Heriot-Watt University

Masters Thesis

# A Framework for Formally Verifying 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, 'A Framework for Formally Verifying 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 is 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 there is a need for formal verification methods that can prove the robustness of a system.

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

This project will investigate how robust programming infrastructure 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 Descrimination & Neural Networks . . . 13
- 2.2 Formal Verification . . . 14
  - 2.2.1 Background . . . 14
  - 2.2.2 Theorem Provers . . . 14
  - 2.2.3 Model Checkers . . . 15
- 2.3 Formal Verification of Neural Networks . . . 15
- 2.4 Programming Paradigms for Deep Learning . . . 16
  - 2.4.1 Automatic Differentiation . . . 17
- 2.5 Conclusions . . . 17

**3 Methodology** **18**

**4 Implementation** **19**

**5 Analysis** **20**

**6 Conclusions** **21**

**A Appendix Title Here** **22**

**Bibliography** **23**

# List of Figures

1.1 Google's adversarial 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
2.7 Demonstration of the effects of adversarial examples in real world 12
2.8 Example of negative bias effects in neural networks . . . . . . . . 13

# List of Tables

# List of Abbreviations

| | |
|---|---|
| AD | Automatic Differentiation. 17 |
| AI | Artificial Intelligence. 3, 4, 15 |
| AIV | Artificial Intelligence Verification. 3, 4, 6 |
| BGD | Batch Gradient Descent. 10, 11 |
| CNN | Convolutional Neural Network. 9, 16 |
| CV | Computer Vision. 7–9, 16 |
| DNN | Deep Neural Network. 9, 11, 13, 16, 17 |
| FP | Functional Programming. 4 |
| GAN | Generative Adversarial Network. 13 |
| MAS | Multi-Agent System. 3, 15 |
| MBGD | Mini-Batch Gradient Descent. 10, 11 |
| ML | Machine Learning. 1–4, 6, 9, 16, 17 |
| MLP | Multi-Layer Perceptron. 9 |
| NN | Neural Network. 1–13, 15–17 |
| RBFN | Radial Basis Function Network. 12 |
| ReLU | Rectified Linear Unit. 7, 8 |
| RNN | Recurrent Neural Network. 9, 16 |
| SD | Symbolic Differentiation. 17 |
| SGD | Stochastic Gradient Descent. 10, 11 |
| SMT | Satisfiability Modulo Theories. 4, 16 |

# 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.

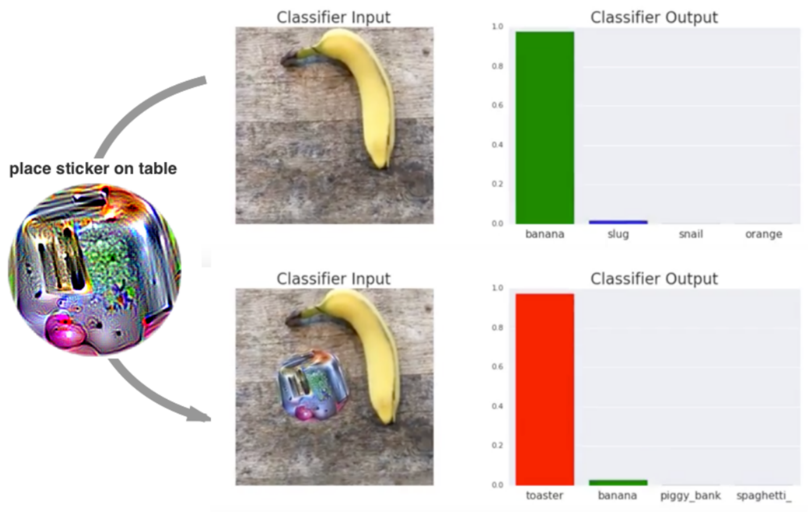

FIGURE 1.1: **Google's Adversarial Patch** – An example of a method to create targeted adversarial attacks on NNs by adding carefully designed noise via a physical patch [Brown et al., 2018].

