Heriot-Watt University

Masters Thesis

Formal Verification of Neural Networks in Go

Author:
Arran Dinsmore

Supervisor:
Ekaterina Komendanskaya

A thesis submitted in fulfilment of the requirements for the degree of MSc. Robotics

in the

School of Electrical, Electronic & Computer Engineering

&

School of Engineering & Physical Sciences

April 2021

Declaration of Authorship

I, Arran Dinsmore, declare that this thesis titled, 'Formal Verification of Neural Networks in Go' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Arran Dinsmore

Date: April 2021

"Program testing can be used to show the ***presence*** *of bugs, but never to show their* ***absence****!"*

Edsger W. Dijkstra

Abstract

As machine learning for safety critical applications such as autonomous vehicles are starting to be developed beyond proof of concepts, and enter into production within society, there is a need to ensure these systems do not fail.

Traditional rigorous testing methods are not a viable approach for such black box systems, and thus a need for formal verification methods that can prove the robustness of a system are required.

Additionally, the choice of programming language used for these tasks has grown with new machine learning extensions being developed on existing languages.

This project will investigate how robust programming infrastructures can be used to enhance formal verification approaches for machine learning tasks, with the main objective of developing a formal methods framework for verifying neural networks in the Go programming language.

Contents

Declaration of Authorship **i**

Abstract **iii**

Contents **iv**

List of Figures **vi**

List of Tables **vii**

List of Abbreviations **viii**

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 3
- 1.3 Aims & Objectives . . . 4

2 Background & Literature Review **6**
- 2.1 Neural Networks & Deep Learning . . . 6
  - 2.1.1 Overview . . . 6
  - 2.1.2 Gradient Descent & Backpropagation . . . 10
  - 2.1.3 Vulnerabilities to Adversarial Attacks . . . 11
  - 2.1.4 Discrimination & Neural Networks . . . 12
- 2.2 Formal Verification . . . 12
  - 2.2.1 Background . . . 13
  - 2.2.2 Current Frameworks . . . 13
- 2.3 Formal Verification of AI . . . 13
  - 2.3.1 Overview . . . 13
  - 2.3.2 Sapphire . . . 13
- 2.4 Programming Paradigms for Machine Learning . . . 13
  - 2.4.1 Computational Graphs . . . 13
  - 2.4.2 Auto Differentiability . . . 13
- 2.5 The Go Programming Language . . . 13
  - 2.5.1 Brief History . . . 13
  - 2.5.2 Go for ML . . . 13
  - 2.5.3 Go for Formal Verification . . . 13
- 2.6 Conclusions . . . 13

3 Methodology **14**

4 Implementation **15**

5 Analysis **16**

6 Conclusions **17**

A Appendix Title Here **18**

Bibliography **19**

List of Figures

1.1 Google's Aversarial Patch 2

2.1 Example of an artifcial neuron 7
2.2 Examples of Activation Functions 8
2.3 Example of a Neural Network 8
2.4 Complex decision boundary from deep neural network 9
2.5 Example visualisation of gradient descent 10
2.6 Demonstration of the effects of adversarial examples 12

List of Tables

List of Abbreviations

AI	Artificial Intelligence. 3, 4
AIV	Artificial Intelligence Verification. 3, 4, 6, 13
BGD	Batch Gradient Descent. 10, 11
CNN	Convolutional Neural Network. 9
CV	Computer Vision. 7–9
DNN	Deep Neural Network. 9, 11
FP	Functional Programming. 4
GD	Gradient Descent. 10, 11
MAS	Multi-Agent System. 3
MBGD	Mini-Batch Gradient Descent. 10, 11
ML	Machine Learning. 1–4, 9
MLP	Multi-Layer Perceptron. 9
NN	Neural Network. 1–13
ReLU	Rectified Linear Unit. 7, 8
RNN	Recurrent Neural Network. 9
SGD	Stochastic Gradient Descent. 10, 11
SMT	Satisfiability Modulo Theories. 4

Chapter 1

Introduction

1.1 Context

Machine Learning (ML) algorithms are becoming increasingly present in systems that operate within shared environments with humans, or involve direct interaction with humans themselves [Pereira and Thomas, 2020]. These systems are often defined as safety-critical, such that their failures lead to unintended and potentially harmful behaviours [Amodei et al., 2016]. Examples of these systems include autonomous automotive systems, traffic control systems, medical devices, aviation software, industrial robotics, and many more cyber-physical systems that interact with our environment. Many of these systems have so far only existed as proof of concepts, but are steadily approaching commercial use within our society.

