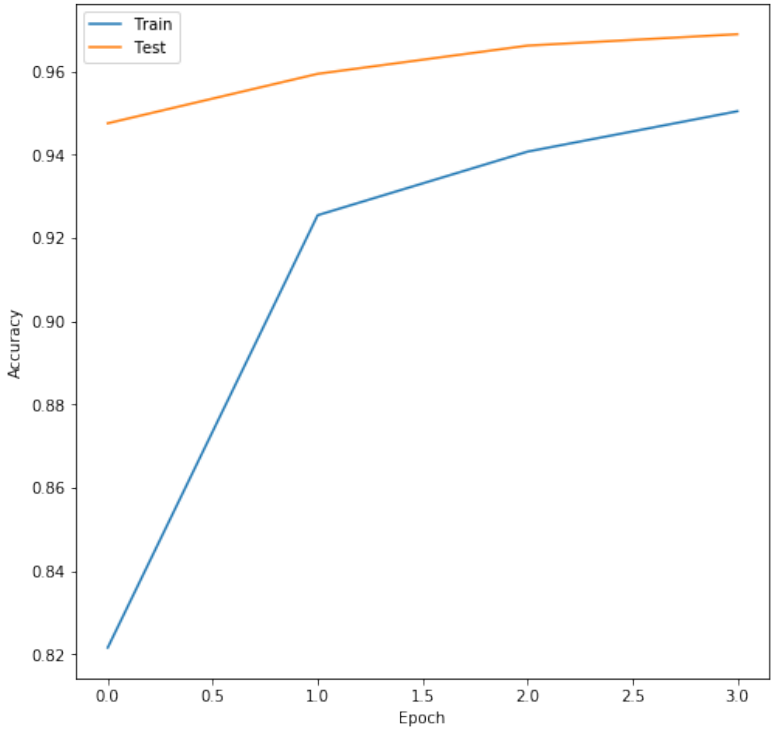

Figure 12: Ensemble model accuracy

Figure 13 displays the loss of the ensemble model during training and testing, collected and averaged using the same method which was used for the accuracy of the ensemble model.

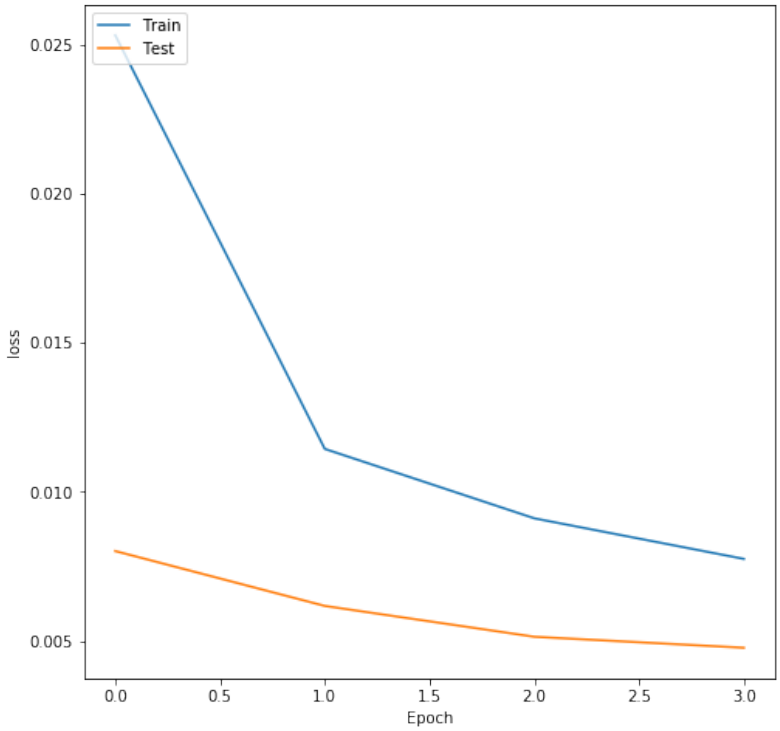

Figure 13: Ensemble model loss

Result Data

Once configured and trained, both models were predicted to perform well on standard testing as the images within the test set are similar in form to those the model processed during training. The results shown in Table 1 align with that assumption, with the ensemble model performing slightly worse with just over a 0.5% loss in accuracy when tested against standard examples. Notably, each of the nested models perform slightly worse still, which is likely due to the decrease in both total training data as well as a strong decrease in the total epochs being trained for.