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 lower interpretability 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 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 ML 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 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 not 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, which was 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 comparatively quicker speeds than its Python counterparts (Theano and TensorFlow) [GoLang, 2020].

A good 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 model queries 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 example training data sets 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 developed 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 traditional and ML software 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

### 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].

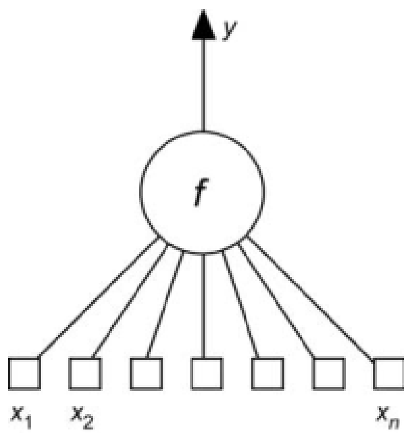

FIGURE 2.1: **Artificial Neuron** – a nonlinear bounded function $y = f(x_1, x_2, \ldots, x_n; w_2, \ldots, w_n)$ where the $x_i$ are the input values and the $w_i$ are the weights of the neuron [Dreyfus, 2005].

A weight is assigned to each of a neuron's inputs. They are the coefficients 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 is the Rectified Linear Unit (ReLU). Although computed as a piecewise linear function, ReLU also adds non-linearity to the output. The ReLU function maps an input to a value within the range of 0 and $\infty$ (*see Fig. 2.2 right*) [Malik, 2019a].

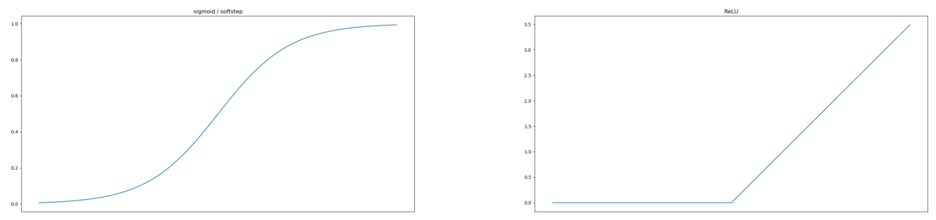

FIGURE 2.2: *Left*: The Sigmoid Function is one type of activation function. 'A bounded, differentiable, real function that is defined for all real input values and has a non negative derivative at each point' [Han and Moraga, 1995]. *Right*: An example of a ReLU activation function transforming $x$ to a value between 0 and $\infty$ [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 $\approx 1$, this will indicate that the network is confident the handwritten digit is 0, and so on [Nielsen, 2015].