Additionally, recent research has exposed broad vulnerabilities to adversarial attacks within data driven ML algorithms, including Neural Networks (NNs); where applying small but intentional perturbations to an input which are not noticible to humans, can lead to a model outputting an incorrect classification with high confidence [Goodfellow et al., 2014]. An example of such an attack can be seen in *Fig. 1.1*. Consequently, the testing and verification of ML for the use of controlling safety-critical systems has become a focused area of research in recent years.

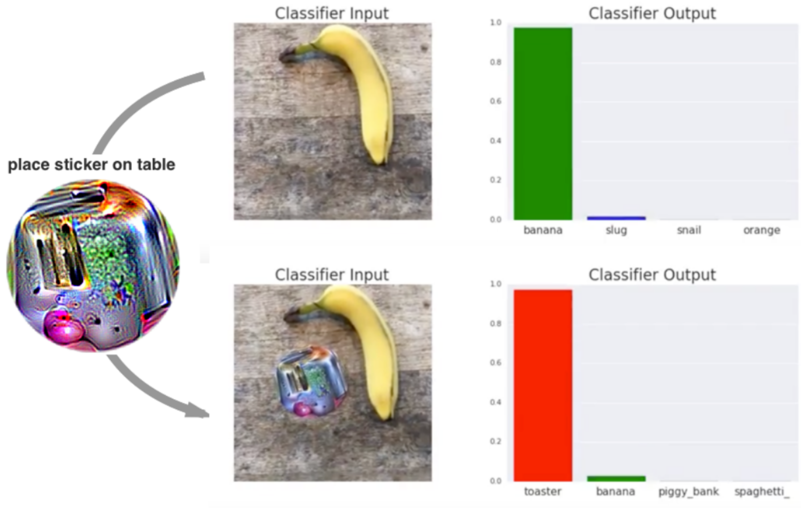

FIGURE 1.1: **Google's Adversarial Patch** – An example of a method to create targeted adversarial attacks on NNs by adding carefully designed noise via a physical patch [Brown et al., 2018].

This thesis will use the following definitions for software testing and verification. Software testing, or validation, is defined as the evaluation of a system under various conditions and observing its behaviour while looking for defects [Pereira and Thomas, 2020]. In the context of ML development, testing is used to ensure that a trained model generalises accurately to some previously unseen test data.

Verification is defined as the process of determining whether the products of a phase of the software development process fulfill the requirements established during the previous phase [Ammann and Offutt, 2008]. Formal verification in other words, formulates logical arguments that a system will not act abnormally under a wide range of circumstances, and can be used to determine not only generality, but also the robustness and correctness of a system.

The challenges regarding verification of ML models stem from the typically less deterministic and more statistically-oriented nature of their algorithms, which lead to a lower degree of understanding than software that is explicitly programmed to perform a specific task [Bishop, 2006]. These types of systems are commonly referred to as *black box* systems, where the internal mechanisms are not revealed; in other words, it is impossible to understand a model just by looking at its parameters [Molnar, 2019].

1.2 Motivation

Public calls for *sensible* or *verifiable Artificial Intelligence (AI)* have been raised in recent years due to ever increasing development of complex and pervasive systems that are entering into our everyday lives [Russell et al., 2016].

Formal verification of deterministic software systems has seen significant progress since the early verification systems. These early systems [Boyer and Moore, 1990, Guaspari et al., 1993, Polak, 1979] often struggled to be widely adopted into industry applications. However, due to the ever increasing complexity of deployed software, new verification tools have been developed with the intent of being accessible to a wide range of industry software engineers [Fisher et al., 2017].

On the other hand, verification of non-deterministic systems has seen relatively little progress, with the exception of Multi-Agent Systems (MASs) [Kouvaros and Lomuscio, 2016, Lomuscio et al., 2017]. Indeed, due to the nature of Artificial Intelligence Verification (AIV) research, there are limited resources with regard to the programming tools available for researchers in this area. This is especially true for work within ML, as the programming languages and tools commonly used for *traditional* verifcation are often disparate from those widely adopted by the ML communities.

