THE UNIVERSITY *of* EDINBURGH

Verification using Formalised Mathematics and Theorem Proving of Reinforcement and Deep Learning

Mark Chevallier

Doctor of Philosophy
Artificial Intelligence Applications Institute
School of Informatics
University of Edinburgh
2024

Abstract

In modern artificial intelligence research, frequently there is little emphasis on mathematical certainty; results are often shown by experimentation, and understanding precisely why a particular method works, or the guarantees that they will be effective, is often constrained to speculation and discussion.

Formal mathematics via theorem proving brings a precision of explanation and certainty that can be missing in this field. We present work that applies the benefits of formal mathematics to two different fields of artificial intelligence, in two different ways.

Using the Isabelle theorem prover, we formalise Markov Decision Processes (MDPs) with rewards, fundamental to reinforcement learning, and use this as the basis for a formalisation of Q learning, a significant reinforcement learning algorithm. Q learning attempts to learn the reward function of an unknown MDP by estimation, correcting its estimates as it navigates the MDP repeatedly. We also formalise the Dvoretzky Stochastic Approximation theorem, a result fundamental to many stochastic processes. It is especially relevant to our work as it is necessary to prove that (given certain assumptions) the estimates of the Q learning algorithm converge to the true values of the reward function.

Secondly, we use theorem proving to integrate a formalised logical system with deep learning, into a neurosymbolic process. We formalise Linear Temporal Logic over finite paths (LTL_f), and develop a loss function (and its derivative) over it that returns a real value corresponding to the satisfaction of a given LTL_f constraint over a given path. We prove that this is sound with respect to the semantics of LTL_f. We use the code generation capabilities of Isabelle to then integrate this into a PyTorch deep learning process designed to learn trajectories. Lastly, we demonstrate experimentally that we can use the resulting neurosymbolic process to learn using LTL_f constraints on the trajectories as well as by imitation of a demonstrator.

Acknowledgements

Many thanks to my supervisor Jacques Fleuriot, my second supervisor Paul Jackson, and the many others who have helped me with my research, especially my fellows in the AI Modelling Lab. In particular, I would like to thank Imogen Morris and Jake Palmer, whose assistance both early on and later was very helpful and deeply appreciated, and all of those who assisted with proofreading and checking this thesis. I would like to thank my friends and family also, whose support has been invaluable, especially my late mother, who I hope would be proud of what I have achieved.

I am also grateful to EPSRC for funding my work and making it possible.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise below or in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Chapter 4 describes joint-work with Matthew Whyte, under the supervision of J. Fleuriot. Mark Chevallier formalised the deep embedding of LTL_f, its evaluation function, the loss function and its components, the smooth maximum and minimum functions, the proofs against them, and all Isabelle work except that in section 4.3.3. These proofs demonstrated both that the LTL_f formalisation provided expected results, and also showed that the loss function was sound with respect to the evaluation function as the smooth functions tend to their non-smooth counterparts. He also worked on part of the code generation process in Isabelle and OCaml, fine-tuning the Isabelle specifications to result in workable code. Matthew Whyte worked on the formalisation of the derivative of the loss function, the method used to integrate it into PyTorch for backpropagation during the training of the neural network by using a subclass of the `autograd.Function` class, the Isabelle work found in section 4.3.3 and part of the code generation process. Some aspects of the work have been published in the proceedings of Overlay 2022, AIxIA 2022 [30].

(*Mark Chevallier*)

Table of Contents

1 Introduction **1**

1.1 Formal mathematics and logic 2
1.2 Reinforcement learning and deep learning – a brief overview 6
1.3 Formal verification of modern AI 10
1.3.1 Adversarial inputs and robustness 11
1.3.2 Safe outputs 12
1.3.3 Theorem proving based formal verification 14
1.3.4 Formal verification of reinforcement learning 14
1.4 Goal and contributions of the work 15
1.5 Organisation 16

2 Markov Decision Processes **19**

2.1 Introduction 19
2.2 Isabelle/HOL 21
2.3 Background 24
2.3.1 Markov decision processes 24
2.3.2 Hölzl's formalisation of MDPs 26
2.4 The fundamentals of our formalisation 28
2.4.1 Formalising finite Markov Reward Processes 29
2.4.2 A slice-based approach to agent paths through an MDP 31
2.4.3 Proving convergence properties 35
2.4.4 Deriving the Bellman equation 37
2.5 Proving the existence of an optimal policy 38
2.5.1 Why is an optimal policy important? 38
2.5.2 Functions on states as a vector space 39
2.5.3 Operators over functions on states as matrices 42
2.5.4 Stochastic matrices 46

2.5.5 Spectral radius using Gelfand's formula 47
2.5.6 Casting functions to vectors and matrices 48
2.5.7 An optimal policy definition 50
2.5.8 The L_π operator and its supremum, $\mathscr{L}_{max}$ 52
2.5.9 Final steps 55
2.6 Value and policy iteration 57
2.6.1 Finding an optimal policy 57
2.6.2 Value iteration 57
2.6.3 Policy iteration 61
2.7 Case study 64
2.7.1 The game - Lavaworld 65
2.8 Related work 71
2.9 Conclusion 73

3 Fundamentals of reinforcement learning in Isabelle 77
3.1 Introduction 77
3.1.1 Q learning 77
3.2 Measure theory and probability theory in Isabelle 83
3.2.1 A very brief introduction to measure theory and probability theory 83
3.2.2 Measure spaces and probability spaces in Isabelle 86
3.2.3 Integrability of random variables 88
3.2.4 A brief introduction to filtrations of sequences of random variables 91
3.3 Dvoretzky's stochastic approximation in Isabelle 97
3.3.1 A brief introduction to Dvoretzky's stochastic approximation . 97
3.3.2 Preparatory lemma 99
3.3.3 Kolmogorov's inequality 102
3.3.4 A set of convergence criteria for sequences of random variables 104
3.3.5 The Du Bois-Reymond test for convergence 108
3.3.6 Dvoretsky's stochastic approximation theorem 113
3.4 Related work 118
3.5 Conclusion 118

4 Formalised logical constraint satisfaction implemented into a neural network **121**
4.1 Introduction . . . 121
4.2 Formalising linear temporal logic . . . 123
4.2.1 States and paths . . . 123
4.2.2 LTL and finite traces of states . . . 124
4.3 A LTL-based loss function and its derivative . . . 129
4.3.1 Soft functions and their derivatives . . . 130
4.3.2 Formalising the loss function . . . 132
4.3.3 Derivative of the loss function . . . 135
4.4 A PyTorch-compatible LTL loss function . . . 137
4.4.1 OCaml code generation . . . 138
4.4.2 PyTorch integration . . . 139
4.5 Experiments . . . 140
4.5.1 Domain setup . . . 140
4.5.2 Unconstrained training . . . 141
4.5.3 Constrained training with LTL_f . . . 141
4.5.4 Results . . . 142
4.5.5 Altering the η weighting value . . . 144
4.6 Related work . . . 148
4.6.1 Comparison with Innes and Ramamoorthy's work . . . 149
4.7 Conclusion . . . 150

5 Conclusion **153**
5.1 Future work . . . 156

References **159**

Chapter 1

Introduction

In modern artificial intelligence research, frequently there is little emphasis on mathematical certainty in engineering new approaches. Experimentation is typically how a successful algorithm is demonstrated, with little effort in rigorously proving its effectiveness. Understanding *precisely* why a particular method works, or the certainty that they will be effective, is often constrained to speculation and discussion [87]. Indeed, it has been argued that artificial intelligence is best viewed as an empirical science, where experiment rather than deduction is key to verification of results [142]. This leads to the potential for doubt over results, especially if high stochasticity is part of an algorithm or the source code is unavailable, leading to an inability to reproduce the results and a lack of verifiability [60, 97, 54].

This mathematical understanding was present in the early days of the development of artificial intelligence: for example, the development of the artificial neuron in 1943 was accompanied by ample mathematical background [104]. It is understandable, though, that the empirically verifiable effectiveness of modern approaches may have led to a reduced priority on the mathematical understanding of that effectiveness.

Formal mathematics and theorem proving brings with it a certainty in their correctness that can be missing in this field, even from pen-and-paper mathematical proofs. One of the primary purposes of formalising mathematics in the early 19th and 20th centuries was to bring an increased certainty [55, 58]. Informal – a word we use here merely to mean "not expressed using formal language and methods" – pen-and-paper proofs are arguments intended to persuade experts that their theorems must be true. But even now experts disagree on the validity of certain proofs, sometimes due to increasingly specialised mathematics, and sometimes due to a lack of trust in certain methods (i.e. geometric proofs) [159]. This occurs even on important mathematical

issues, where experts in the field can disagree (for example, Mochizuki's proof of the *abc* conjecture) [1].

Formal mathematics tries to increase the certainty in results by formalising both the language those results are expressed in, and the methods by which inferences can be made. By "formalising" here, we mean using a symbolic language with very strict syntactic rules for inference, each reflecting logical arguments that are nearly universally acceptable. By allowing inferences to proceed only by these means, any theorem proved in this way carries significantly greater certainty than an informal proof, as we discuss in Section 1.1.

Additionally, results proven this way can reliably be re-used to develop further proofs. Thus, the development of mathematics can follow the same path, building a chain of results none of whose certainty is in doubt.

1.1 Formal mathematics and logic

Logical reasoning is the process of finding conclusions, from a set of given premises, in such a way that the truth of those conclusions cannot be in doubt without doubting the premises. This distinguishes it from other kinds of reasoning, based on experiences or beliefs, where there is room for doubt in one's conclusions.

Valid logical reasoning preserves the truth of its premises, and in this way underlies mathematics. A mathematical proof is an argument – a social task to convince the reader that its conclusion cannot be in doubt – and a successful proof persuades using the tools of logic. Informal mathematical proofs, though, can be flawed, and mathematical history has examples of proofs that were persuasive but ultimately invalid. For example, there are several claimed proofs of the four colour theorem that turned out to be false [44].

The aim of using formal logic to arrive at a proof is to eliminate the risk of flaws by expressing statements in a formal language and restricting reasoning to a limited number of formal logical steps. For example, instead of saying "it will either rain or be sunny tomorrow, but not both; it will be sunny tomorrow; therefore it will not rain tomorrow", one might say something like:

$$\begin{array}{l} P \oplus Q \\ Q \\ \therefore \neg P \end{array} \tag{1.1}$$

Here, P means "it will rain tomorrow", Q means "it will be sunny tomorrow", $P \oplus Q$ means "either P or Q, but not both", $\therefore$ means "therefore" and $\neg$ means "not". It should be apparent that if we replace the meaning of P and Q with anything else, and the premises remain true, that the conclusions also remain true. This represents a valid argument, then, that preserves the truth of its premises.

Each rule of inference used in formal logic represents a valid argument, and if we keep the set of such rules small, anyone should be able to go over them all and trust that any argument constructed with such rules must be valid. These rules are defined by their syntax, whereby simply having a collection of symbols in a particular form is sufficient to allow the rule to be carried out. While they are syntactical, though, each rule represents a real, semantic argument that can be accepted by the reader through an understanding of the meaning assigned to each symbol used.

By presenting an argument in such a form, it becomes very difficult to deny its truth, unless one can assert the falsehood of one of the premises. The argument is more certain than an informal proof, that often leaves things unsaid in context, relying on prior knowledge, and not being explicit about assumptions that may actually be necessary [55].

We can prove the soundness of formal mathematical systems against known sets of axioms in mathematics: for example, we can show that if the system used by theorem prover HOL Light[57] (HOL here means higher order logic, a system of formal logic used by the theorem prover that we will discuss shortly below) is inconsistent (meaning that it can produce a logical contradiction), then Zermelo–Fraenkel set theory with the axiom of choice (ZFC) must be also [55]. This means that anyone who can accept the axioms of ZFC – widely accepted in the mathematical community – should logically accept a proof written in HOL Light.

Two mathematicians might both accept the ZFC axioms but disagree over an informal written proof, due to differing levels of expertise on the subject matter, or differing acceptance of some unstated axiom used within it. However, they cannot reasonably doubt the same proof presented in a formal fashion, provably sound with respect to ZFC, without doubting the explicit premises. This is what we mean when we talk

about increasing certainty.