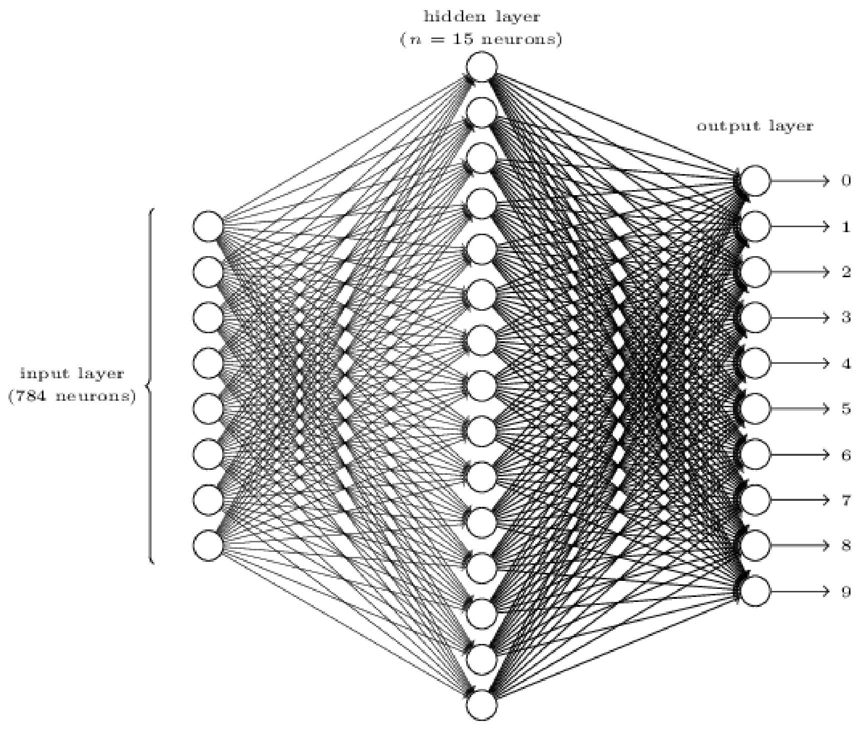

FIGURE 2.3: Neural Network. Example of a NN to classify handwritten digits. The input is a single vector of 28x28 pixels, i.e. 784 neurons, and outputs 10 neurons representing digits 0-9 [Nielsen, 2015].

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].

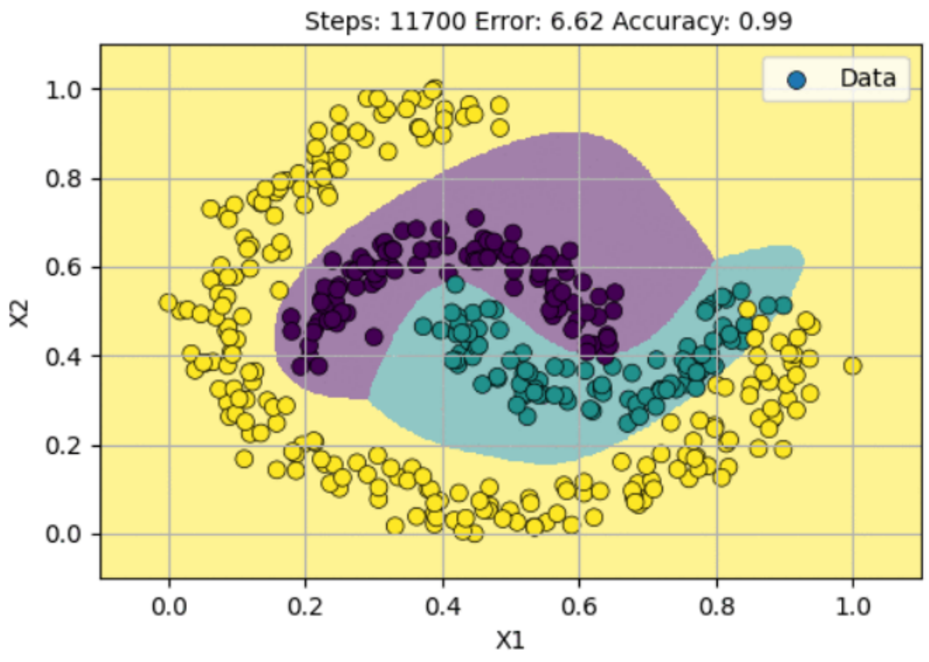

FIGURE 2.4: **Complex Decision Boundary** – Example of a decision boundary made capable by deep learning [Sapkota, 2020]

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 to use some flavour of *gradient descent* together with a technique for efficiently computing partial derivatives within a directed graph called *backpropagation*.

There are three main variants of gradient descent; vanilla gradient descent 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].