Popular programming languages used for ML such as Python or Matlab currently have comparitively less formal verification tools available than those concerned with system infrastructure or embedded applications. Additionally, AIV toolkits for ML tasks in these languages are still in early stages of development, and mainly focused on the verification of NNs [Kokke, 2020].

Furthermore, the landscape of ML programming itself is forever shifting, and while there is yet a programming language dedicated for ML tasks, huge efforts from programming language designers have been made in developing ML libraries for existing languages. This is necessary in order to handle the extremely high computational demands, and to simplify model languages to make them easier to add domain-specific optimisations and features [Innes et al., 2017].

A prime example of such development can be seen in the Go programming language, or *GoLang*. A relatively new language, originally developed by Google in 2009 with the intention of creating a modern general-purpose language similar to C. GoLang has seen a surge in popularity within the ML community since the release of its first extensive ML package, *Gorgonia*, in 2016, which heavily relies on the use of expression graphs [Chew, 2016]. This package allows GoLang developers to take advantage of automatic and symbolic differentiation, gradient descent optimisations, numerical stabilisation, added support for CUDA/GPGPU computation, and comparitively quick speeds than its Python counterparts (Theano and TensorFlow) [GoLang, 2020].

Another example of a programming paradigm shift towards dedicated ML languages, is Microsoft's efforts in developing an efficient differentiable version of the Functional Programming (FP) language F [Shaikhha et al., 2019].

Consequently, as programming languages continue to develop ML capabilities, there is a need for exploring new and scalable approaches for developing AIV tools in these languages. This is especially important for programming languages which are being adopted by industry to implement ML models for the use within safety-critical or pervasive systems.

1.3 Aims & Objectives

The aim of this project is to investigate the current programming paradigms within ML development, and to explore the suitability of current formal verification toolkits available to them. Subsequently, this thesis will aim to design and implement a GoLang formal methods framework for Gorgonia NNs, providing GoLang ML developers with a set of tools which will allow them to produce safe and fair AI applications.

This framework will extend upon the work made by [Kokke, 2020], and the Sapphire library implemented in Python which successfully translates TensorFlow feed-forward NN models to the Z3 Satisfiability Modulo Theories (SMT) solver created by Microsoft Research [De Moura and Bjørner, 2008].

To achieve this project's aims, the following objectives should be met:

- *Objective 1* - Conduct a feasibility study with regards to developing a formal methods framework for NNs in Go.
- *Objective 2* - Implement bindings that map the parameters of a Gorgonia NN model to Z3 variables.
- *Objective 3* - Select data in order to train and verify NN models using this project's formal methods framework.

- *Objective 4* - Implement a series of NN models in Gorgonia using the data sets mentioned in *Objective 3*.
- *Objective 5* - Verify the correctness of Gorgonia NNs using the bindings mentioned in *Objective 2*.
- *Objective 6* - Make conclusions about the developed framework's benefits and limitations, and discuss future improvements to the methodology as described in *Objective 1*.

Chapter 2

Background & Literature Review

This chapter will provide a background understanding to the important concepts that are required by this thesis, and explore the current trends within AIV research. This includes an introduction to formal verification, both within deterministic and non-deterministic systems; an overview of the current state of NN and deep learning research, and the programming paradigms used for their development; and finally an investigation into the Go programming language infrastructure and the feasibility of using it for verifying NNs.

2.1 Neural Networks & Deep Learning

This section aims to clarify the concepts of NNs and deep learning, as well as to show the successes and failures of the field, and to demonstrate the need for formal approaches for developing such algorithms.

2.1.1 Overview

NNs are learning algorithms based on a loose analogy of how the human brain functions. They consist of nodes, or neurons (*see Fig. 2.1*), which act as functions that output a nonlinear combination of weighted inputs and a bias [Dreyfus, 2005]. Learning is achieved by adjusting the weights on the connections between nodes, which are analogous to synapses and neurons in nature [Sammut and Webb, 2010].