Using a very small kernel of logical rules to write formal proofs can both be laborious and lead to proofs of thousands of lines that are nearly unreadable. In fact, one of the earliest attempts to formalise mathematics in the *Principia Mathematica* suffered from exactly this problem [165].

Computers can interpret lengthy formal proofs rather easily, by simply checking that the rules of inference – remember, these are rules of syntax, ideally suited for computerised checking – are being followed properly. Even the lengthiest formal proof can be confirmed by computer (if it has sufficient storage and memory), such as the formal sections of the original complete proof of the four colour theorem (while large portions were necessarily and laboriously formalised, it retained some pen-and-paper elements), but it may remain opaque to human mathematicians [5].

To make the process both easier and also more likely to result in readable proofs, we use computer software in the form of interactive theorem provers. These can collect proofs in lemmas, to make them easier to refer to later, and simplify the writing of a proof without compromising the strict adherence to the formal logical system being used, by aggregating the logical steps involved and thus, simplifying the presentation of the proof. Proofs become easier to write, more interpretable by humans and can bring insight in the same way as an informal proof. Such systems allowed Gonthier to finish a fully formal version of the four colour theorem that is more readable than the formal portions of the original proof [49].

There are several theorem proving systems in use today – we have mentioned HOL Light above, for example. Common contemporary theorem provers are Coq[19] and Isabelle[114]. Each has differences in its approach to logic, its syntax, and how proof structures are written within it.

Throughout our work, we use the Isabelle theorem prover, because of its extensive libraries and readable, natural proof-style. We go over the syntax of Isabelle at length in Section 2.2, but here we present our reasons for selecting it over the most likely alternative, Coq, before discussing its logical and historical foundations.

Coq is comparable to Isabelle, in many ways, with large libraries representing different areas of mathematics. The chief differences are the lack of support for a backward proof method (where one can state explicit goals as one proceeds through a proof, resulting in increased readability and a more natural approach to proof for those from a mathematical background) and that Coq uses intuitionist logic rather than propositional logic at its core, preventing the use of the axiom of excluded middle. Addi-

tionally, Coq's limited support for mathematical notation makes its proofs slightly less readable [169]. These properties made Isabelle our choice of theorem prover for this work.

Isabelle was developed from the Logic for Computable Functions (LCF) theorem prover developed in Stanford and Edinburgh by Milner [106]. It has as its core, a weak type theory (a system of classifying terms into types with operations associated with each) used as a meta-logic to implement other logical systems. The LCF approach has theorems as terms of a particular abstract type with operations corresponding to the rules of inference. Once proven, a proof (which may be lengthy and complicated) does not need to be retained in memory for the associated theorem to be trusted for later use. This makes working with lengthy proofs that build upon each other much more practical.

For our work in Isabelle, we used higher order logic (HOL) as our logical system, which underpins the majority of Isabelle projects. Higher order logic is a type-theory based system of logic, which is more fully described in Paulson's overview of it for Isabelle [116]. First order logic allows us to express quantifiers over individuals – for example, "$\forall n.\, P\, n$" means "$P\, n$ is true for all n". Second order logic extends those quantifiers over sets – for example, "$\forall n \in A$" means "for all n in set A". Higher order logics extends that scope arbitrarily to sets of sets, and so on.

An important notion in HOL is that we do only make conservative extensions to existing material by adding new definitions rather than new axioms. In this way, inconsistency cannot be introduced. In Isabelle, there is a concept called "locales" (discussed in more detail in Section 2.2), by which new axioms can be introduced, but these axioms are restricted to the locale: thus, any potential inconsistency is likewise restricted.

There are extensive libraries for Isabelle/HOL covering a wide variety of mathematical fields, which was our primary reason for picking this tool. Secondarily, Isabelle's declarative style of proof leads to much more readable proofs: it builds forwards towards the proof's goal rather than backward from it. This style is much more natural to read and more typical of the informal mathematical process.

Additionally, Isabelle has a proven track record in formal verification of software and mathematical theorems. For example, the operating system kernel for seL4 was formally verified using Isabelle, and a number of bugs uncovered via this process [78]. This demonstrates that theorem proving techniques (using Isabelle) can find problems even in complicated systems and verify when they have been fixed.

1.2 Reinforcement learning and deep learning – a brief overview

Reinforcement learning and deep learning are two approaches to artificial intelligence that have enjoyed substantial success in recent times as public awareness of artificial intelligence has grown [40]. Deep learning, especially, is a technique at the heart of much of what the public recognises as artificial intelligence. However, reinforcement learning, too, is increasingly at the forefront of public awareness of artificial intelligence with technologies like AlphaGo, the system that famously was able to outplay the world's best Go player [140, 141]. AlphaGo combines deep learning and reinforcement learning techniques (along with symbolic learning techniques), as most reinforcement learning systems that deal with large state spaces do, but reinforcement learning is key to its performance. As these two methods are key to this work, and as both are so critical in the modern world, a brief overview of both is needed.

Deep learning began with the development of the artificial neuron in 1943 by McCulloch and Pitts [104]. The artificial neuron was later implemented as the perceptron by Rosenblatt[129], and the artificial neuron, with modifications, is at the core of deep learning and artificial neural networks today.

The perceptron is an algorithm that takes in numerical inputs, weighting each in a sum, then outputs the total through an "activation function" (originally, the Heaviside step function which returns "1" for positive inputs and "0" for all others). This final output is taken as the result of the algorithm. For example, if a perceptron is attempting to classify toddlers versus dogs, an output of "0" might represent a toddler, and of "1", a dog.

For the toddlers versus dogs example, the perceptron might have inputs of an entity's size, speed, volume of body hair, and the number of words it speaks per day. Each input is weighted by some value – from a human perspective, we know that if any words at all are spoken, chances are fairly good one is dealing with a toddler instead of a dog, so the weight associated with that input, given our prior knowledge, might be a large negative value, forcing the output to be negative. However, a perceptron has no knowledge, in the human sense, that dogs don't speak: it has no information about the world, except as represented by the weights it associates with each input.

So how does the perceptron learn and improve its ability to classify toddlers versus dogs? It is supplied with training data; a set of inputs, each associated with the correct output. For each piece of training data, it carries out its calculation. If the result is

incorrect, the weights for each piece of input data are slightly adjusted to correct for the error. Repeat this process many times, with many pieces of training data, and its ability to use its inputs to classify toddlers versus dogs will improve, assuming the training data is sufficiently general to represent the underlying problem.

There are many statistical means of attempting classification of data, and deep learning can be seen simply as one of these. However, its ability to "learn" and improve its own performance, and its ability to succeed with general types of data with no prior estimates of distributions, means that it has stood out.

In fact, the perceptron convergence theorem mathematically proves that if a set of data is linearly separable (in our example, if all toddlers spoke and no dogs did), a perceptron is guaranteed to learn how to correctly classify them in some finite time given sufficent training data [110].

However, most classification problems are substantially more complicated than linearly separable pieces of data. The perceptron cannot help with these; at least not by itself. Once combined with other artifial neurons, though, each taking in inputs, weighting and summing them, producing an output, then passing that on to some other neuron in a new layer, we have an artificial neural network. If that network has layers sandwiched between the input layer and output layer then we have deep learning.

We know that such a network can in theory accurately approximate many more complicated functions thanks to the universal approximation theorem [68]. But there is no general mathematical proof that a neural network is guaranteed to converge to an approximation for a given function through its learning process. Experimentation shows that under a wide variety of circumstances, neural networks can learn to approximate a wide variety of arbitrary functions. The mathematics of deep learning are not well-understood in the sense that we cannot generally mathematically verify the behaviour of a deep learning process except by running it. From the outside, they are commonly thought of as black boxes: inputs are passed in one end, and outputs come out of the other, but the interior is difficult to mathematically grasp.

There are issues with the learning process carried out by deep learning networks. In particular, they can take a substantial length of time to learn, especially when dealing with a complicated process. This can require very large volumes of training data, something that is not always practical to collect. Whilst learning, errors are likely, and if we are dealing with some safety-critical process, that can be dangerous. Deciding when such a system has learned enough to be safe is not a trivial task. The training data used to train a deep learning network might not be a good representation of the

underlying problem and this can be difficult to detect. This can lead to unanticipated problems with fairness of outcomes that rely upon the network [105].

Coupled with this are issues of reproducibility in deep learning research: as discussed above, results are typically shown by experiment rather than being proven, and these experiments may not be easily reproduced. Missing or incomplete source code, training data (which may not be shareable for privacy or financial reasons), unreported hyperparameter tuning and the randomness of what can be a random process all contribute to difficulties in reproducing results [97]. If an experiment cannot be reproduced, trust in the veracity of the reported results is inevitably weakened. This lack of trust can delay life-saving development in critical areas, such as medicine [54].

Part of the solution here, of course, is transparency in publishing results: making training data, source code, and hyperparameters available. But another component to the solution could be to introduce mathematical proof of new developments, ideally formally verified.

Moving on to reinforcement learning, this is a set of related approaches to problem solving whereby a reward is provided when negotiating a particular environment, and the goal is to maximise this reward by learning the best choice of action to take in each state – the optimal policy [147].

In reinforcement learning, an agent takes actions in an environment that it may be able to perceive only partially. These actions cause the agent to transition into new states within the environment, and to receive numerically valued rewards, as illustrated in Figure 1.1. The goal of the agent is to maximise the reward received over the total time the agent runs – not just at its current state. Many problems can be framed in this way, from game-playing [140, 141, 139, 90], through finance [9, 81, 94], to robot control systems [144, 79, 27].

The roots of the reinforcement learning approach were laid by approaches to optimal control theory and dynamic programming from the 1950s, in particular by Bellman[17], as discussed in Chapter 2. Here, though, there was no concept of *learning* the best policy for negotiating an environment, but simply proving that an optimal policy existed and computing it.

The idea of learning such an optimal policy by interacting with the environment via a process of trial and error originates from psychology and animal psychology. The notion first found expression in Edward Thorndike's "Law of Effect" [151], a means of stating that discomfort discouraged associated animal behaviours, while rewards reinforced them. This notion is a precursor to the idea of reinforcement learning as a

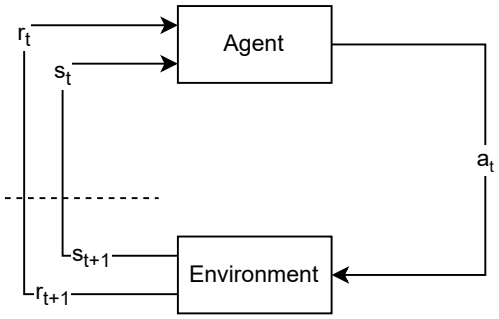

Figure 1.1: A simple illustration of the reinforcement learning process. Here, a_t is the action taken, r_t is the reward earned, and s_t is the state of the environment at time t.

mathematical process.

An important difference between reinforcement learning and deep learning is that normally reinforcement learning requires no training data. The agent learns solely by exploring the environment and seeking to maximise its reward. Thus, it is suited for particular kinds of learning, where outcomes can be quantified and more positive outcomes associated with larger quantities.

Another important distinction, is that for simpler reinforcement learning systems, the mathematics are well-understood and informally proven. There is ample mathematical literature on the fundamentals of reinforcement learning, as we will discuss in Chapter 2 and 3. However, the mathematical background again is often ignored in reinforcement learning research in favour of experimentation to verify results.

The reinforcement learning process is fundamentally stochastic in several ways. Firstly, the agent's exploration of the environment is typically done with an element of random choice. Secondly, an action transitions the agent into a state determined probabilistically. These two factors contribute to a difficulty in reproducing experiments with similar challenges as for deep learning – lack of source code and hyperparameter information [60].

By formalising the mathematics underlying reinforcement learning, modelling reinforcement learning systems could be made easier, and thus the mathematical underpinnings of some modification or algorithm more easily evaluated and used to build trust in the results. Ideally, this would lead to a greater confidence in novel research backed (in whole or in part) by mathematical proof.

It's important to recognise that the two approaches to learning can be combined. One such common approach recently is deep reinforcement learning, where a deep

learning network estimates the rewards, typically where the state space is very large [149]. The deep learning process can introduce more uncertainty given its black box nature, but the power of such systems to learn how to maximise reward in more complicated environments has been established.

