# Knowledge Distillation before Formal Verification of Deep Neural Networks

by

**Jordan Perr-Sauer**

B.S., University of Colorado, Denver 2016

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2022

Committee Members:

Ashutosh Trivedi, Chair

Claire Monteleoni

Fabio Somenzi

Perr-Sauer, Jordan (MS., Computer Science)

Knowledge Distillation before Formal Verification of Deep Neural Networks

Thesis directed by Prof. Ashutosh Trivedi

This thesis explores the potential in applying knowledge distillation to overcome the scaling problem in formal verification of deep neural networks. Instead of verifying the large neural network directly, we first compress the neural network with knowledge distillation and then verify the compressed model, which can then be deployed in safety critical applications. In support of this framework, we provide background information on data-driven modeling with deep neural networks, neural network verification, and neural network compression. A proof of concept using the XOR problem is provided, before a more detailed computational experiment is performed on the benchmark neural network, ACAS-Xu. Compression with knowledge distillation is shown to provide an exponential speedup in formal verification for models of both XOR and ACAS-Xu by reducing the size of these neural networks. While knowledge distillation is able to compress a neural network model of the XOR problem with high fidelity, this was not the case for ACAS-Xu. Overall, we provide evidence to suggest that knowledge distillation can be used to reduce the runtime of formal verification algorithms by first compressing the neural network models.

## Dedication

To the pursuit of knowledge and truth through science.

## Acknowledgements

I thank my committee chair: Ashutosh Trivedi, and the members of my committee: Claire Monteleoni, and Fabio Somenzi. I thank Mateo Perez, Sriram Sankaranarayanan, Sierra Niemiec, and the CU Boulder CLIMB group for providing advice and suggestions along the way. I thank my graduate advisor, Rajshree Shrestha, for helping me to meet the deadlines and requirements. Finally, I thank the National Renewable Energy Laboratory and my colleagues there who supported and enabled me to attend and complete this program.

# Contents

# Tables

**Table**

# Figures

**Figure**

# Chapter 1

# Introduction

Deep neural networks are successful data-driven models, widely used in the fields of machine learning and artificial intelligence to solve a variety of problems. Despite their success and wide deployment in many real-world applications, deep neural networks are considered to be uninterpretable "black-box" models. Such models are unacceptable in safety critical and scientific applications. Sub-fields in machine learning research, such as interpretability, verification, and compression, are producing work to make deep neural networks more interpretable. A fundamental problem facing neural network verification is that it does not scale to very large neural networks. Unfortunately, large neural networks are very common, since they have favorable training characteristics and often lead to more accurate and generalizable models. However, it has been found that large neural networks may contain redundant information which is not strictly necessary to achieve good performance. In this thesis we consider the use of a neural network model compression technique, knowledge distillation, to compress neural network models before formal verification. By verifying a compressed representation of the neural network, we circumvent the need to verify the large neural network in the first place.

Deep neural networks have been used to solve a wide range of problems, especially those problems with unstructured or "raw" data inputs, such as speech and image recognition [36]. One drawback to using deep neural networks is that they are considered to be uninterpretable "black-box" models, which are resistant to inspection of their internal workings [40]. This may be acceptable for some low-stakes applications, such as email filtering, directing a smart speaker to

play a song, or detecting and labeling faces in your phone's photo library, but may not be acceptable for all applications.

Neural networks are increasingly being used in safety critical and other high-stakes applications. In these applications, treating neural networks as black-box models is no longer sufficient. In the case of aircraft collision avoidance, we may be interested in obtaining a proof, or a certificate, that a given control system can avoid collisions under a wide range of circumstances [34, 28]. In the case of machine learning for scientific applications, the explanation is the very output which is being sought, with the accuracy of the model being less important than the robustness and interpretability of its output [6].

A sub-field in the program verification literature has emerged to apply formal verification techniques, which were developed for program verification, to the verification of deep neural networks [2, 39]. These methods can provide formal guarantees of the neural network's behavior by exhaustively enumerating the functional structure of the neural network. These formal verification techniques suffer from high computational complexity (exponential in the size of the network, in the worst case), and many of them are restricted to a subset of neural network architectures [32]. Techniques such as abstraction and refinement are used to help scale the formal verification algorithms by producing over-approximations of the original network, iteratively refining the abstraction until the problem is solved [2, 39]. Unfortunately, most of the useful and popular deep neural networks are still too large for formal verification, even when using the most sophisticated abstraction and refinement techniques. This limits the usefulness of neural network verification today.

At the same time, there are efforts in the machine learning literature around neural network compression, with most applications in embedded systems and low-power-consumption use-cases. Over-parameterized neural networks are known to generalize better and train in fewer epochs [20, 9]. Compression may reduce the redundancy in overparametrized deep neural networks, the removal of which provides a potential pathway to reduce the size of these models without impacting their accuracy. The idea of generating a smallest, simplest model to describe a data problem has appeared many times throughout literature. Sparsity-promoting techniques have been used in the physics-

informed machine learning literature to create simpler, more parsimonious models, that exposing governing equations of the underlying system [12]. Recently, work has been done to enumerate the Rashomon set of models satisfying a given problem, showing that many tabular problems admit simple, sparse, and interpretable models [47].

While large, overparametrized networks may be useful in the training stage, they are difficult to verify [32] and possibly unnecessary for the inference stage. We posit that compressing a neural network using a technique called knowledge distillation before verification can be an effective approach to improve the scale of the verification algorithms while maintaining acceptable accuracy in the compressed network.

### Thesis Statement

By separating the training phase from the verification phase with a knowledge distillation step, we hypothesize it is possible to take advantage of an overparametrized network in the training stage, while leveraging the benefits of a smaller representation in the verification stage. We further hypothesize that this may be applicable to a wide variety of deep neural networks used in practice. We support these hypotheses by designing experiments to answer the following questions.

(RQ1) Can knowledge distillation help scale the neural network verification problem by providing an exponential speedup in verifying deep neural networks?

(RQ2) Can the smaller, verified networks reach a similar level of performance as the larger, original networks, allowing them to be used in applications?

(RQ3) Do compressed networks with high performance satisfy the same formal properties as the original network?

## Statement of Novelty

The knowledge distillation algorithm and formal verification algorithm used in this work are existing ideas coming from the neural network compression [26, 23], and the neural network verification literature [2, 39, 3]. The general idea of training neural networks in a way which makes verification easier is an active area of research [35, 55]. We are not aware of an attempt to perform knowledge distillation before formal verification. The novel aspects of this thesis are: (1) Using knowledge distillation to compress deep neural networks before performing safety verification on the compressed network using a geometric path enumeration algorithm, and (2) results from a computational experiment to explore the efficacy of this method using the benchmark neural network, ACAS-Xu [30]. Both of these contributions are novel in the literature to the author's best knowledge at the time of publication.

# Chapter 2

# Preliminaries

We assume knowledge of the basics of statistical modeling, machine learning and deep neural networks. For a detailed treatment of these concepts, see [22]. In this chapter, we review some of these preliminaries and define the notation which will be used throughout the thesis. We start by defining data-driven models, of which which neural networks are universal approximators that have been successful in practice. We will then discuss techniques in the interpretability literature, leading up to neural network verification and compression in the subsequent chapters.

## 2.1 Data-Driven Models

Here, we define a **mathematical model** for a data-set $X = \{(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}\}_{i=0}^{n}$ as a function $F : \mathcal{X} \to \mathcal{Y}$ which closely matches and describes $X$ such that for all $(x, y) \in X$ we have that $F(x) = \hat{y} \approx y$ (read $F$ models $X$, or $\hat{y}$ approximates $y$, up to some error). Mathematical models are often **parameterized** by parameters $\theta$, which are tuned or optimized through mathematical optimization. We write a parameterized model as $F(\theta; x)$.

A good example of a mathematical model is the linear model,

$$F_{\text{linear}}(x; \theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n,$$

where $\theta, x \in \mathbb{R}^n$. The parameters of the linear model are most commonly optimized using linear-least-squares, which admits relatively efficient algorithms for exact computation. In this case, we first define a loss function as the mean squared error: $\mathcal{L}_{\text{mse}} = ||\hat{y} - y||^2$, and minimize this loss

function by finding optimal parameters $\theta$. If the model fits the data well, we can interpret the model parameters $\theta_i$ as the proportion of which the output $\hat{y}$ changes for any change in input $x_i$. If $\theta_i < 0$, we know $x_i$ is negatively correlated with $\hat{y}$, and vice versa. Due to its well known interpretability and simplicity, linear models are the preferred model if the data allows for it.

A model which describes a data-set well is said to be valid, or accurate. Once an accurate model is created from the data, it can be used for inference (predicting the output for yet-unseen input). Predicting in-sample data is known as interpolation, whereas predicting out-of-sample data is known as extrapolation. A model which provides insight into the system, such as the parameters of the linear model which are proportional to the output, are said to be interpretable. Of course, many physical systems and data-sets are non-linear (they can not be accurately described by a linear model), and therefore the linear model would not be valid in these cases.

The pursuit of interpretable mathematical models for such non-linear systems found in nature is a foundational activity in modern science. These models typically involve algebraic relationships between variables and perhaps their derivatives, and much of scientific discovery is based on finding new models which more accurately describe experimental data. The distinction between searching for a new model structure and optimizing parameters is fuzzy, but in general we will consider those parameters which dictate the structure of a model to be called **hyperparameters**. There is a sub-field of machine learning (AutoML for the more generic case, and Neural Architecture Search in the case of neural networks) which concerns this hyperparameter optimization, or automatic search for an optimal model architecture [27]. Another approach, in contrast to searching over hyperparameters describing the optimal model structure is to instead optimize the parameters of a universal approximation model.

## 2.2 Universal Approximation with Deep Neural Networks

There are many cases where the underlying data does not admit an obvious mathematical model, where one is not yet known, or where traditional modeling techniques have failed to produce accurate models. Two major examples of these types of types of problems include image recognition

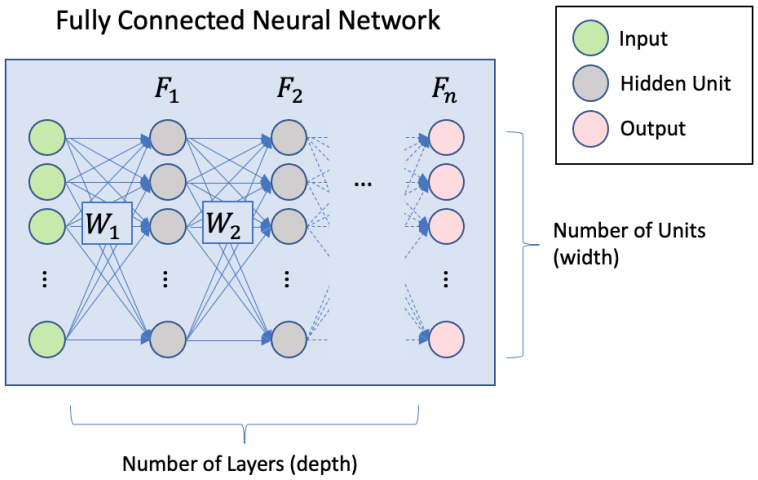



Figure 2.1: **A fully connected neural network.** It features the input layer, the output layer, the hidden layers ($F_i$), and weight matrices ($W_i$). We do not show the bias and non-linearity, which would be computed inside the hidden units.

and natural language processing. In these cases, practitioners employ universal approximation, or surrogate modeling, to generate accurate models. These models are built to achieve high accuracy and are tuned until they achieve good performance on inference tasks, but typically do not provide good interpretability. Some examples of universal approximators include random forests (decision trees) and deep neural networks. The structure of these universal approximators may have little in common with the structure of the domain problem, yet have a large enough number of parameters which allow them to be very flexible. Further, the structure of these universal approximators are built to admit fast optimization algorithms, such as gradient descent.