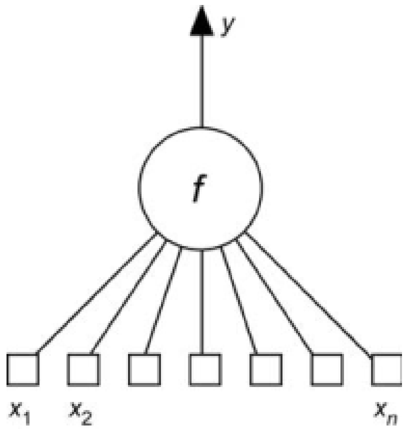

FIGURE 2.1: **Artificial Neuron** – a nonlinear bounded function $y = f(x_1, x_2, \ldots, x_n; w_2, \ldots, w_n)$ where the x_i are the input values and the w_i are the weights of the neuron [Dreyfus, 2005].

A weight is assigned to each of a neuron's inputs. They are the co-efficients of a neuron's equation and therefore reflect the importance of individual inputs. A bias is a constant value assigned to each neuron. They are used to shift a neuron's activation function output in a positive or negative direction [Malik, 2019b].

A NN is made up of a series of layers; an input layer, a number of hidden layers, and an output layer. Each layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer. Each neuron within a single layer does not share connections with, and operates completely independently from one another [Stanford Vision and Leaning Lab, 2012].

Using the case of Computer Vision (CV) as an example, the input layer of a NN consists of neurons encoding the values of image pixels (RGB or greyscale intensities). The encoding is typically achieved by passing the raw input value through an activation function which outputs a normalised value. Often, activation functions in modern NNs output non-linearities, an example is to use a Sigmoid Function which maps an input to a value between 0 and 1 (*see Fig. 2.2 left*) [Nielsen, 2015].

However a more common activation function found in current NN models for CV is the Rectified Linear Unit (ReLU). It also adds non-linearity to the output, however it maps the input to a value within the range of 0 and ∞ (*see Fig. 2.2 right*) [Malik, 2019a].

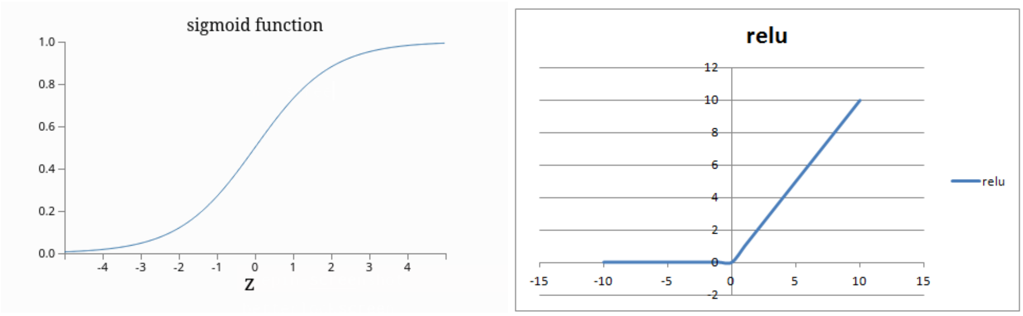

FIGURE 2.2: *Left*: The Sigmoid Function is one type of activation function. 'A bounded, differentiable, real function that is defined for all real input values and has a non negative derivative at each point' [Han and Moraga, 1995]. *Right*: An example of a ReLU activation function transforming x to a value between 0 and ∞ [Malik, 2019a].

The output layer of a CV classification network contains neurons representing the class scores of the task (*see Fig. 2.3*). For example, in a NN attempting to classify handwritten digits, the output layer would contain 10 neurons, representing the digits 0 - 9. If the first neuron fires, i.e. has an output $\approx l$, this will indicate that the network is confident the handwritten digit is 0, and so on [Nielsen, 2015].