1.3 Formal verification of modern AI

In this section, we will discuss work carried out in recent years on formal verification of deep and reinforcement learning. Little of this work has been done using formal theorem proving: we will discuss existing work in this specific area that is directly related to our own work in Chapters 2, and 3 and work that is unrelated to our own directly below in section 1.3.3.

We will discuss formal methods applied to deep learning and reinforcement learning separately. While reinforcement learning methods involving a large state-space often use deep learning methods to assist their understanding of similarities within the state-space, the fundamental learning process remains quite distinct from that of a deep learning process. In both cases, formal verification is very rarely general, typically being intended to work with a subset of methods or domains.

Moving to deep learning first, it is important first to clarify what is being formally verified. Typically, a deep learning process learns using a large volume of training data, and is used after this training is complete. Most formal verification methods focus on verifying a trained neural network and its behaviour.

Beginning with those methods that verify trained neural networks, we must ask ourselves what verification of a neural network means. Work by Kurd and Kelly put forward a large set of goals for verification [85] of neural networks. The majority of verification work on neural networks focuses on goal G4: establishing that the neural network can robustly handle faulty input. This can be interpreted as input that is not designed to be handled by the neural network, or input that is in some ways distorted from the input that the neural network is designed to handle. Some work also focuses on goal G5: ensuring that the output of the neural network is not hazardous. This might be interpreted as ensuring output falls within a guaranteed safe or correct range.

1.3.1 Adversarial inputs and robustness

The idea of faulty input from Kurd and Kelly was shown to be important in the work of Szegedy et al [148]. They were able to formally show that adversarial examples exist near successfully handled inputs for a neural network. This laid the groundwork for many of the attempts to use these adversarial examples to train a neural network to avoid input problems and thus verify property G4 from Kurd and Kelly's work.

This discovery was re-examined by Goodfellow, Shlens and Szegedy [50], who concluded that the existence of adversarial inputs near to properly classified inputs was inherent to the linear nature of neural networks. They also discussed the generalisation of these adversarial inputs, showing that the same adversarial input could be misclassified even by neural networks of quite different architectures and processes.

A lot of other formal verification work focuses on finding adversarial examples of input. The network can then be trained using these examples until it can handle them correctly. Bastani shows a method for finding the nearest adversarial example [14] to a successfully handled input. However, in common with many formal verification methods applied to neural networks, this is limited to neural networks with particular characteristics: in this case, it must use the rectified linear unit (ReLU) activation function.

In contrast, Huang et al. [70] focuses on proving that no adversarial example exists within a certain neighbourhood of a given input. This method is flexible with respect to the activation method used, using the SMT solver z3 [35] to evaluate the network. Unfortunately, the approach can have issues with complicated inputs.

Wang et al. [156] likewise provides work that focuses on checking no adversarial example inputs exist within a certain L norm of a given correctly handled input. This improved on existing work in this area using interval analysis and linear relaxation. This combination of the two approaches is claimed to correct for over-estimation of both lower and upper bounds of the predicted intervals. While more efficient, this method is again limited to neural networks with ReLU activation functions.

Translating a neural network safety query to a mixed integer linear program (MILP) [95] is suggested by Tjeng et al. [152]. This method then uses MILP verification to find adversarial examples to a given input. This is limited to a ReLU activation neural network.

Cheng et al. [29] again use MILP programming to establish the resilience of a neural network to adversarial inputs. They use a different approach for ReLU acti-

vation functions and approximate $\tan^{-1}$ activation functions using signal processing techniques. The scalability of the approach seems to be an issue, with experiments demonstrating results only on small networks.

Another method to find adversarial examples was proposed by Weng et al. [160]. They claim that their method finds an approximate lower bound to the closest adversary to a given input much more quickly – the paper claims 33-14,000 times faster – than many previous attempts, with the caveat that the lower bound is inexact. They also establish the theoretical best complexity of any algorithm that can achieve this, assuming $P \neq NP$.

Bunel et al. [26] re-frame the problem of formal verification using adversarial examples as a special case of branch and bound optimisation methods [24]. By honing these methods, they claim to improve on the speed of existing methods by two orders of magnitude. Their method is reasonably robust to changes in the neural network itself, and so can handle large, more realistic, networks more easily than some other methods.

Gher et al. [46] present a method of evaluating the robustness of neural networks that circumvents scalability issues at the cost that it finds false positives when verifying. The method, called AI^2 does this by making domain-dependent approximations of the neural network's computations. These approximations are carried forward through further approximations of the network's layers to the output layer. Performance is good on even relatively large neural networks, at the cost of incompleteness.

Balunovic and Vechev's work [12] finds the nearest adversarial example to an input, and uses that to train a neural network to establish a convex space around the input in which no further adversary can be found. Once more, this method is restricted to those that use ReLU activation functions. Experiments using the CIFAR-10 image classification dataset [83] show promising results.

1.3.2 Safe outputs

A different approach is taken by those looking to verify that the output of a neural network is non-hazardous. Pullina and Tuchella [123] produced an early work in this area using SMT solvers (in the experiments, the HySAT solver [43]) with neural networks. A multi-layer perceptron (MLP) network [121] – a common variety of neural network – is abstracted to a combination of linear arithmetic constraints. If the abstraction returns safe results, the neural network is guaranteed to do so; if not, the counter-example

can be used to correct the network automatically. This work is challenged by problems of scaleability, being rather restricted in the size of neural network it can verify in a practical timescale.

They followed up this work by experimenting with different SMT solvers to see which can most efficiently check an MLP [124]. SMT solvers were evaluated primarily by the time taken to verify if a given MLP satisfied a set of constraints. Differences both in time taken and in the satisfiability of MLPs were noted.

Katz et al. [74] produced a specialised SMT solver called Reluplex. This abstracts a feedforward neural network, restricted to those with ReLU activation functions, as the name implies. The simplex algorithm [107] is the basis for the method, which rewrites the verification test as a set of SMT constraints. The method is applied to some rather small, but realistic, networks with positive results. The method can be slow, with some tests taking up to two days on larger networks.

The scaleability problem was addressed in Katz et al.'s follow up work [75]. Building on Reluplex [74], this presents a new framework, Marabou, for checking neural networks safety properties, again by evaluating them via constraint satisfaction. It expands on the variety of neural networks that can be verified, beyond the relatively simple ReLU activations that Reluplex can verify. The performance is also optimised, reducing the lengthy time taken to verify large networks somewhat.

Ehlers' work [39] attempts to provide scale that previous work using SMT solvers could not achieve. It uses approximation of the deep learning process to achieve this, which leaves open the possibility that such an approximation may inaccurately reflect the true system, but demonstrates good results with more practical deep learning processes than the work it builds upon.

Dutta et al. [37] produced work that, like Reluplex, works by establishing the range that the output of a ReLU activated neural network can fall into. Instead of taking an SMT based approach, the Sherlock tool evaluates the problem as a Mixed Integer Linear Program, or MILP [95]. Local search techniques are used to optimise performance. The resultant tool, Sherlock, is shown to outperform both Reluplex [74] and a more naïve MILP approach.

Other work on formal verification of deep learning systems provide verifiability in limited areas, such as a robot equipped with LIDAR or computer vision networks. The LIDAR work here again proceeds by abstracting the deep learning network, leaving open the same potential issues as Ehlers's work [145]. The computer vision work has performance issues, and even on a relatively simple dataset (the MNIST dataset

used to train neural networks to recognise numbers), the process could timeout before completing [80]. However, it was able to verify that the network tested was not robust to transformations of the images it attempted to classify.

Singh et al. [143] argue for analysing the neurons of neural networks with ReLU activations jointly by later, rather than separately, using a method they call k-ReLU. This method can be applied in combination with other verification methods, and the argument is that by considering a layer at a time, there is less redundancy in the process.

Corsi et al. [31] produced work that somewhat crosses over between verification of deep learning and reinforcement learning, by focusing the neural networks used in deep reinforcement learning. They introduce a measure for safety-critical neural network systems, the *violation rate*. This measures the proportion of the input domain that for a given neural network function results in a breach of some safety property. This is calculated using the ProVe algorithm, which iteratively divides the input area and uses these divisions to over-estimate the violation rate. If ProVe results in a violation rate of 0, then the neural network is formally verified with respect to the safety property.

1.3.3 Theorem proving based formal verification

Bentkamp et al. [18] present a formalisation of the expressiveness of neural networks. They formalise a theory of tensors and explain the greater expressive capacity of deep learning networks compared to shallow, single layer networks.

There has also been work by Brucker and Stell [25] on directly verifying feed-forward neural networks in Isabelle/HOL. Their paper discusses a process to embed a neural network into a formalisation that can then have properties proved against it. Their method is adapted to permit imports of neural networks written using Tensorflow [111], a development framework widely used with neural network models. The work is illustrated with a small example network analysing pixelated numerical digits.

Aleksandrov and Völlinger [4] work in the theorem prover Coq to formalise affine activation functions for neural networks. The formalisation implements a ReLU function using the formalisation and proving properties against it.

1.3.4 Formal verification of reinforcement learning

Works discussing formal verification of reinforcement learning tend to focus on specific domains to a greater extent than works on formal verification of neural networks. For example, Pore et al. [122] present work focusing specifically on deep reinforce-

ment learning applied to surgical robots performing tissue retraction. It uses reward shaping in combination with a formally verified estimate of the likelihood of safety properties being violated after a given amount of training.

Also in a specific domain, Corsi et al. in a 2020 paper [32] use formal verification over a specific type of deep reinforcement learning network used in generating trajectories, specifically those for the Franka Emika "Panda" robotic arm. The primary innovation is the use of adaptive discounting factor in the reinforcement learning process, using earlier work by Wang et al. [156] to formally prove the safety of the process.

Work by Fulton provides verification of agent behaviour for reinforcement learning in cyber physical systems [45]. It does this primarily by run-time monitoring of a system, fitting its current state to a differential dynamic logic formula and verifying that the formula explains it. It also formally verifies a policy to ensure it cannot breach a given safety condition. However, this requires the environment be modelled accurately via differential dynamic logic, and this may be a complicated and difficult task as well as running counter to the notion that a reinforcement learning agent may be exploring an unknown environment. For simple environments, though, this work can guarantee a safe policy and monitor a learning process to revert its agent to the safe policy when conditions are breached.

1.4 Goal and contributions of the work

The primary goal of this work is to demonstrate that formal mathematics, using mechanical theorem proving tools, can be of benefit, in terms of building confidence in results, to reinforcement learning and deep learning research. We present work that builds to this goal in two main ways:

1. Formal verification of the mathematics underlying some artificial intelligence method has two main benefits. Firstly, it increases the certainty of the theorems on which it is founded, and secondly, it develops a formal framework for those theorems that can be built upon by later work. The main problem with this approach is that with artificial intelligence methods whose mathematical foundations remain undeveloped, even informally, this verification requires new mathematics as well as new formalisation. It is more achievable to focus on an approach with a well-understood mathematics – in our case, reinforcement

learning.

To this end, we formalise and verify the fundamental mathematics of discrete reinforcement learning, and present a formal model of Q learning, a significant reinforcement learning algorithm. We focus specifically on extending an existing formalisation of Markov decision processes, a mathematical model of discrete processes, and a formalisation of Dvoretzky's stochastic approximation theorem, a result in probability theory that is important to proving that discrete reinforcement learning is able to find the optimal policy for a Markov decision process. We demonstrate the potential to build on this work by formalising a simple game and showing that it can be interpreted as a MDP and thus inherit the properties we have proven against them.

2. We implement a loss function that trains a neural network to avoid breaching arbitrary linear temporal logic constraints, using code generation from formal specifications after formally proving their correctness. This enables a neural network to learn not only from imitation of training data but via explicit rules as well. With our method, the code generated matches the formal specification and thus carries the certainty of its proofs into the implementation [52]. This greatly reduces the risk of human error in code, as well as providing mathematical certainty in the loss function's behaviour, modulo practical issues such as floating point approximations of real numbers.

In the process of carrying out this work, we contributed novel research in several areas underpinning our two main objectives. Further details and discussion on these contributions are detailed in the conclusions of Chapters 2, 3 and 4. All our contributions are summarised in our final conclusion in Chapter 5.