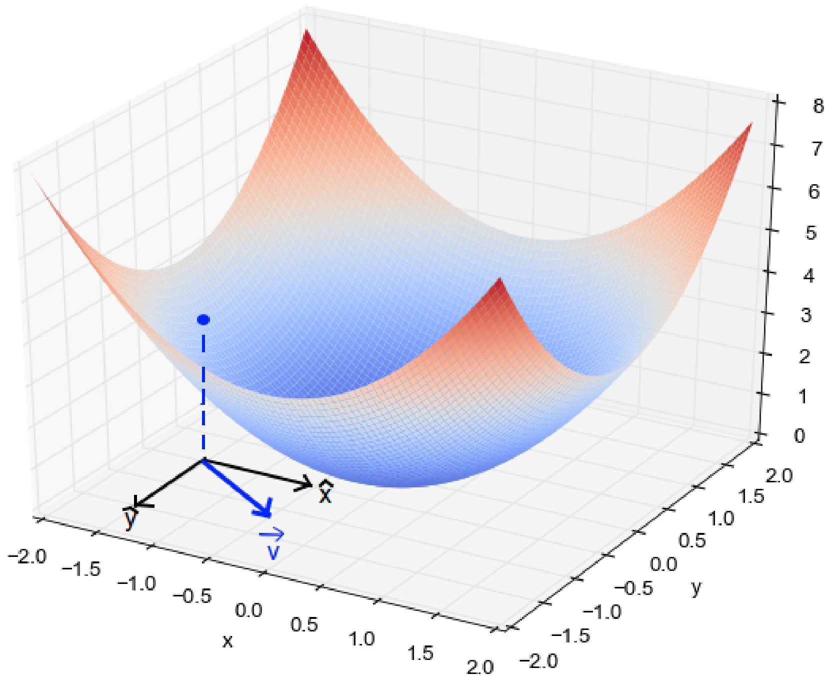

Figure 2.5: **Visualisation of Gradient Descent Search Space** – An example of an *ideal* search space, where the vertical $z$ axis shows the cost function $f(x, y)$, and $\vec{v}$ represents the resulting direction of the maximum gradient applied to the parameter in question [Bendersky, 2016].

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 to 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 gradient descent 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 complex tasks. A more detailed investigation into this technique will be discussed later in this chapter (*Section 2.4*).

### 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 made 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 data sets altogether. A tentative explanation for this phenomenon was to blame the non-linear nature of NNs, and cases of poor generalisation on test data.

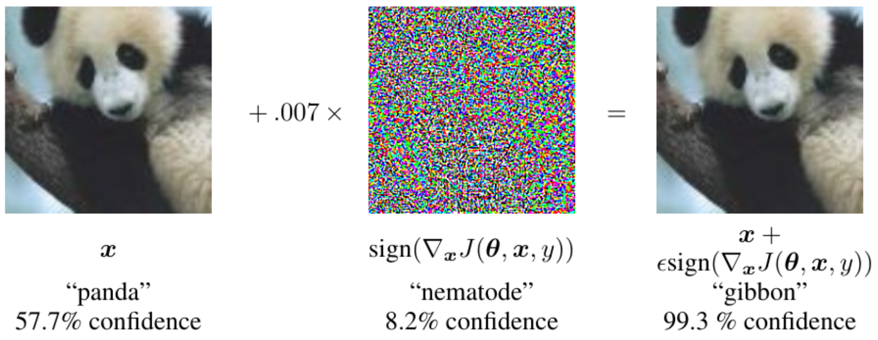
$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

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 behaviour of NNs which is sufficient to cause adversarial examples [Goodfellow et al., 2015]. This claim was supported by the authors' demonstration that leveraging non-linear NN families such as Radial Basis Function Networks (RBFNs) can significantly reduce the vulnerabilities to adversarial examples.