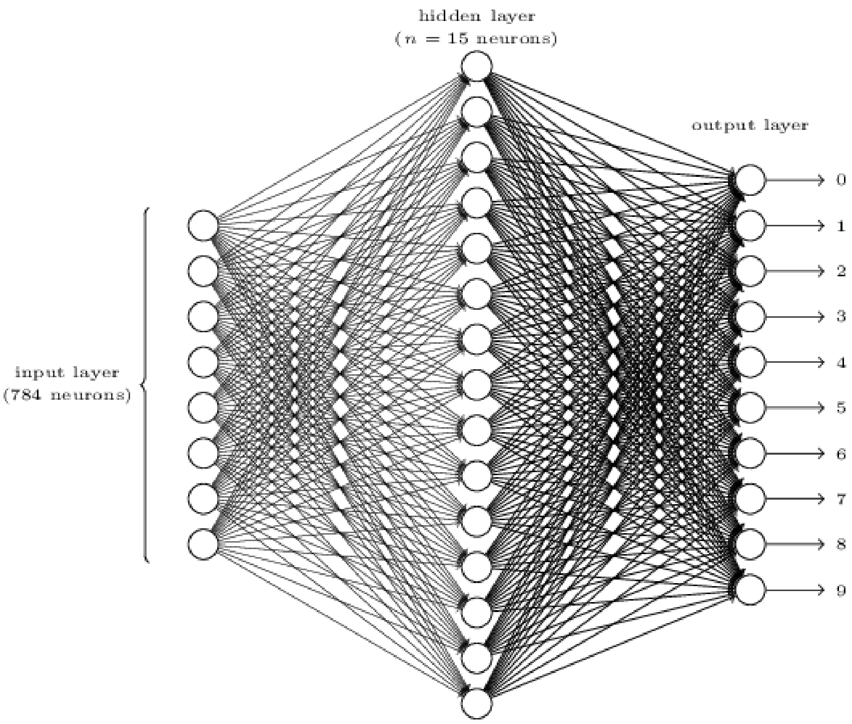

FIGURE 2.3: Neural Network. Example of a NN to classify handwritten digits. The input is a single vector of 28x28 pixels, i.e. 784 neurons, and outputs 10 neurons representing digits 0-9 [Nielsen, 2015].

NNs with a single hidden layer are able to approximate functions that contain any continuous mapping from one finite space to another, whereas with no hidden layers a NN model would only be able to represent linear functions or decision boundaries [Hornik, 1991].

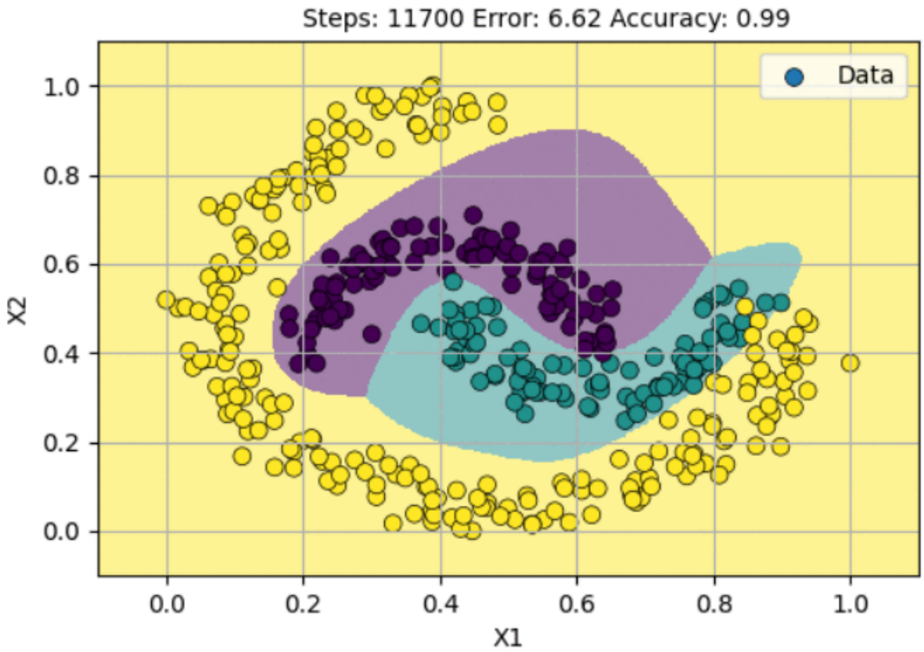

FIGURE 2.4: **Complex Decision Boundary** – Example of a decision boundary made capable by deep learning [Sapkota, 2020]

NNs are especially powerful when additional hidden layers are added to a network's architecture. By doing so, a model can not only approximate continuous functions to a high accuracy with less computational cost, but it can also represent complex composite functions [Sapkota, 2020]. An example of the complex decision boundaries that are possible from NNs with more than one hidden layer can be seen in *Fig. 2.4*.