Table 2: Standard testing results

Model	**Loss**	**Accuracy**
Control	0.0029	0.9828
Ensemble	0.0036	0.9769
- Nested Model 1	0.0045	0.9708
- Nested Model 2	0.0050	0.9667
- Nested Model 3	0.0046	0.9697
- Nested Model 4	0.0044	0.9709
- Nested Model 5	0.0047	0.9681

The main component being analyzed is the accuracy of these models because they determine how often the model is likely to misclassify an input. Thus, when tested against the initial generated set of 10,000 adversarial examples, the results shown highly support the robustness of the ensemble model as seen in Table 2. The control model only managed about 30% accuracy against the adversarial set while the ensemble correctly classified over 90% of examples. However, a closer look at the accuracies of each nested model shows the bias of this initial testing procedure. While the ensemble was predicted to perform better than the control, the nested models were predicted to only perform as well as the control, if not slightly worse due to reduced training time. This prediction turned out not to be the case due to the bias in initial testing procedures as discussed in the *Recording Procedures* section of this chapter.

Table 3: Initial adversarial testing

Model	**Loss**	**Accuracy**
Control	0.1225	0.2968
Ensemble	0.0150	0.9128
- Nested Model 1	0.0166	0.9070
- Nested Model 2	0.0164	0.9120
- Nested Model 3	0.0160	0.8960
- Nested Model 4	0.0211	0.8598
- Nested Model 5	0.0210	0.8634

The next testing procedure corrected for this error, yet still showed a massive increase in accuracy from the control model to the ensemble model as can be seen in Table 3. In this procedure, the testing set for the control model remained unchanged, as did the results. The Ensemble model however performed quite differently as a result of the targeted compound adversarial set. The nested model accuracies and losses shown below were produced by evaluating each nested model along with its 1,000-image segment of the compound adversarial set, and the ensemble model results were generated by evaluating the ensemble model against the full collection of 5,000 targeted images created by combining each 1,000-image segment.

Table 4: Targeted adversarial results

Model	**Loss**	**Accuracy**
Control	0.1225	0.2968
Ensemble	0.0355	0.7940
- Nested Model 1	0.1035	0.3130
- Nested Model 2	0.0750	0.5080
- Nested Model 3	0.0684	0.5410
- Nested Model 4	0.0806	0.4870
- Nested Model 5	0.1244	0.2710

In an effort to further evaluate the resiliency of the ensemble model against adversarial examples when compared to the control model, the final procedure discussed in the previous section was implemented. In Table 5 below, the adversarial set of 5,000 images used against the control was combined with the compound adversarial set used to generate the results in Table 3. This composite dataset totals 10,000 images and was used against both the control and ensemble models to produce yet another measure of effectiveness against adversarial examples. These results show that even with a dataset covering all of the models, the ensemble model still performs over 35% better in terms of accuracy. When considering the size of the test set, this increase in accuracy is statistically significant.

While each of the nested models were also tested against the composite adversarial dataset, the results recorded for them likely suffer from the same bias discussed in the initial testing procedure since the total images targeted at them make up only 10% of the overall dataset.

Table 5: Composite adversarial testing

Model	**Loss**	**Accuracy**
Control	0.0847	0.5023
Ensemble	0.0253	0.8534
- Nested Model 1	0.1035	0.3130
- Nested Model 2	0.0750	0.5080
- Nested Model 3	0.0684	0.5410
- Nested Model 4	0.0806	0.4870
- Nested Model 5	0.1244	0.2710

Regardless of the testing procedures, the ensemble model performed with significantly higher accuracy than the control with the same total training time and data.

CONCLUSION

The share of total research publications taken up by AI research has increased over 300% in the past two decades, leading AI to become a dominant field of technical research (Stanford, 2019). Despite this massive increase in publications, the security of these models has only recently become a serious concern for researchers.

As people rely more and more on these systems for simple preference matching in the form of recommendation systems by Facebook and Google or for life endangering activities such as driving an autonomous vehicle like a Tesla, it is not an understatement to say that lives can be saved or lost as a result of AI security research. While AI systems are becoming more readily adopted and accepted globally (Liu, 2019), this lack of security can drastically worsen perception of AI as more and more examples of simple alterations in input cause dire consequences (Winder, 2020).

While researchers have recently developed ways to reduce the impact of adversarial examples on model accuracy, few approach the problem with constraints that this increase in security must not come at the cost of training time or data needed. This research shows that a structural change in a deep learning model can reduce the effectiveness of adversarial examples while satisfying these constraints.