The foundation laid down can be built upon by later work, and provides tools for researchers to extend the formalisation to new methods.

The theory files associated with the work described in this thesis can be downloaded from the AI Modelling Lab website.[1]

1.5 Organisation

The two main methods of demonstrating our goal in this work are:

[1] https://aiml.inf.ed.ac.uk/wp-content/uploads/2023/02/PhD_isabelle_theories.zip

1. Formalising the fundamental mathematics of reinforcement learning.

2. Formalising a logic constraint based loss function for deep learning and using code generation to implement it.

For the first method, we focus our work on formal verification of the mathematics of discrete reinforcement learning in Chapters 2 and 3. Our reasons for choosing this particular set of artificial intelligence methods is that the mathematics underpinning them and demonstrating their properties is well-understood: we can formalise existing pen-and-paper proofs [126, 7, 36]. Our focus here is on establishing certainty in the idea that a reinforcement learning agent is guaranteed to find the best set of actions to successfully complete some task.

To this end, we begin by formalising a set of mathematical models known as Markov Decision Processes with rewards in Chapter 2. These are the environments that discrete reinforcement learning agents typically act in. Our primary focus here is in showing that a "best choice of action" (an optimal policy) exists, and that it is possible to algorithmically find it with perfect knowledge of the environment.

In Chapter 3, we focus on the reinforcement learning agent itself, beginning by modelling the algorithm it uses to try to find the optimal policy, and then proving results in probability theory that are necessary to show that this algorithm works – in particular, the Dvoretzky stochastic approximation theorem.

We present our work on the second method in Chapter 4. We formalise an adapted loss function for a particular, path-learning, network and prove that its mathematical underpinning is correct, before implementing its formal specification faithfully via code generation and demonstrating via experimentation that the code successfully produces the expected results.

As each chapter represents a distinct body of work, we will discuss existing work related to each within the chapter itself, ending with a final conclusion in Chapter 5.

Chapter 2

Markov Decision Processes

2.1 Introduction

Mathematical models of real-life problems are frequently useful in providing insights into developing strategies to deal with those problems. Markov Decision Processes (MDPs) are a mathematical model of environments in which an agent can act over a discrete set of states (each representing a different set of conditions in the environment) by taking particular actions in discrete time [164]. An agent taking an action a in a state s probabilistically transitions into a subsequent state s'. The distribution of subsequent states depends only on the initial state and the action taken..

Importantly, an MDP has the following Markov property: the result of moving between time steps t and $t+1$ depends only on the state and actions taken at time step t; the history of the agent and the MDP before that is irrelevant. Thus, the MDP is memoryless.

Taking a particular action and obtaining a result from it gives a reward to the agent. In general, the aim of modelling an environment in this way is to suppose that by optimising the reward gained, we optimise an agent's behaviour to some preferred end. An agent negotiating the MDP model behaves optimally if it acts in such a way as to maximise its reward over time.

MDPs were developed from optimisation theory [17] and have applications in many fields, including medical, financial and logistical [3, 15, 150, 163]. In more recent years, they have frequently been used as environments over which a reinforcement learning agent or other autonomous agent can act [138, 158]. Many reinforcement learning approaches over discrete states and times use MDPs (such as Q learning or TD-λ learning which we will discuss in Chapter 3), and having a formal model with

rigorously proven properties is the first step to any formalisation of these methods.

Our longer-term intention, as discussed in Chapter 1, is for our framework to be usable by others to formalise work in this area. If researchers can readily relate to our models and formalise their ideas, they should be able to rigorously verify properties of their algorithms, more easily finding greater certainty in their results. In order to achieve this, we keep our formalisation rooted in mathematics that will be familiar to those exposed to MDPs and reinforcement learning from a machine learning perspective.

We begin in Section 2.2 by going over the theorem proving tool, Isabelle, using higher order logic. We discuss syntax and structure of proofs and definitions, and this material may be useful to refer to whilst reading the rest of this thesis.

In Isabelle, there is an existing mechanisation of MDPs [64] and their precursors, Markov chains. A Markov chain is a simpler process where no choice of action is permitted and instead agents simply transition from one state to another probabilistically [109]). This existing mechanisation is anchored in a monadic, category-theoretic approach. In the current work though, in order to achieve the accessibility mentioned above, we provide an alternative formalisation based on linear algebraic concepts that is more typical of mathematical writings about them. We will discuss this further in Section 2.3, where we briefly examine the mathematical background to MDPs and review the existing formalisation and why it is not suitable for our goals.

In Section 2.4, we introduce the specifics of our model and our preliminary proofs based on it. Our aim here is to demonstrate that this model meets the mathematical characteristics of an MDP, and we do so by deriving the Bellman equation [13].

In Section 2.5, we discuss a pen-and-paper proof of the existence of an optimal choice of actions for an agent (a *policy*) on any MDP and its mechanisation. We discuss some specific issues we had finding appropriate type representation in Isabelle and how we resolved these difficulties. Finally, we explain how we built our formal proof of the existence of an optimal policy on our model of MDPs. In doing so, we discovered an error in the pen-and-paper proof we were using as our basis and we disclose how we addressed this in the context of our finite MDPs.

In Section 2.6, we look at value and policy iterations. These are algorithms intended to provide a means to compute the optimal policy (or an arbitrarily close approximation) in finite time. We examine the steps for each of the algorithms and then formally prove that both work as intended.

We conclude with a summary of the work done and a discussion of some of the

issues that arose.

2.2 Isabelle/HOL

We now briefly review some aspects of Isabelle/HOL, a higher-order-logic proof assistant, [108], which we use throughout this work to formalise our models and prove their properties. Mathematical theories written in Isabelle are a collection of formal definitions of various kinds (algebraic objects, types, functions, etc.), and theorems that prove properties against them. We say "formal" here because formal statements made using the language can have their truth value determined (when possible) via deterministic, formal rules of higher order logic, as discussed in Section 1.1.

We write our proofs in the structured proof language Isar, which depicts proofs in human-readable form and where steps in a proof typically follow those that would be used in a naturally written mathematical proof [161].

```
lemma policies_not_empty:
  shows "policies ≠ {}"
proof -
  have "∧s. ∃a. a ∈ K s"
    using K_f
    by blast
  then have "(λs'. (SOME a. a ∈ K s')) ∈ policies"
    by (auto intro: someI_ex simp add: policies_def)
  then show ?thesis
    by auto
qed
```

Listing 2.1: An example of a proof written in Isar

An example of an Isar proof may be found in Listing 2.1, wherein we prove that the set of policies on a valid Markov Decision Process is not empty. The details of the proof are not important and the example is simply meant to illustrate some of the basic keywords and structure of Isar; more on the meaning can be found in Chapter 2. We first state the goal that we are trying to prove after the `lemma` keyword, labeling the lemma as `policies_not_empty`. Our statement here is to show that `policies` (a set we have defined earlier) does not equal the empty set `{}`.

We begin a `proof` block, first establishing the fact (via the keyword `have`) that all states have at least one choice of valid action. `∧s.` may be taken to understand "for a fixed and arbitrary element `s`". `s`, in the context of the statement that follows, is

understood to belong to the type of states: the `K` function found in the same statement returns the set of valid actions from a state-typed parameter. So this first statement can be understood as "for a fixed and arbitrary state, there always exists an action in the set of valid actions for that state". We prove this using an existing assumption named `K_f` and finishing the proof of this statement by means of one of Isabelle's automatic methods called `blast` [115] – the `using` keyword introduces the facts we will need to prove the statement, and the `by` keyword tells Isabelle what method to follow to establish the proof.

The next statement begins `then`, meaning it inherits the immediately prior proven statement as a fact to be used. We assert in this statement that a function we define using Isabelle/HOL's Hilbert choice operator `SOME` (which selects (using `K`) an arbitrary valid action for each state), will be a valid policy. Instead of the `using` statement from the previous line, we introduce our needed facts `someI_ex` and `policies_def` with the method that we are asking Isabelle to use, `auto`.

The automatic provers that Isabelle uses to complete a proof goal, of which `blast` and `auto` are examples, use a variety of methods to try to complete the proof given the facts present. Each method results in a proof that follows the syntactic rules of the formal logic system being used (higher order logic in this instance) and can make use of previously proven facts. In this way, one can be sure that anything proved by one of them is as certain as anything proved via formal logic.

Lastly, we show that the fact that this function is in the set of policies means that this set must therefore be non-empty. The keyword `then`, as before, means that this statement inherits the fact just established, but instead of using `have` we use `show` to indicate that we believe this statement finishes the proof we are working on. The keyword `?thesis` is used by Isabelle to refer back to the lemma's original proof goal. The `auto` method finishes the proof.

Isabelle/HOL is a typed logic that supports type classes [162]. A type class has assertions of properties applied against type variables to constrain and interpret them as abstract algebraic concepts: for example, as a vector or metric space. These are then known as *sort* constraints. In addition, you may define functions that must be present for a given type class. If we assert that a type variable fulfils a sort constraint as part of the definition of a theorem, then that theorem is true for any concrete type that meets this constraint.

A type can be shown to be an *instance* of a type class, by defining all required functions and proving that all asserted properties are met. This means, for instance,

that one can assert that `int`, the type of the integers, forms a group under a given definition for the group operator by proving that it is an instance of the `group` type class. Then theorems which are true for all groups become available for the integers under this interpretation.

When defining a constant or function, it must be assigned a type ("`t::`τ" states that `t` is of type τ). Function types are written using "⇒", from the type on the left-hand of the arrow to the type on the right-hand. Type variables are written with a prefixed apostrophe, like "`'s`". Sort constraints can be specified against types using suffixed "`::`" notation, followed by the required constraints. We provide some quick illustrative examples:

1. `test_sequence :: "nat ⇒ real"` tells us that `test_sequence` is a function from the `nat` type, the natural numbers including zero, to the real numbers – a sequence of reals, in other words.

2. `location :: "real ⇒ real ⇒ 'l"` tells us that `location` is a function of two real numbers which produces a result of type variable `'l`.

3. `state_space :: "'s::{finite} set"` tells us that `state_space` is a set of values of type `'s`, which we assert is finite using the Isabelle sort constraint `finite`.

Within Isabelle, one can create a context within whose scope certain assumptions are held and certain constants have a declared type and fixed value. This is called a *locale* [73]. All theorems proven within the locale depend upon and have access to its assumptions and declared constants. As with any set of axioms, care must be taken to ensure consistency in the locale's assumptions; however, the encapsulation provided by a locale ensures that an inconsistency can never propagate to Isabelle's top level: as discussed in Section 1.1, the axioms of the locale are restricted to that locale, so if there are axioms which lead to an absurdity, it is also restricted to that locale. An example locale definition is as follows:

```
locale Sample_Locale =
  fixes sequence :: "nat ⇒ real"
    and ϕ :: real
  assumes sample_assumption: "sequence 0 = ϕ"
```

Listing 2.2: An example locale

This locale fixes two parameters: `sequence` is a function from the natural numbers to the reals and ϕ is a real number. It then specifies an assumption that the value of the sequence at index 0 is ϕ by asserting `sequence 0` $= \phi$.

We rely throughout this work on Isabelle's extensive libraries of existing work. In particular, our formalisation of Markov Decision Processes in this chapter makes use of several existing theories from the Isabelle standard library extended by the Archive of Formal Proof (AFP) [99], on probability, linear algebra and analysis, which we introduce and discuss as they arise in the rest of that chapter. Our work on Dvoretzky's stochastic approximation theorem in Chapter 3 likewise uses many results in measure theory, probability theory, and functional analysis.

It is possible to generate computable code from the formal specification of functions in Isabelle into Standard ML (PolyML), OCaml, Haskell and Scala [52]. This mechanism provides a rigorous link between Isabelle concepts and their automatically-generated counterparts, whose computational behaviour (modulo implementational details such as translating reals to floats) can then be expected to respect the properties that were formally proven as theorems in Isabelle. This is a vital component of our work in Chapter 4 and will be discussed further in Section 4.4.

2.3 Background

We introduce Markov decision processes by their mathematical description and then go on to give a brief discussion of the existing formalisation of MDPs within Isabelle.

