© 2018 Karan Ganju

INFERRING PROPERTIES OF NEURAL NETWORKS WITH INTELLIGENT DESIGNS

BY

KARAN GANJU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Carl Gunter

ABSTRACT

Neural networks have become popular tools for many inference tasks nowadays. However, these networks are functions derived from their training data and thorough analysis of these networks reveals information about the training dataset. This could be dire in many scenarios such as network log anomaly classifiers leaking data about the network they were trained on, disease detectors revealing information about participants such as genomic markers, facial recognition classifiers revealing data about the faces it was trained on, just to name a few alarming cases. As different industries employ this technology with open arms, it would be wise to be aware of the privacy impacts of openly shared classifiers.

To that measure, we perform the first study of property inference attacks specifically for deep neural networks - deriving properties of the training dataset with no information apart from the classifier parameters and architecture. We implement and compare different techniques to improve the effectiveness of the attack on deep neural networks. We show how different interpretations and representations of the same classifier - 1) after sorting and normalizing the vector representation, 2) as a graph and 3) as a group of sets - are able to increase leakage of data from the classifier, with the latter being the most effective. We compare effectiveness on a synthetic dataset, the US Census Dataset and the MNIST image recognition dataset, showing that critical properties such as training conditions or bias in the dataset can be derived by the attack.

To

my professor, for his guidance and support,

my friends, for the inspiration, and

my family, for their love

ACKNOWLEDGMENTS

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000780.

I would like to express my gratitude to my professor, Carl Gunter, for his advice and support throughout my studies at the university. I cannot stress sufficiently how much I appreciate the confidence he has shown in me and the freedom he has provided me to pursue my interests. It has truly been an enriching experience because of him.

I also would like to thank my friends, especially, Unnat Jain, for all the discussions we have had, which have aided me a lot in my research.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Neural Networks have gained great popularity both in academia as well as industry due to their proven ability to solve difficult tasks with reassuring capability. They are already being employed as part of conversational agents[1, 2] and autonomous vehicles[3]. Likewise, they have also shown great promise for sensitive operations such as facial recognition[4, 5], speech recognition[2], personalized medicine[6, 7] and many others. In fields like malware analysis and network anomaly detection, there is a growing drive to share learned models within the community, which can help in serving quicker and more standardized responses.

However, it is important to draw the distinction between sharing hard-coded rules, such as antivirus policies and signatures, and learned anomaly classifiers. The former can be easily interpreted and understood; its logic can be reasoned about and the impact of sharing it can be quantified. On the other hand, anyone debugging a neural network knows that the logic it serves often lacks transparency. It is not uncommon for the community to be skeptical about why certain tricks work and others don't. This often leads to neural networks being treated as black boxes that just solve the problem - no questions asked. This draws the question - Do we really know what we are sharing when we share classifiers?

As a simple but illustrative example, consider the case of Support Vector Machines[8], a linear model that is deemed very effective for classification or regression. The classifier learned by the SVM, a hyperplane which divides the domain space into two, can be represented as a weighted sum of its training data points. Often, when sharing the SVM classifier, the data points and their weights are shared instead of the derived linear classifier. This sort of data leakage could implicitly be happening in neural networks too, although, their black box nature makes deriving this a challenge too. Hence, while sharing classifiers, one must keep in mind that there is a risk of some leakage of data, and consequently privacy, as well.

Different privacy attacks have previously been launched on machine learning models with varying degrees of success. Model extraction attacks[9] seek to obtain the parameters of

a model given the outputs it predicts for a chosen set of inputs by the attacker. Model Inversion attacks[10, 11, 12] output some of the possible training data samples that the machine learning model could have been trained on. Membership inference attacks[13, 12], very similar to Model Inversion, predict the likelihood of a given data sample to be in the training data, given the model. We specifically focus on Property Inference attacks.

Property Inference is the task of inferring properties about the training dataset or process using only the parameters of the trained classifier as prior knowledge. These properties could range from bias in the dataset to existence of certain types of samples in the same. This could be useful knowledge to an adversary, for example, predicting the existence of vulnerable systems from a network anomaly detector trained on it. Prior work by Ateniese et al.[14] has demonstrated this attack for classifiers such as Hidden Markov Models or Naive Bayes models and has laid down the general framework of the attack. We specialize the attack to work on dense neural networks. There are challenges in using the prior work out-of-the-box to attack dense neural networks. We provide potential reasons for this which have to do with the inability of the previous frameworks to generalize well for neural networks and also introduce different extensions to the attacks which tackle this issue.

Note that although we only demonstrate the improvements for the specific attack, the improvements are not specific to the attack but generally in improving the representation of the target classifier so that we can infer the most out of it. As this suggests, the improvements we suggest are not just limited to property inference but can also potentially be applied to other tasks, from other privacy attacks to interpretability based tasks for neural networks.

The contributions of this work can be broadly categorized in three main points -

1. We highlight existing problems with using the attack as-is on neural networks.

2. We introduce a data preprocessing step and a data augmentation step which can be potentially used to improve effectiveness of the attack.

3. We introduce two different representations of neural network classifiers and architec-

tures to deal with both.

- Representing neural networks as graphs
- Representing neural networks as collections of sets

The rest of the thesis is divided into the following chapters -

- We discuss about relevant background for understanding this paper, theory for neural networks and Property Inference Attack, followed by related work in Chapter 2.
- Then, in Chapter 3, we discuss what we believe are issues with using the attack out-of-the-box on neural networks.
- We start with our extensions in Chapter 4, where we discuss the data preprocessing and data augmentation techniques to improve the attack.
- In Chapters 5 and 6, we discuss the new architectures which improve the attack. These are respectively the graph-based and set-based architectures.
- We highlight all experiments and results in Chapter 7.
- Finally, we conclude in Chapter 8.

CHAPTER 2: BACKGROUND

Since we focus on Property Inference attacks on neural networks, we provide background knowledge for both neural networks and property inference attacks in the following sections, followed by a discussion on related work.

2.1 NEURAL NETWORKS

Neural networks are machine learning models that are built of several layers of computational units that process or transform the output of the previous layer and produce input for the next layer. The first computational layer receives input from an additional input layer and the last layer is known as the output layer. Mathematically, the output of the neural network y, for input x, can be written as

$$y = F_n(F_{n-1}(\cdots F_2(F_1(x))\cdot) \tag{2.1}$$

where the number of layers in the neural network are n, each representing a function F_i. The type of transformation between layers often depends on the type of the task at hand. For example, convolutional layers are often employed for images while recurrent layers have been traditionally employed for text. The most commonly used layers are linear layers built up of multiple computational units called perceptrons[15], each possessing a multiplicative weight, an additive bias and a non-linear activation function. Hence, for the input to the layer, o_{i-1}, the output of the perceptron o_{ij} with weight column vector $w_{ij} \in \mathbb{R}^{|o_{i-1}|}$, bias $b_{ij} \in \mathbb{R}^1$ and a non-linear activation function γ is

$$o_{ij} = \gamma(w_{ij}^T \cdot o_{i-1} + b_{ij}) \tag{2.2}$$

and the layer output, o_i is given as

$$o_i = [o_{i1}, o_{i2}, \cdots o_{in}] \tag{2.3}$$

Putting all of this together, the layer output can be written as

$$o_i = F_i(o_{i-1}) = \gamma(w_{i*}^T \cdot o_{i-1} + b_{i*}) \tag{2.4}$$

where w_{i*} represents all the weight column vectors of the constituent perceptrons stacked together and likewise for b_{i*}.The non-linear activation function γ is chosen to increase the representational power of the neural network, so that it is able to learn non-linear functions too, and is often one of the following

$$ReLU(x) \stackrel{\text{def}}{=} \begin{cases} 0 & x < 0 \\ x & x >= 0 \end{cases} \tag{2.5}$$

$$LeakyReLU(x) \stackrel{\text{def}}{=} \begin{cases} \alpha x & x < 0, 0 < \alpha < 1 \\ x & x >= 0 \end{cases} \tag{2.6}$$

$$tanh(x) \stackrel{\text{def}}{=} \frac{1-e^{-2x}}{1+e^{-2x}} \tag{2.7}$$

$$sigmoid(x) \stackrel{\text{def}}{=} \frac{1}{1+e^{-x}} \tag{2.8}$$

For each layer, the choice of number of perceptrons is a design choice and is treated as a hyperparameter to be tuned. The number of perceptrons in the final layer could range from 1 for regression and 2-way classification tasks to k for k-class/label classification. Similarly, the choice of the activation function, γ, is also a design choice.