The intention of this research was to confront the gap in research surrounding structural security in deep learning. A targeted adversarial generation algorithm was implemented algorithmically and used to generate further data for both a control and an ensemble model to be tested on. Next, an ensemble approach to algorithm structure was analyzed against a traditional model of the same layer configuration. An ensemble of nested models trained on subsamples of

the overall dataset was then constructed. Both models were tested on a suite of test data meant to target the structural weaknesses of each model.

Test data was collected on the models through three different routines. First, a test set generated for best effect against the control model was tested, showing preference towards the ensemble model, but also raising questions of bias. Next, two comprehensive test sets were generated and compiled to more accurately reflect the effectiveness of the experimental model against adversarial examples. The first test was composed of the original test set alongside an additional test set generated to target the ensemble model. The second test was composed of adversarial examples targeted at both the control model as well as each of the nested models in the ensemble structure.

Further research on ensemble structured models can reveal more about the impact of structural combinations on a given model's resilience against adversarial examples. One such experiment could test the effectiveness of further segmenting the ensemble model into smaller nested models. Such an experiment would likely deepen understanding of the impact ensemble size has on adversarial resilience. Another example could involve an additional dense layer that takes the nested ensemble models as input and outputs the ensemble output. This model will likely develop specific weights and biases to account for potential imbalances in nested model reliability. Finally, additional research could illuminate the impact of including varied optimization algorithms for training the nested models rather than using the same one for each.