2.3.1 Markov decision processes

A finite Markov decision process is a tuple

$$(S, A, P, R)$$

where:

- S is a finite set of states representing differing conditions in the environment. States are "terminal" if no actions are possible from them.
- $A(s)$ is a function on the states that returns a finite set of possible actions that can be taken from state s.

- $P(s,a,s')$ is a function that gives the probability of an agent transitioning from state s into state s' if it performs action a. Each choice of action gives a probability distribution over the states, so we have:

$$\forall s,a.\ \left(\sum_{s'\in S} P(s,a,s')\right) = 1$$

$$\forall s,a,s'.\ P(s,a,s') \geq 0$$

- $R(s,s',a)$ is a function that gives the reward for transitioning from state s to state s' by performing action a. Rewards are real valued.

The goal for an agent navigating an MDP is to maximise the total reward it earns. For a general MDP, there is no restriction on $R(s,s',a)$, so we cannot guarantee that the sum of rewards received over time will converge. If there is no cap on the number of actions an agent may perform, this is a sum of an infinite sequence of rewards and could be divergent. Throughout this chapter it is our assumption that there is no such finite cap, and this assumption is commonly called an infinite horizon [126].

Consequently, it is common to introduce a discounting factor γ, where $0 \leq \gamma \leq 1$ (note that if $\gamma = 1$ this is the same as having no discounting factor) [126]. Its effect is that at time step n, any rewards earned from time step $n+1$ onwards are multiplied by γ before evaluation, and this discounting is compounded for future time steps. Its purpose is to enable us to estimate total future rewards by introducing a mechanism by which we can guarantee that the sum of rewards will converge for any MDP that offers bounded rewards. We can interpret it as meaning that we value distant future rewards as being worth less to us than immediate rewards.

If we index the state of an agent at time step t as s_t, and its actions at t as a_t, the total discounted value of the agent performing a choice of actions from an initial state s_0 would therefore be:

$$R(s_0,s_1,a_0) + \gamma R(s_1,s_2,a_1) +$$
$$\ldots + \gamma^t R(s_t,s_{t+1},a_t) + \ldots$$

Normally when we consider an agent navigating an MDP, we view it as following a particular policy π that chooses the action for the agent to perform in each state. Thus, $\pi(s)$ is a function on the states where $\forall s.\ \pi(s) \in A(s)$.

When we want to evaluate potential policies to estimate the rewards an agent might earn from pursuing them, we use the concept of value, which is the expected rewards accounting for the discount. Thus, $V(s,\pi)$ is a function of states and policies that gives us the total expected discounted future reward for an agent pursuing policy π from state s – we call this the V-value function.

Similarly, $Q(s,a,\pi)$ is the total discounted expected future reward for an agent choosing action a from state s and subsequently pursuing policy π – the Q-value is distinct from the V-value by its allowance that the initial action taken may differ from $\pi(s)$. Thus, the V-value is a special case of the Q-value.

Obviously, these can only be evaluated when we can be sure that the sum of rewards will converge, so this condition is typically met by having $\gamma < 1$. It is possible as well to ensure convergence by having a finite horizon limiting the number of steps an agent can perform, or by designing an MDP that inevitably ends in some terminating state.

The value function is typically depicted in MDP literature using the Bellman equation [13], which evaluates it recursively (recall that V is a special case of Q where the action is chosen by the policy):

$$Q(s,a,\pi) = \sum_{s'} P(s,a,s') \left[R(s,a,s') + \gamma V(s',\pi) \right] \tag{2.1}$$

To summarise the equation, recall that $Q(s,a,\pi)$ is the total expected future reward, with discounting, for taking action a in state s and *subsequently* pursuing policy π. $P(s,a,s')$ is the probability of transitioning to state s' by taking action a from state s. $R(s,s',a)$ is the reward earned for transitioning from state s to s' by performing action a. The discount factor is given by γ and, finally, $V(s,\pi)$ is the total expected future discounted reward for pursuing policy π from state s, and is equal to $Q(s,\pi(s),\pi)$.

2.3.2 Hölzl's formalisation of MDPs

As previously mentioned, we are indebted to the work by Hölzl on Markov Decision Processes [64], which informed the current formalisation.

Hölzl's work was performed with an emphasis on proving properties regarding the traces that an agent might make in negotiating an MDP or that might be found using its purely stochastic precursor, the Markov chain [109]. These included properties such as reachability of given sets of states, hitting times and probabilities of such sets of states, and other properties that are illuminated using distributions of program traces. The idea of reward is not really considered, except as one attached to a particular probabilistic

program run across the MDP [64], which does not resemble the general reward that is typically considered with MDP optimisation problems.

Hölzl models finite MDPs using a type variable `'s` to represent the states of the MDP. He represents actions purely by their result, via a probability distribution over the states it might take an agent to. To achieve this, he uses the type `'s pmf` [67], which represents probability mass functions (PMFs) over the type variable `'s`. A PMF is a function over a countable set (in this instance, a set of states) that returns the probability of a transition to a given member of the set. Thus an action is defined here as the probability distribution over the transitions to new states that it might cause. Hölzl's formalisation is:

```
locale Finite_Markov_Decision_Process = Markov_Decision_Process
  K for K :: "'s ⇒ 's pmf set" + fixes S :: "'s set"
  assumes S_not_empty: "S ≠ {}"
  assumes S_finite: "finite S"
  assumes K_closed: "⋀s. s ∈ S ⟹ (⋃D∈K s. set_pmf D) ⊆ S"
  assumes K_finite: "⋀s. s ∈ S ⟹ finite (K s)"
```

Listing 2.3: Hölzl's `Finite_Markov_Decision_Process` locale

where the following applies:

- `set_pmf` here is a function that returns a set of all of the results of a probability mass function which have non-zero probability.
- Hölzl does not assume that the variable type `'s` is finite because he is inheriting part of the locale definition from another one where there are possibly infinite states. This means that the parameter `S`, assumed to be finite and not empty, is the set of *valid* states considered to be part of the MDP. The finiteness of this set defines the finiteness of the MDP.
- `K` is a function on states that returns a finite set of all valid actions that may be taken from the specified state. Note the assumptions that K returns a finite set only, and that any actions in `K s` (where `s ∈ S`) only lead to other states in `S`.

We also note that Hölzl uses a generalisation of the idea of policy called "configurations" [64]. These can model policies that vary over time in a way that is not required for our work: our primary goal is to provide a formalisation of MDPs suitable for modelling reward optimisation methods. Stationary policies, where the choice of

action does not vary over time, are sufficient for optimality as we shall prove in Section 2.5 and is proven informally in the literature [126]. Configurations can also model stationary policies as a special case, but the underlying definition remains unsuited for our model since the extra definition required for configurations is not needed for our approach and would only complicate our proofs.

Clearly, there is no general reward function or discounting factor defined as part of Hölzl's locale, as it is not concerned with questions of reward or optimisation (in the general case). Our work, by contrast, will model real-valued rewards (see the next section). This allows us to prove the necessary results in optimisation theory that lead into dynamic programming and reinforcement learning.

In addition, the use of the parameter `S` in the above locale to denote the set of valid states, which is then constrained via an explicit assumption of finiteness instead of using the Isabelle sort `finite` over the type, leads to clumsiness in further proofs. More specifically, since functions in Isabelle are total, we must then always specify the behaviour of any function over valid as well as invalid states and account for this in proofs.

Additionally, the locale makes extensive use of the Giry monad [48]. However, our position is that the model we are looking to build should be accessible, using the mathematics that is typically encountered in the literature. So instead of the Giry monad, our mechanisation uses transition matrices and stochastic matrices, which are by far the most common elements in texts on this material e.g. in Puterman's classic textbook [126].

This emphasis on linear algebra demands more preliminary work in finding suitable type representations for the matrices we will work with, and in establishing the matrix properties we will make use of, but the improved accessibility it offers by comparison to a Giry monadic approach is worthwhile.

Consequently, while using the ideas in Hölzl's formalisation as the inspiration for our own work, we decided to define our own model with some crucial, necessary differences to focus on the optimisation questions that interest us.

2.4 The fundamentals of our formalisation

In this section, we describe the basic concepts that underlie our approach and review their mechanisation in Isabelle. We begin by discussing the specific differences between our approach and that of Hölzl before introducing our "slice" based approach to

understanding agent paths through an MDP.

2.4.1 Formalising finite Markov Reward Processes

As stated, our model is built around a locale inspired by Hölzl's work, but with some important changes. Our definition is as follows:

```
locale Finite_Markov_Reward_Process =
  fixes K :: "'s::{finite_discrete_topology} ⇒ 's pmf set"
    and R :: "('s × 's × 's pmf) ⇒ real"
    and γ :: real
  assumes K_f: "∀s. finite' (K s)"
    and gamma_range:"0 ≤ γ ∧ γ ≤ 1"
```

Listing 2.4: The `Finite_Markov_Reward_Process` locale

where:

- Similarly to Hölzl's `Finite_Markov_Decision_Process` locale, we use the type variable `'s` used to represent states, although we assert it belongs to the `finite_discrete_topology` type class described below. This models the S from the informal definition of an MDP given in section 2.3.1.
- We also use the `K` function that returns a set of valid actions from a state, which again we assume is finite and non-empty for all `s` – the Isabelle `finite'` predicate asserts both finiteness and non-emptiness. This function modes the A function, and the action distributions themselves model the P function from the informal definition.
- `R` is a function on a 3-tuple of two states and an action, which returns a real valued reward, which is the R function of the informal definition.
- γ is a discounting factor such that `0 ≤ γ ≤ 1`.

When first building our formalisation, we tried to inherit directly from Hölzl's `Finite_Markov_Decision_Process` locale. However, as alluded to in the previous section, its use of `S` as a finite set to contain all valid states in the MDP, and the consequent theoretical presence of *invalid* states, greatly complicated the proof process. Every theorem required assumptions that any state was a member of `S`, and functions on states needed definition on invalid states as well as valid ones. To overcome these

issues, we asserted that our type variable belongs to the `finite` type class, circumventing the need to consider the possibility of invalid states while retaining the finiteness properties we require.

Subsequently, when we began working with bounded continuous functions over the state type (discussed in detail in sub-section 2.5.2), we needed to show that functions on states were continuous. As states are a finite, discrete type, the only way we can show functions on them are continuous is to equip them with the discrete topology sort, and later to assert they form a metric space in order to use the Banach fixed point theorem (see sub-section 2.5.9 for more detail).

It is possible in Isabelle to assert that a type `'t` belongs to multiple type classes. However, when proving that a type derived from a type variable is an instance of a type class, one cannot assert multiple type classes on the type variables; if that is necessary for the proof, it will fail. As we will need to prove that several types derived from our state type are Banach spaces, we combine the type classes `finite`, `discrete_topology` and `metric_space` into a single type class. We call this combined type class `finite_discrete_topology`:

```
class finite_discrete_topology
  = finite + discrete_topology + metric_space
```

Listing 2.5: The `finite_discrete_topology` type class

Now that we have defined the MDP with rewards, we need to build representations that match the discounted Q and V-values discussed previously. We formalise Q-values, because if we have the Q-value function for a policy π, we can always find the V-value function by assuming we choose action $\pi(s_0)$ in our initial state s_0.

Recall that Hölzl used configurations as a generalisation of policies. As we are not inheriting directly from his locale, we can simplify our definitions. We define `policies` as the set of all functions from the state to probability distributions on the state (recall from Section 2.3.2 that the latter is the type used for actions in the current context), where the function only returns a member of the set of valid actions on any state:

```
definition policies :: "('s ⇒ 's pmf) set" where
"policies = {p. (∀s. p s ∈ K s)}"
```

Listing 2.6: The `policies` definition

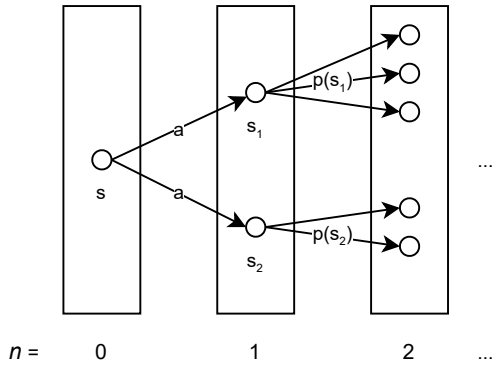

Figure 2.1: An illustration of the slice-based approach. Each rectangle, or slice, at time step n contains every possible state (e.g. s_1 and s_2 for slice 1) that an agent might be in by taking initial action a then following policy p. The possible transitions from one time step to another are indicated by the labelled arrows (e.g. $p(s_1)$, $p(s_2)$ between step 1 and 2), with multiple arrows from a single action indicating that subsequent states are stochastically determined, with several possibilities.