NNs with two or more hidden layers fall under the category of deep learning, and are often referred to as Deep Neural Networks (DNNs) or Multi-Layer Perceptrons (MLPs). This subset of ML has become increasingly powerful with the rise of powerful variations of DNNs, namely Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) in recent years due to their successes within the field of CV.

2.1.2 Gradient Descent & Backpropagation

Training a NN consists of iteratively adjusting the values of weights at each neuron in order to minimise the model's output error. Although there are many algorithms available for determining the optimum values of weights, a common approach is by using some flavour of Gradient Descent (GD) together with a technique for efficiently computing partial derivatives within a directed graph called *backpropagation*.

There are three main variants of GD; vanilla GD or Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD), and Mini-Batch Gradient Descent (MBGD).

BGD computes the gradients of a cost function with regards to the weights within an entire training set. The cost function can take many forms depending on the architecture of the NN and the task it is concerned with, however the main principle behind it is to map the different values of each weight to a score which determines how well the model performs [Shung, 2018].

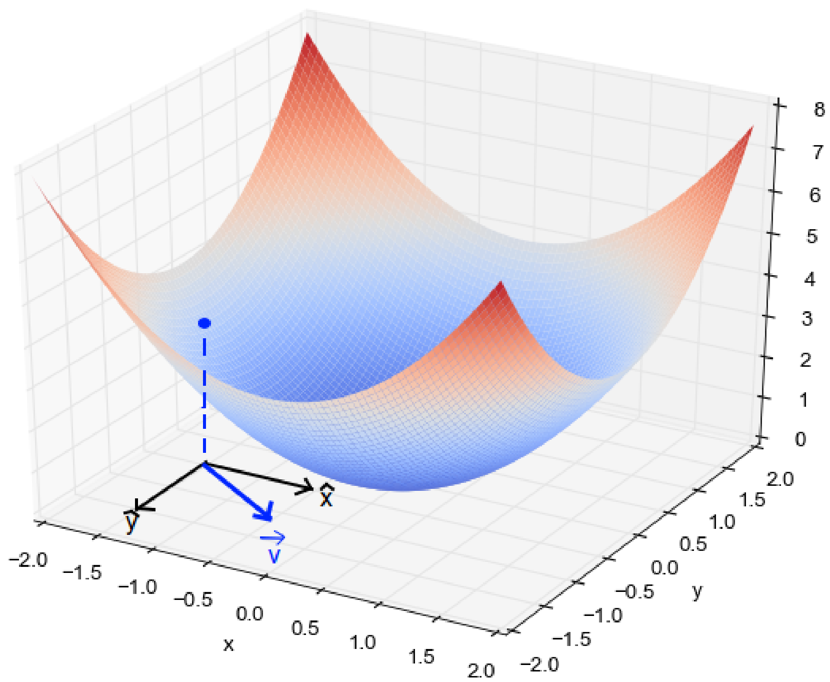

FIGURE 2.5: **Visualisation of GD Search Space** – An example of an *ideal* search space, where the vertical z axis shows the cost function $f(x, y)$, and $\vec{v}$ represents the resulting vector applied to the parameter in question [Bendersky, 2016].

A search space can then be defined by plotting the output of a cost function against the values of the weights it is concerned with, an example of such a search space with two weights can be seen in *Fig. 2.5*. As the number of weights increases, the harder it becomes to visualise the contours of a multi-dimensional plane. BGD then computes the directional derivative of this plane given a set of weight values, and uses this value as a vector with a magnitude defined by a *learning rate* hyper-parameter to update the weights of the network [Ruder, 2017].

SGD attempts to reduce the number of computations during training by only performing updates to weights for each training example instead of recomputing gradients for similar weights at each iteration. By removing these redundant calculations, SGD typically decreases the time taken to converge to an optimum solution of weights. Additionally, due to the high variance of each update, and so long as the learning rate is steadily decreased at each iteration, SGD has an equal chance at finding the global minimum than BGD [Ruder, 2017].