We define the **fully connected, feed forward, deep neural network** recursively as

$$F_n(x;\theta) = \phi\left(F_{n-1}^T W_n + b_n\right) \tag{2.1}$$

with parameters $\theta = ((W_1, b_1), (W_2, b_2), \ldots, (W_n, b_n))$, where $(W_i, b_i)$ are the weights matrices and biases vectors of layer $i$, respectively, and the number of hidden layers $n > 1$. We define $\phi : \mathbb{R} \to \mathbb{R}$ as the **activation function**, also known as the non-linearity, which can be any differentiable nonlinear function. The activation function is applied element-wise to an input vector. A common activation function is the rectified linear unit (ReLU), defined as $\phi(x) = \max(0, x)$. Figure 2.1

shows a fully connected neural network. We call the output of each hidden unit in layer $i$ the activation of that layer, $F_i$, and note that is a vector. It is worth noting that the fully connected neural network becomes a linear model in the degenerate case of $n = 1$ and $\phi = \mathbb{I}$ the identity function.

The parameters $\theta$ are fit by minimizing a **differentiable loss function** $\mathcal{L}$ using an **optimization algorithm**, also known as the optimizer. One of the most common loss functions used for multidimensional, continuous output, is the mean squared error (mse), which can be written using the $l_2$ norm of the prediction in the same way as for a linear model

$$\mathcal{L}_{\text{mse}}(y, \hat{y}) = ||\hat{y}_i - y||^2 = \frac{1}{|y|} \sum_{i=0}^{|y|} (\hat{y}_i - y_i)^2 \tag{2.2}$$

where $|y|$ is the number of data points in the data-set. Another common loss function, used for classification, is the categorical cross-entropy loss:

$$\mathcal{L}_{\text{cce}}(y, \hat{y}) = \frac{1}{|y|} \sum_{i=0}^{|y|} \sum_{c=0}^{|y_i|} y_{i,c} \log(\hat{y}_{i,c}), \tag{2.3}$$

where $y_{i,c}$ is the $c^{\text{th}}$ component of $y_i$. This loss contains a double sum. The inner sum is across all categories $c$ and assumes a one-hot vector for all outputs $y$ (that is, zeros for every dimension except for that which corresponds to the desired class, which is takes the value of one). The outer sum then takes the average of this value across all data points.

The most common method for optimizing the parameters $\theta$ with respect to the loss function is gradient descent, made possible by the **backpropagation algorithm** [46] which efficiently computes the gradient of the loss function (including the entire deep neural network) with respect to each parameter to be optimized. Two variants of gradient descent commonly used in practice are stochastic gradient descent (SGD) and adaptive moment estimation (ADAM) [33].

It has been shown that neural networks can approximate any smooth function to arbitrary precision in the infinite-width case [15]. In other words, given a wide enough network, it is technically possible to pick parameters $\theta$ to approximate any data-set to arbitrary accuracy. However, in the real-world case with finite networks, the architecture of a network (such as the number of hidden

layers, the width of each layer, and the type of activation function) are treated as hyperparameters which are tuned by the practitioner using a search method.

One counter-intuitive finding from the deep learning literature is that large, overparametrized networks both learn faster and tend to generalize better than small networks [36]. The dynamics of this are still not well understood. The most widely accepted explanation has to do with the abundance of saddle points in the loss landscape of a highly overparametrized network [16]. Other explanations include the lottery ticket hypothesis [20] and deep double descent phenomenon [9]. Modern neural networks may have billions of parameters, and specialized architectures that are optimized for the structure of data in a given problem domain, such as the convolutional neural network for images and transformer network for natural language processing.

## 2.3 Interpretability of Deep Neural Networks

Although deep neural networks with high accuracy have solved problems in many application domains, they are large, dense structures which do not admit a natural interpretation. Deep neural networks have been described as "black-box" models. This is due, in part, to the large number of parameters inherent in the definition of the neural network (Equation 2.1). In this definition, the parameters $\theta$ contain multiple matrices and vectors, of which the real-valued elements each have a nonlinear relationship with the output. This non-linearity prevents a clear interpretation of each parameter with respect to the output, as we have for linear models. This resistance to interpretability and introspection limits the applicability in safety-critical scenarios.

A recent book by Molnar describes current methods in machine learning interpretability [40], with a chapter towards the end detailing the application of these methods to deep neural networks. According to Molnar, interpretability of machine learning models can be categorized in two ways: Extrinsic methods and intrinsic methods. In an extrinsic method, the model is treated as a black-box, and the changes and sensitivities of the output to perturbations of the inputs are used to study the model. Examples of these methods include sensitivity analysis, feature importance, individual conditional expectation (ICE) plots, SHAP scores, and salience maps. The Intrinsic methods rely

on the internal structure of the black-box model and may share mathematical characteristics with the extrinsic methods otherwise. Examples of intrinsic methods for deep neural networks include linear probes, layer-wise correlation analysis, and decomposition of hidden layers.

Feature visualization is another promising approach to interpret deep neural networks [19]. DeepDream, or inceptionism, was one popular example of feature visualization, producing mesmerising images that captured the imagination of many [41]. Feature visualization techniques work by searching for an input which maximizes the activation of a selected unit in the neural network. Related to feature visualization are salience maps, which visualize the gradient of some selected hidden unit with respect to each element for a given input [48]. Recently, a concept called circuits has been introduced for feature visualization for multiple neurons that potentially form higher level concepts in the neural network [43].

In another family of techniques, the singular value decomposition is used to decompose the activation space of the hidden layers ($F_i$, where $1 < i < n$) with respect to the input data. In a method called singular vector canonical correlation analysis (SVCCA) [44], the response of each component is correlated across layers, producing a measure of the similarity of the internal representation of each layer. Similarly, linear probes fit linear models from each hidden layer to the output, in order to quantify the improvement of the internal representation at each layer [1].

It has been show that image classification networks fail to be robust to small perturbations in the input image [51]. For example, a small sticker placed on a stop sign may cause an autonomous vehicle to miss the sign completely, potentially causing a dangerous accident. This has created a sub-field of deep learning literature to study the so-called adversarial robustness of deep neural networks. These papers study the types of small changes that can modify the output of a deep neural network, and techniques to generate networks that are more robust.

Neural network verification is sometimes included as a concept within the space of neural network interpretability techniques. In neural network verification, we desire some formal guarantee, certificate, or proof about the behavior of a neural network. Techniques to accomplish neural network verification are being developed in a sub-field of the formal program verification literature,

which we will describe in the next chapter.

# Chapter 3

## Verification of Deep Neural Networks

The verification of deep neural networks shares concepts and techniques from the broader field of program verification, of which deep neural networks are a special case. The techniques considered here, such as path enumeration and abstraction and refinement, have parallels in the program verification field. For an introduction to the concepts of neural network verification, see [2]. A thorough review paper with specific details about the implementation of different neural network verification techniques in the literature can be found in [39].

All verification algorithms take as input the neural network $F$, a Boolean predicate $p$ (which is known as a **property**), and return whether or not $F$ satisfies $p$. In the next section, we discuss the types of properties that have been used in the verification literature. Later in this chapter, we will describe the different approaches that are taken to perform this verification, and present a deep dive into one such algorithm. The goal is to provide the reader with an overview of this field, and to explain the fundamental bottleneck preventing practitioners from scaling these verification algorithms to large neural networks.

A fundamental open problem in the field of neural network verification is that of scaling to large neural networks. The problem of neural network verification has been shown to be NP-Complete in the size of the network for neural networks with the ReLU activation function [32]. This means that in the worst case the verification problem scales exponentially with the size of the network, making it infeasible for most modern deep learning models which may have billions of parameters.

## 3.1 Verifiable Properties of Deep Neural Networks

We may verify a trained neural network $y = F(\theta; x)$ with fixed parameters $\theta$ and an input domain $\mathcal{X}$ and output range $\mathcal{Y}$, in the sense that we can prove that the network will always satisfy a property $p$ for all possible inputs $x \in \mathcal{X}$. There are several types of properties that we can verify in a neural network which have received attention in the literature: Safety (open loop), robustness, and reachability (closed loop safety) properties are some of the most commonly studied today. We will enumerate and provide examples for some of these types of properties.

**Safety properties**, (further defined as **open-loop safety properties** in some sources) are broadly stated as guarantees that the output remains in some safe region over all inputs of interest. Common applications of safety properties include guaranteeing that a network will not produce outputs that are not physically possible, or that might damage a physical system. Given a safe subset of the input $\mathcal{Y}_{\text{safe}} \subseteq \mathcal{Y}$ and a safe subset of the output $\mathcal{X}_{\text{safe}} \subseteq \mathcal{X}$, a safety property is:

$$p_{\text{safety}} = [\forall x \in \mathcal{X}_{\text{safe}} \text{ we have } F(x) \in \mathcal{Y}_{\text{safe}}] \tag{3.1}$$

A visualization of such a safety property is given in Figure 3.1.

In the literature, the input and output sets $\mathcal{X}_{\text{safe}}, \mathcal{Y}_{\text{safe}}$ may be restricted to be **convex polytopes**, which can be defined as the intersection of a set of half-spaces. This is a convenient form for verification algorithms that operate over monotonically increasing activation functions and feed-forward fully connected neural networks. In order to verify properties across arbitrary regions of the input domain, we decompose the input domain into a set of such polytopes and then verify the property for each input polytope individually. In a recent review paper, six types of input sets are considered: Halfspace, Halfspace-polytope, Vertex-polytope, Hyper-rectangle, Zonotope, and Star-set [39]. Different algorithms in the literature take advantage of the special properties provided by each of these sets.

**Robustness properties** are most studied in the context of uncovering the existence of adversarial examples, which are small changes to an input that cause large changes in the output. Common applications of adversarial examples include the ability to fool image recognition systems

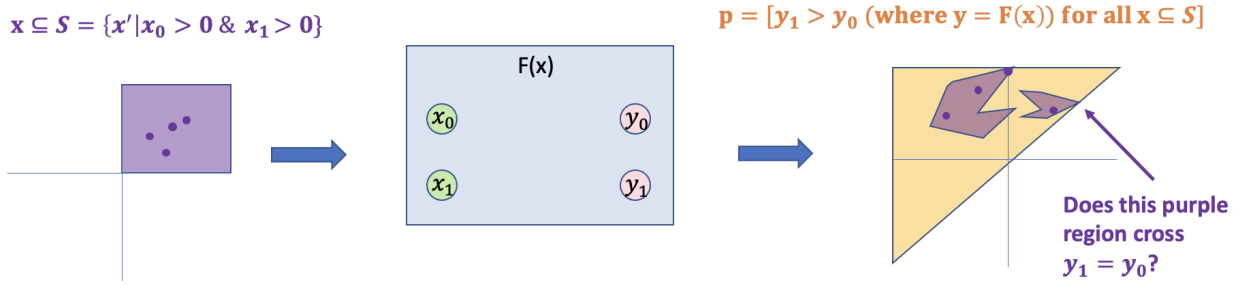



Figure 3.1: **The open-loop safety verification problem**. On the left, the safe input region $\mathcal{X}_{\text{safe}}$ is denoted in purple. The region is propagated through the neural network and shown projected into the output space on the right. The yellow region represents the safe output region represented in the verification property $p$. The goal of open-loop safety verification is to determine if every point inside $\mathcal{X}_{\text{safe}}$ gets mapped to a point inside the safe output region.

by modifying a small number of pixels of the input image. The robustness property $\phi_{\text{robust}}$ for a neural network $F$ characterizes the condition that the output will not change within an $\epsilon$-ball ($\mathbb{B}(x, \epsilon)$) around a given input $x$, i.e.

$$p_{\text{robust}}(x) := [\forall x' \in \mathbb{B}(x, \epsilon) \text{ we have } F(x') = F(x)]. \tag{3.2}$$

**Reachability analysis**, also known as **closed-loop safety** properties in some sources, is an extension of open-loop safety. Here, we consider a neural network controller $F(\theta; x)$ which is embedded within some dynamical system evolving in time. We may want to ensure that it is impossible for the system as a whole to ever reach a bad state. Reachability analysis is common for physical controllers, such as guidance systems. We can state complex reachability properties in terms of a temporal logic. In [29], Julian et al. use a reachability analysis of an aircraft guidance system, ACAS-Xu, to ensure that a collision is not possible between aircraft at any future point in time, as long as the control signals of the guidance system are followed.