Given this functional representation, we pass input x through the input layer and obtain the final output, y through the output layer. In order to train the neural network, i.e. tune

the parameters of the model so that it produces useful output, we have to first define a metric for how well or bad the network is already doing. This is known as the loss function. The loss should be large if the output is far from the correct output and smaller as we move towards the correct output. The specific choice of the loss function depends a lot on the type of problem, the type of architecture of the neural network and other contextual settings. For example, for 2-way classification, binary cross entropy is a common loss function employed while mean squared error is common for regression tasks. Given this loss function, a technique called stochastic gradient descent[16] is employed to tune the parameters in order to reduce the current loss. Hence, at the end of training, we obtain a tuned neural network and use this for the test samples without performing the gradient descent steps, i.e. holding the parameters, w and b, constant.

2.2 PROPERTY INFERENCE

The Property inference attack by Ateniese et al.[14] explores the general problem of inferring any kind of interesting property about the training dataset from prior knowledge of the parameters of a classifier trained on the same. This property could be, for example, the accent of the speakers employed to train a speech recognition model or the inherent properties of network traffic, such as whether it is from Google or not, used to train a network traffic classifier.

In principle, we want to build a meta-classifier (which is how we describe a classifier that infers on classifiers), say $\mathbb{MC}$, which predicts a property $\mathbb{P}$ or its absence $\bar{\mathbb{P}}$ from input $\mathbb{C}$, which is the feature representation of the classifier. This classifier could be any learning model, from a Naive Bayes model[17] to a decision tree[18], a Hidden Markov Model[19](HMM) or in our case, a deep neural network. Correspondingly, the feature representation varies accordingly. It could be a set of probabilities for Naive Bayes or a HMM, or it could be a collection of weights and biases for a neural network. The meta-classifier, too, need not be

restricted to any particular kind of learning model and Ateniese et al. use decision trees in their attacks.

In order to do this, we have to train $\mathbb{MC}$ and for that, we need a dataset of feature representations for it, for both $\mathbb{P}$ and $\bar{\mathbb{P}}$ properties. Essentially, we create or obtain a dataset adhering to property $\mathbb{P}$ and another dataset adhering to property $\bar{\mathbb{P}}$. This dataset could be obtained from the same distribution and/or source that the target network was trained from or a similar one. We then train multiple classifiers for both datasets, known as shadow classifiers. These are used to train the meta-classifier which, after training, can be used to infer the same property about the target network. Algorithm 2.1 from the original paper[14] enumerates the same process.

Algorithm 2.1 Property Inference Attack[14]

Input:
D: the array of training sets
l: the array of labels, where each $l_i \in \{\mathbb{P}, \overline{\mathbb{P}}\}$
Output: the meta-classifier $\mathbb{MC}$
TrainMC(D, l)

begin
  $D_c = \{\phi\}$
  for each $D_i \in D$ **do**
    $C_i \leftarrow$train(D_i)
    $F_{C_i} \leftarrow$ getFeatureVector(C_i)
    $D_c \leftarrow D_c \cup \{F_{C_i}, l_i\}$
  end for
  $\mathbb{MC} \leftarrow$ train(D_c)
  return $\mathbb{MC}$
end

2.3 RELATED WORK

Apart from Property Inference, there is already existing work on privacy attacks on shared classifiers. Model inversion is a kind of attack that aims to derive training data attributes or samples from a trained classifier. Fredrikson et al.[10] launch a model inversion attack on a linear regression model that predicted patients' appropriate Warfarin dosage given their

demographics, genomic markers and medical information. With the attack, they were able to successfully predict the patients' genomic markers given their corresponding output from the regression model and some auxiliary demographic information about them. In following work, Fredrikson et al.[11] were able to perform model inversion for decision trees and neural networks using the confidence values along with the prediction itself. With this attack, they were able to extract faces in the training data of a facial recognition system and extract lifestyle choices in the data of a lifestyle prediction system.

Another class of attacks, known as membership inference attacks is able to infer, given a black-box model, whether a particular data point was present in its training data set. A black-box model in this context refers to one where the adversary is not able to obtain model parameters but can only observe the outputs for chosen inputs. This attack was developed by Shokri et al.[13] who use a similar attack strategy as the Property Inference attack in that they train multiple shadow models to help train an attack classifier, which ultimately is trained to perform this membership inference task for a target classifier. More recent work by Yeom et al.[12] show that the membership inference attack and the model inversion attack are related, along with showing their link to overfitting and influence of variables.

Tramèr et al.[9] show another model of attack known as model extraction, where the adversary attempts to obtain the model parameters of a black-box model. They demonstrate the attack on popular model classes - linear regression, neural networks and decision trees. These attacks open the potential for follow-up attacks. For example, an adversary could obtain a near-equivalent model of the black-box model that is to be attacked and use property inference or other white-box attacks on the white-box equivalent. Oh et al.[20] also use a attack classifier in order to infer properties about the training process to aid in 'whitening' a black-box model. They are able to infer important architectural, data and training properties. Although their attack is very similar to an application of the property inference attack, their method is significantly different, making predictions on target network outputs as opposed to the target network parameters.

A variety of other attacks on classifiers have also been reported, ranging from adversarial attacks on object detectors[21] or classifiers[22, 23, 24, 25] to malicious algorithms which discreetly memorize the training data samples[26] or malicious classifiers which misclassify in presence of an input trigger[27]. However, while these also motivate why vendors must consider risks involved in sharing classifiers, they differ from the kind of attack we explore.

CHAPTER 3: EXISTING LIMITATIONS OF METHODS

While the prior work by Ateniese et al.[14] is demonstrated to work well for classifiers like Support Vector Machines[8] and Hidden Markov Models[19], it is not able to perform very well out-of-the-box on deep neural networks. Similar to proposed methods for other classifiers, one could attempt to use a flattened vector of all the weights and biases of the neural network as a feature representation. However, this does not perform well in practice. Scaling the meta-classifier architecture does not solve this issue. We provide reasons for why we believe the simple feature representation is at fault here.

3.1 AFFECTED BY ROTATIONS

A flattened vector will have specific meta-classifier weights associated to each feature for every position. However, consider the two neural networks shown in Figure 3.1. They perform the same function but have different flattened representation. We call this the rotation equivalence of linear layers. More formally,