MBGD on the other hand, attempts to combine the benefits from BGD and SGD by performing an update for every mini-batch of n training examples. Therefore, allowing for the precision of BGD with similar speeds as SGD.

backpropagation is a computational technique commonly used within NN training for calculating partial derivatives used for GD algorithms in linear time with respect to the number of weights being optimised. This is an important step in order to train NNs within a sensible timeframe, considering the potentially high volume of weights that are needed for difficult tasks. A more detailed investigation into this technique will be discussed later in this chapter.

2.1.3 Vulnerabilities to Adversarial Attacks

NNs and DNNs have been adopted and deployed within a wide range of industry applications for tasks such as speech recognition or facial recognition, and have shown to perform adequately for many of these tasks. However, as mentioned in *Chapter 1*, NNs have been shown to be vulnerable to adversarial attacks. Specifically, by adding small, imperceptable changes to the input features, can lead to abnormal behaviours such as missclassification in the output layer.

This observation was first discovered in 2014, which found properties of NNs that cause them to learn uninterpretable solutions that could have counter-intuitive properties when imperceptable non-random pertubartions are made to a test input, known as *adversarial examples* [Szegedy et al., 2014]. Interestingly, these examples were shown to be robust, such that they have the same effect across models with varying architectures, activation functions, or trained on different datasets altogether. A tentative explanation for this phenomenon was to blame the non-linear nature of NNs, and cases of overfitting on trained data.

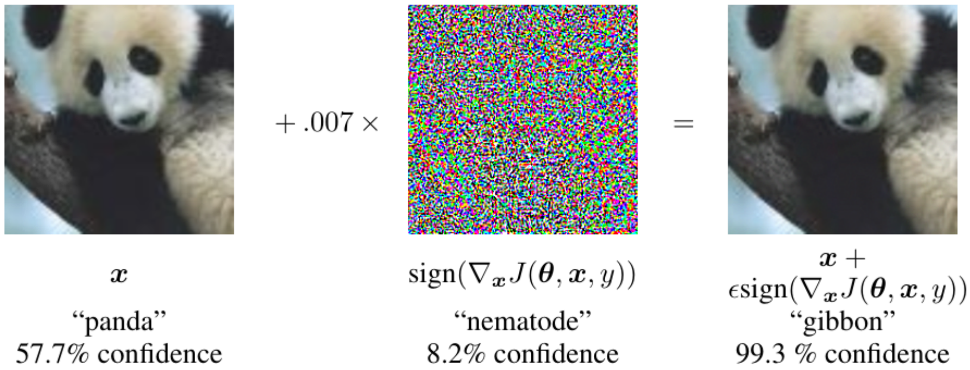

FIGURE 2.6: **Effects of Adversarial Examples** – A demonstration of the effects of adversarial examples; an input image of a panda with added noise causes the model to missclassify the image as a gibbon [Goodfellow et al., 2015].

However in 2015, further attempts to explain NN vulnerabilities to adversarial examples argued that it was not the non-linear nature, but rather the linear properties of NNs [Goodfellow et al., 2015]. This claim was supported by the authors' novel method for generatating new adversarial examples such that a model can undertake adversarial training.

2.1.4 Discrimination & Neural Networks

2.2 Formal Verification

Formal verification is an extensive field which has seen development in many areas of software engineering. As such, this section will attempt to provide a succinct overview of the ideas behind formal verification while keeping the focus on areas related to this thesis.

2.2.1 Background

2.2.2 Current Frameworks

2.3 Formal Verification of AI

This section will provide a more detailed investigation into the current research undertaken within AIV, with a focus on NNs and deep learning tasks.

2.3.1 Overview

2.3.2 Sapphire

2.4 Programming Paradigms for Machine Learning

2.4.1 Computational Graphs

2.4.2 Auto Differentiability

2.5 The Go Programming Language

2.5.1 Brief History

2.5.2 Go for ML

2.5.3 Go for Formal Verification

2.6 Conclusions

Chapter 3

Methodology

Chapter 4

Implementation

Chapter 5

Analysis

Chapter 6

Conclusions

Appendix A

Appendix Title Here

Write your Appendix content here.

Bibliography

Ammann, P. and Offutt, A. J. (2008). Introduction to software testing.