## 3.2 Methods for Neural Network Verification

The choice in verification algorithm depends on the type of property, the architecture of the neural network, and the desired guarantees (statistical vs. formal guarantees). Formal verification

strives to produce a proof that the property will always be satisfied under every possible input.

Verification algorithms takes as input the neural network $F$ and a property $p$, and return one of three possible outputs: **Valid** (the property is satisfied for all possible inputs), **invalid** (there exists some input which falsifies the property), and **unknown** (the algorithm has failed to determine whether or not the property is valid or invalid). Furthermore, the verification algorithm may have any of the following properties: **Completeness** (for all properties, the algorithm will return valid or invalid), **soundness** (if the algorithm outputs valid or invalid, then the property is actually valid or invalid, respectively), **termination** (the algorithm will terminate, providing an output, in finite time).

There are several different approaches in the literature to develop and scale verification algorithms. We provide a high level overview of some approaches found in the literature. First, we have formal methods and statistical methods. Within formal methods, a recent review paper lists four sub-types [39]:

(1) **Formal Verification** include methods which exhaustively explore the functional structure of the network, providing formal guarantees that apply for every possible input.

   (a) **Reachability approach**, also called geometric or abstraction-based techniques, are methods which formulate the problem as propagation of sets with certain geometries through the neural network function.

   (b) **Optimization approach**, also called constraint-based, are methods which formulate the problem in a mathematical optimization framework, representing the functional structure of the network as constraints, and then apply techniques from the mathematical optimization literature (e.g, representing the problem in its dual form) to solve the problem. The optimization approaches are typically SMT-based, or MILP-based.

   (c) **Search approach** are methods which add hyperparameters to control the level of abstraction, or other aspect of the formal verification algorithm. This creates an outer-loop search over these hyperparameters which can be used to tune the verification.

(d) **Training + Verification approach** are methods which augment the training of the neural network to simplify the verification problem. Some such techniques include compressing or pruning the network before verification [35], training to minimize the number of ReLUs in the posneg case [55], or training to enforce a Lipschitz-type smoothness property over the output of the network.

(2) **Statistical Verification** are methods which provide only probabilistic statements about the validity of a network [54, 7]. Such methods may rely on random algorithms such as a Monte Carlo sampling of data points within the input region.

One type of algorithm that falls into the search and reachability-based sub-types is called range propagation [17]. There is a family of software which implements geometric path enumeration, including Reluplex [32], NNV, Marabou, and Sherlock [49]. In this section, we describe the method of geometric path enumeration and some of the optimizations which are implemented in NNenum [3]. This software implements sound and complete verification, capable of verifying safety properties in ReLU networks. We focus here on the method behind NNenum because we will use this software in later chapters to perform the verification step in our knowledge distillation before verification pipeline. NNenum was able to verify the most properties in the ACAS-Xu benchmark neural network, achieving highest score for this benchmark, in the VNN-Comp 2021 competition [4].

## 3.3 Geometric Path Enumeration with NNenum

Let us explore one algorithm that solves the open-loop safety verification property (Equation 3.1) using techniques from the reachability and search subtypes, to make the verification problem more concrete. Let the safe output region $\mathcal{Y}_{\text{safe}}$ be defined by a set of linear constraints, so it is possible to restate the property in terms of a single real variable,

$$p_{\text{safety}} = [\forall x \in \mathcal{X}_{\text{safe}} \text{ we have } y* > 0 \text{ where } y* = b^T y] \tag{3.3}$$

for some $b \in \mathbb{R}^n$. This is functionally equivalent to adding an extra layer to the end of the neural network with an identity activation function and one output neuron. By performing this linear

transformation on the output of the network, we have projected the output into one dimension. Now, we can state the open-loop safety verification problem as a minimization problem:

$$\begin{aligned} \min_{x} \quad & y* = F_n \\ \text{s.t.} \quad & \hat{F}_n = W_n F_{n-1}(x) + b_n && (3.4) \\ & F_n = \phi(\hat{F}_n) && (3.5) \\ & F_0 = x, x \in \mathcal{X}_{\text{safe}} && (3.6) \end{aligned}$$

The interesting constraints in this minimization problem are the non-linearities $\phi$ (Equation 3.5) and the definition of the safe input set $\mathcal{X}_{\text{safe}}$ (Equation 3.6). Geometric path enumeration works by making simplifications to both of these constraints while preserving the soundness of the algorithm. In the extreme case where the non-linearity $\phi = \mathbb{I}$, and $S$ being a convex polytope, then we are left with only linear constraints (Equation 3.4), and the problem simplifies into a linear program. The main idea of these neural network verification algorithms is to linearize the problem, either by splitting it into multiple linear sub-problems (path enumeration), or by over-approximating the nonlinear constraints with linear ones (abstraction).

The geometric path enumeration algorithm is described in [5], and is based on the Reluplex algorithm [32]. We begin with an input set $\mathcal{Y}_{\text{safe}}$ defined as a star-set (or a collection of star sets). Star-sets are used due to their efficient support of affine transformation and intersection with the half-space. The star set is defined as an abstraction of a polytope $P$

$$\mathcal{X}_{\text{star}} = \{x \in \mathbb{R}^n \text{ such that } x = c + Gx', x' \in P\}$$

where $c$ is a vector called the center of the star-set, and $G$ is called the generator matrix. In the case $\mathcal{X}_{\text{safe}}$ is not already a star-set, a conversion must be made. The input set must be represented as the intersection of star-sets.

One the input set $\mathcal{X}_{\text{safe}}$ is defined, we propagate the set through the neural network, iterating over one neuron at a time, starting from neurons in the first hidden layer. The affine transformation by the weight matrix is efficient with star sets, so this step is relatively easy. To deal with the ReLU

non-linearity, recognize that each neuron will be in one of three cases based on the projection of the input set in the previous layer. For each non-linearity, the input set either be bounded completely below zero (negative case), completely above zero (positive case), or have some positive and some negative regions (posneg case).

In the positive case, the input region propagates through the ReLU undisturbed (since the ReLU is the identity function in the positive case). In the negative case, the input region is projected to zero. In the posneg case, where some parts of the input region are located below zero and others are located above zero, the problem is split into to two sub-problems. In one sub-problem a constraint is added restricting the input of this neuron to the negative region and the output is again projected to zero. In the other sub-problem, a constraint is added restricting the input of this neuron to the positive region, and the output is again unmodified. Both sub-problems must be solved, with the original problem being valid only if both sub-problems are valid. It is precisely this split into two sub-problems which gives path enumeration its exponential time complexity in the size of the network. A diagram of this technique is provided in Figure 3.2.

NNenum [5, 3] is a software package which uses this technique [52] to solve the neural network verification problem. NNenum was the only tool able to verify all properties of the ACAS-Xu network, and placed 8th overall in a combined score across all benchmark problems tested, in the VNN-Comp 2021 competition [4].

## 3.4 Abstraction-based Techniques

In general, the path enumeration problem has an exponential worst-case computational complexity, $O(2^n)$ [32]. One approach to scaling the verification algorithms and overcoming the exponential search is through abstraction. Here, the neural network is simplified in a way that preserves the soundness of the verification query. One may consider abstraction as a special type of compression, one which is limited in the type of error it can introduce. After abstraction, networks are smaller and require less steps to verify. However, there is a chance that the abstracted network will fail to verify when the original network should have verified. Thus, abstraction-based methods

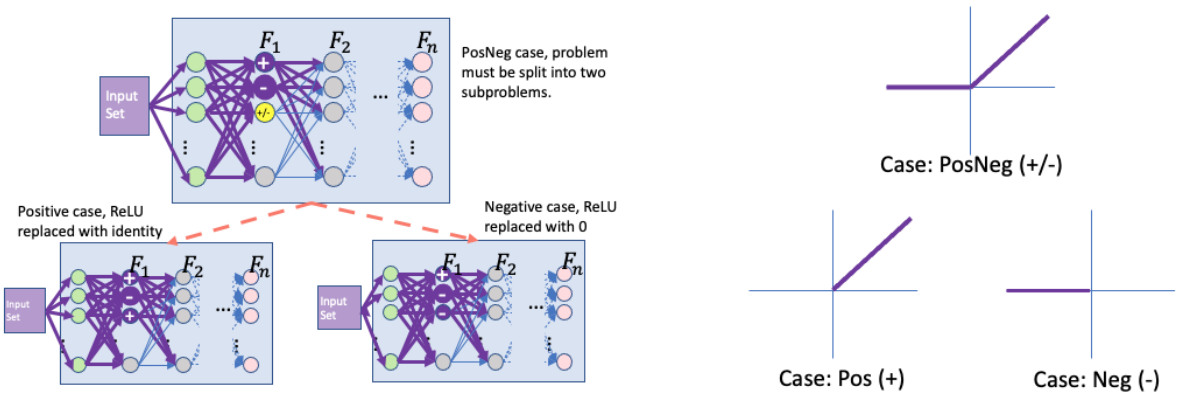



Figure 3.2: **The geometric path enumeration algorithm**. In the case of a posneg ReLU (highlighted in yellow), the problem is split into two sub-problems where this neuron is restricted to be strictly positive or strictly negative. A diagram of the ReLU function over these three cases is shown to the right.

may be incomplete. One examples of an abstraction-based method is the linear relaxation of the ReLU in Neurify [53].

Some researchers have experimented with embedding abstraction criteria into neural network training methods. For example, Xiao et al. [55] studies the effect of adding sparsity constraints in a deep, fully-connected network with ReLU activations, finding speed-ups of multiple orders of magnitude with only 10% impact on model accuracy. Elboher et al. [18] describes a counter-example guided abstraction-refinement framework for verifying ReLU-based networks. In another paper, Gokulanathan et al. [21] describes a method to abstract fully-trained ReLU networks by combining hidden units within a given layer.

## 3.5 State-of-the-art, Community, and Software Packages

There is a competition held each year to test the best neural network verification tools against a set of benchmark problems. The 2nd International Verification of Neural Networks Competition (VNN-Comp) was held 2021 in conjunction with the 4th Workshop on Formal Methods for ML-Enabled Autonomous Systems (FoMLAS) which itself was held as part of the 33rd International

Conference on Computer Aided Verification (CAV). A report is available which lists the attributes of the best performing networks [4]. The top score in this competition went to Alpha-Beta Crown.