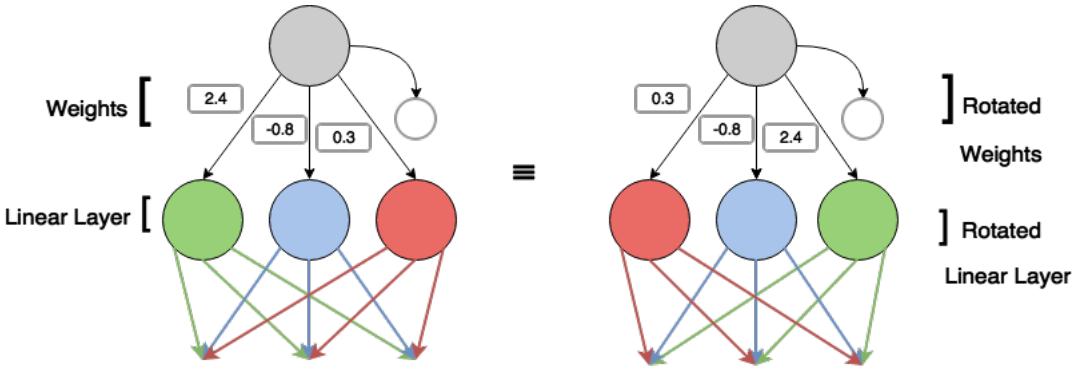


Figure 3.1: Rotation Equivalence

Theorem 3.1. *Consider a hidden linear layer m with neurons $[f_1^m, f_2^m, f_3^m, \cdots f_{n_m}^m]$ with each neuron f_i^m connected to the layer $m-1$ by weights w_i^m. If we replace the layer with the linear layer, for some permutation σ, $[f_{\sigma(1)}^m, f_{\sigma(2)}^m, f_{\sigma(3)}^m, \cdots f_{\sigma(n_m)}^m]$ such that 1) the weights and biases of layer m are shifted so that the neuron $f_{\sigma(i)}^m$ is connected to the layer $m-1$*

with weights $w^m_{\sigma(i)}$ and bias $b^m_{\sigma(i)}$ and 2) the elements of the weights of layer $m+1$ for every neuron f^{m+1}_j, previously $w^{m+1}_j = [w^{m+1}_j(1), w^{m+1}_j(2) \cdots w^{m+1}_j(n_m)]$, are now rotated to $rot(w^{m+1}_j) = [w^{m+1}_j(\sigma(1)), w^{m+1}_j(\sigma(2)) \cdots w^{m+1}_j(\sigma(n_m))]$, then the output of the neural network remains exactly the same.

Proof. Consider the output of a neural network with k layers, o_k, for input x,

$$o_k = F_k(F_{k-1} \cdots (F_1(x)) \cdot) \tag{3.1}$$

In order to prove Theorem 3.1, it would suffice to show that, for some rotated layer m which now represents G^m instead of F^m, the output of the the neural network for the same input x is still o_k. Note that while rotating the layer, we only rotate the current layer nodes, its weights and the weights connecting it to the next layer. Since the other layers, apart from m and $m+1$, have not changed, it is sufficient to prove that

$$G^{m+1}(G^m(x_{m-1})) = F^{m+1}(F^m(x_{m-1})) \tag{3.2}$$

for

$$x_{m-1} = F^{m-1}(F^{m-2}(\cdots(F^1(x))\cdot) \tag{3.3}$$

Now, let's consider the output of layer m.

$$F^m(x_{m-1}) = [f^m_1(x_{m-1}), f^m_2(x_{m-1}), \cdots f^m_{n_m}(x_{m-1})] \tag{3.4}$$

$$G^m(x_{m-1}) = [f^m_{\sigma(1)}(x_{m-1}), f^m_{\sigma(2)}(x_{m-1}), \cdots f^m_{\sigma(n_m)}(x_{m-1})] \tag{3.5}$$

Now consider any neuron i in the layer $m+1$. Its output would be, for the first case,

$$\begin{aligned} f_i^{m+1}(F^m(x_{m-1})) &= \gamma(w_i^{m+1} \cdot F^m(x_{m-1}) + b_i^{m+1}) \\ &= \gamma(w_i^{m+1} \cdot [f_1^m(x_{m-1}), f_2^m(x_{m-1}), \cdots f_{n_m}^m(x_{m-1})] + b_i^{m+1}) \\ &= \gamma(\sum_{j=1}^{n_m} w_i^{m+1}(j) * f_j^m(x_{m-1}) + b_i^{m+1}) \\ &= \gamma(\sum_{j=1}^{n_m} w_i^{m+1}(\sigma(j)) * f_{\sigma(j)}^m(x_{m-1}) + b_i^{m+1}) \\ &= \gamma(rot(w_i^{m+1}) \cdot [f_{\sigma(1)}^m(x_{m-1}), f_{\sigma(2)}^m(x_{m-1}), \cdots f_{\sigma(n_m)}^m(x_{m-1})] + b_i^{m+1}) \\ &= \gamma(rot(w_i^{m+1}) \cdot G^m(x_{m-1}) + b_i^{m+1}) \end{aligned} \tag{3.6}$$

The final equation obtained is the output of the ith node of the $m+1$th layer. Hence, the shuffling that we performed does not change the output of any neuron of the $m+1$th layer and so, does not change the output of the neural network. □

Another way to think about this is that if we shift or rotate around the nodes within a layer, with their edges maintained, the output should be unaffected. Hence, any inference on nodes within a layer should be permutation-invariant. That highlights the problem with using a flattened vector to represent a neural network. There are specific weights associated with each position which makes it very tough for the meta-classifier to be insensitive to permutations, without taking any specific measure for making it so.

3.2 AFFECTED BY CONSTANT MULTIPLICATIONS

Apart from rotation equivalence, network layers with ReLU[28] or LeakyReLU[29] activations also show, what we call the property of multiplicative equivalence. This property states that if we multiply the weights and bias of a neuron by some constant β and divide the weights connecting it to the next layer by the same constant, the output of the neural network remains the same, as long as the activation is ReLU or LeakyReLU - both very

commonly used activation functions. This is illustrated in Figure 3.2. More formally,

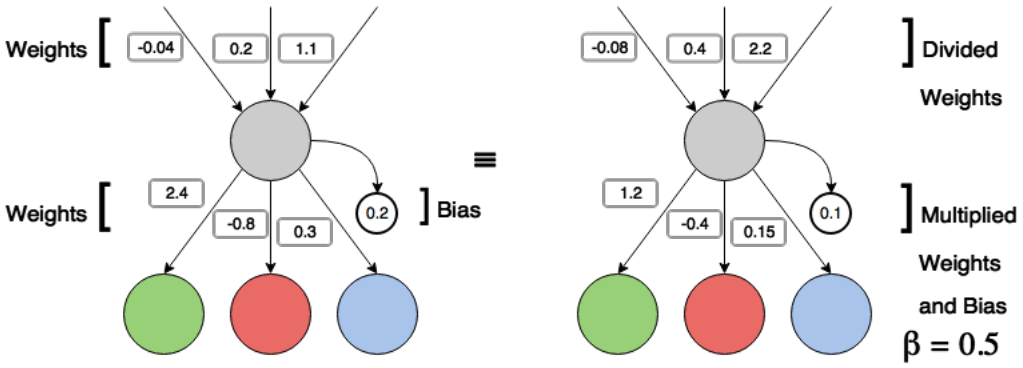


Figure 3.2: Multiplicative Equivalence for ReLU activation

Theorem 3.2. *Consider an artificial neuron i within the mth hidden layer of the neural network with an activation of ReLU or Leaky ReLU, with weight vector w_i^m, bias b_i^m and weights connecting it to the next layer $w_*^{m+1}(i)$. Then, if we replace this weight vector with $\beta \times w_i^m$, the bias with $\beta \times b_i^m$ and the connecting weights with $w_*^{m+1}(i)/\beta$, the output of the neural network remains exactly the same.*

Proof. Let's consider the output of the mth layer. Prior to multiplying by β, it was

$$F^m(x_{m-1}) = [f_1^m(x_{m-1}), f_2^m(x_{m-1}), \cdots f_{n_m}^m(x_{m-1})] \tag{3.7}$$

After multiplying, the output of node i is β times larger because both the weight and bias have been multiplied by β and the output of layer m is now

$$[f_1^m(x_{m-1}), f_2^m(x_{m-1}), \cdots \beta * f_i^m(x_{m-1}) \cdots f_{n_m}^m(x_{m-1})] \tag{3.8}$$

Note that we can write this precisely because with the ReLU or Leaky ReLU as the activation function γ the following equation holds

$$\gamma(\beta w_i^m \cdot x_{m-1} + \beta b_i^m) = \beta\gamma(w_i^m \cdot x_{m-1} + b_i^m) = \beta f_i^m(x_{m-1}) \tag{3.9}$$

Now, take any node j from the $m+1$th layer. Its output was previously

$$f_j^{m+1}(F^m(x_{m-1})) = \gamma(w_j^{m+1} \cdot [f_1^m(x_{m-1}), f_2^m(x_{m-1}), \cdots f_{n_m}^m(x_{m-1})] + b_j^{m+1}) \tag{3.10}$$