2.4.2 A slice-based approach to agent paths through an MDP

In order to model value, we need a way of showing the expectation of rewards assigned over time. To do this, we need to mechanise the probability of various paths being followed by an agent pursuing a policy over time. Hölzl's method was to build a probability measure over the paths an agent might take – his primary concern being reachability and other properties of the states the agent might hit.

We need more concrete definitions for our purposes, so we take a different approach. We model the paths of an agent over finite time steps, and then consider the limit of those paths as the number of time steps approaches infinity. We introduce *slices* as a snapshot of an agent's possible path through an MDP, each slice capturing the possibilities at a given time step. For an illustration of this approach, see Figure 2.1.

We begin by defining the basics, namely a collection of functions that all take four parameters: the time step `n` being considered, the policy `p` that an agent is following, the initial state `s` that the agent is in, and the action `a` chosen in the first time step. After action `a` is performed, the agent always chooses action `p s'` for all subsequent states `s'`.

The function `Qslice n p s a` returns the set of all the states that the agent might be in at time step n:

```
fun Qslice :: "nat ⇒ ('s ⇒ 's pmf) ⇒ 's ⇒ 's pmf ⇒ 's set"
where
  "Qslice 0 p s a = {s}"
| "Qslice (Suc 0) p s a = set_pmf a"
| "Qslice (Suc (Suc n)) p s a
    = (⋃s'∈Qslice (Suc n) p s a. set_pmf (p s'))"
```

Listing 2.7: The `Qslice` definition

This function proceeds by recursing downward from the time step parameter `n`. If `n` is 0, then the set of possible states only consists of the initial state `s`. If the time step is 1 (represented above by `Suc 0` where `Suc` is Isabelle's successor function on natural numbers), then the set of possible states are defined using the `set_pmf` function on the initial action `a`. This function returns all results from a PMF which have a probability greater than 0 – in this case, all possible states that `a` could transition into. If the time step is higher than that, we look at all states in the preceding `Qslice`. For each state in that slice, we choose an action using policy `p`, then use `set_pmf` again to form a set of those actions. The function returns a union over all of these sets using `⋃s'∈Qslice (Suc n) p s a`.

`Qslicep n p s a s'` returns the total probability that, by any possible route given initial action `a` and subsequent policy `p`, an agent will be in state `s'` at time step `n`.

```
fun Qslicep :: "nat ⇒ ('s ⇒ 's pmf) ⇒ 's ⇒ 's pmf ⇒ 's ⇒ real"
where
  "Qslicep 0 p s a s' = (if (s' = s) then 1 else 0)"
| "Qslicep (Suc 0) p s a s' = pmf a s'"
| "Qslicep (Suc (Suc n)) p s a s'
    = (∑s''∈a. pmf a s'' * Qslicep (Suc n) p s'' (p s'') s')"
```

Listing 2.8: The `Qslicep` definition

This works in a similar manner to the `Qslice` function. We find our values of interest using the `pmf a s` function, which returns the probability of state `s` in action `a`. For any given route to the state of interest, we multiply the transition probabilities together for the different time steps to arrive at the total probability of arriving via that route. These routes are summed across the slices, considering any possible route to the state in question.

`RQslice n p s a` returns the reward expected by an agent at time step `n` for taking a single action and transitioning to a new state at time step `n+1`. We do not take discounting into account when calculating `RQslice` – it is the expected reward earned, not the value we place on it. Note that it is a single step's worth of reward only.

```
fun RQslice :: "nat ⇒ ('s ⇒ 's pmf) ⇒ 's ⇒ 's pmf ⇒ real"
  where
  "RQslice 0 p s a = (∑s'∈a. (pmf a s') * R (s, s', a))"
| "RQslice (Suc n) p s a =
    (∑s'∈Qslice (Suc n) p s a. Qslicep (Suc n) p s a s' *
    (∑s''∈p s'. (pmf (p s') s'') * R (s', s'', (p s'))))"
```

Listing 2.9: The `RQslice` definition

Again, this works recursively, albeit slightly more simply than our preceding examples. The initial reward is calculated using the fixed choice of action `a`, and subsequent time steps use the `Qslice` and `Qslicep` functions to probabilistically weight the reward for each possible action.

To ensure that these are consistent and represent our expectations for MDPs, we prove some basic properties of these functions. We begin by showing that for each function the recursive definition can be evaluated in different ways. We use these to give us the expressiveness we need to show that these simple functions can be re-expressed as the Bellman equation [13]:

```
lemma Qslice_rec:
  fixes p :: "'s ⇒ 's pmf" and s :: 's and a :: "'s pmf"
  shows "Qslice (Suc m) p s a
    = (⋃s'∈a. Qslice m p s' (p s'))"

lemma Qslicep_rec:
  fixes s' s''' :: 's and a' :: "'s pmf" and p :: "('s ⇒ 's pmf)"
  assumes "a'∈K s'"
  shows "Qslicep (Suc m) p s' a' s'''
    = (∑s''∈a'. pmf a' s'' * Qslicep m p s'' (p s'') s''')"

lemma Qslicep_rec2:
  fixes s s' :: 's and a :: "'s pmf" and p :: "('s ⇒ 's pmf)"
  assumes a_in_Ks:"a∈K s" and p_in_pol:"p∈policies"
  shows "Qslicep (Suc (Suc m)) p s a s'
    = (∑s''∈Qslice (Suc m) p s a.
      Qslicep (Suc m) p s a s'' * pmf (p s'') s')"

lemma RQslice_rec:
  fixes s s' :: 's and a :: "'s pmf" and p :: "('s ⇒ 's pmf)"
  assumes a_in_Ks:"a∈K s" and p_in_pol:"p∈policies"
  shows "RQslice (Suc n) p s a
```

```
    = (∑s'∈a. (pmf a s') * RQslice n p s' (p s'))"
```

Listing 2.10: Lemmas showing we can express the recursion of the slice based functions

These lemmas are useful for proving the theorems we will go on to establish as they show the equivalence of several different definitions for the functions.

Proving that it is possible to evaluate the slice functions across all the subsequent states in a slice at time step `n` and the set of current states in the slice at the subsequent time step `n+1`, and produce identical results takes a number of non-trivial lemmas. As an example of this type of lemma:

```
lemma Qslice_pol_summation2:
  fixes s s'' :: 's and a :: "'s pmf" and p :: "('s ⇒ 's pmf)"
  assumes a_in_Ks:"a∈K s" and p_in_pol:"p∈policies"
    and m_gr_0:"m > 0" and s''_in_a:"s''∈a"
  shows "(∑s'''∈Qslice m p s'' (p s'').
    pmf a s'' * Qslicep m p s'' (p s'') s''' * pmf (p s''') s')
      = (∑s'''∈Qslice (Suc m) p s a.
        pmf a s'' * Qslicep m p s'' (p s'') s'''
          * pmf (p s''') s')"
```

Listing 2.11: The `Qslice_pol_summation2` lemma

This specific lemma shows that summing the total probability of being in a given state at a given time step of a particular set of paths is the same regardless of whether we measure the paths from an initial state s or from some subsequent state s' after accounting for the transition between s and it.

We next show that `Qslicep` produces a probability distribution over the states at any time step `n`:

```
lemma Qslicep_nn: "Qslicep m p s a s' ≥ 0"

lemma Qslicep_1:
 fixes s :: 's and a :: "'s pmf" and p :: "('s ⇒ 's pmf)"
   and m :: nat
 assumes a_in_Ks:"a∈K s" and p_in_pol:"p∈policies"
 shows "(∑s'∈Qslice (Suc m) p s a. Qslicep (Suc m) p s a s') = 1"
```

Listing 2.12: Lemmas showing `Qslicep` forms a probability distribution over the states and that `s' ∈ Qslice n p s a` if and only if `Qslicep n p s a s' > 0`:

```
lemma Qslice_Qslicep:
  fixes s s' :: 's and p :: "('s ⇒ 's pmf)" and a :: "'s pmf"
```

```
  assumes a_in_Ks:"a∈K s" and p_in_pol:"p∈policies"
  shows "s' ∈ Qslice m p s a = (Qslicep m p s a s' > 0)"
```

Listing 2.13: The `Qslice_Qslicep` lemma

2.4.3 Proving convergence properties

These basic properties proven, we go on to build our definition of value on our MDP model. We do this using two functions, as described next.

`Qexpectedn n p s a` gives us the expected value up to time step `n` of an agent beginning in state `s`, taking action `a` and subsequently following policy `p`. The discount factor γ is taken into account in this definition.

```
definition
 Qexpectedn :: "nat ⇒ ('s ⇒ 's pmf) ⇒ 's ⇒ 's pmf ⇒ real"
  where
"Qexpectedn n p s a = (∑i=0..n. γ^i * RQslice i p s a)"
```

Listing 2.14: The `Qexpectedn` definition

where γ^i indicates γ^i.

`Qexpected p s a` takes the limit of the `Qexpectedn` function as `n` approaches infinity. We will prove that this function represents the Q-value as used in MDP literature (see section 2.3.1) by showing that we can derive the Bellman equation using it in section 2.4.4.

```
definition Qexpected :: "('s ⇒ 's pmf) ⇒ 's ⇒ 's pmf ⇒ real"
where
"Qexpected p s a = lim (λn. Qexpectedn n p s a)"
```

Listing 2.15: The `Qexpected` definition

We define a terminal state in our representation as one where all possible actions produce a reward of zero and do not transition the agent into a different state. In other words, if an agent is in a terminal state, neither total reward earned nor the state of the agent will change in subsequent slices.

```
definition terminal_state :: "'s ⇒ bool" where
  "terminal_state s =
      (∀s' a'. R (s, s', a') = 0 ∧ (⋃a∈K s. set_pmf a) = {s})"
```

Listing 2.16: The `terminal_state` definition

We say that an MDP has inevitable terminal states under a given policy `p` if and only if for any agent pursuing that policy, after some finite time, the slice of possible states of the agent only includes terminal states:

```
definition inevitable_term_state :: "('s ⇒ 's pmf) ⇒ bool" where
  "inevitable_term_state p =
      (∀s. ∀a∈K s. ∃n. ∀s'∈Qslice n p s a. terminal_state s')"
```

Listing 2.17: The `inevitable_term_state` definition

We prove that `Qexpectedn n p s a` converges as `n` $\to \infty$ next, and thus `Qexpected p s a` has a definite value, under either of two different assumptions:

- When `inevitable_term_state` is true under policy `p`.
- When γ<1. This second assumption is the one that we will use in our later proofs throughout section 2.5, but specifically in subsection 2.5.9, as it is fundamental, in the context of a finite MDP, to proving an optimal policy exists.

The proof under the first assumption is essentially trivial, as knowing that the agent can only be in some terminal state at some finite time step means we simply must sum our rewards until the point where `Qslice n p s a` is a set of only terminal states. As there exists an `n` where this must be true, convergence is guaranteed. The Isabelle theorem is thus:

```
theorem convergence_Qexpectedn_term_state:
  assumes "inevitable_term_state p" and "a∈K s"
      and "p∈policies"
    shows "convergent (λn. Qexpectedn n p s a)"
```

Listing 2.18: The `convergence_Qexpectedn_term_state` theorem

The proof under the second assumption is more complicated and depends on the reward function `R` being bounded (above and below) by some `M`, which as it is acting on a finite set of states and actions, it must be. That established, our discounted `RQslice` function must be less than $\sum_{i=0}^{n} \gamma^i M$ for any n. As γ<1, this is a geometric series, so we know it converges. By the usual comparison and absolute value series tests [8, 42], we know that our `Qexpectedn` value converges too as `n` approaches infinity. The resulting Isabelle theorem is as follows:

```
theorem convergence_Qexpectedn_discount:
  assumes "a∈K s" and "p∈policies" and "γ < 1"
```

```
  shows "convergent (λn. Qexpectedn n p s a)"
```

Listing 2.19: The `convergence_Qexpectedn_discount` theorem

2.4.4 Deriving the Bellman equation

Our final step here is to show that our method of calculating the Q-value of a given state-action-policy combination matches the Bellman equation (see section 2.3) [13].