Conference and workshop venues in this field include the aforementioned FoMLAS at CAV, the Federated Logic Conference (FLOC), the NASA Formal Methods Symposium, the International Conference on Software Engineering (ICSE), the ACM Principles of Programming Languages conference (POPL). Conferences which may also feature submissions on neural network verification include the International Conference on Autonomous Agents and Multi-agent Systems (AAMAS), the International Conferences on Learning Representations (ICLR), the International Conference on Machine Learning (ICML), and the International Conference on Neural Information Processing Systems (NEUR-IPS), and the Association for the Advancement of Artificial Intelligence (AAAI).

The field is starting to produce standard data formats around the development of neural network verification engines. One effort in this space is the VNNLIB standard for benchmarks [8], which recommends the ONNX file format for neural networks, and the SMT-LIB format for property specification. The two benchmark example problems listed in this standard are ACAS-Xu (which will be used in this thesis) for safety properties and MNIST for robustness properties.

State-of-the-art research is focusing on how to scale verification algorithms to larger neural networks. We believe that neural network compression techniques, such as knowledge distillation, can help. Therefore, we introduce the field of neural network compression in the next chapter.

# Chapter 4

# Compression of Deep Neural Networks

The goal of compression algorithms is to reduce the number of bits of information it takes to encode some data. Here, we are compressing neural network models. The reduction in model size may reduce the number of computational steps involved at the inference stage, speeding up the computation. Much of the literature on model compression has applications in embedded systems, where practitioners are trying to fit ever larger models on mobile devices and other portable systems where low power consumption and memory requirements are critical. It is important to distinguish between two related fields: **Model compression** (which is the subject of this chapter), and **neural compression** (which refers to the use of neural networks to themselves compress the information in a data-set). Here, we describe model compression for neural networks, where the aim is to reduce the size of a previously trained neural network without impacting the accuracy. This section will provide background for, motivate, and introduce the method of data-free knowledge distillation. This method will be used to perform model compression before verification in Chapter 5.

## 4.1 Methods for Neural Network Compression

Several high-level approaches for model compression are identified throughout the literature [14, 24]. Compression algorithms, in general, may be classified as **lossy** or **lossless**, based on whether or not they preserve exactly the behavior of the original model. Metrics used to judge the effectiveness of a compression algorithm include the **compression ratio** and the **speedup rate**, which describe the decrease in size (in bits) to describe the model, and the decrease in computation

time to use the model on an inference task, respectively.

Some of the techniques used for model compression include: Parameter pruning, structural matrix constraints, quantization and encoding, decomposition and low-rank approximation of the weight matrices, training with sparsity-promoting regularization, and knowledge distillation [23]. Most practical methods of compression in the literature rely on alternating model reduction (which reduces accuracy) and re-training steps (which increases the accuracy again).

Parameter pruning has a long history. The idea of removing weights with low magnitude, or low contribution to the gradient with respect to the loss function, have been in the literature for decades [37, 42]. A recent survey paper by Blalock et al. [10] identifies problems comparing against papers with different parameter pruning techniques, and suggests a suite of benchmark problems, baseline pruning algorithms, and quality metrics called ShrinkBench. Recent work in the parameter pruning literature frames pruning within the Koopman operator theory [45] to identify hidden units that have little impact on the training process.

There has been some cross-over between the neural network verification literature and the model compression literature. Gokulanathan et al. [21] used formal verification to identify hidden units which do not impact the output of the network (within some tolerance $\epsilon$), and then pruned these units out of the network. By using this technique, they showed the ability to remove up to 10% of the hidden units in an ACAS-Xu network. Further, a paper by Han et al. [24] which used a three-step pipeline (pruning, quantization, and Huffman coding) to compresses AlexNet by 35x and VGG-16 by 49x without any loss in accuracy has been cited as an exciting future direction in the formal verification literature.

For the purposes of this thesis, we pick one method of model compression, knowledge distillation, to explore further in our verification experiment. We choose knowledge distillation due to its relative simplicity, the empirical success of the method in the literature, and the number of applications of knowledge distillation that have focused on interpretability.

## 4.2 Knowledge Distillation

Knowledge Distillation is a method to compress neural networks, popularized by Hinton et al. [26], based on earlier work exploring model compression by Bucilua and Caruana [13] and Ba and Caruana [38]. Knowledge distillation involves using transfer learning to train a smaller **student** network with the help of a larger, overparametrized, pre-trained **teacher** network (or, as studied in [13], an ensemble of more generic teacher models). There are many variants of knowledge distillation that vary in the implementation details. Here, we describe Hinton's formulation of knowledge distillation, some theoretical justifications for why the method should work, and cover one variant, data-free knowledge distillation.

Let the teacher network be defined as $F_{\text{teacher}} : \mathcal{X} \to \mathcal{Y}$. We will train a student model over the same domain and range $F_{\text{student}} : \mathcal{X} \to \mathcal{Y}$, to approximate the teacher model, so $F \approx \hat{F}$. In [26], Hinton limits $\mathcal{Y} \subset \mathbb{Z}$, so that the teacher is a discrete classifier network. In this case, the output layer of each network have a semantically meaningful interpretation, the output logits, and represent the probability or weight that a given sample belongs to a given class. In Hinton's original formulation, the student network is trained using a loss function which is a combination of the output logits of the teacher network and the output class. This combination is controlled using the parameter $\alpha$. Following the implementation in the Keras Documentation [11], we can write loss function for Hinton knowledge distillation as

$$\mathcal{L}_{\text{kd}} = \alpha \mathcal{L}_{\text{cce}}(y, \hat{y}) + (1 - \alpha)\mathcal{L}_{\text{cce}}(\text{Softmax}(z), \text{Softmax}(\hat{z})).$$

Where the softmax function for a given temperature $T$ is defined as

$$\text{Softmax}(z) = \frac{e^{z/T}}{\sum_i e^{z_i/T}}$$

We provide a diagram of Hinton's distillation method for classifier network in Figure 4.1.

The work of Hinton et al. [26] provides empirical evidence that knowledge distillation can compress neural network models without sacrificing accuracy. A survey paper by Gou et al. [23] provides tables with dozens of examples from the literature that show knowledge distillation working

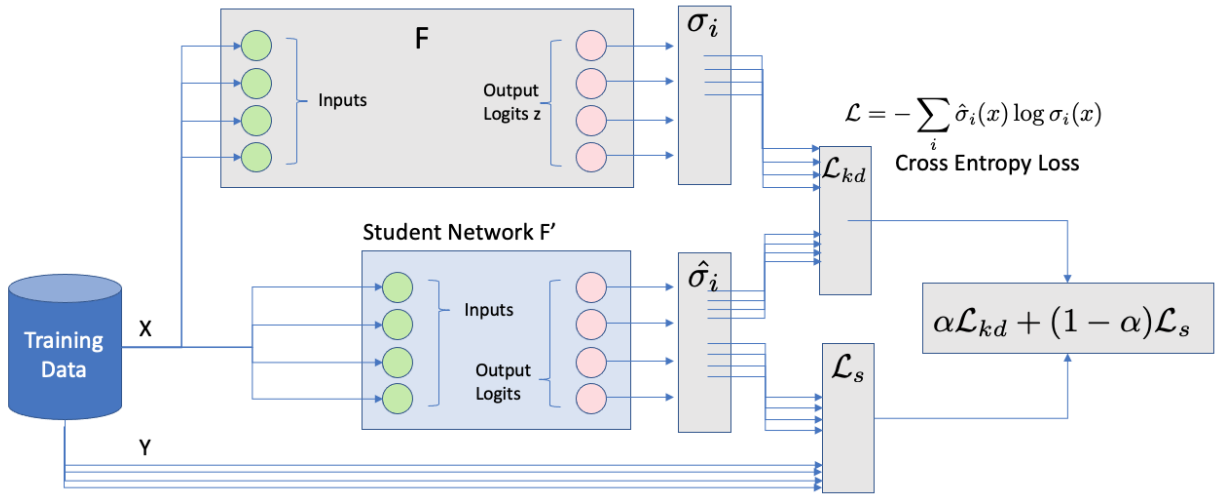



Figure 4.1: **Classic knowledge distillation of a classifier network**. The input training data $X$ is used to activate the teacher network $F$ and the student network $F'$, producing outputs which are fed into the softmax function $\sigma_i$. The cross-entropy loss $\mathcal{L}_{\text{kd}}$ is then used to compare the outputs of the student and teacher networks. Finally, the outputs of the training data $Y$ are compared with the outputs of the student network using a separate cross-entropy loss function $\mathcal{L}_s$, which is combined with the $\mathcal{L}_{\text{kd}}$ using a parameter $\alpha$.

better to compress neural networks than re-training from the training data alone.

Some theoretical justifications for the success of knowledge distillation include (1) access to potentially infinite number of training samples by the student network, (2) access to so-called "dark knowledge" from the teacher network, which is described as the correlation in output logits of different classes, and (3) a potential regularizing effect of using the teacher network to augment the classification problem, instead of sparse input data, which has been compared to label smoothing.

Still other work is skeptical of knowledge distillation. Stanton et al. shows that knowledge distillation often fails to produce students that can replicate the teacher network in high fidelity [50]. Still, the generalization performance of student networks is shown to be improved. There are many variations on the knowledge distillation concept which have been explored in the literature. One such variant is data-free knowledge distillation, which will be described in the next section.

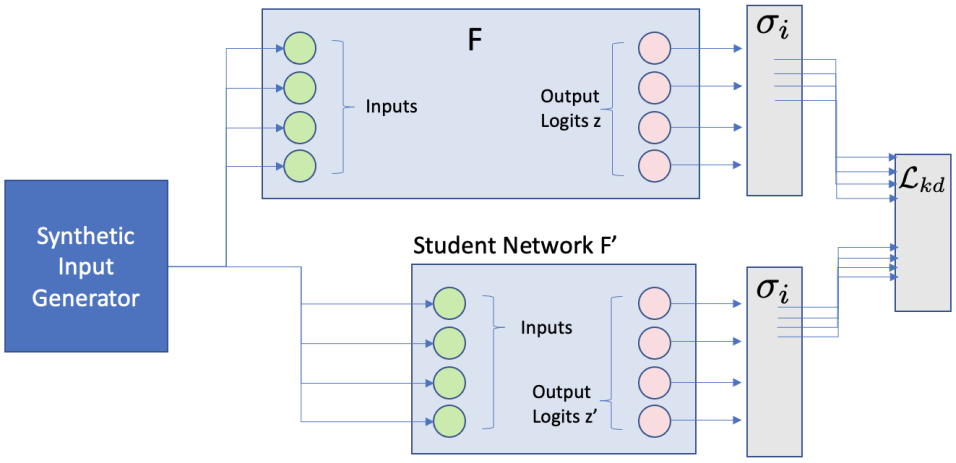



Figure 4.2: **Data-free knowledge distillation for a classifier network**. Synthetic (transfer) data is generated to mimic the input distribution of data-set used to train the teacher network. The knowledge distillation loss $\mathcal{L}_{kd}$ is computed between output logits of the student and teacher network. The class prediction output for each network is generated using a softmax function $\sigma_i$.