After multiplying the weight and bias of node i and dividing the connecting weights to layer $m+1$ by β, we get the following output of node j

$$\begin{aligned}
&\gamma([w_j^{m+1}(1), w_j^{m+1}(2), \cdots w_j^{m+1}(i)/\beta \cdots w_j^{m+1}(n_m)] * [f_1^m, f_2^m, \cdots \beta f_i^m \cdots f_{n_m}^m] + b_j^{m+1}) \\
&= \gamma([w_j^{m+1}(1), w_j^{m+1}(2), \cdots w_j^{m+1}(i) \cdots w_j^{m+1}(n_m)] * [f_1^m, f_2^m, \cdots f_i^m \cdots f_{n_m}^m] + b_j^{m+1}) \\
&= \gamma(w_j^{m+1} \cdot F^m(x_{m-1}) + b_j^{m+1}) \\
&= f_j^{m+1}(F^m(x_{m-1}))
\end{aligned} \tag{3.11}$$

Hence, multiplying and dividing parameters in this manner does not change the output of the $m+1$th layer and consequently does not affect the output of the neural network. □

In such cases the meta-classifier should also be invariant to multiplicative equivalents and again, it is too much to ask for the meta-classifier to learn this by itself without specifically training it in a way that it recognizes this invariance.

3.3 NO ARCHITECTURE INFORMATION

A flattened vector does not give any information about the architecture of the neural network. Any shared classifier would also need to release its architecture along with the weights and biases to render the classifier usable. Hence, it would be highly likely that the adversary is able to obtain the architecture, if the weights and biases are public knowledge. This might not have been a drawback for other types of classifiers because of their inherent structural simplicity. For example, neural networks have a much more complex architecture than SVMs which are linear classifiers and HMMs which are generally smaller models. By

representing the neural network as a flattened vector, the attacker is not utilizing this useful piece of information.

Another advantage of the using the architecture as additional information while inferring from the network is that a generalized meta-classifier could then potentially be able to train on smaller shadow models and infer properties of a larger target network.

CHAPTER 4: USING SORTING AND NORMALIZING

The requirement of having a meta-classifier invariant to rotations and multiplicative operations can be seen as similar to a requirement commonplace in vision-based tasks. Take the case of a face recognition task. One would desire a face recognition tool to still recognize the same face despite of rotations and/or changes in brightness in the image. The same requirement follows for a lot of other image based tasks. Part of this problem, in the case of images, is solved through the use of Convolutional Networks which are shift-invariant and so tolerate some movement or some small rotations of the image. However, that alone would not be sufficient to deal with large rotations or changes in brightness.

There are two common ways this is dealt with which we can also use to deal better with rotated or multiplicative-constant-shifted classifiers - Preprocessing and Augmentation, which we explain in the following sections.

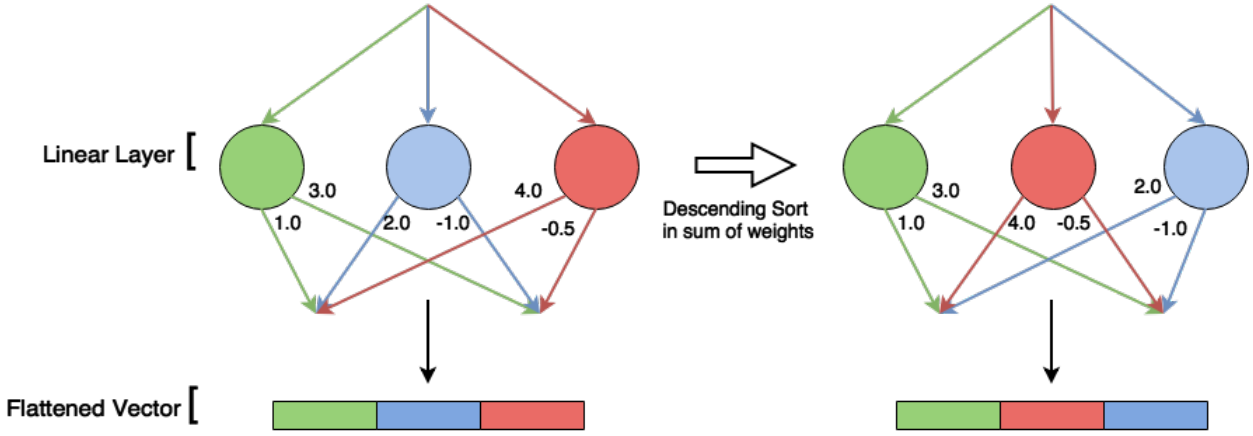
4.1 PREPROCESSING

A commonly applied approach to this problem in the vision domain is to preprocess rotated images to a canonical pose. In face recognition, the face is aligned either manually or with a classifier to set the face along the edges of the image. This preprocessing helps improve the accuracy of other tasks. Similarly, in our case, we can preprocess the classifier representation to bring it to a canonical form in order to make the inference task easier.

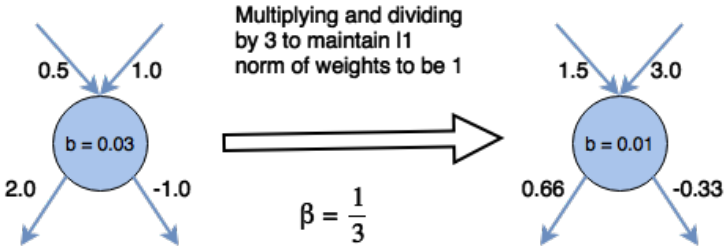
Because of the rotation equivalence, we are free to rotate nodes within a dense layer as long as we rotate the weights accordingly. Hence, we can obtain some metric for all nodes and then sort them within the layer based on this metric to obtain a 'canonical' form. We try multiple different metrics such as the sum of weights, the sum of absolute values of weights, etc and observed that the sum of weights yielded the best results.

We can also perform a similar step to handle multiplicative-constant-invariance. Note again that this is specific to neural layers with ReLU or Leaky ReLU activations, which are

very common activations. In this case, recall that we can multiply a neurons weight and bias by any constant as long as we counteract the multiplication with a division by the same constant along weights connecting it to the next layer. We can choose a constant so that the neuron conforms to a more canonical form by our chosen definition of 'canonical'. For example. we could ensure that its weights have a norm of 1. We try multiple approaches and observe that enforcing that the sum of absolute values of weights to be 1 performs the best.