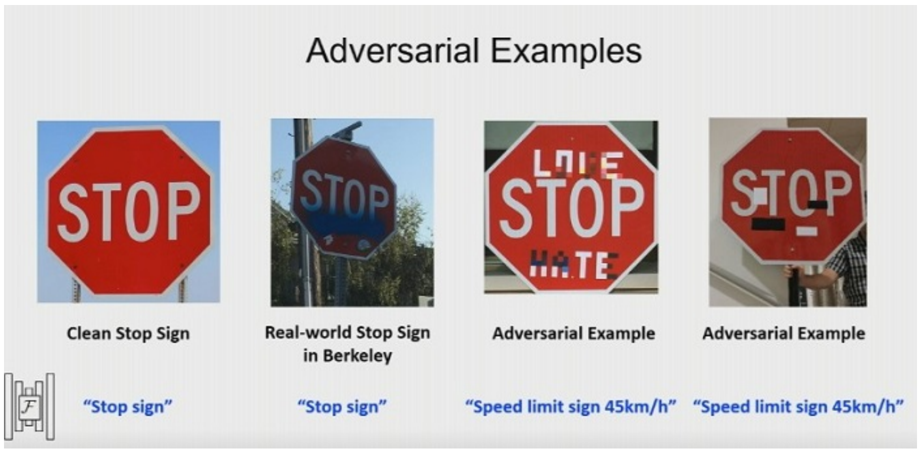

FIGURE 2.7: **Effects of Adversarial Examples in Real World** – A demonstration of how adversarial examples can be used in real world situations to cause misclassification of stop signs [Eykholt et al., 2018].

Research in 2018 showed that adversarial examples can be used in real world situations in order to fool a DNN used for street sign recognition within a self-driving car's navigation system by placing black and white stickers on street signs (*see Fig. 2.7*) [Eykholt et al., 2018].

These vulnerabilities have demonstrated that NN technology has yet to reach the level of maturity necessary for applications in safety-critical systems, and have raised concerns over the robustness of NNs in general.

### 2.1.4 Descrimination & Neural Networks

Aside from vulnerabilities caused by intrinsic properties of NNs, there are issues which stem from the data being used to train supervised models too. That is, if the training data has inherent bias towards or against a specific class within the domain, the output of the model trained on that data will reflect these biases (*see Fig. 2.8*). Subsequently, when bias data is used to train NNs used in applications which have either a direct or indirect affect on people's lives, there is a chance that they can result in simulated descrimination of certain social groups.

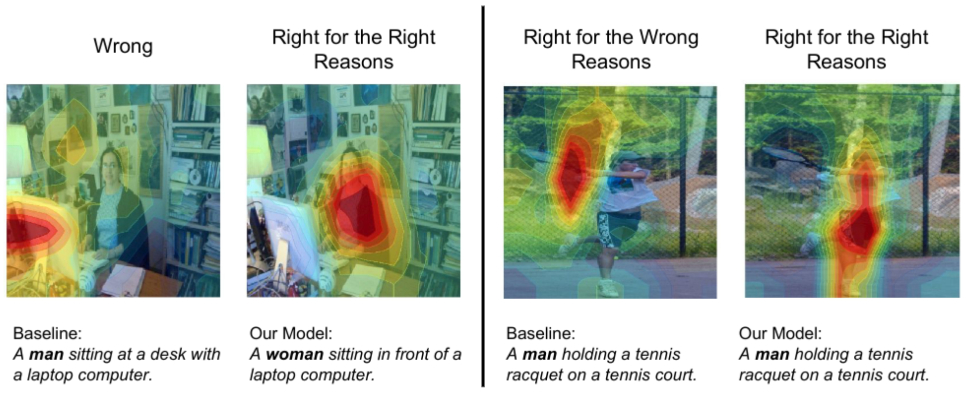
Wrong
Right for the Right Reasons
Right for the Wrong Reasons
Right for the Right Reasons
Baseline: A **man** sitting at a desk with a laptop computer.
Our Model: A **woman** sitting in front of a laptop computer.
Baseline: A **man** holding a tennis racquet on a tennis court.
Our Model: A **man** holding a tennis racquet on a tennis court.

FIGURE 2.8: **Discriminating Bias in Training Data** – An example of how image captioning NNs use the features present in a bias data set to generate either incorrect captions (*left*), or use the wrong features to generate correct captions (*right*) [Burns et al., 2019].