We do this by first showing that it is possible to evaluate `Qexpectedn` recursively:

```
lemma Qexpectedn_rec:
  fixes s s' :: 's and a :: "'s pmf" and p :: "('s ⇒ 's pmf)"
  assumes a_in_Ks:"a∈K s" and p_in_pol:"p∈policies"
  shows "Qexpectedn (Suc n) p s a =
    (∑s'∈a. (pmf a s') * (R (s, s', a) +
        γ * Qexpectedn n p s' (p s')))"
```

Listing 2.20: The `Qexpectedn_rec` lemma

We then use this to show that it is possible to evaluate `Qexpectedn` by "splitting" the value as such:

```
lemma Qexpectedn_split:
  fixes s :: 's and a :: "'s pmf" and p :: "('s ⇒ 's pmf)"
  assumes a_in_Ks:"a∈K s" and p_in_pol:"p∈policies"
  shows "Qexpectedn (Suc n+m) p s a = Qexpectedn n p s a +
    (∑s'∈Qslice (Suc n) p s a. γ^(Suc n) *
         Qslicep (Suc n) p s a s' * Qexpectedn m p s' (p s'))"
```

Listing 2.21: The `Qexpectedn_split` lemma

This (along with a few minor lemmas showing properties of limits) in turn leads to the proof that `Qexpected p s a` produces the same result as the Bellman equation:

```
theorem Qexpected_split_Bellman:
  fixes a :: "'s pmf" and s :: 's and p :: "'s ⇒ 's pmf"
  assumes "a∈K s" and "p∈policies"
    and "∧s a'. a'∈K s ⟹ convergent (λn. Qexpectedn n p s a')"
  shows "Qexpected p s a = (∑s'∈a. (pmf a s') * (R (s, s', a) +
                              γ * Qexpected p s' (p s')))"
```

Listing 2.22: The `Qexpected_split_Bellman` theorem

Note that for this proof, we assume convergence of `Qexpectedn` rather than any specific assumption which proves that convergence. From this general proof, we can

derive the Bellman equation under either the assumption that γ<1 or the assumption that `inevitable_terminal_states p`.

We then define `Vexpected p s = Qexpected p s (p s)`; in other words, the V value is simply the Q value where we replace the chosen action by that dictated by our choice of policy.

We have now shown that we can derive Bellman's equation in full from our locale assumptions and the model of value we built based on a naïve understanding of an agent gathering reward over time. This (along with our other results) gives us confidence that our model is not flawed and accurately represents a Markov decision process.

2.5 Proving the existence of an optimal policy

In this section, we cover the non-trivial mechanisation of the proof of the existence of an optimal policy on all finite MDPs with a discounting factor less than one.

2.5.1 Why is an optimal policy important?

The purpose of using an MDP is typically to come up with the ideal strategy for optimising some discretisable process: to find some policy that, regardless of an agent's starting state or initial action, we can expect to produce the highest possible total reward if pursued. As noted before (see section 2.3), we usually evaluate this potential total reward using the concept of a discounted value. We call this policy an optimal policy.

A very important consideration, then, is whether such an optimal policy exists. Clearly, given there are finitely many states and actions, our choice of policies is finite too. For any individual choice of initial state-action combination, there will be a policy that produces the highest value when followed subsequently. But the question of the existence of a policy that produces a maximal result for *any* choice of initial state and action is critical to knowing that an MDP can fulfil its purpose.

We use Puterman's proof of the existence of an optimal policy [126] as the basis for our own formal proof. It is important to note though that Puterman approaches this question from a more general starting point – his proof is adequate for finite or infinite MDPs (using slightly different methods at various points), and considers non-stationary policies (policies whose choice function might vary over time) before prov-

ing that a stationary policy is adequate for optimality. We begin by assuming that all policies are stationary and that our MDP is finite – as previously noted, stationary policies are sufficient to find optimality (as Puterman proved), and the scope of our formalisation is over finite MDPs as that is what the informal proofs of the reinforcement learning properties of interest demand (see the discussion in Sections 3.1.1.4 and 3.1.1.5).

Note that we are skipping the detail of the individual proofs themselves, instead giving the outline of the structure of Puterman's proof so that it is clear what is needed to follow the description of our formal version. Moreover, in many cases when discussing the mathematical background, we will state that we mechanise particular statements without providing the formal proof, except as a sketch at most. The interested reader can find details of the pen-and-paper proof in chapters 5 and 6 and appendix C of Puterman's book, and can consult our Isabelle theories for additional details pertaining to the formalisation.

As we review the various elements of the proof, we will begin by going over the mathematics and then discuss the details of the formalisation.

2.5.2 Functions on states as a vector space

First, note that all real functions on our states are over a finite set and are therefore bounded in magnitude. Consider them as belonging to the vector space V of bounded real-valued functions on the states. In V, each component of a vector represents the result of the function on a particular state. Note that V is closed under addition and scalar multiplication as expected [82]. The dimension of the vector space is equal to the number of states that we have.

Equip V with the supremum norm, which is the supremum in the reals of the norms of its components (recall each component is $v(s)$ for some state s). Each supremum is attained by each $v \in V$ as we have finite states. The definition is then as follows:

$$\begin{aligned} \forall v \in V.\ \|v\| &= \sup_{s \in S} |v(s)| \\ &= \max_{s \in S} |v(s)| \end{aligned} \tag{2.2}$$

Then equip V with a partial order where a vector v is less than or equal to another v' if and only for all states s, $v(s) \leq v'(s)$. Because this is a partial order, we have no guarantee that arbitrary vectors in V are comparable using them – if there exists s, s'

such that $v(s) < v'(s)$ and $v'(s') < v(s')$, then v and v' cannot be ordered with respect to each other. This partial ordering is given by:

$$\forall v, v' \in V.\ v \leq v' \iff \forall s.\ v(s) \leq v'(s) \tag{2.3}$$

A *Cauchy sequence* in V is a sequence v_n of vectors where

$$\forall \varepsilon > 0.\ \exists N.\ \forall n, m \geq N.\ \|v_m - v_n\| < \varepsilon$$

and a *complete space* is one in which the limit of all Cauchy sequences in the space is attained within the space [82]. A *Banach space* is simply a complete, normed vector space, and thus we know that V is a Banach space: this will be critical to the later proof in section 2.5.9 when we make use of the Banach fixed point theorem, a property of Banach spaces and certain functions (contraction mappings) over them.

Isabelle has several formalisations of vectors but none that exactly matches what we need. There is an often-used vector type, `vec`, within the "Finite Cartesian Product" theory of the main Isabelle/HOL analysis library, and it would allow us to define vectors from our real functions over the finite state type. Unfortunately, it lacks the supremum norm required by Puterman's proof and has a Euclidean norm instead.

However, Isabelle's analysis library has a formal theory of bounded continuous functions, with a type of the same name which has a supremum norm defined over it. This type also belongs to the normed vector space type class and the complete space type class, so we can perform addition and scalar multiplication over it and we know it is a Banach space.

In this theory, `('a, 'b) bcontfun` represents bounded continuous functions from a type variable `'a` (which must be an instance of the `topological_space` type class) to the type variable `'b` (which must be an instance of the `metric_space` type class).

If we equip our state space with the discrete topology (where any subset is both an open and closed set), any function on it is continuous through the topological definition of continuity (where a function is continuous if the preimage of any open set is open), and the type variable we use to represent it would be an instance of the `topological_space` type class. As the state space is finite, we also know that functions on it must be bounded. So we can represent real functions over a finite state space as bounded continuous functions this way, with all the properties of a Banach space and the supremum norm we are looking for.

We still have to define a partial ordering over the bounded continuous functions, which we do. Note that `apply_bcontfun` accesses the underlying function from a

`bcontfun` and obtains its result.

```
definition less_eq_bcontfun ::
  "('a, 'b) bcontfun ⇒ ('a, 'b) bcontfun ⇒ bool"
  where "less_eq_bcontfun a b
    = (∀i. apply_bcontfun a i ≤ apply_bcontfun b i)"

definition less_bcontfun ::
  "('a, 'b) bcontfun ⇒ ('a, 'b) bcontfun ⇒ bool"
  where "less_bcontfun a b = (a ≤ b ∧ ¬ b ≤ a)"
```

Listing 2.23: The functions needed to form a partial ordering on bounded continuous functions

2.5.3 Operators over functions on states as matrices

Next, we return to Puterman's proof: we consider linear operators over V as matrices in their own vector space W, with the usual understanding of matrix-vector multiplication representing its operation. As normal, a matrix A may be shown as invertible with an inverse A^{-1}, if $AA^{-1} = A^{-1}A = I$, where I is the identity matrix [96].

For these operators, we define their norm as the operator norm [16] against the result of their operation on vectors in V. For a matrix A, this is the least upper bound on $\frac{\|Av\|}{\|v\|}$, meaning the greatest scaling we expect its operation to have on the norm of any vector v. It can be shown that the operator norm for A can be found by taking the supremum of the norm of any vectors of at most norm 1 after they are multiplied by A [16]:

$$\begin{aligned} \forall A \in W.\ \|A\| &= \inf\{c \geq 0 : \forall v \in V : \|Av\| \leq c\|v\|\} \\ &= \sup\{\|Av\| : \|v\| \leq 1, v \in V\} \end{aligned} \tag{2.4}$$

In this case, as we are using the supremum norm for vector space V, we can show that the operator norm is equal to the highest sum of absolute values in any row of A. If we take $A_{i,j}$ as being the component in row i and column j of the matrix A, then:

$$\forall A \in W : \|A\| = \sup_i \{\sum_j |A_{i,j}|\} \tag{2.5}$$

In the Isabelle analysis library's "Finite Cartesian Product" theory, matrices are represented as vectors-of-vectors (abbreviated at `mat`). Again, unfortunately, the norm definition does not match what we need (the Euclidean norm is used), and in any case

the vector type we are using is not defined for multiplication by members of the `mat` type.

In order to match Puterman's usage of matrices, we have two main requirements:

1. The matrix type should be equipped with the operator norm.
2. It should provably be a complete space.

Unfortunately, no matrix type in Isabelle's current libraries has exactly what we need. However, the `sq_mtx` type of square matrices in the "SQ MTX" theory, part of the "Matrices for ODEs" entry in the Archive of Formal Proof [168] does have a notion of operator norm, which we can adapt to meet our requirements. Note that if `A` is of this type, we use the notation `A$$i$j` to refer to its component in row `i` and column `j`. This `sq_mtx` type is restricted to representing real-valued square matrices, but that is all we need for our work.

The `sq_mtx` type has multiplication defined against the `vec` type, not the bounded continuous function type that we are using. In order to use this definition for matrix multiplication and the various theorems that prove its properties, we demonstrate that a bijection exists between `vec` (which does have matrix multiplication defined against it) and bounded continuous functions.

Vectors, represented as `('a, b') vec` type, can be defined in Isabelle using Cartesian products, where `'a` gives the type of the vector components and `'b` is a finite type variable whose cardinality represents the dimension of the vector. The second type, by virtue of its finiteness, provides an index into the vector, enabling us to refer to its components. Thus, a `('a, b') vec` for a type variable `'b` is essentially the function space `'a` $\Rightarrow$ `'b` [56]. For a vector `v` of type `vec`, the notation `v$i` then represents its i^{th} component and the notation `(`χ `i. f i)` (where `f i` can be any function) defines a vector whose i^{th} component is equal to `f i`.

A similar notation, namely `('a, 'b) bcontfun` is used for bounded continuous functions, although, in this case, this type refers to the underlying function space `'a::topological_space` $\Rightarrow$ `'b::metric_space` as previously noted.