(a) Sort Canonicalization



(b) Multiplicative Canonicalization

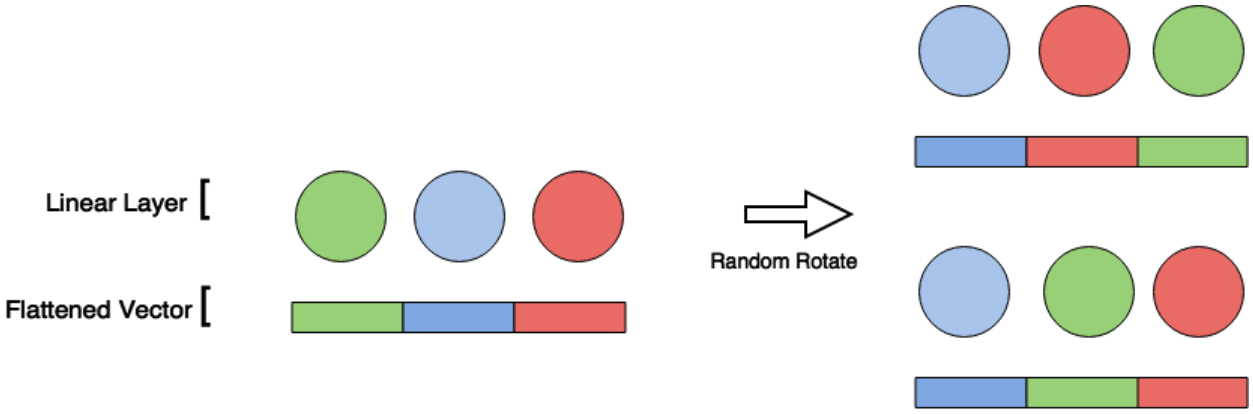
Figure 4.1: Preprocessing

4.2 AUGMENTATION

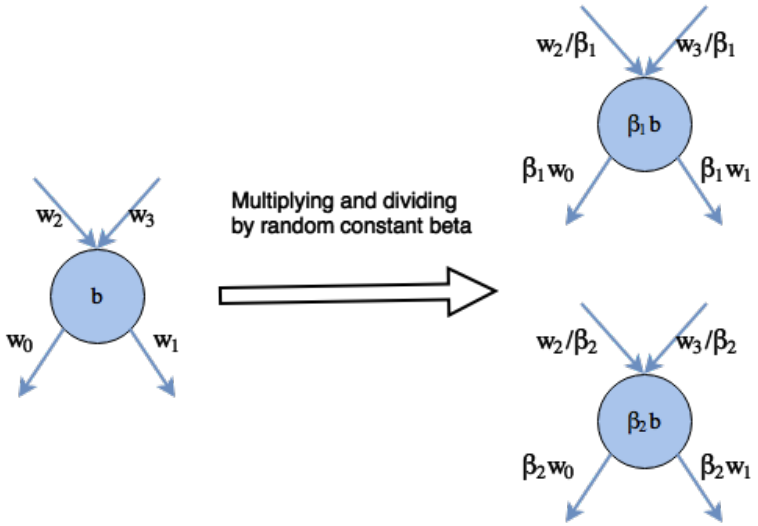
Data Augmentation is the alternative approach to help generalize a classifier for different versions of the same data. Essentially, one can generate different versions of the same data,

rotated, multiplicative-constant-shifted, etc, and pass them to the classifier to learn during training and hope that the classifier becomes invariant to these changes during testing. There is a fundamental difference between the augmentation approach and the preprocessing approach, where in the latter you do some of the work for the classifier and in the former you expect the classifier to be able to capture these changes by itself.

In our case, this translates to training the meta-classifier with different rotated and multiplicative constant shifted versions of the same classifier. Here, we hope for the classifier to be able to become invariant to these changes over the course of training. In order to make the inference task simpler for the meta-classifier, we limit the multiplicative constant to be between 0.1 and 10. Not performing this step results in very small and very large weights, which are both difficult to deal with. We report the results in Chapter 7. This process is also illustrated in Figure 4.2.



(a) Sort Augmentation



(b) Multiplicative Augmentation

Figure 4.2: Augmentation

CHAPTER 5: USING GRAPH-BASED REPRESENTATIONS

While processing and augmentation are steps that may partially aid in inferring attributes about classifiers, they do not capture any information about the architecture of the classifier. In order to exploit this additional information we have about the architecture, we need a more intelligent design.

One way to address this issue is to change the representation of the classifier from a flat vector to that of a graph, with different neurons being the nodes, connected to each other with directed weighted edges and having an inherent bias feature. However, we also need a machine learning construct which will be able to deal with graphs and Graph Convolutional Networks by Kipf et al.[30] do exactly that.

5.1 WORKING

Convolutions on functions f and g are defined as $f * g$ where

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\inf}^{\inf} f(\tau)g(t-\tau)$$

In discrete spaces, convolutions are defined as

$$(f * g)(t) \stackrel{\text{def}}{=} \sum_{-\inf}^{\inf} f(\tau)g(t-\tau)\, d\tau$$

These operators can be used to apply filters (function f) to images (function g) and offer the following advantages -

1. Parameter Sharing - the same filter is applied to all points
2. Sparse Connections - if the filter is of size $h * w$, the output at a point will only depend on $h * w$ inputs, instead of the entire input

Both of the mentioned advantages make convolutions very fast to use leading to their widespread adoption in data intensive fields like Computer Vision.

Traditional Convolutional Networks[31] compute convolutions over pixels and their neighboring pixels within a window defined by the filter size. Likewise, Graph Convolutional Networks[30](GCN), compute convolutions over graph nodes and their neighboring nodes within a window. The GCN takes as input an adjacency matrix A and a feature matrix H. It then progressively returns advanced feature matrices which are fed again and again into the GCN to obtain more and more advanced features. For N, the number of nodes, and $k(l)$, the number of features of a node, or neuron in our case, in the lth layer of the GCN, we have $H^{(l)} \in \mathbb{R}^{N*k(l)}$, the feature matrix in the same layer and $A \in \mathbb{R}^{N*N}$, the adjacency matrix. Then, the output of a GCN layer on these two matrices is

$$H^{(l+1)} = \gamma(AH^{(l)}W)$$

for the learned weight vector, $W \in \mathbb{R}^{k(l)*k(l+1)}$ and activation function, γ.

This learned vector $H^{(l+1)}$ can be thought of as a processed feature vector that represents a function computed over local features of each node and its immediate neighbors, providing a more global inference.

5.2 APPLICATION TO PROPERTY INFERENCE

Figure 5.1 shows the process of representing a dense neural network in the form of an adjacency matrix and a feature matrix. The following steps briefly capture the same process.

1. All neurons from all layers, including the input and output layer, are treated as nodes with directed edges pointing away from the input layer and towards the output layer.

2. The edge weights for two nodes in the adjacency matrix are derived from weights connecting the corresponding two neurons in the neural network.

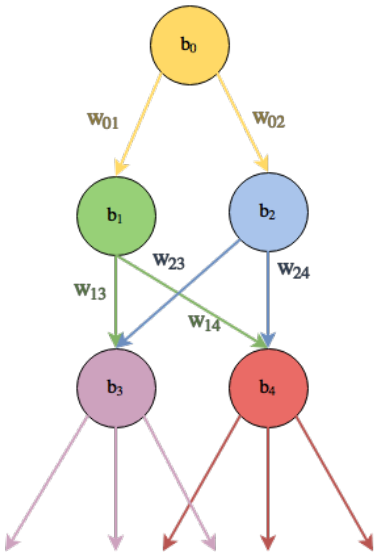
Feature	Type
Bias of Neuron	Float
Is Input Neuron?	Binary
Activation Type	Category/Int
Has Dropout?	Binary

Table 5.1: Feature Matrix Constituents

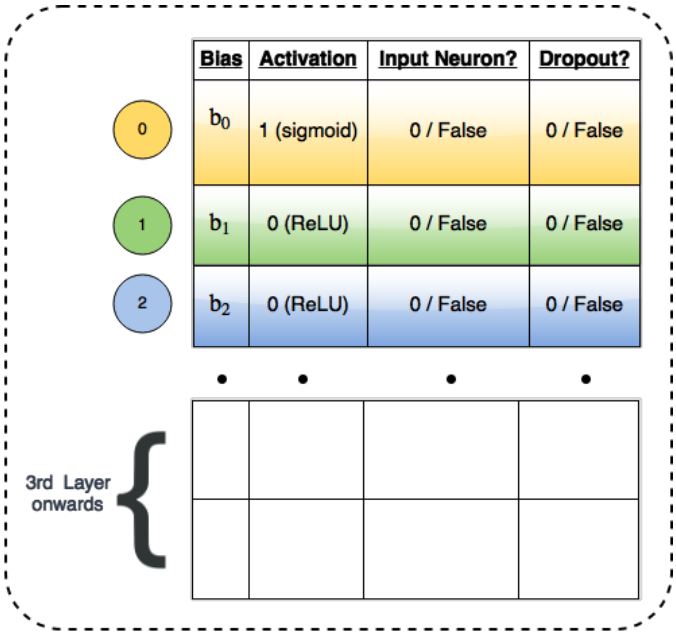
3. Self edges of weight 1 are added to the adjacency matrix as prescribed in the GCN paper.