APPENDIX

Source Code

```
#Imports
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist, cifar10, cifar100
from tensorflow.keras import Sequential
from tensorflow.keras.callbacks import LambdaCallback
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout, Dense,
Flatten, Activation

import numpy as np
import random

import matplotlib.pyplot as plt
import matplotlib as mpl

mpl.rcParams['figure.figsize'] = (8, 8)
mpl.rcParams['axes.grid'] = False

#Define Config Variables
numEpochs = 20
numAdversaries = 1000

#Inital Data Processing
(x_train, y_train), (x_test, y_test) = mnist.load_data()

labels = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven', '
eight', 'nine']

img_rows, img_cols, channels = 28, 28, 1
num_classes = 10

x_train = x_train / 255
x_test = x_test / 255

x_train = x_train.reshape((-1, img_rows, img_cols, channels))
x_test = x_test.reshape((-1, img_rows, img_cols, channels))

y_train = tf.keras.utils.to_categorical(y_train, num_classes)
```

```
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

print("Data shapes", x_test.shape, y_test.shape, x_train.shape, y_train.sh
ape)

#Create Model
def create_model():
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3), strides=(3, 3), padding='same
', activation='relu', input_shape=(img_rows, img_cols, channels)))
    model.add(Conv2D(64, kernel_size=(3, 3), strides=(3, 3), padding='same
', activation='relu'))
    model.add(Conv2D(64, kernel_size=(3, 3), strides=(3, 3), padding='same
', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(32))
    model.add(Dropout(0.2))
    model.add(Dense(32))
    model.add(Dropout(0.2))
    model.add(Dense(num_classes, activation='softmax'))

    model.compile(optimizer='adam', loss='mse', metrics=['accuracy'])

    return model

model = create_model()

#Fit Model
cR = model.fit(x_train, y_train,
          batch_size=32,
          epochs=numEpochs,
          validation_data=(x_test, y_test))

#Evaluate Control Model
print("Base accuracy:", model.evaluate(x=x_test, y=y_test, verbose=0))

#Split MNIST Data into Overlapping Subsets
def stagger_split(x, k, ratio):
    N = x.shape[0]
    chunksize = int(N * ratio)
    stagger = int((N - chunksize) / (k - 1))

    splits = []
```

```
    for i in range(k):
        idx = i * stagger
        splits.append(x[idx:idx + chunksize])

    return splits

x1,x2,x3,x4,x5 = stagger_split(x_train, k=5, ratio=0.6)
y1,y2,y3,y4,y5 = stagger_split(y_train, k=5, ratio=0.6)

#Create Nested Ensemble Model
model1 = create_model()
model2 = create_model()
model3 = create_model()
model4 = create_model()
model5 = create_model()

#Configure Epoch Size
if (numEpochs == 1):
  nestEpochs = 1
else:
  nestEpochs = 4

m1R = model1.fit(x1, y1,
          batch_size=32,
          epochs=nestEpochs,
          validation_data=(x_test, y_test))

m2R = model2.fit(x2, y2,
          batch_size=32,
          epochs=nestEpochs,
          validation_data=(x_test, y_test))

m3R = model3.fit(x3, y3,
          batch_size=32,
          epochs=nestEpochs,
          validation_data=(x_test, y_test))

m4R = model4.fit(x4, y4,
          batch_size=32,
          epochs=nestEpochs,
          validation_data=(x_test, y_test))

m5R = model5.fit(x5, y5,
          batch_size=32,
          epochs=nestEpochs,
```

```
          validation_data=(x_test, y_test))

#Evaluate each nested model on control data
print("Base accuracy:", model1.evaluate(x=x_test, y=y_test, verbose=0))
print("Base accuracy:", model2.evaluate(x=x_test, y=y_test, verbose=0))
print("Base accuracy:", model3.evaluate(x=x_test, y=y_test, verbose=0))
print("Base accuracy:", model4.evaluate(x=x_test, y=y_test, verbose=0))
print("Base accuracy:", model5.evaluate(x=x_test, y=y_test, verbose=0))

#Nest Models in Ensemble
inputs = keras.Input(shape=(28, 28, 1))

m1 = model2(inputs)
m2 = model1(inputs)
m3 = model3(inputs)
m4 = model4(inputs)
m5 = model5(inputs)

outputs = layers.average([m1, m2, m3, m4, m5])
ensemble_model = keras.Model(inputs=inputs, outputs=outputs)

#Compile Ensemble Model
ensemble_model.compile(optimizer='adam', loss='mse', metrics=['accuracy'])

#Evaluate Ensemble Model
print("Ensemble accuracy:", ensemble_model.evaluate(x=x_test, y=y_test, ve
rbose=0))

index = 0
image = x_train[index]
image_label = y_train[index]

adversarial = image

if channels == 1:
    plt.imshow(adversarial.reshape((img_rows, img_cols)))
else:
    plt.imshow(adversarial.reshape((img_rows, img_cols, channels)))
plt.show()

print("Base Label:",labels[model.predict(image.reshape((1, img_rows, img_c
ols, channels))).argmax()])

#Generate Adversarial Example for Given Image
def adversarial_pattern(image, label):
```

```
    image = tf.cast(image, tf.float32)

    with tf.GradientTape() as tape:
        tape.watch(image)
        prediction = model(image)