In recent years, there have been many attempts to mitigate the effects of unwanted bias in data being fed to NNs. These include work in facial recognition, where a model is trained separately on demographic representations prior to being trained on face attribute data, thus preserving potential users' demographic privacy [Ryu et al., 2018]; or more recently, developing new regularisation techniques for Generative Adversarial Networks (GANs) which allow a model to categorise classes while ignoring bias features in the training data [Kim et al., 2019].

Although there has been significant progress in this area of research and that discussed in *Section 2.1.3*, many of the proposed solutions have not yet become ubiquitous in industry. As such, there is a need for creating formal methods which allow developers to verify the fairness and robustness of their systems in a wide range of programming environments.

## 2.2 Formal Verification

Formal verification is a mature and extensive discipline 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

Although formal verification stems from a long history of development within the fields of classical and first-order logic [Boole, 2009, Russell, 1937, Smith, 2011], this thesis is concerned with modern formal methods used for software engineering tasks, and thus will use the following definition.

Formal verification is defined by a set of mathematical tools used to analyse the space of possible system states both in hardware and software [Seligman et al., 2015]. In other words, checking that a system will not produce abnormal behaviours given a specific input, or to verify the correctness of hardware or software design [Grout, 2008].

Early software systems were typically verified by hand when necessary. However, as the size and complexity of software increased over time, so too did the computational cost of exhaustively verifying them. As such, huge efforts have been made over the past few decades to develop formal methods frameworks which utilise the power of computing for software engineers.

### 2.2.2 Theorem Provers

The earliest approach for formally verifying software, and still commonly used, is with theorem provers. Theorem provers were initially developed as interactive, or computer assisted frameworks, however over time the much harder task of automated theorem provers were introduced.

These tools allow developers to formalise theorems using some variant of mathematical logic, such as propositional logic, predicate logic, or first-order logic to name a few. Then, by using *proof checking*, one can establishes the validity of a theorem by mechanically checking the proof [Geuvers, 2009].

### 2.2.3 Model Checkers

Model checking is another branch of formal verification for software that came along after theorem provers. This technique consists of three main tasks, *modelling*, *specification*, and *verification*.

Modelling is the process of converting a system design into a formalism accepted by a model checking tool. Once this process has finished, a specification of the properties in which a design should satisfy must be provided before verification. Once a model and a specification have been provided, the process of verifying the system can occur. In an ideal world, the final stage can be automated, but often it is necessary to manually analyse the verification results [Clarke et al., 2018].

In other words, given a program $P$ and a specified property $\phi$, the primary goal of a model checker is to search the space of possible states of $P$, and ensure that $\phi$ holds in all scenarios [Zhang et al., 2019].

## 2.3 Formal Verification of Neural Networks

In recent years, there has been growing interest in using formal verification tools within the field of AI [Russell et al., 2016]. Although there have been promising advancements in areas such as MAS [Kouvaros and Lomuscio, 2016, Lomuscio et al., 2017], there is significantly less research regarding the formal verification of NNs.

The most common methods for evaluating NNs rely on testing models on previously unseen data, which can provide statistical guarantees regarding accuracy and generalisation. However this approach tends to be incomplete as it does not provide an exhaustive search of all possible inputs to the network [Akintunde et al., 2020].

Initial attempts at formally verifying NNs focused on using *reachability analysis* as a verifiable property. *Reachability analysis* is concerned with determining whether a given state is reachable in a number of steps from an initial state of a system [Akintunde et al., 2018]. The idea behind conducting such an analysis, is that it is then possible to verify that an unwanted state of a system such as those discussed in *Sections 2.1.3 & 2.1.4* is never reached during the lifecycle of the system.

Thus far, this area of research is still in its early stages, however fast progress has been made in demonstrating how it is possible to achieve real-world scalability *w.r.t* verifying NNs [Pulina and Tacchella, 2010, Xiang et al., 2018], including a handful of deep structures such as CNNs [Kouvaros and Lomuscio, 2018] and RNNs [Zhang et al., 2020].