4. The bias of a neuron is taken as a feature of the node it represents, along with other features given in Table 5.1.

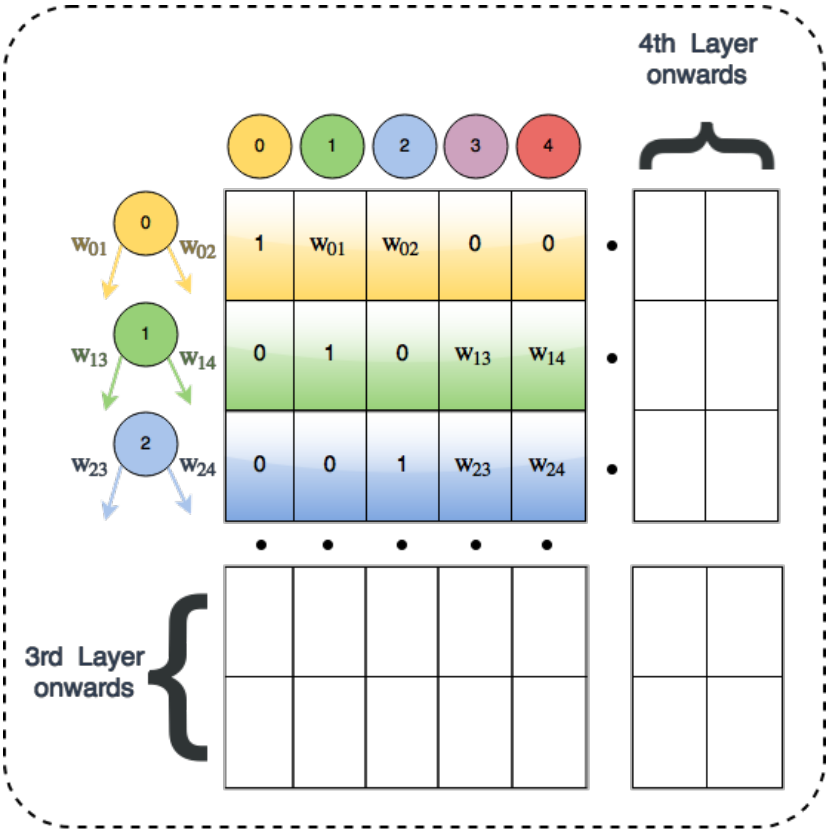
After representing the neural network in this form, we can use the GCN to process it. However, the GCN outputs a set of processed feature vectors for each node, i.e. a processed feature matrix with more global representations. Our task deals with making a classification decision for each graph, as opposed to one for each node. In order to obtain the final classification decision, we flatten out the processed feature vector and pass it through some dense layers.



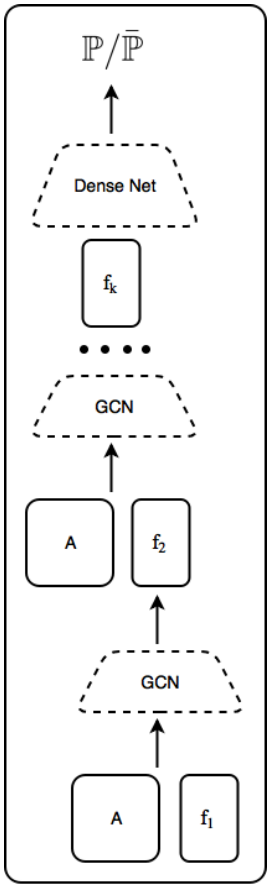
(a) Classifier



(b) Feature Matrix(f_1)



(c) Adjacency Matrix(A)



(d) Using GCN Architecture

Figure 5.1: GCN Architecture

CHAPTER 6: USING SET-BASED REPRESENTATIONS

Another way of representing neural networks is to represent each layer of neurons as a set of neurons. In this case, each set is processed separately in the virtue of being different layers, whereas each neuron within a layer receives no order preference. As we shall see, this simplistic yet representative model is quite a powerful tool to reason about neural networks.

Just like we found a functional approximator to work for graphs, we now have to find a functional approximator to work on sets. The key point here is that no element within a set gets any different treatment based on its order. In order to have such a function on sets, we use a powerful architecture recently introduced by Zaheer et al.[32] called DeepSets.

6.1 WORKING

Zaheer et al. introduce a new architecture that enforces the following property in order to compute a function on the set X.

Property 6.1. *A function f acting on sets must be permutation invariant to the order of objects in the set, i.e.,*

$$f(x_1, x_2, \cdot\cdot, x_n) = f(x_{\sigma(1)}, x_{\sigma(2)}, \cdot\cdot, x_{\sigma(n)})$$

This leads to the next theorem -

Theorem 6.1. *A function S(X) operating on a set X can be a valid scoring function, i.e., it is permutation invariant to the elements in X, if and only if it can be decomposed in the form* $\rho\left(\sum_{x \in X} \phi(x)\right)$*, for suitable transformations* ρ *and* ϕ

We ask the readers to refer to the paper[32] for the proof. Since, we choose to represent these functions using neural networks, both ρ and ϕ are represented using neural networks. Figure 6.1 taken from the same paper illustrates the new architecture which is capable of

taking sets as inputs. We use a neural network ϕ to obtain processed feature representations for all the elements of the set individually, which we add and pass to the neural network ρ to obtain a representation of the set in its entirety.

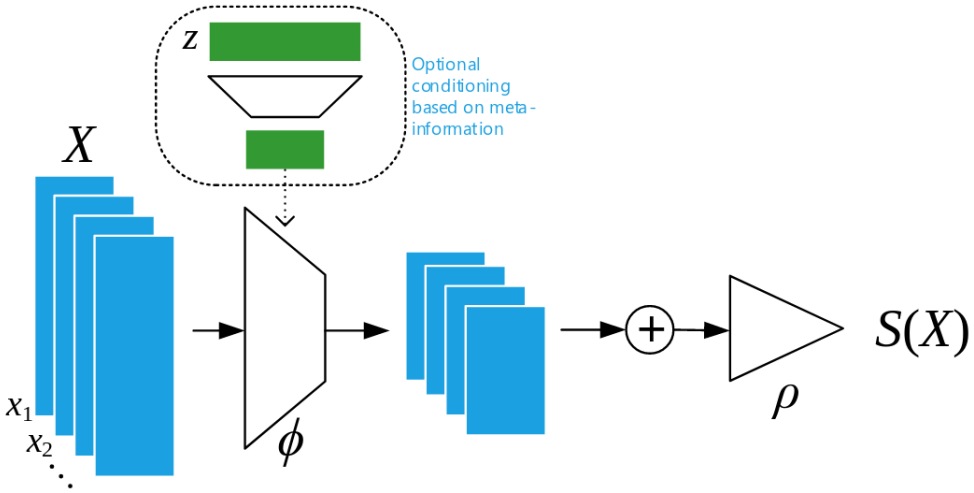


Figure 6.1: DeepSets Architecture from Zaheer et al.[32]

It is interesting to note here that the two functions, ϕ and ρ, can be very lean. This is because ϕ only deals with elements of the set and not the entire set itself. Similarly, ρ deals with the sum of the processed features of each element, which again is a smaller data representation. This allows the DeepSets architecture to be highly computationally efficient with respect to the other architectures used and this is evident in the Results in Table 7.4.

6.2 APPLICATION TO PROPERTY INFERENCE

The main advantage of this architecture is its ability to efficiently solve functional representations for sets. Property 6.1, which the DeepSets architecture possesses, is also a property that we wish any meta-classifier architecture to have - to be invariant to permutations as discussed in Section 3.1.

We do not necessarily want the meta-classifier to be permutation-invariant to the entire flattened parameter vector. We instead want it to be permutation-invariant for neurons of the same layer. In order to do this, we construct ϕ architectures for each layer separately.

That means there will be as many ϕ networks as there are layers in the target classifier or shadow classifiers. Each ϕ network takes as input the weights and the bias of the neuron. Additionally, to provide context for the latter layers of the classifier, we append the weights of each neuron to the ϕ processed feature representation of the neuron the weight is connected to. Finally, the ϕ processed element features for each node in a layer are summed, and we obtained summed feature vectors for each layer. We concatenate these vectors and pass them through a single ρ network that takes the final inference decision. Figure 6.2 illustrates this process.