        loss = tf.keras.losses.MSE(label, prediction)

    gradient = tape.gradient(loss, image)

    signed_grad = tf.sign(gradient)

    return signed_grad

#change index for different image
index = 0
image = x_train[index]
image_label = y_train[index]

perturb_effect = 0.1

perturbations = adversarial_pattern(image.reshape((1, img_rows, img_cols,
channels)), image_label).numpy()
adversarial = image + perturbations * perturb_effect


if channels == 1:
    plt.imshow(adversarial.reshape((img_rows, img_cols)))
else:
    plt.imshow(adversarial.reshape((img_rows, img_cols, channels)))
plt.show()

print("Base Label:",labels[model.predict(image.reshape((1, img_rows, img_c
ols, channels))).argmax()])
print("Predicted:",labels[model.predict(adversarial).argmax()])

#Ensemble adversarial generation
def adversarial_pattern(image, label):
    image = tf.cast(image, tf.float32)

    with tf.GradientTape() as tape:
        tape.watch(image)
        prediction = model(image)
        loss = tf.keras.losses.MSE(label, prediction)

    gradient = tape.gradient(loss, image)
```

```
    signed_grad = tf.sign(gradient)

    return signed_grad

    #Ensemble adversarial generation
def adversarial_pattern1(image, label):
    image = tf.cast(image, tf.float32)

    with tf.GradientTape() as tape:
        tape.watch(image)
        prediction = model1(image)
        loss = tf.keras.losses.MSE(label, prediction)

    gradient = tape.gradient(loss, image)

    signed_grad = tf.sign(gradient)

    return signed_grad

    #Ensemble adversarial generation
def adversarial_pattern2(image, label):
    image = tf.cast(image, tf.float32)

    with tf.GradientTape() as tape:
        tape.watch(image)
        prediction = model2(image)
        loss = tf.keras.losses.MSE(label, prediction)

    gradient = tape.gradient(loss, image)

    signed_grad = tf.sign(gradient)

    return signed_grad

    #Ensemble adversarial generation
def adversarial_pattern3(image, label):
    image = tf.cast(image, tf.float32)

    with tf.GradientTape() as tape:
        tape.watch(image)
        prediction = model3(image)
        loss = tf.keras.losses.MSE(label, prediction)

    gradient = tape.gradient(loss, image)
```

```
    signed_grad = tf.sign(gradient)

    return signed_grad

    #Ensemble adversarial generation
def adversarial_pattern4(image, label):
    image = tf.cast(image, tf.float32)

    with tf.GradientTape() as tape:
        tape.watch(image)
        prediction = model4(image)
        loss = tf.keras.losses.MSE(label, prediction)

    gradient = tape.gradient(loss, image)

    signed_grad = tf.sign(gradient)

    return signed_grad

    #Ensemble adversarial generation
def adversarial_pattern5(image, label):
    image = tf.cast(image, tf.float32)

    with tf.GradientTape() as tape:
        tape.watch(image)
        prediction = model5(image)
        loss = tf.keras.losses.MSE(label, prediction)

    gradient = tape.gradient(loss, image)

    signed_grad = tf.sign(gradient)

    return signed_grad

def generate_adversarials(batch_size):
    while True:
        x = []
        y = []
        for batch in range(batch_size):
            N = random.randint(0, 100)

            label = y_train[N]
            image = x_train[N]
```

```
            perturbations = adversarial_pattern(image.reshape((1, img_rows
, img_cols, channels)), label).numpy()

            #set modifier to limit perturbation effect
            epsilon = 0.1
            adversarial = image + perturbations * epsilon

            x.append(adversarial)
            y.append(y_train[N])


        x = np.asarray(x).reshape((batch_size, img_rows, img_cols, channel
s))
        y = np.asarray(y)

        yield x, y

def generate_adversarials1(batch_size):
    while True:
        x = []
        y = []
        for batch in range(batch_size):
            N = random.randint(0, 100)

            label = y_train[N]
            image = x_train[N]

            perturbations = adversarial_pattern1(image.reshape((1, img_row
s, img_cols, channels)), label).numpy()

            #set modifier to limit perturbation effect
            epsilon = 0.1
            adversarial = image + perturbations * epsilon

            x.append(adversarial)
            y.append(y_train[N])


        x = np.asarray(x).reshape((batch_size, img_rows, img_cols, channel
s))
        y = np.asarray(y)

        yield x, y

def generate_adversarials2(batch_size):
```

```
    while True:
        x = []
        y = []
        for batch in range(batch_size):
            N = random.randint(0, 100)

            label = y_train[N]
            image = x_train[N]

            perturbations = adversarial_pattern2(image.reshape((1, img_row
s, img_cols, channels)), label).numpy()

            #set modifier to limit perturbation effect
            epsilon = 0.1