Amodei, D., Olah, C., Steinhardt, J., Christiano, P. F., Schulman, J., and Mané, D. (2016). Concrete problems in AI safety. *CoRR*, abs/1606.06565.

Bendersky, E. (2016). Understanding gradient descent - eli bendersky's website.

Bishop, C. (2006). *Pattern Recognition and Machine Learning*, volume 16, pages 140–155.

Boyer, R. S. and Moore, J. S. (1990). A theorem prover for a computational logic. In Stickel, M. E., editor, *10th International Conference on Automated Deduction*, pages 1–15, Berlin, Heidelberg. Springer Berlin Heidelberg.

Brown, T. B., Mané, D., Roy, A., Abadi, M., and Gilmer, J. (2018). Adversarial patch.

Chew, X. (2016). Gorgonia.

De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg. Springer-Verlag.

Dreyfus, G. (2005). *Neural Networks: An Overview*, pages 1–83. Springer Berlin Heidelberg, Berlin, Heidelberg.

Fisher, K., Launchbury, J., and Richards, R. (2017). The hacms program: Using formal methods to eliminate exploitable bugs. *Philosophical Transactions of The Royal Society A Mathematical Physical and Engineering Sciences*, 375:20150401.

GoLang (2020). Golang machine learning libraries.

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

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015). Explaining and harnessing adversarial examples.

Guaspari, D., Marceau, C., and Polak, W. (1993). Formal verification of ada programs. In Martin, U. and Wing, J. M., editors, *First International Workshop on Larch*, pages 104–141, London. Springer London.

Han, J. and Moraga, C. (1995). The influence of the sigmoid function parameters on the speed of backpropagation learning. In Mira, J. and Sandoval, F., editors, *From Natural to Artificial Neural Computation*, pages 195–201, Berlin, Heidelberg. Springer Berlin Heidelberg.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257.

Innes, M., Barber, D., Besard, T., Bradbury, J., Churavy, V., Danisch, S., Edelman, A., Karpinski, S., Malmaud, J., Revels, J., and et al. (2017). On machine learning and programming languages.

Kokke, W. (2020). A library for translating tensorflow models to z3.

Kouvaros, P. and Lomuscio, A. (2016). Parameterised verification for multi-agent systems. *Artif. Intell.*, 234(C):152–189.

Lomuscio, A., Qu, H., and Raimondi, F. (2017). Mcmas: an open-source model checker for the verification of multi-agent systems.

Malik, F. (2019a). Neural network activation function types. https://medium.com/fintechexplained/neural-network-activation-function-types-a85963035196. accessed: 06.11.2019.

Malik, F. (2019b). Neural networks bias and weights. https://medium.com/fintechexplained/neural-networks-bias-and-weights-10b53e6285da. accessed: 06.11.2019.

Molnar, C. (2019). *Interpretable Machine Learning*. https://christophm.github.io/interpretable-ml-book/.

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

Pereira, A. and Thomas, C. (2020). Challenges of machine learning applied to safety-critical cyber-physical systems. *Machine Learning and Knowledge Extraction*, 2(4):579–602.

Polak, W. (1979). An exercise in automatic program verification. *IEEE Transactions on Software Engineering*, (5):453–458.

Ruder, S. (2017). An overview of gradient descent optimization algorithms.

Russell, S., Dewey, D., and Tegmark, M. (2016). Research priorities for robust and beneficial artificial intelligence.

Sammut, C. and Webb, G. I., editors (2010). *Neural Networks*, pages 716–716. Springer US, Boston, MA.

Sapkota, S. (2020). Perceptron to deep-neural-network — rough ai blog. https://tsumansapkota.github.io/algorithm/2020/06/06/Perceptron-to-DeepNeuralNets/. Accessed: 2021-04-05 05:32:32.

Shaikhha, A., Fitzgibbon, A., Vytiniotis, D., and Peyton Jones, S. (2019). Efficient differentiable programming in a functional array-processing language. *Proc. ACM Program. Lang.*, 3(ICFP).

Shung, K. P. (2018). Gradient descent: Simply explained? — by koo ping shung — towards data science.

Stanford Vision and Leaning Lab (2012). Cs231n convolutional neural networks for visual recognition. http://cs231n.github.io/convolutional-networks/. accessed: 04.11.2019.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2014). Intriguing properties of neural networks.