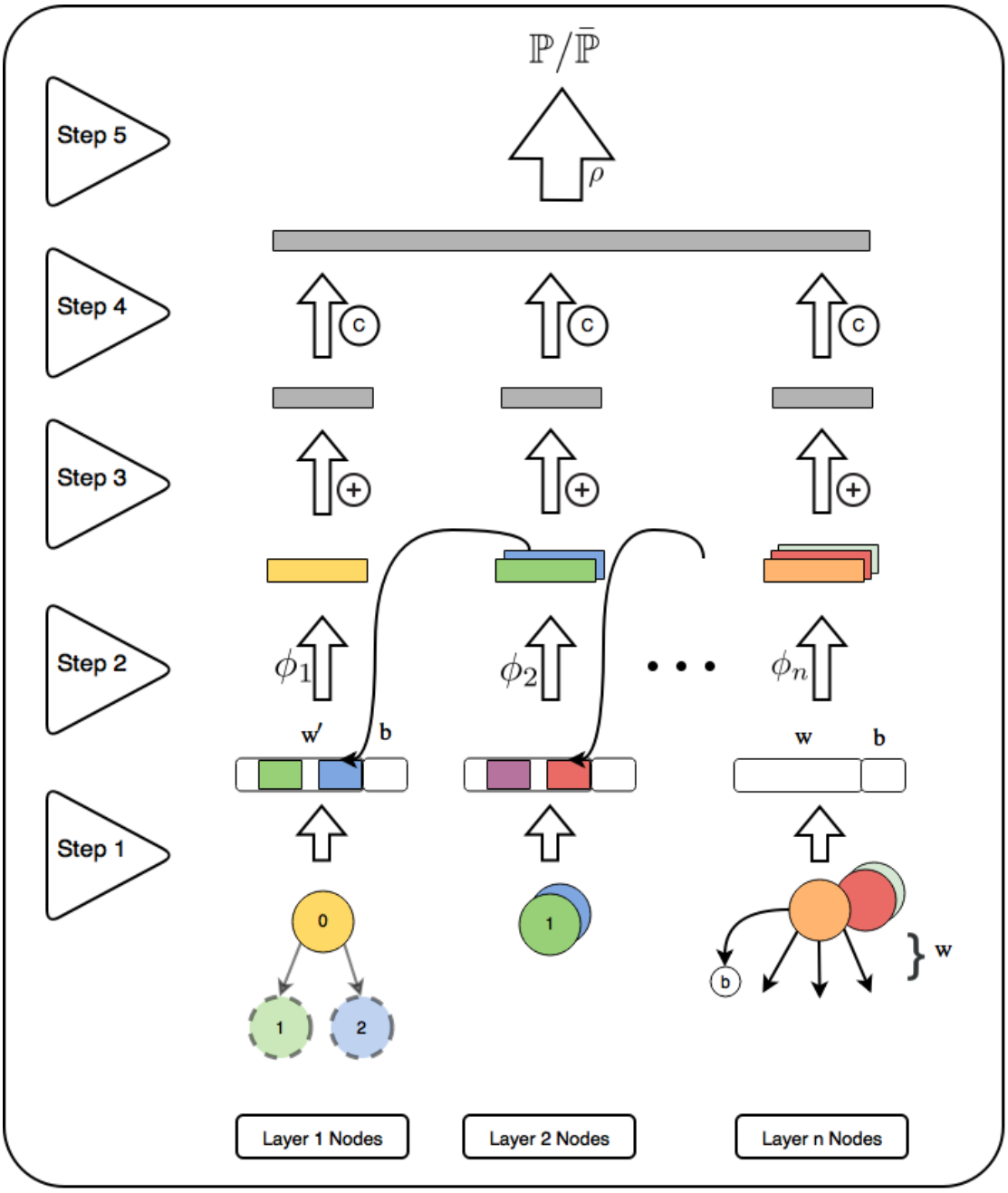


Figure 6.2: Using the DeepSet Architecture (Refer Figure 5.1a for classifier) Steps :

1. For each node, obtain initial feature representation by concatenating its weights and bias. For nodes belonging to layers after the input layer, each weight is also appended with feature representation of the node it connects to in the lower layer.
2. Pass each node through corresponding layer's ϕ network
3. Add feature representations for all nodes in a layer
4. Concatenate feature representations for all layers
5. Pass concatenated feature representation through ρ network

CHAPTER 7: RESULTS

In order to compare the effects of using these different adjustments to the original attack, we use three different datasets ranging from an synthetic dataset to the MNIST digit recognition dataset. We describe the datasets and the properties to be inferred, followed by results in the following sections. For all of the experiments, we generate 2048 classifiers for each property (4096 in total) for the training dataset and 256 classifiers for each property (512 in total) for the test dataset. We end the chapter with a discussion on the results.

7.1 DATASETS

7.1.1 Synthetic Dataset

The first dataset we created was meant to perform a sanity check for the inference tasks. Each data sample within the dataset is a vector of 40 binary digits (which are represented as either 0/1 or -1000/+1000 for different experiments). Each classifier predicts the task of whether or not there are more than a pre-determined number, k, of 1s or +1000s in the data sample. We use a 2 layer neural network classifier in each case. The following are the experiments we ran for inference -

- Property of Training - Predicting if the classifier was trained for 50 or 200 epochs
- Property of Data - Predicting if the classifier was trained on a vector belonging to $\{-1000, 1000\}^{40}$ or to $\{0, 1\}^{40}$
- Property of Problem - Predicting if classifier is trained on $k = 15$ or $k = 3$

For this test, we report the test accuracy of the different trained architectures on the different property inference tasks. The tasks were chosen to differentiate in the type of inference that the meta-classifier was performing, e.g. data property inference could have more to do with

Table 7.1: Comparison of Different Techniques on Synthetic Dataset

Meta-Classifier	Training Property	Data Property	Problem Property
Baseline	83.30%	93.80%	81.60%
Preprocessing	98.00%	98.60%	93.20%
Augmentation	92.60%	97.60%	89.60%
Graph-Based	92.39%	97.46%	92.28%
Set-Based	94.10%	100.00%	97.85%

inspecting earlier layers whereas problem inference could have more to do with inspecting latter layers. The results are shown in Table 7.1.

The preprocessing extension and the set-based representation are the clear winners in this case. All the extensions perform better than the baseline and show at least 7% improvement in accuracy in all tasks.

7.1.2 US Census Dataset

The US Census Dataset[33] is a dataset with anonymized features aggregated from data from the 1994 and 1995 US Census. It contains 41 attributes with possibly sensitive data such as race, sex, citizenship, employment status, etc. Each classifier is trained to predict whether a person with those attributes is likely to earn above $50,000 income or not. We only accept classifiers that score more than 80% accuracy for the test set and use those for the inference task. In this case, we tested inference tasks which predicted a bias towards any particular attribute. Hence, we produced classifiers trained on biased versions of the Census Dataset. The following biases were added to produce different datasets of classifiers

- Normal/Unbiased Dataset - No artificial bias was produced (any inherent bias in the dataset would still be preserved)
- Bias towards Women - Data samples with the female attribute were upsampled and data samples with the male attribute were downsampled
- Bias towards Men

- Bias towards Low Income
- Removing Samples with White racial attribute from dataset - This can be thought of as an extremely biased dataset

The results for different architectures predicting different biases is provided in Table 7.2.

It is clearly noticeable that some inference tasks are tougher than the others, e.g. the task of inferring if a neural network was trained on a dataset biased towards lower income or the normal dataset. Except for the augmentation method, all methods perform better than the baseline in more than half of the tasks. The set-based representation performs impeccably well, having extremely high accuracies for all experiments except for one, where it still performs very well. The next best method is the preprocessing method which scores just slightly below the sets-based representation. The graph-based architecture does not fare better than the two methods, although it provides satisfactorily good results for most of the experiments. The augmentation step proves very tough to train in a lot of cases and ends up blindly returning a single label, which results in 50% accuracy.