            adversarial = image + perturbations * epsilon

            x.append(adversarial)
            y.append(y_train[N])


        x = np.asarray(x).reshape((batch_size, img_rows, img_cols, channel
s))
        y = np.asarray(y)

        yield x, y

def generate_adversarials3(batch_size):
    while True:
        x = []
        y = []
        for batch in range(batch_size):
            N = random.randint(0, 100)

            label = y_train[N]
            image = x_train[N]

            perturbations = adversarial_pattern3(image.reshape((1, img_row
s, img_cols, channels)), label).numpy()

            #set modifier to limit perturbation effect
            epsilon = 0.1
            adversarial = image + perturbations * epsilon

            x.append(adversarial)
            y.append(y_train[N])
```

```
        x = np.asarray(x).reshape((batch_size, img_rows, img_cols, channel
s))
        y = np.asarray(y)

        yield x, y

def generate_adversarials4(batch_size):
    while True:
        x = []
        y = []
        for batch in range(batch_size):
            N = random.randint(0, 100)

            label = y_train[N]
            image = x_train[N]

            perturbations = adversarial_pattern4(image.reshape((1, img_row
s, img_cols, channels)), label).numpy()

            #set modifier to limit perturbation effect
            epsilon = 0.1
            adversarial = image + perturbations * epsilon

            x.append(adversarial)
            y.append(y_train[N])


        x = np.asarray(x).reshape((batch_size, img_rows, img_cols, channel
s))
        y = np.asarray(y)

        yield x, y

def generate_adversarials5(batch_size):
    while True:
        x = []
        y = []
        for batch in range(batch_size):
            N = random.randint(0, 100)

            label = y_train[N]
            image = x_train[N]
```

```
            perturbations = adversarial_pattern5(image.reshape((1, img_row
s, img_cols, channels)), label).numpy()

            #set modifier to limit perturbation effect
            epsilon = 0.1
            adversarial = image + perturbations * epsilon

            x.append(adversarial)
            y.append(y_train[N])


        x = np.asarray(x).reshape((batch_size, img_rows, img_cols, channel
s))
        y = np.asarray(y)

        yield x, y


#Generate Batch of Adversarials
x_adversarial_test, y_adversarial_test = next(generate_adversarials(numAdv
ersaries*5))
x_adversarial_test1, y_adversarial_test1 = next(generate_adversarials1(num
Adversaries))
x_adversarial_test2, y_adversarial_test2 = next(generate_adversarials2(num
Adversaries))
x_adversarial_test3, y_adversarial_test3 = next(generate_adversarials3(num
Adversaries))
x_adversarial_test4, y_adversarial_test4 = next(generate_adversarials4(num
Adversaries))
x_adversarial_test5, y_adversarial_test5 = next(generate_adversarials5(num
Adversaries))

print(x_adversarial_test.size)
print(y_adversarial_test.size)

x_ad_ensemble = np.concatenate((x_adversarial_test1, x_adversarial_test2,
x_adversarial_test3, x_adversarial_test4, x_adversarial_test5), axis=0)
y_ad_ensemble = np.concatenate((y_adversarial_test1, y_adversarial_test2,
y_adversarial_test3, y_adversarial_test4, y_adversarial_test5), axis=0)

x_total = np.concatenate((x_ad_ensemble, x_adversarial_test), axis=0)
y_total = np.concatenate((y_ad_ensemble, y_adversarial_test), axis=0)

print(x_ad_ensemble.size)
print(y_ad_ensemble.size)
```

```
print(x_total.size)
print(y_total.size)

#Evaluate each nested model on control data
print("m1 - Base accuracy:", model1.evaluate(x=x_test, y=y_test, verbose=0
))
print("m2 - Base accuracy:", model2.evaluate(x=x_test, y=y_test, verbose=0
))
print("m3 - Base accuracy:", model3.evaluate(x=x_test, y=y_test, verbose=0
))
print("m4 - Base accuracy:", model4.evaluate(x=x_test, y=y_test, verbose=0
))
print("m5 - Base accuracy:", model5.evaluate(x=x_test, y=y_test, verbose=0
))

#Evaluate on adversarial data
print("m1 - Adversarial accuracy:", model1.evaluate(x=x_ad_ensemble, y=y_a
d_ensemble, verbose=0))
print("m2 - Adversarial accuracy:", model2.evaluate(x=x_ad_ensemble, y=y_a
d_ensemble, verbose=0))
print("m3 - Adversarial accuracy:", model3.evaluate(x=x_ad_ensemble, y=y_a
d_ensemble, verbose=0))
print("m4 - Adversarial accuracy:", model4.evaluate(x=x_ad_ensemble, y=y_a
d_ensemble, verbose=0))
print("m5 - Adversarial accuracy:", model5.evaluate(x=x_ad_ensemble, y=y_a
d_ensemble, verbose=0))

#Control - Calculate Base Accuracy
print(" Control - Base accuracy:", model.evaluate(x=x_test, y=y_test, verb
ose=0))

#Ensemble - Calculate Base Accuracy
print("Ensemble - Base accuracy:", ensemble_model.evaluate(x=x_test, y=y_t
est, verbose=0))

print("Initial Set Targeted at Control")
#Calculate Inital AdAcc
print(" Control - Adversarial accuracy:", model.evaluate(x=x_adversarial_t