Additionally, progress has been made in developing tools designed to utilise the many formal methods that have been worked on for decades which allow developers to analyse and verify their models easily. This includes tools such as the Maribou Framework for DNNs [Katz et al., 2019], which builds on the authors' previous work in using SMT-based techniques to verify DNNs; and the Sapphire Python library [Kokke, 2020], which translates TensorFlow feed-forward NN model queries to the Z3 SMT solver created by Microsoft Research [De Moura and Bjørner, 2008].

However, as NN development is starting to become prevalent across a wider range of programming languages and paradigms, it is important for further work to be made in developing new frameworks for formally verifying NNs, and to investigate how different programming language infrastructures can affect this development.

## 2.4 Programming Paradigms for Deep Learning

ML has to date offered incredibly powerful software tools for developing complex systems quickly compared to *traditional* programming techniques. However, as these systems grow in size to deal with harder tasks such as CV or natural language generation, so to does the risk of incurring large ongoing computational costs [Sculley et al., 2015]. As such, an enormous amount of work has gone into ensuring the toolkits used by ML engineers allow for scalable and efficient development of their models.

### 2.4.1 Automatic Differentiation

The biggest challenge when computing the necessary steps of training NNs, lies in the need for calculating hundreds, if not thousands, of partial derivatives in order to find the directional gradient for optimising the model's weights [Steinitz, 2013].

Although Symbolic Differentiation (SD) is used within computing [Maxima, 2020, Trott, 2006], since it can be useful when there is a need to produce a legible symbolic description of a derivative, it has shown poor performance when handling common data structures which appear in ML algorithms [Achyutuni, 2020].

Automatic Differentiation (AD) is a technique which has been adopted by the most successful ML frameworks such as TensorFlow and Theano in Python, as well as emerging frameworks such as Gorgonia in Go.

The idea behind AD stems from the ability to break down any arbitrary function, in this case a NN, within a program into a computational graph of *atomic* mathematical functions. This in turn allows the derivative of the function to be represented as a computational graph, which can be built simultaneously as a NN is being constructed [Baydin et al., 2015].

## 2.5 Conclusions

NNs and DNNs are becoming an ubiquitous part of a software engineer's toolkit. As these programming paradigms become increasingly complex, so too does the nature of formally verifying them. This section has highlighted the current approaches being used to tackle this issue, and how these techniques are becoming more available to developers through tools like Maribou and Sapphire [Katz et al., 2019, Kokke, 2020].

As ML capabilities are being developed across an extended variety of programming languages [GoLang, 2020, Shaikhha et al., 2019], it is important to ensure that formal methods of verifying these complex models are also explored and ultimately implemented.

# Chapter 3

# Methodology

# Chapter 4

# Implementation

# Chapter 5

# Analysis

# Chapter 6

# Conclusions

# Appendix A

# Appendix Title Here

Write your Appendix content here.

# Bibliography

Achyutuni, K. (2020). The role of automatic differentiation in machine learning — by kiran achyutuni — deep dives into computer science — medium.

Akintunde, M., Lomuscio, A., Maganti, L., and Pirovano, E. (2018). Reachability analysis for neural agent-environment systems. In *KR*, pages 184–193.

Akintunde, M. E., Botoeva, E., Kouvaros, P., and Lomuscio, A. (2020). Formal verification of neural agents in non-deterministic environments. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '20, page 25–33, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

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.

Baydin, A. G., Pearlmutter, B. A., and Radul, A. A. (2015). Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767.

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

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

Boole, G. (2009). *An Investigation of the Laws of Thought: On Which Are Founded the Mathematical Theories of Logic and Probabilities.* Cambridge Library Collection - Mathematics. Cambridge University Press.

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.

Burns, K., Hendricks, L. A., Saenko, K., Darrell, T., and Rohrbach, A. (2019). Women also snowboard: Overcoming bias in captioning models.