## 4.3 Data-Free Knowledge Distillation

In the neural network verification problem we are not guaranteed to have access to the original training data. Therefore, we use a variant of knowledge distillation called data-free knowledge distillation. In data-free distillation, we do not consider the true output classes when training the student network, only the output of the teacher network. Data-free distillation is achieved by generating a transfer set of data (also known as pseudo data, or synthetic data) $X'$ from the teacher model, and use this data to train the student model $\hat{F}$. There are different approaches to generating this transfer set $X'$. These methods aim to reproduces the distribution of inputs in the original training data-set, so $X' \sim X$. Some cutting-edge techniques to generate the synthetic transfer data-set include generative adversarial networks (GANs). [25]. A diagram of data-free knowledge distillation for classifier networks is shown in Figure 4.2. The loss function we are using for data-free knowledge distillation therefore does not include the training data, only the output of the teacher network $z$:

$$\mathcal{L}_{\text{kd\_data\_free}} = \mathcal{L}_{\text{cce}}(\text{Softmax}(z), \text{Softmax}(z')).$$

## 4.4 Variants of Knowledge Distillation

There are many variants of knowledge distillation in the literature. Gou et al. in a recent survey paper on the field [23] categorize the types of knowledge distillation along four axes: Knowledge type, distillation scheme, student-teacher architecture, and distillation algorithm. The knowledge type axis indicates from which part of the teacher network knowledge is extracted. In response-based knowledge distillation, for example, the information from the teacher network is extracted from the output neurons (or logits) as described in [26]. In feature-based knowledge distillation, information may be extracted from the hidden layers of the teacher network. The distillation scheme axis describes in what order the teacher and student models are updated. For example, in the offline distillation scheme the student network is trained after the teacher network has already been fully trained, whereas in online distillation the student and teacher networks are trained simultaneously. The student-teacher architecture describes the model architecture of the teacher and student networks. For example, we may study the model-capacity gap between the student and teacher networks. Finally, the distillation algorithm describes any other differences in the knowledge transfer process. For example, in data-free knowledge distillation, the original training data is not accessible during the distillation.

Knowledge distillation has been applied to a large number of different domains, including "visual recognition, natural language processing, and recommendation systems." [23]. Knowledge distillation has been explored as a means to achieve privacy-preserving machine learning, such as in federated learning. To the best of our knowledge, there has been no application of data-free knowledge distillation to the scaling of verification methods for large neural networks, which we discuss in the next chapter.

# Chapter 5

## Knowledge Distillation before Verification

In this section, we combine the neural network verification technique of geometric path enumeration (Section 3.3) and the neural network compression technique of data-free knowledge knowledge distillation (Section 4.3), to compress a deep neural network before verification. We describe the technique and provide evidence that the method works using a toy problem (the XOR problem). Next, we report on our attempt to apply data-free knowledge distillation to a neural network model of the ACAS-Xu benchmark problem [30]. We then apply the same method to speed up the verification ten properties of the ACAS-Xu network which were part of VNN-Comp 2021 [4], as described in [32], over a variety of hyperparameter choices. This experiment is designed to answer the following research questions in the context of the ACAS-Xu network, which are repeated here from Chapter 1:

(RQ1) Can knowledge distillation help scale the neural network verification problem by providing an exponential speedup in verifying deep neural networks?

(RQ2) Can the smaller, verified networks reach a similar level of performance as the larger, original networks, allowing them to be used in applications?

(RQ3) Do compressed networks with high performance satisfy the same formal properties as the original network?

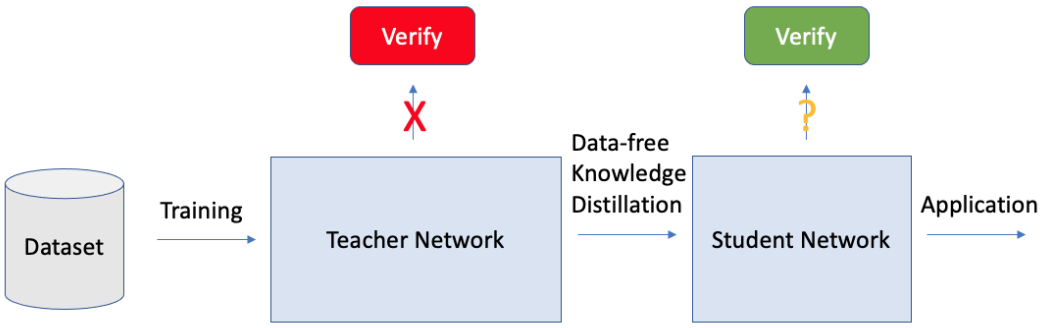



Figure 5.1: **Method of Knowledge Distillation before Verification**. The teacher network may be too big to verify, but the compressed representation (student network) is smaller and potentially easier to verify. This student network is then used in the application.

## 5.1 Description of the Method

Figure 5.1 shows a block diagram of the method. Training data is used to generate a teacher network, which in this case is too big to verify. Offline, data-free knowledge distillation is then used to compress the teacher network into a smaller student network. Next, the student network is verified using geometric path enumeration technique implemented in NNenum [3]. The compressed, verified student network can then be deployed in a safety critical application.

We use Tensorflow with the Keras API to perform the data-free knowledge distillation. All code used to perform the knowledge distillation and produce the experimental figures in this thesis are available on Github.[1] The ACAS-Xu implementation used here is distributed in the ONNX file format as part of VNN-Comp 2021, and so the ONNX libraries were used to load these networks. We use NNenum [3, 5], available through Github,[2] to verify the networks in the context of open-loop safety properties. We use NNenum because it had the best performance on the ACAS-Xu network in VNN-Comp 2021 [4]. Another reason we use NNenum in this work is due to the excellent packaging and software documentation around it, and because it is written in Python.

[1] Code to perform knowledge distillation and reproduce the experimental results:
`https://github.com/jordanperr/masters-thesis/tree/52d5b23`

[2] NNenum software used to perform the geometric path enumeration:
`https://github.com/stanleybak/NNenum/tree/fa1463b`

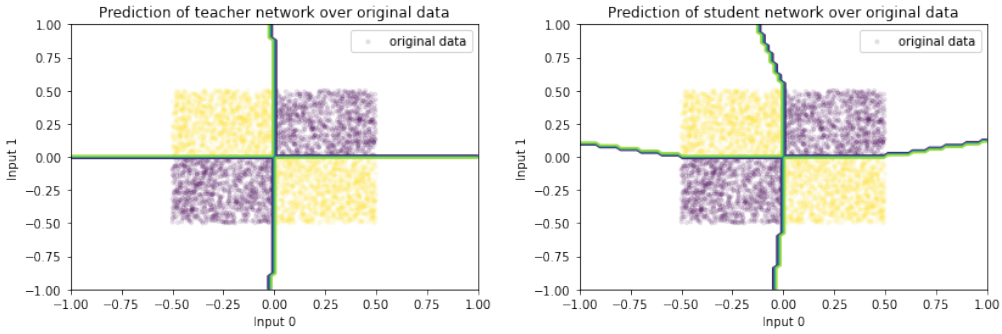



Figure 5.2: **Decision regions of the teacher and student network for the XOR problem**. Training data is in the domain (-0.5,0.5), and we plot the decision region to (-1.0, 1.0) to show the extrapolation region. We can see that both networks model the problem well in the interpolation region.

## 5.2 Proof of Concept with the XOR Problem

The XOR function underpins an infamous problem in the machine learning literature, after it was shown that the Perceptron was theoretically incapable of learning this simple logical function, as the problem is not linearly separable. It was this problem which initially motivated the use of deep neural networks, and so we find it fitting to use that problem here to motivate knowledge distillation before verification. It is possible to represent the XOR function exactly using three neurons in two hidden layers with manually designed weight matrices. Thus, learning XOR with an overparametrized network represents an extreme example of using a large dense network to represent a nonlinear function with a sparse and interpretable representation.

To model the XOR function, we first generate 4096 uniformly distributed samples in $x \in [0.5, -0.5]$, before applying the logical XOR on the sign of the input. We then train a teacher network with 300 units width and 6 layers of depth using the categorical cross entropy loss. This network is trained for 250 epochs using the ADAM optimizer with learning rate $\alpha = 0.01$ and parameters $\beta_1 = 0.9$, and $\beta_2 = 0.999$. After training, the teacher network achieves an accuracy of 0.996 on the original data-set.

Next, we train a student network with 2 hidden layers of 10 hidden units each on 4096 synthetic data points, uniformly distributed in $x \in [-0.5, 0.5]$, and $y = F_{n,\text{teacher}}(x)$, (the output

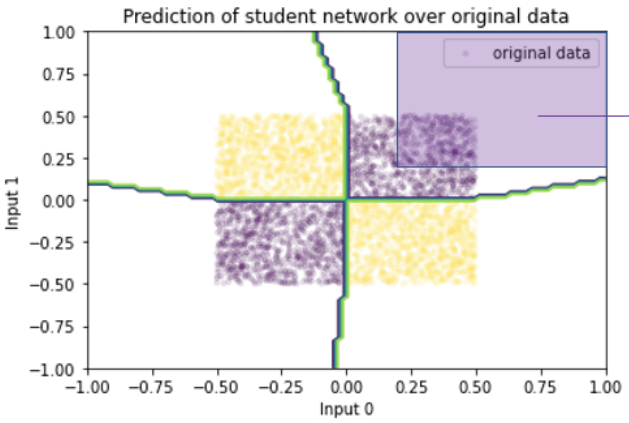
logits of the teacher network). This network is trained for 250 epochs using the same optimizer settings and achieves an accuracy of 0.972 on the classification of the synthetic transfer set.

The decision regions are shown in Figure 5.2, overlaid on top of the original 4096 data points upon which the teacher network was trained. It is interesting to compare the output regions of the student and teacher networks. The teacher network, which is highly overparametrized, seems to have a decision region which better extrapolates in the region which is 0.5 units outside of the original data.

We design a simple property to verify by inspecting these output regions, and choosing a rectangular region in quadrant 1 which both networks classify correctly. The region we choose is $\mathcal{X}_{\text{safe}} = ([0.2, 1], [0.2, 1])$, and is represented by a translucent purple rectangle in Figure 5.3. The code representing this region in the VNNLIB format is also provided in the same figure.

We run NNenum using this query against the two networks, teacher and student, and report the results in the table at the bottom of Figure 5.3. Both networks satisfy the property, and are marked as safe. However, we can see that the runtime of NNenum on the student network is 80% less than the runtime of NNenum on the teacher. The verification software NNenum was also able to use fewer star sets.

Here, we have used the XOR network as a proof of concept, to demonstrate that compression with knowledge distillation before verification with a tool like NNenum has the potential to speed up the verification. However, this example is quite extreme. It is unlikely that we would encounter a network so severely overparametrized as our XOR teacher network. In the next section, we apply the same technique to the ACAS-Xu benchmark data-set to see if this proof of concept can generalize to a benchmark data-set which is more representative of a real-world problem.




```
; Quadrant 1 Property

(declare-const X_0 Real)
(declare-const X_1 Real)

(declare-const Y_0 Real)
(declare-const Y_1 Real)

; If we are in quadrant 1
(assert (>= X_0 0.2))
(assert (<= X_0 1.0))
(assert (>= X_1 0.2))
(assert (<= X_1 1.0))

; Probability of output 1
should be higher than
output 0
(assert (>= Y_1 Y_0))
```

NNEnum Output

| Network | Q1 Query | Runtime | Number of Stars | Number of Loops |
|---|---|---|---|---|
| Teacher | SAFE | **0.5 Sec** | 71 | 5 |
| Student | SAFE | **0.1 Sec** | 3 | 10 |

Figure 5.3: **Results of the verification of the student XOR network**. Property $p$ enforces the value of logit $Y_1$ must be greater than the value of logit $Y_0$ for all inputs in a rectangular subset of quadrant 1. We run NNenum on both the student and teacher network for this property, reporting the results in the table. The student network verifies much faster than the teacher network.