est, y=y_adversarial_test, verbose=0))
print("Ensemble - Adversarial accuracy:", ensemble_model.evaluate(x=x_adve
rsarial_test, y=y_adversarial_test, verbose=0))

print("")
print("5,000img Targeted Set for Each Model")
#Calculate Targeted AdAcc
```

```
print(" Control - Targeted accuracy:", model.evaluate(x=x_adversarial_test
, y=y_adversarial_test, verbose=0))
print("Ensemble - Targeted accuracy:", ensemble_model.evaluate(x=x_ad_ense
mble, y=y_ad_ensemble, verbose=0))

print("")
print("10,000img Total Adversarial Set")
#Calculate Total AdAcc
print(" Control - TotalTest accuracy:", model.evaluate(x=x_total, y=y_tota
l, verbose=0))
print("Ensemble - TotalTest accuracy:", ensemble_model.evaluate(x=x_total,
 y=y_total, verbose=0))

# list all data in history
print(cR.history.keys())
# summarize history for accuracy
plt.plot(cR.history['accuracy'])
plt.plot(cR.history['val_accuracy'])
plt.title('Control Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(cR.history['loss'])
plt.plot(cR.history['val_loss'])
plt.title('Control Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()


#Combine data from nested models for composite graph
compositeAccTrain = [((a + b + c + d + e)/5) for a, b, c, d, e in zip(m1R.
history['accuracy'], m2R.history['accuracy'], m3R.history['accuracy'], m4R
.history['accuracy'], m5R.history['accuracy'])]
compositeAccTest = [((a + b + c + d + e)/5) for a, b, c, d, e in zip(m1R.h
istory['val_accuracy'], m2R.history['val_accuracy'], m3R.history['val_accu
racy'], m4R.history['val_accuracy'], m5R.history['val_accuracy'])]

compositeLossTrain = [((a + b + c + d + e)/5) for a, b, c, d, e in zip(m1R
.history['loss'], m2R.history['loss'], m3R.history['loss'], m4R.history['l
oss'], m5R.history['loss'])]
```

```
compositeLossTest = [((a + b + c + d + e)/5) for a, b, c, d, e in zip(m1R.
history['val_loss'], m2R.history['val_loss'], m3R.history['val_loss'], m4R
.history['val_loss'], m5R.history['val_loss'])]

# summarize history for accuracy
plt.plot(compositeAccTrain)
plt.plot(compositeAccTest)
plt.title('Ensemble Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

# summarize history for loss
plt.plot(compositeLossTrain)
plt.plot(compositeLossTest)
plt.title('Ensemble Model Loss')
plt.ylabel('loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

#List all data for nested models

# list all data in history
print(m1R.history.keys())
# summarize history for accuracy
plt.plot(m1R.history['accuracy'])
plt.plot(m1R.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(m1R.history['loss'])
plt.plot(m1R.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()


# list all data in history
```

```
print(m1R.history.keys())
# summarize history for accuracy
plt.plot(m1R.history['accuracy'])
plt.plot(m1R.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(m1R.history['loss'])
plt.plot(m1R.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()


# list all data in history
print(m2R.history.keys())
# summarize history for accuracy
plt.plot(m2R.history['accuracy'])
plt.plot(m2R.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(m2R.history['loss'])
plt.plot(m2R.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()


# list all data in history
print(m3R.history.keys())
# summarize history for accuracy
plt.plot(m3R.history['accuracy'])
plt.plot(m3R.history['val_accuracy'])
plt.title('Model Accuracy')
```

```
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(m3R.history['loss'])
plt.plot(m3R.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()


# list all data in history
print(m4R.history.keys())
# summarize history for accuracy
plt.plot(m4R.history['accuracy'])
plt.plot(m4R.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(m4R.history['loss'])
plt.plot(m4R.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()


# list all data in history
print(m5R.history.keys())
# summarize history for accuracy
plt.plot(m5R.history['accuracy'])
plt.plot(m5R.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
# summarize history for loss
```

```
plt.plot(m5R.history['loss'])
plt.plot(m5R.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

BIBLIOGRAPHY

Carlini, N., & Wagner, D. (2017). Towards evaluating the robustness of neural networks. *2017 IEEE Symposium on Security and Privacy (SP)*. doi: 10.1109/sp.2017.49

Dietterich, T. G. (2002). Ensemble learning. *The handbook of brain theory and neural networks*, 2, 110-125.

Frans, K., Ho, J., Chen, X., Abbeel, P., & Schulman, J. (2017). Meta learning shared hierarchies. arXiv preprint arXiv:1710.09767.

Gemici, M., Hung, C. C., Santoro, A., Wayne, G., Mohamed, S., Rezende, D. J., ... & Lillicrap, T. (2017). Generative temporal models with memory. arXiv preprint arXiv:1702.04649.

Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572.