Chew, X. (2016). Gorgonia.

Clarke, E., Grumberg, O., Kroening, D., Peled, D., and Veith, H. (2018). *Model Checking, second edition.* Cyber Physical Systems Series. MIT Press.

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.

Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Rahmati, A., Xiao, C., Prakash, A., Kohno, T., and Song, D. (2018). Robust physical-world attacks on deep learning visual classification. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1625–1634.

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.

Geuvers, J. (2009). Proof assistants : history, ideas and future. *Sadhana : Academy Proceedings in Engineering Sciences (Indian Academy of Sciences)*, 34(1):3–25.

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.

Grout, I. (2008). Chapter 1 - introduction to programmable logic. In Grout, I., editor, *Digital Systems Design with FPGAs and CPLDs*, pages 1–41. Newnes, Burlington.

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.

Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D. L., Kochenderfer, M. J., and Barrett, C. (2019). The marabou framework for verification and analysis of deep neural networks. In Dillig, I. and Tasiran, S., editors, *Computer Aided Verification*, pages 443–452, Cham. Springer International Publishing.

Kim, B., Kim, H., Kim, K., Kim, S., and Kim, J. (2019). Learning not to learn: Training deep neural networks with biased data.

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.

Kouvaros, P. and Lomuscio, A. (2018). Formal verification of cnn-based perception systems.

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.

Maxima (2020). Maxima, a computer algebra system.

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.

Pulina, L. and Tacchella, A. (2010). An abstraction-refinement approach to verification of artificial neural networks. In *International Conference on Computer Aided Verification*, pages 243–257. Springer.

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

Russell, B. (1937). *Principles of Mathematics*. Routledge.

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

Ryu, H. J., Adam, H., and Mitchell, M. (2018). Inclusivefacenet: Improving face attribute detection with race and gender diversity.

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.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J., and Dennison, D. (2015). Hidden technical debt in machine learning systems. In *NIPS*.

Seligman, E., Schubert, T., and Kumar, M. V. A. K. (2015). Chapter 1 - formal verification: From dreams to reality. In Seligman, E., Schubert, T., and Kumar, M. V. A. K., editors, *Formal Verification*, pages 1–22. Morgan Kaufmann, Boston.

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.

Smith, R. (2011). Aristotle. prior analytics book 1. *Ancient Philosophy*, 31:417–424.

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

Steinitz, D. (2013). Backpropogation is just steepest descent with automatic differentiation — maths, stats & functional programming.

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

Trott, M. (2006). The mathematica guidebook for symbolics. with dvd. *tThe Mathematica GuideBook for Symbolics*.

Xiang, W., Tran, H.-D., and Johnson, T. T. (2018). Output reachable set estimation and verification for multilayer neural networks. *IEEE transactions on neural networks and learning systems*, 29(11):5777–5783.

Zhang, H., Shinn, M., Gupta, A., Gurfinkel, A., Le, N., and Narodytska, N. (2020). Verification of recurrent neural networks for cognitive tasks via reachability analysis. In De Giacomo, G., Catala, A., Dilkina, B., Milano, M., Barro, S., Bugarin, A., and Lang, J., editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, including 10th Conference on Prestigious Applications of Artificial Intelligence, PAIS 2020 - Proceedings*, Frontiers in Artificial Intelligence and Applications, pages 1690–1697. IOS Press BV. Publisher Copyright: © 2020 The authors and IOS Press. Copyright: Copyright 2020 Elsevier B.V., All rights reserved.; 24th European Conference on Artificial Intelligence, ECAI 2020, including 10th Conference on Prestigious Applications of Artificial Intelligence, PAIS 2020 ; Conference date: 29-08-2020 Through 08-09-2020.

Zhang, J. M., Harman, M., Ma, L., and Liu, Y. (2019). Machine learning testing: Survey, landscapes and horizons.