## 5.3 The ACAS-Xu Benchmark Problem

ACAS-Xu is a control advisory system for unmanned aircraft, developed by Kochenderfer et al. in [34]. The model takes five inputs, which are computed from sensors on the airplane (sensor noise is filtered in a separate pre-processing step, not modelled here). The system consists of a quantized lookup table of solutions to the Bellman equation, which predicts the future positions of two aircraft (the ownship, and an intruder in their airspace) and recommends control actions to avoid a collision. The output is five floating point numbers, which can be interpreted as a score for the advisory of a given control action. The five actions are: Clear of conflict (CoC, Do Nothing), strong left (3 degrees/sec), weak left (1.5 degrees/sec), strong right (3 degrees/sec), weak right (1.5 degrees/sec). The input domain is discretized into 120,000,000 different possible states, distributed uniformly across the input domain of the neural network. These lookup tables are over 2GB in size uncompressed, which is too large to fit on aircraft control computers, and therefore a compression scheme was motivated. In [30], Julian et al. compresses the ACAS Xu lookup tables using a fully connected neural network.

There are multiple implementations of ACAS Xu available in the literature. The most recent implementation is called HorizontalCAS [28]. For this thesis, we use an implementation described in [32], which was also used in the VNN-Comp 2021 [4]. This implementation contains 45 separate networks, discretized along the $\tau$ and $a_{\text{prev}}$ dimensions with 9 and 5 levels, respectively. Each network contains 300 hidden units in total, organized into six fully connected layers of 50 hidden units. Each hidden unit uses the ReLU activation function. This implementation of ACAS-Xu, along with the ten properties used in VNN-Comp 2021, are available for download on Github.[3]

We make a few simplifying assumptions. First, we choose to only inspect one of the 45 neural networks (in particular, $\alpha = 1, \tau = 1$). Second, we interpret the output of the neural network as output logits in a categorical classifier. Julian et al. recommends taking the maximum score of the

[3] ACAS-Xu benchmark networks and properties used here:
`https://github.com/stanleybak/vnncomp2021/tree/90419aa/benchmarks/acasxu`

output and interpreting this as the output decision:

$$y(x) = \text{argmax}(F_n(x))$$

However, this simplifying assumption ignores the magnitude $|F_n(x)|$, which varies based on the confidence of the network in recommending an output action. Using this setup, we next use knowledge distillation to compress this model even further.

## 5.4 Knowledge Distillation of an ACAS-Xu Neural Network

We proceed to distill the ($\alpha = 1, \tau = 1$) ACAS-Xu network, trying out three different loss functions. The ACAS-Xu network is trained on a uniformly dense grid of data points, and therefore has an input distribution which can be modelled by a uniform random distribution. This is an atypical input distribution which is trivial to model. Therefore, we can assume that the input distribution of ACAS-Xu is uniform for each dimension and not concern ourselves with trying to generate that distribution. For more general problems this will not be the case, and the input distribution will have to be estimated using one of the many techniques in the data-free knowledge distillation literature. These inputs are scaled to $[-0.5, 0.5]$ to match the input distribution of the teacher network. We generate another 20% more synthetic data-points than required to serve as the validation set.

We use a rectangular, fully connected neural network with ReLU activation functions. Layer width is set to 50 hidden units (matching the ACAS-Xu implementation). The teacher network has six hidden layers, and the student network has four hidden layers. Weights and biases are initialized using the Glorot uniform distribution. The optimizer used is ADAM (as implemented in Keras) with a learning rate $\alpha = 0.001$, and parameters $\beta_1 = 0.9$, and $\beta_2 = 0.999$. We use an early stopping criteria on the validation loss with a patience of 20 epochs and a maximum of 500 epochs and a batch size of 128 data points. We tried using label weights to adjust for imbalance in the synthetic data-set. In fact, the class labels were imbalanced, with the largest category claiming around 60% of the data. However, using weights scaled inversely proportionally to the class counts

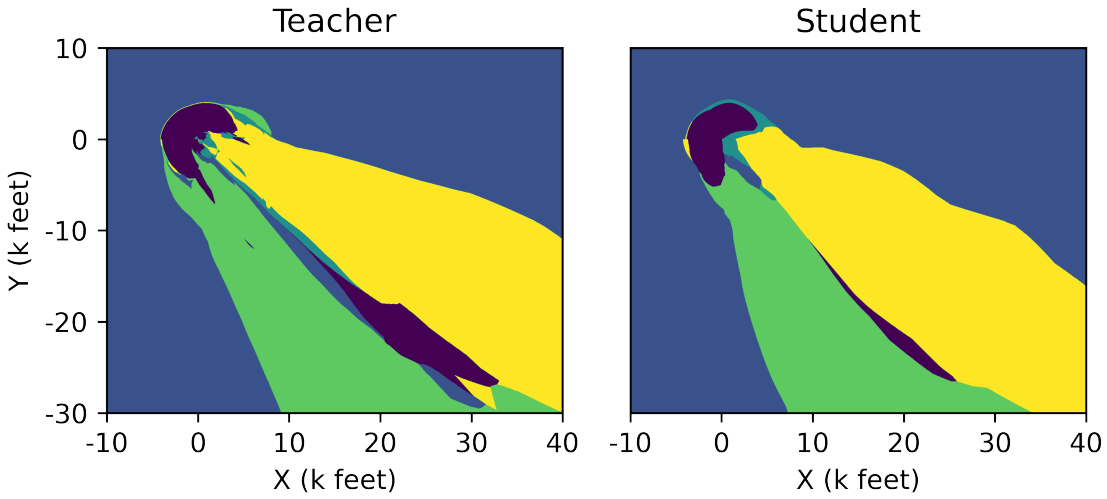



Figure 5.4: **Output classification regions of the ACAS-Xu student and teacher networks.** This covers one combination of hyperparameters and is plotted across the downrange and cross-range axes. We can see that the student network does a reasonable job reproducing the classification regions of the teacher, however there remain some noticeable differences between the two.

did not have any impact on the overall accuracy, and so we did not use label weights.

We tried three different loss functions to distill the ACAS-Xu network. Unlike in the XOR example, the data-free knowledge distillation using the categorical cross entropy loss did not work well. We tried tuning the temperature parameter with many values between 0.1 and 100. We also tried normalizing the output logits around zero before scaling them with the softmax. However, we were unable to achieve an accuracy over about 60% using Hinton distillation between the teacher and student network. Therefore, we also tried using the mean squared error loss between output logits from the student and teacher network. This loss worked better than the categorical cross-entropy, yielding an accuracy of around 80%.

Finally, we tried using the sparse categorical cross-entropy loss and outputs from the teacher network which have gone through the argmax function. This method yielded the best accuracy, at around 93%, which is comparable to the accuracy of the original networks with respect to the training data. The output regions of the teacher network and a student network trained in this way is shown in Figure 5.4. This was an interesting result, since it means that knowledge distillation was

| Experiment | Hidden layers | Synthetic data points | Repetition |
|---|---|---|---|
| Network Size | 2...8 | $2^{12}$ | 1...30 |
| data-set Size | 4 | $2^9...2^{15}$ | 1...30 |

Table 5.1: **Hyperparameters scanned across two computational experiments**. The first experiment is designed to test the effect of neural network size on the accuracy and verification time. The second experiment is designed to test the effect of transfer data-set size on the accuracy and verification time. Each experiment searches 7 different hyperparameter values and is repeated 30 times, for a total of 210 repetitions in each experiment.

largely unhelpful for the ACAS-Xu distillation. We believe this happened because the ACAS-Xu network violates some of the assumptions made in justifying the use of knowledge distillation.

One assumption which is violated is that the correlations between class labels should provide meaningful information. Here, the output categories are sequential (-3, -1.5, 0, 1.5, and 3 degrees), and there is no evidence to suggest that there should be complicated interactions between these categories.

## 5.5 Hyperparameter Study

In the previous section, we described the challenges in performing knowledge distillation on the ACAS-Xu network, and describe a set of hyperparameters which does a reasonable job at distilling the six layer network from [30] into a four layer network. In this section, we run this task repeatedly, searching across the number of synthetic data points $\{2^9...2^{15}\}$ and the number of layers in the student network $\{2...8\}$. Each of these configurations is run 30 times with different random seeds, as described in Table 5.1.

These experiments were run on the Summit supercomputer at the University of Colorado, Boulder Research Computing. Each experiment was run on a single node in the Haswell partition of the cluster, finishing the entire workflow in under two hours. The nodes had two Intel(R) Xeon(R) E5-2680 v3 CPUs running at clock speed of 2.50GHz. The experimental workflow was laid out as a flat pipeline. The knowledge distillation step and the network verification step were each paralellized into four ranks using Python's multiprocessing library.

From each experiment we record the following metrics from the distillation phase: Accuracy

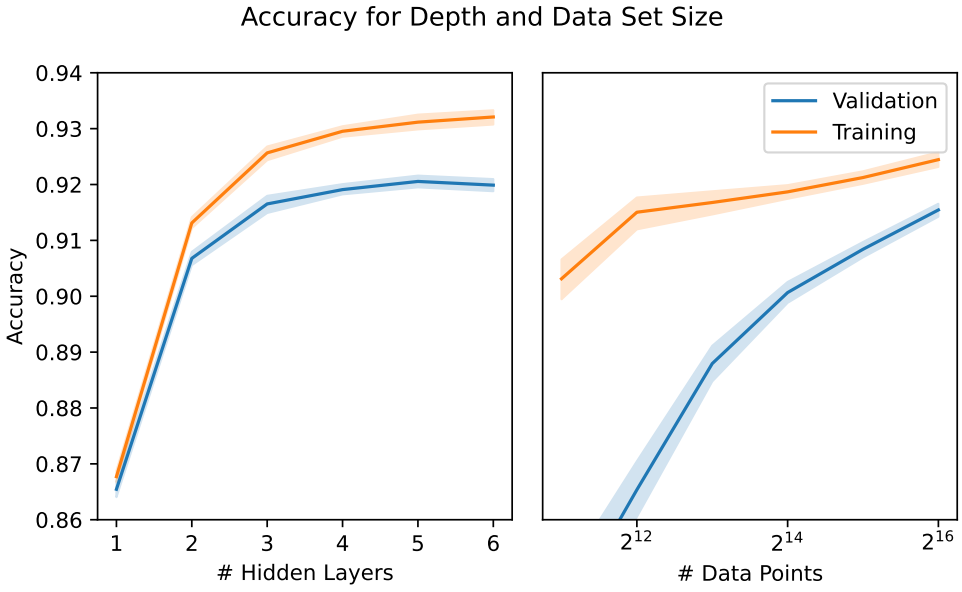



Figure 5.5: **Accuracy vs number of hidden layers and synthetic data-set size**. The accuracy increases with both independent variables.

and loss for both validation and training data-sets at each training epoch, wall-time to train the student network, and the final weights and biases in the form of an ONNX file. We also record the following metrics from the verification phase: Wall time to verify, reported time to verify from NNenum, the result of verifying each of the ten properties, and a file for each property that contains the standard output of NNenum.

In Figure 5.5, we can observe how the accuracy of the student network changes as we both add more hidden layers to the student network and add more datapoints to the synthetic data-set. As expected, the accuracy of the student network increases with both. We find that accuracy increases with the number of hidden layers and the number of data points, which is to be expected. Furthermore, the accuracy tends to increase logarithmically with these variables. This is in line with literature identifying a scaling law between the size of a neural network and its accuracy [31].