7.1.3 MNIST Dataset

The MNIST dataset[34] is a widely used digit recognition dataset. Each data point with the dataset is a 28x28 grayscale image of either of the 10 digits. Each classifier is trained on a 10-way classification task. Although most vision tasks generally utilize convolutional networks to solve tasks, we employ fully connected dense classifiers to do this 10-way classification. Since the MNIST dataset is amongst the simpler vision datasets to solve, a fully connected neural network is also able to obtain more than 90% accuracy for the task. For MNIST, we ran the task of inferring whether the provided dataset was noisy (had a jitter in brightness) or not. The results are tabulated in Table 7.3.

The MNIST task is clearly much tougher than the previous tasks we have considered. Due to increase in the size of the classifiers, the meta-classifiers have to be made larger to be able

Table 7.2: Comparison of Different Techniques on US Census Income Dataset

Bias Prediction	Baseline	Preprocessing	Augmentation	Graph-Based	Set-Based
Normal vs. More Women	55%	89%	50%	81%	97%
Normal vs. More Men	100%	100%	100%	96%	100%
Normal vs. Lower Income	63%	85%	70%	78%	88%
Normal vs. No Whites	93%	100%	50%	97%	100%
More Women vs. More Men	99%	100%	100%	98%	100%
More Women vs. Lower Income	68%	95%	50%	88%	98%
More Women vs. No Whites	93%	100%	50%	96%	100%
More Men vs. Lower Income	100%	100%	100%	98%	100%
More Men vs. No Whites	99%	100%	100%	98%	100%
Lower Income vs. No Whites	95%	100%	100%	97%	100%

Table 7.3: Comparison of Different Techniques on MNIST Dataset

Meta-Classifier	Predicting Noisy Input
Baseline	58%
Preprocessing	64%
Augmentation	50%
Graph-Based	52%
Set-Based	85%

to handle them, as is evident in Table 7.4. The set-based representation performs much better than any of the other methods in this case. We believe that this is because its architecture is inherently more capable to infer properties on neural networks even without having a large number of parameters. The augmentation extension and the graph-based approach prove tough to train and do not obtain good accuracies. The preprocessing extensions provides marginal gains over the baseline.

7.2 DISCUSSION

Except for the augmentation method, most of our discussed extensions prove to perform better than the established prior work. We feel that the reason for the augmentation method not performing well is that the meta-classifier is not representationally powerful enough to learn similarities between all different equivalents of the same classifier.

The graph-based representation performs mildly better than the other methods. We believe that the complexity of the network often makes it tough to train well and so in some of the tougher tasks, it performs poorly.

The preprocessing method performs better than the baseline in all the explored tasks and is very simple to implement as well. We recommend this method for anyone performing inference tasks on neural networks who does not want to employ the better performing but specialized set-based architecture.

Overall, the set-based representation shows the most promise for inference. It is able to

Table 7.4: Comparison of Parameters of Different Architectures

Meta-Classifier	Synthetic Dataset	Census Dataset	MNIST Dataset
Linear	$47.3K$	$1.3M$	$435.7M$
Graph-Based	$217.4K$	$3.3M$	$256.5M$
Set-Based	385	$29.9K$	$270.7K$

obtain very high accuracies for all the tasks and also has the smallest, and consequently easiest to train, architecture. It is often two to three orders of magnitude smaller in number of parameters than the other methods, as can be observed in Table 7.4. Hence, we regard the set-based representation as our best model.

CHAPTER 8: CONCLUSION

While data gains more and more prominence, we realize that a lot of the problems that we faced before can be solved using advanced data processing techniques such as neural networks, which seem to be the major driving forces behind this shift. However, as a wise fictitious man once claimed, with great power comes great responsibility, and so, while harnessing the great advantages of neural networks, it would also be wise to know more about what the neural network exactly represents, how it is able to solve the problems it does and more specifically in our case, what it reveals about the underlying training data.

We provide different architectures and techniques in order to demystify the privacy implications of using neural networks using a property inference attack. We show that preprocessing and augmentation are two quick steps anyone can use to make the inference task simpler. Graph-based representations using GCNs involve larger networks that have the ability to do away with classifier architecture assumptions but take longer to train and need not always perform better than other extensions we have provided. Set-based representations using the DeepSets architecture seems best suited to tackle these problems, except for its assumption of the classifier architecture. It has consistently shown a better performance than the other meta-classifiers for almost all of the inference tasks.

We encourage the community to try these techniques on other inference problems based on neural networks. Property Inference can help us reveal inherent properties of the dataset such as biases, training flaws, etc and is one of the many steps in addressing transparency and interpretability of neural networks. Our work has significantly improved the effectiveness of Property Inference attacks on neural networks and has opened up techniques for further improving inference tasks on neural networks.

REFERENCES

[1] "Wavenet launches in the google assistant." [Online]. Available: https://deepmind.com/blog/wavenet-launches-google-assistant/

[2] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[3] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Computer Vision (ICCV), 2015 IEEE International Conference on*. IEEE, 2015, pp. 2722–2730.

[4] "An on-device deep neural network for face detection - apple." [Online]. Available: https://machinelearning.apple.com/2017/11/16/face-detection.html

[5] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.

[6] "Deepmind health and research collaborations." [Online]. Available: https://deepmind.com/applied/deepmind-health/working-partners/health-research-tomorrow/

[7] "Ibm watson for oncology," Jan 2017. [Online]. Available: https://www.ibm.com/watson/health/oncology-and-genomics/oncology/

[8] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the fifth annual workshop on Computational learning theory*. ACM, 1992, pp. 144–152.

[9] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis." in *USENIX Security Symposium*, 2016, pp. 601–618.

[10] M. Fredrikson, E. Lantz, S. Jha, S. Lin, D. Page, and T. Ristenpart, "Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing."

[11] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1322–1333.

[12] S. Yeom, M. Fredrikson, and S. Jha, "The unintended consequences of overfitting: Training data inference attacks," *arXiv preprint arXiv:1709.01604*, 2017.

[13] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 3–18.

[14] G. Ateniese, L. V. Mancini, A. Spognardi, A. Villani, D. Vitali, and G. Felici, "Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers," *International Journal of Security and Networks*, vol. 10, no. 3, pp. 137–150, 2015.

[15] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, pp. 65–386, 1958.

[16] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[17] P. Langley, W. Iba, K. Thompson et al., "An analysis of bayesian classifiers," 1992.

[18] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[19] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state markov chains," *The annals of mathematical statistics*, vol. 37, no. 6, pp. 1554–1563, 1966.

[20] S. J. Oh, M. Augustin, B. Schiele, and M. Fritz, "Whitening black-box neural networks," *arXiv preprint arXiv:1711.01768*, 2017.

[21] J. Lu, H. Sibai, and E. Fabry, "Adversarial examples that fool detectors," *arXiv preprint arXiv:1712.02494*, 2017.

[22] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 506–519.

[23] S. M. Moosavi Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, no. EPFL-CONF-218057, 2016.

[24] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations."

[25] I. Evtimov, K. Eykholt, E. Fernandes, T. Kohno, B. Li, A. Prakash, A. Rahmati, and D. Song, "Robust physical-world attacks on machine learning models," *arXiv preprint arXiv:1707.08945*, 2017.

[26] C. Song, T. Ristenpart, and V. Shmatikov, "Machine learning models that remember too much," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 587–601.

[27] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," 2017.

[28] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[29] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models."

[30] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[31] Y. LeCun, Y. Bengio et al., "Convolutional networks for images, speech, and time series."

[32] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola, "Deep sets," in *Advances in Neural Information Processing Systems*, 2017, pp. 3394–3404.

[33] D. Dheeru and E. Karra Taniskidou, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[34] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits." [Online]. Available: http://yann.lecun.com/exdb/mnist/