He, W., Wei, J., Chen, X., Carlini, N., & Song, D. (2017). Adversarial example defenses: Ensembles of weak defenses are not strong. *WOOT*. Retrieved from https://arxiv.org/pdf/1706.04701.pdf

Liu, S. (2019, November 25). Global AI software market growth 2019-2025. Retrieved from https://www.statista.com/statistics/607960/worldwide-artificial-intelligence-market-growth/

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2017). Towards deep learning models resistant to adversarial attacks. arXiv preprint arXiv:1706.06083.

THE MNIST DATABASE. (n.d.). Retrieved from http://yann.lecun.com/exdb/mnist/

Moosavi-Dezfooli, S.-M., Fawzi, A., & Frossard, P. (2016). DeepFool: A simple and accurate method to fool deep neural networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. doi: 10.1109/cvpr.2016.282

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

Polikar R. (2012) Ensemble learning. In: Zhang C., Ma Y. (eds) *Ensemble Machine Learning*.

Qiu, X., Zhang, L., Ren, Y., Suganthan, P., & Amaratunga, G. (2014). Ensemble deep learning for regression and time series forecasting. *2014 IEEE Symposium on Computational Intelligence in Ensemble Learning (CIEL)*. doi: 10.1109/ciel.2014.7015739

Shaham, U., Cheng, X., Dror, O., Jaffe, A., Nadler, B., Chang, J., & Kluger, Y. (2016, June). A deep learning approach to unsupervised ensemble learning. In International conference on machine learning (pp. 30-39).

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Lillicrap, T. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815.

Sitawarin, C., Bhagoji, A. N., Mosenia, A., Chiang, M., & Mittal, P. (2018). Darts: Deceiving autonomous cars with toxic signs. arXiv preprint arXiv:1802.06430.

Su, J., Vargas, D. V., & Sakurai, K. (2019). One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, *23*(5), 828–841. doi: 10.1109/tevc.2019.2890858

Tencent Keen Security Lab. (2020, March 18). Tencent Keen Security Lab: Experimental security research of tesla autopilot. Retrieved from

https://keenlab.tencent.com/en/2019/03/29/Tencent-Keen-Security-Lab-Experimental-Security-Research-of-Tesla-Autopilot/

ACADEMIC VITA

Ethan Adams
epa5093@psu.edu

Education

Major: B.S. Information Sciences and Technology (Design and Development), Security and Risk Analysis (Information and Cyber Security)
Honors in Information Sciences and Technology
Graduation: May 2020

International Education

HUM 200H - Explorations in the Humanities: The Quest
China (Beijing, Shanghai, Suzhou, Hangzhou)
March 2018

Research

Thesis: Deep Ensemble Learning for Robust Defense Against Adversaries
Thesis Supervisor: Dr. Mahdi Nasereddin
Honors Adviser: Dr. Sandy Feinstein
Grant Received: Erickson Discovery Grant (2019)

Presentations

ASEE 2017 Mid-Atlantic Conference Session
Nittany AI Challenge, Top 3 Pitch
Berks Technology Club

Awards

IST Leadership Award (2020)
Weidenhammer Systems SRA Award (2020)
Simplr AI and Technology Scholarship (2019)
Boscov Honors Scholarship (2019)
Sproesser Honors Scholarship (2018)
Dean's List Award (2016-2020)

Relevant Work Experience

June 2019 – August 2019
Solution Summer Scholar (Software Engineering)

- Developed a front-end solution for state benefits mobile application with 100,000+ downloads
- Presented MVC architecture overhaul to an executive audience, reducing redundant API calls by 100%

Deloitte Consulting LLC, Mechanicsburg, PA
Richard Abel III

May 2018 – May 2019
Software Engineering Intern

- Designed the front-end User Experience for a translation application with Angular 2 followed by React Native
- Handled nested JSON API response, parsing into parent and child menus for thousands of languages and dialects
- Integrated the front-end application with a back-end REST API affecting 300+ international students on campus

Traduki Technologies LLC, Reading. PA
Ryan Morris

October 2017 – April 2018
Data Validation Specialist

- Confirmed over 20,000 services and datasets for a large pest control client transitioning to a new system

Weidenhammer Systems Corporation, Wyomissing, PA
Kurt Bauder

August 2017 – December 2017
Peer Tutor

- Provided one-on-one tutoring for 20+ students visiting the writing center at Penn State Berks

Pennsylvania State University, Berks, Reading, PA
Dr. Holly Ryan

Leadership
March 2018 – May 2020
Berks Technology Club President

- Coordinating speakers from our faculty, local businesses, and large tech companies for our 70+ active members
- Organized trip to Google NYC HQ for 33 students to foster connections for Penn State Berks technology students

Summer 2017 & Summer 2018
GenCyber Teacher's Assistant

- Presented and assisted in the NSA/NSF funded summer camp program for 40+ K-12 students
- Created several dozen hands-on activities using a VMware virtual machine environment for ethical hacking