In Figure 5.6, we see the effect of the same hyperparameters (number of hidden layers, and number of data points) on the time to verify each network, as reported by NNenum. Here, we see that the time to verify increases exponentially with the number of hidden layers. This was

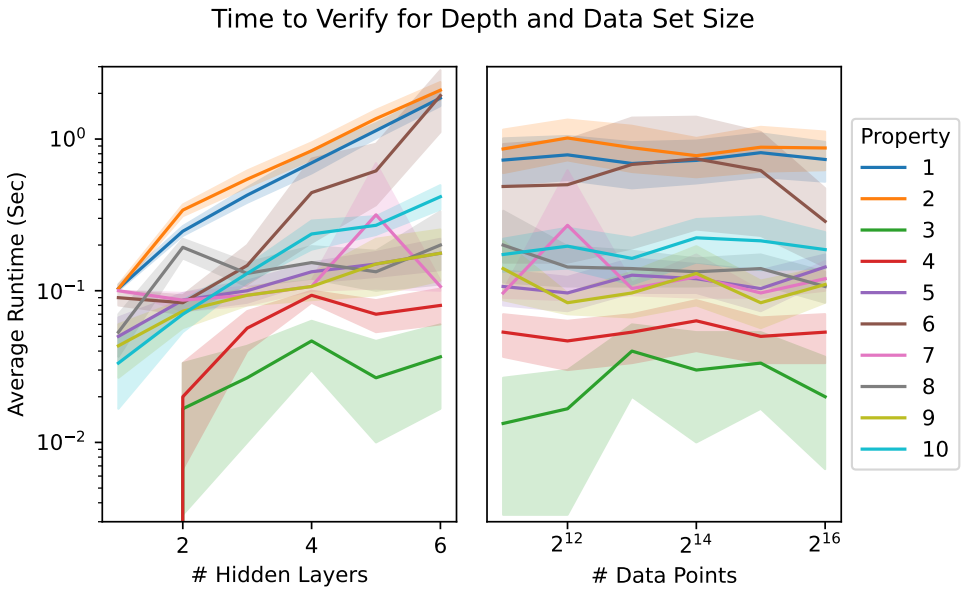



Figure 5.6: **Time to verify vs number of hidden layers and synthetic data-set size**. We observe that the runtime increases exponentially with the number of hidden layers, and is constant with respect to the number of data points.

expected, since the verification algorithm NNenum is exponential in the worst case. This confirms the first research question, which states that the time to verify can be reduced exponentially by reducing the size of the network using knowledge distillation.

Figure 5.6 also shows that the time to verify is not impacted by the number of data points in the synthetic data-set. This makes intuitive sense, since the number of data points used in the training data-set is independent of the size of the network. It could have been the case that more training data points, and higher accuracy, would have resulted in more ReLUs in the posneg case, and therefore more splits during verification. However, this did not turn out to be the case. Our data shows that the number of synthetic data points has no impact on the time to verify.

Figure 5.7 shows the time to verify student networks relative to the time to verify the teacher network. Using this plot, we can confirm that most properties verify more quickly in the student network than in the teacher network. Every property except properties seven and eight verify more quickly in the student network than the teacher network, provided the number of hidden layers in the student is less than four layers. However, for student networks with more than four layers,

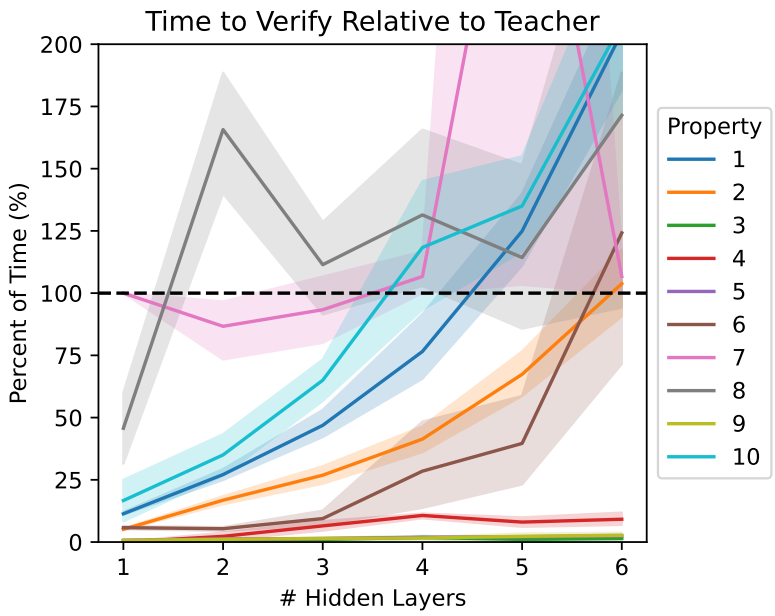



Figure 5.7: **Relative time to verify (in percent) vs the number of hidden layers in the student network**. We observe that most properties verify more quickly in the smaller student network (less than 100%). However, there are a few exceptions which we believe may be caused by a small base rate in the percentage.

properties ten, five, four, and two begin to verify slower than the teacher. Additionally, properties seven and eight do not in general verify much faster than the teacher even below four hidden layers. We believe the reason for this unexpected result is that the base rate in these proportions (i.e, the time it took the teacher networks to verify) was quite small for these properties. This would make the proportion a numerically unstable quantity in these cases.

Finally, in Figure 5.8, we plot the miss rate of properties with respect to the number of hidden layers and the number of data points, for those properties which were valid in the teacher network. A miss rate of 0 means that none of the thirty student networks violated the property, whereas a miss rate of 1 means that they all violated the property. We can see that properties one, two, and three are relatively well preserved in the student networks, with miss rates of under 0.2 for all hyperparameters. The miss rate of these properties does not change as we sweep the hyperparameters, with the notable exception of the case with one hidden layer. All of these properties

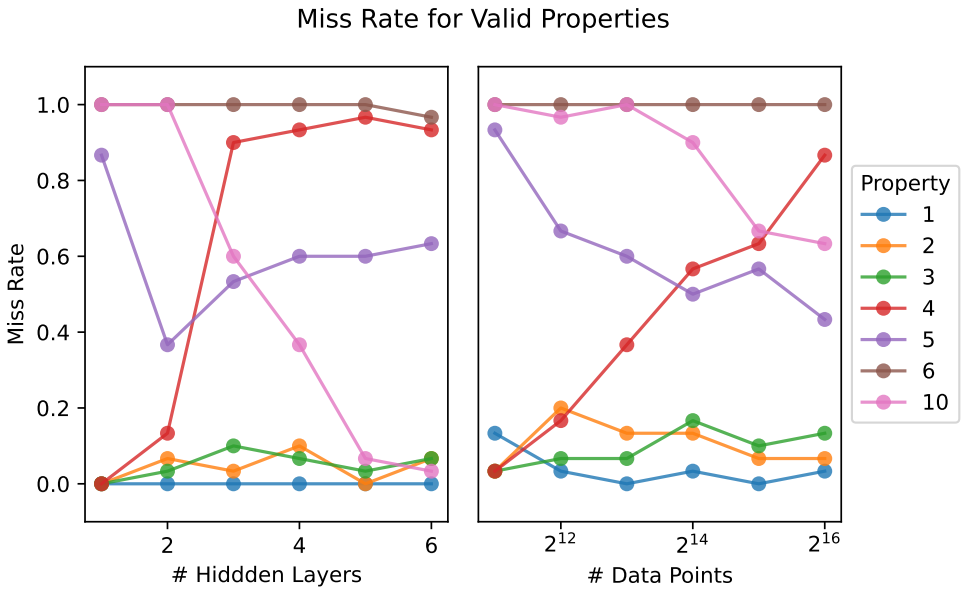



Figure 5.8: **Miss rate for properties in student networks compared to the teacher network**. A miss rate of 1.0 indicates that every student network (out of the 30 networks tested in a given experiment) failed to verify for this property. A miss rate of 0.0 indicates that every student network did verify for this property. We can see that the results are mixed for different properties.

(plus property four) has miss rates of zero in this case. Properties five and ten have a miss rate of 1 for small networks and small number of data points, and these trend lower both hyperparameters increase. Property four has a higher miss rate as both hyperparameters increase. Finally, property six maintains a miss rate of 1 in nearly all hyperparameter configurations searched.

We believe that Figure 5.8 demonstrates that there is a complex relationship between accuracy in the student networks and the probability of a student network satisfying a formal verification property, helping to answer our third research question. We expected that with increased categorical accuracy on the synthetic data-set, the properties should have a higher chance of being satisfied, but this is not what was observed in the data. In the next and final chapter, we summarize the results from this chapter in the context of our research questions.

# Chapter 6

# Conclusion

A fundamental challenge in formal verification of deep neural networks is that of scaling the algorithms to work for large neural networks. Such neural networks are common in practical applications, since large networks tend to have better training and generalization performance. We explored a method of first compressing the large neural network with knowledge distillation. Using this method, it may be possible to produce smaller, verifiable networks that perform satisfactorily for an application, obviating the need to verify the large neural network directly. Throughout this thesis, we have described the scaling problem in formal verification and illustrated the key issue through one verification technique (geometric path enumeration) which causes the runtime algorithm to scale exponentially in the size of the network. We described a neural network model compression technique (data-free knowledge distillation) to produce compressed networks with high accuracy that can be verified more quickly. We then showed a proof of concept experiment using the XOR problem, before showing a more detailed computational experiment and hyperparameter study on the benchmark neural network ACAS-Xu. Our results on the XOR problem and ACAS-Xu show promise, but we also have some interesting negative results in the case of ACAS-Xu.

## 6.1 Summary of the Results

We set out to explore the potential in using knowledge distillation to compress neural networks before formal verification. In particular, we set out to answer three research questions:

**(RQ1) Can knowledge distillation provide an exponential speedup in verifying a deep neural network?** Overall, our answer to the question is yes. We saw a large speedup in the verification time of the XOR model, and we have demonstrated empirically that the verification time of an ACAS-Xu model is reduced exponentially as the size of the compressed model is decreased. This exponential scaling can be seen in Figure 5.6. This result was expected theoretically [32] and was shown here empirically.

**(RQ2) Can the compressed networks reach a similar level of performance as the original networks, allowing them to be used in applications?** We believe that we have shown promising results in this direction. We found this to be true for the XOR model, with data-free knowledge distillation able to produce a compressed representation with an accuracy of 0.972 compared with the original model. The decision regions of both the teacher and student models of the XOR problem can be seen in Figure 5.2. However, we were unable to produce similar levels of accuracy students of models of the ACAS-Xu problem using the same data-free knowledge distillation technique. We were able to overcome this issue by modifying the data-free knowledge distillation technique (applying the argmax function to the output of the teacher network), as described in Section 5.4. The students produced by this modified knowledge distillation method achieved a 93% classification accuracy when compared to the outputs of the original ACAS-Xu model. A comparison of output regions from both the original network and the compressed representation are shown in Figure 5.4, and appear to be close based on visual inspection. One conjecture as to why the data-free knowledge distillation method was unable to compress ACAS-Xu is that ACAS-Xu is not a true classification problem, but a discretized regression problem.

**(RQ3) Do compressed networks with high performance satisfy the same formal properties as the original network?** Our results are mixed here, and do not provide a good answer to this question. Our compressed representation of the XOR model was able to satisfy the single safety property which was tested. However, student models of the ACAS-Xu network were unable to satisfy many of the desired safety properties. This is illustrated in Figure 5.8, which shows some

safety properties being missed by nearly all of the student networks. The fact that student networks of ACAS-Xu missed several safety properties reduces the usefulness of the method as implemented today, since those student networks which do not satisfy the relevant safety properties are not useful in safety critical applications. We do not believe this limitation is caused by any fundamental issue with knowledge distillation. On the contrary, it motivates a clear pathway towards future work to improve the method.

## 6.2 Future Directions

We have presented evidence that knowledge distillation before formal verification can be used to speed up the verification time. However, the compressed representations of ACAS-Xu did not, in general, satisfy the safety properties defined for the original network. One exciting next step is to explore different ways to ensure that the compressed networks satisfy the same properties as the original ones. Such a study could correct the poor verification performance seen in the compressed representation of ACAS-Xu, producing compressed networks that would be usable in practice. Another exciting direction could be to expand upon our encouraging results by including more and varied knowledge distillation and verification techniques, as well as a wider variety of benchmark problems and property types.

**Compressed model refinement.** It may be interesting to incorporate the knowledge distillation and verification inside of a larger refinement loop (perhaps using counter-examples generated from the verification step to refine the distillation). Such an outer loop could lead to a sound method for verification of the teacher. One can use counterexamples from the formal verification method to guide the knowledge distillation step in refining those regions with violated constraints.

**Compression and verification algorithms.** Future work could include the use of different compression techniques than knowledge distillation, different variants of knowledge distillation, and different neural network verification approaches. Knowledge distillation has many variants, and it may be possible to improve the accuracy of the compressed networks by using different

knowledge types, and more sophisticated distillation techniques. It would also be interesting to explore online distillation techniques, or other training + verification techniques with access to the original training data.

**Benchmark problems and properties.** While ACAS-Xu is a state-of-the-art benchmark in the verification literature, it is still relatively small compared to other benchmarks in the larger machine learning literature, such as CIFAR. Applying this technique to other properties such as robustness or fairness constraints may present new challenges and opportunities. The inclusion of such constraints in the loss function of the knowledge distillation step may improve the performance of the compressed networks.

In this thesis, we presented initial evidence to support the use of knowledge distillation to compress deep neural networks before before formal verification. The motivation for this method comes from the idea that many neural networks are overparametrized to improve training dynamics, but these extra parameters may not be necessary for inference. Large neural networks have drawbacks, including higher power consumption and memory requirements, less interpretability, and that they are more difficult to verify. We hope this work provides motivation to seek smaller, more interpretable, and verifiable models instead of relying on overparametrized models in applications.

## Bibliography


[1] Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. November 2018. arXiv:2109.00498 [cs].

[2] Aws Albarghouthi. Introduction to Neural Network Verification. 2019. `http://verifieddeeplearning.com`.

[3] Stanley Bak. nnenum: Verification of relu neural networks with optimized abstraction refinement. In NASA Formal Methods Symposium, pages 19–36. Springer, 2021.

[4] Stanley Bak, Changliu Liu, and Taylor Johnson. The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results, August 2021. arXiv:2109.00498 [cs].

[5] Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T. Johnson. Improved Geometric Path Enumeration for Verifying ReLU Neural Networks. In Shuvendu K. Lahiri and Chao Wang, editors, Computer Aided Verification, volume 12224, pages 66–96. Springer International Publishing, Cham, 2020. Series Title: Lecture Notes in Computer Science.

[6] Nathan Baker, Frank Alexander, Timo Bremer, Aric Hagberg, Yannis Kevrekidis, Habib Najm, Manish Parashar, Abani Patra, James Sethian, Stefan Wild, Karen Willcox, and Steven Lee. Workshop Report on Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence. Technical Report 1478744, February 2019.

[7] Baluta. Scalable Quantitative Verification For Deep Neural Networks. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, ES, 2021. IEEE.

[8] Clark Barrett, Guy Katz, Dario Guidotti, Luca Pulina, Nina Narodytska, and Armando Tacchella. The VNNLIB standard for benchmarks. page 10. http://www.vnnlib.org/.

[9] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. Proceedings of the National Academy of Sciences, 116(32):15849–15854, August 2019.

[10] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the State of Neural Network Pruning? 2020. Publisher: arXiv Version Number: 1.

[11] Kenneth Borup. Introduction to Knowledge Distillation, September 2020. Part of the Keras Documentation, https://keras.io/examples/vision/knowledge_distillation/.

[12] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data: Sparse identification of nonlinear dynamical systems. 113(15):3932–3937, 2015.

[13] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '06, page 535, Philadelphia, PA, USA, 2006. ACM Press.

[14] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges. IEEE Signal Processing Magazine, 35(1):126–136, January 2018.

[15] G. Cybenko. Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals, and Systems, 2(4):303–314, 1989.

[16] Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization, June 2014. arXiv:1406.2572 [cs, math, stat].

[17] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output Range Analysis for Deep Feedforward Neural Networks. In Aaron Dutle, César Muñoz, and Anthony Narkawicz, editors, NASA Formal Methods, volume 10811, pages 121–138. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science.

[18] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. An Abstraction-Based Framework for Neural Network Verification. In Shuvendu K. Lahiri and Chao Wang, editors, Computer Aided Verification, volume 12224, pages 43–65. Springer International Publishing, Cham, 2020. Series Title: Lecture Notes in Computer Science.

[19] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. Visualizing Higher-Layer Features of a Deep Network. 2009.

[20] Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, March 2019. Number: arXiv:1803.03635 arXiv:1803.03635 [cs].

[21] Sumathi Gokulanathan, Alexander Feldsher, Adi Malca, Clark Barrett, and Guy Katz. Simplifying Neural Networks Using Formal Verification. In Ritchie Lee, Susmit Jha, Anastasia Mavridou, and Dimitra Giannakopoulou, editors, NASA Formal Methods, volume 12229, pages 85–93. Springer International Publishing, Cham, 2020. Series Title: Lecture Notes in Computer Science.

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

[23] Jianping Gou, Baosheng Yu, Stephen John Maybank, and Dacheng Tao. Knowledge Distillation: A Survey. International Journal of Computer Vision, 129(6):1789–1819, June 2021. arXiv:2006.05525 [cs, stat].

[24] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding, February 2016. arXiv:1510.00149 [cs].

[25] Matan Haroush, Itay Hubara, Elad Hoffer, and Daniel Soudry. The Knowledge Within: Methods for Data-Free Model Compression, April 2020. arXiv:1912.01274 [cs, stat].

[26] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. arXiv:1503.02531 [cs, stat], March 2015. arXiv: 1503.02531.

[27] Frank Hutter, Jörg Lücke, and Lars Schmidt-Thieme. Beyond Manual Tuning of Hyperparameters. KI - Künstliche Intelligenz, 29(4):329–337, November 2015.

[28] Kyle D. Julian and Mykel J. Kochenderfer. Guaranteeing Safety for Neural Network-Based Aircraft Collision Avoidance Systems. In 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), pages 1–10, September 2019. arXiv:1912.07084 [cs, eess].

[29] Kyle D. Julian and Mykel J. Kochenderfer. Reachability Analysis for Neural Network Aircraft Collision Avoidance Systems. Journal of Guidance, Control, and Dynamics, 44(6):1132–1142, June 2021.

[30] Kyle D. Julian, Jessica Lopez, Jeffrey S. Brush, Michael P. Owen, and Mykel J. Kochenderfer. Policy compression for aircraft collision avoidance systems. In 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC), pages 1–10, Sacramento, CA, USA, September 2016. IEEE.

[31] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models, January 2020. arXiv:2001.08361 [cs, stat].

[32] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks, May 2017. arXiv:1702.01135 [cs].

[33] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. arXiv:1412.6980 [cs].

[34] Mykel J Kochenderfer. Optimized Airborne Collision Avoidance. In Decision making under uncertainty: theory and application, page 28. MIT Press, 2015.

[35] Lindsey Kuper, Guy Katz, Justin Gottschlich, Kyle Julian, Clark Barrett, and Mykel Kochenderfer. Toward Scalable Verification for Safety-Critical Deep Networks, February 2018. arXiv:1801.05950 [cs].

[36] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. Nature, 521(7553):436–444, 2015.

[37] Yann LeCun, John S Denker, and Sara A Solla. Optimal Brain Damage. In Advances in Neural Information Processing Systems, volume 2, page 8, 1989.

[38] Ba Lei Jimmy and Rich Caruana. Do Deep Nets Really Need to be Deep? In Advances in Neural Information Processing Systems 27 (NIPS 2014), 2014.

[39] Changliu Liu, Tomer Arnon, and Christopher Lazarus. Algorithms for Verifying Deep Neural Networks. page 126, 2019. arXiv:1903.06758 [cs.LG].

[40] Christoph Molnar. Interpretable Machine Learning. 2021. https://christophm.github.io/interpretable-ml-book/.

[41] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. Inceptionism: Going Deeper into Neural Networks, June 2015.

[42] Michael C. Mozer and Paul Smolensky. Using Relevance to Reduce Network Size Automatically. Connection Science, 1(1):3–16, January 1989.

[43] Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. Zoom In: An Introduction to Circuits. Distill, 5(3):10.23915/distill.00024.001, March 2020.

[44] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. SVCCA: Singular Vector Canonical Correlation Analysis for Deep Learning Dynamics and Interpretability. In Proceedings of the 31st International Conference on Neural Information Processing Systems, pages 6078–6087, Long Beach, California, USA, 2017. arXiv: 1706.05806.

[45] William T. Redman, Maria Fonoberova, Ryan Mohr, Ioannis G. Kevrekidis, and Igor Mezic. An Operator Theoretic View on Pruning Deep Neural Networks. arXiv:2110.14856 [cs, math], March 2022. arXiv: 2110.14856.

[46] David E Rumelhart, Geoffrey E Hintont, and Ronald J Williams. Learning representations by back-propagating errors. Nature, 323:533–536, 1986.

[47] Lesia Semenova, Cynthia Rudin, and Ronald Parr. On the Existence of Simpler Machine Learning Models. In 2022 ACM Conference on Fairness, Accountability, and Transparency, pages 1827–1858, June 2022. arXiv:1908.01755 [cs, stat].

[48] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, April 2014. arXiv:1312.6034 [cs].

[49] Dutta Souradeep. Verification of Neural Networks. PhD, University of Colorado, Boulder, Boulder, CO, 2020.

[50] Samuel Stanton, Pavel Izmailov, Polina Kirichenko, Alexander A. Alemi, and Andrew Gordon Wilson. Does Knowledge Distillation Really Work?, December 2021. arXiv:2106.05945 [cs, stat].

[51] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, February 2014. arXiv:1312.6199 [cs].

[52] Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. Star-Based Reachability Analysis of Deep Neural Networks. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, Formal Methods – The Next 30 Years, volume 11800, pages 670–686. Springer International Publishing, Cham, 2019. Series Title: Lecture Notes in Computer Science.

[53] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient Formal Safety Analysis of Neural Networks, November 2018. arXiv:1809.08098 [cs, stat].

[54] Tsui-Wei Weng, Pin-Yu Chen, Lam M. Nguyen, Mark S. Squillante, Ivan Oseledets, and Luca Daniel. PROVEN: Certifying Robustness of Neural Networks with a Probabilistic Approach, January 2019. arXiv:1812.08329 [cs, stat].

[55] Kai Y. Xiao, Vincent Tjeng, Nur Muhammad Shafiullah, and Aleksander Madry. Training for Faster Adversarial Robustness Verification via Inducing ReLU Stability, April 2019. arXiv:1809.03008 [cs, stat].