Given these two representations of vectors and bounded continuous functions, we define the following bijections between them:

```
definition bcontfun_of_vec ::
 "('a::metric_space,
   'b::finite_discrete_topology) vec ⇒ ('b, 'a) bcontfun" where
"bcontfun_of_vec v = Bcontfun(λi. v$i)"
```

```
definition vec_of_bcontfun ::
 "('b::finite_discrete_topology,
   'a::metric_space) bcontfun ⇒ ('a, 'b) vec" where
"vec_of_bcontfun f = (χ i. (apply_bcontfun f) i)"

lemma bcf_inv[simp]:
  shows "vec_of_bcontfun (bcontfun_of_vec v) = v"
  and "bcontfun_of_vec (vec_of_bcontfun f) = f"
```

Listing 2.24: Bijections between vectors and bounded continuous functions

We define a new matrix multiplication operation over a bounded continuous function by first casting it to a vector of the `vec` type, then perform matrix-vector multiplication, and finally casting the result back to a bounded continuous function. This method also ensures that the existing body of proofs regarding matrix-vector multiplication are available to us when we need to prove the properties of our matrix-bounded continuous function multiplication, greatly easing the amount of work we have to do here (note the `sq_mtx_vec_mult` function simply performs a matrix-vector multiplication):

```
definition sq_mtx_bcf_mult ::
  "('m::finite_discrete_topology) sq_mtx ⇒
    ('m, real) bcontfun ⇒ ('m, real) bcontfun"
where
  "A *M x =
    bcontfun_of_vec (sq_mtx_vec_mult A (vec_of_bcontfun x))"
```

Listing 2.25: The `sq_mtx_bcf_mult` definition

Note that we use the notation `r *R A` to represent scalar multiplication of a matrix `A` by scalar `r` and the notation `A *M v` to represent matrix multiplication of a bounded continuous function `v` by a matrix `A`.

We go on to prove a wide variety of typical linear algebra results of matrix-vector multiplication are still true against our bounded continuous function definition. The proofs for these fall out quickly from our use of the existing matrix-vector multiplication in our new definition as we have established a bijection between the two types previously. Some example theorems are the following:

```
lemma mtx_bcf_mult_add_rdistr: "(A + B) *M x = A *M x + B *M x"
  unfolding sq_mtx_bcf_mult_def
  by (simp add: bcontfun_vec_add mtx_vec_mult_add_rdistr)
```

```
lemma sq_mtx_times_bcf_assoc: "(A * B) *_M x = A *_M (B *_M x)"
  by (metis bcf_inv(1) sq_mtx_bcf_mult_def sq_mtx_times_vec_assoc)
```

Listing 2.26: Example theorems in multiplication of square matrices by bounded continuous functions that adapt existing theorems of matrix-vector multiplication

We then equip these matrices with the operator norm against this function on bounded continuous functions. Our definition here performs a matrix multiplication on a bounded continuous function, which gives us a vector that we then cast as a bounded continuous function again. This means that the `om_norm` function, also denoted by $\|\_\|_{om}$, measures the change in the norm on the bounded continuous function using its supremum norm, which is the behaviour we need.

```
abbreviation om_norm ::
  "(('a::{metric_space, real_normed_vector, semiring_1},
    'b::finite_discrete_topology) vec, 'b) vec ⇒ real"
  where "‖A‖_om ≡ onorm (λx. bcontfun_of_vec (bcf_mtx_mult x A))"
```

Listing 2.27: The operator norm for bounded continuous function-matrix multiplication

Note that `bcf_mtx_mult` here is a matrix multiplication over a bounded continuous function interpreted as a vector. This is then used to define the norm in our proof that our square matrix type forms a normed vector space:

```
definition norm_sq_mtx :: "'a sq_mtx ⇒ real"
  where "‖A‖ = ‖to_vec A‖_om"
```

Listing 2.28: The `norm_sq_mtx` definition

We go on to prove its properties as required, beginning with verifying that the operator norm thus defined is equal to the highest sum of the absolute values of the elements of any row in the matrix:

```
theorem sq_mtx_norm_value:
  fixes A :: "('b::finite_discrete_topology) sq_mtx"
  shows "‖A‖ = Max (⋃i. {(∑j. norm (A$$i$j))})"
```

Listing 2.29: Verifying the norm matches operator norm expectations

The next result:

```
theorem sq_mtx_norm_mult:
  fixes A B :: "('b::finite_discrete_topology) sq_mtx"
  shows "‖A * B‖ ≤ ‖A‖ * ‖B‖"
```

Listing 2.30: The `sq_mtx_norm_mult` theorem

is also important to many subsequent proofs, in particular when dealing with stochastic matrices.

The `sq_mtx` type was originally shown to form a complete space, but we reprove this using our new operator norm, establishing that we still have a Banach space.

2.5.4 Stochastic matrices

When using matrix operators, we will usually be dealing with matrices that capture the probability transitions of a particular policy π, denoted T^π. Each component $T^\pi_{i,j}$ represents the probability of transitioning to state j under action $\pi(i)$. As these represent transitions, we refer to them as transition matrices, but they are more generally examples of stochastic matrices [28]. As each row of a matrix represents a probability distribution, each row therefore sums to 1, with each element being non-negative.

We establish the notion of a stochastic matrix in Isabelle via a predicate that verifies this property:

```
definition stochastic_mtx ::
  "'b::finite_discrete_topology sq_mtx ⇒ bool" where
"stochastic_mtx A = ((∀i. (∑j. A$$i$j) = 1) ∧ (∀i j. A$$i$j ≥ 0))"
```

Listing 2.31: The `stochastic_mtx` definition

and then prove some of elementary properties of stochastic matrices, starting with the fact that the identity matrix is a stochastic matrix:

```
theorem stochastic_mtx_1: "stochastic_mtx 1"
```

Listing 2.32: The `stochastic_mtx_1` theorem

Next, we prove that the product of any two stochastic matrices is also a stochastic matrix, and likewise for a exponentiation of a stochastic matrix. We inherit matrix exponentiation and its properties for the `sq_mtx` type from the fact that it is proven as an instance of the `real_normed_algebra_1` type class, which has exponentiation defined against it.

```
theorem stochastic_mtx_mult:
  fixes A B :: "'b::finite_discrete_topology sq_mtx"
  assumes "stochastic_mtx A" "stochastic_mtx B"
  shows "stochastic_mtx (A * B)"
```

```
theorem mat_pow_stochastic_mtx:
  fixes A :: "'b::finite_discrete_topology sq_mtx" and n :: nat
  assumes "stochastic_mtx A"
  shows "stochastic_mtx (A^n)"
```

Listing 2.33: Stochastic matrices closed under multiplication and exponentiation

Note that `A^n` represents a matrix `A` raised to the power `n`. We also show that the operator norm is 1 for any stochastic matrix (which falls very simply out of our definition of a stochastic matrix):

```
theorem stochastic_mtx_norm:
  fixes A :: "'b::finite_discrete_topology sq_mtx"
  assumes "stochastic_mtx A"
  shows "∥A∥ = 1"
```

Listing 2.34: The norm of a stochastic matrix is always 1

2.5.5 Spectral radius using Gelfand's formula

We now build a formal notion of spectral radius on our matrices. We denote the spectral radius of a matrix A by $\rho(A)$. Normally the spectral radius of a matrix is defined as the largest of its eigenvalues [96] – an eigenvalue being the scalar by which A multiplies a vector that otherwise is unchanged by matrix-vector multiplication.

We will not use this approach though, since our purpose here is to further our formalisation of MDPs. The notion of spectral radius only plays a small part in Puterman's proof – namely to establish the invertibility of certain matrices – and for this we only need Gelfand's formula: $\rho(A) = \lim_{n\to\infty} \|A^n\|^{1/n}$ [47]. This is especially important for proving that we can express the expected value vector of a policy in terms of a relatively simple inverse matrix (as expressed later with equation 2.10), and for deriving some other results in section 2.5.8. Using Gelfand's formula, we then need to show that:

$$\rho(I-A) < 1 \implies \exists A^{-1}.\, A^{-1} = \lim_{N\to\infty} \sum_{n=0}^{N} (I-A)^n \tag{2.6}$$

Therefore, we express the spectral radius using the notion of matrix exponentiation to match Gelfand's formula. Thankfully, the latter is already defined for types satisfying the `real_normed_algebra_1` sort [66], within a theory belonging to the "Ergodic Theory" session in the Archive of Formal Proof [51]:

```
definition spectral_radius::"'a::real_normed_algebra_1 ⇒ real"
  where "spectral_radius x = Inf {root n (norm(x^n))| n. n>0}"
```

Listing 2.35: The `spectral_radius` definition

where `root n x` denotes the n^{th} root of real number x and `Inf` denotes the infimum operator. This definition is easily shown to be equivalent to the limit one from Gelfand's formula, which is then used to formalise statement (2.6) as follows:

```
theorem specrad_inverse_matrix:
  fixes A :: "'b::finite_discrete_topology sq_mtx" and L :: real
  assumes "spectral_radius (1-A) < 1"
  shows "mtx_invertible A"
    and "A⁻¹ = lim (λN. ∑n=0..N. (1 - A)^n)"
```

Listing 2.36: The `specrad_inverse_matrix` definition

The formal proof of the above result is lengthy and follows the informal proof in Puterman [126]. We first show that if $\rho(A) < 1$ then $\sum_{i=0}^{\infty} A^i$ converges using the completeness of a Banach space. We go on to construct the matrix $\sum_{i=0}^{\infty} A^i$ and prove that when multiplied by A it results in the identity, using the operator norm.

2.5.6 Casting functions to vectors and matrices

We now define the functions in Isabelle that will allow us to cast any function on states to the bounded continuous function type, and any function on pairs of states to the square matrix type:

```
definition vectify :: "('s ⇒ real) ⇒ ('s, real) bcontfun" where
"vectify f = Bcontfun f"

definition vectify_inv :: "('s, real) bcontfun ⇒ ('s ⇒ real)"
  where "vectify_inv v = apply_bcontfun v"

definition matrify :: "('s ⇒ 's ⇒ real) ⇒ 's sq_mtx" where
"matrify f = to_mtx (χ i j. f i j)"

definition matrify_inv :: "'s sq_mtx ⇒ ('s ⇒ 's ⇒ real)" where
"matrify_inv M = (λi j. vec_nth (sq_mtx_ith M i) j)"
```

Listing 2.37: Functions to cast between bounded continuous functions and functions on states, and functions on pairs of states and square matrices

When a type is defined from another underlying type in Isabelle, functions for casting to and from that underlying type can be created automatically. Our `vectify` and `matrify` functions are all straightforward adaptations of these automatically created casting functions. The `matrify` definition uses the function parameter `f` to populate the matrix rows and columns with the function's result on corresponding states. We then prove that these functions can be used as inverses of each other:

```
lemma "vectify (vectify_inv v) = v"

lemma "vectify_inv (vectify f) = f"

lemma "matrify (matrify_inv A) = A"

lemma "matrify_inv (matrify f) = f"
```

Listing 2.38: Lemmas showing these casting functions can be used as inverses of each other

We note that `vectify (Vexpected p)` is the vector of the expected reward under policy `p` against the states. Using `vectify`, we define another vector which holds the expected reward for taking a single step from a state under policy p, which we can find using the function `single_reward_vec`:

```
definition single_reward_vec :: "('s ⇒ 's pmf) ⇒ 's ⇒ real"
  where
"single_reward_vec p = (λs. RQslice 0 p s (p s))"
```

Listing 2.39: The `single_reward_vec` definition

Using `matrify` we define a matrix which forms the transition matrix under policy `p`, using a function called `transition_matrix`. We then show that this transition matrix is a stochastic matrix:

```
definition transition_matrix :: "('s ⇒ 's pmf) ⇒ 's sq_mtx"
  where
"transition_matrix p = matrify (λs s'. pmf (p s) s')"

theorem trans_matr_stochastic:
  fixes p :: "'s ⇒ 's pmf"
  assumes "p∈policies"
  shows "stochastic_mtx (transition_matrix p)"
```

Listing 2.40: The `transition_matrix` definition and proof that its results are stochastic

We have now built the concepts needed for our proof of the existence of an optimal policy. The groundwork thus laid, we go on to the proof itself.

2.5.7 An optimal policy definition

Define P as the set of all valid policies. As we have noted before, this is a finite set as we have finite states and finite actions we can take from each one. Define v_π as the vector of the function which returns the value (the total expected discounted reward) of each state under policy π (see section 2.3). Then define v^* as the supremum in the vector space V of the set $\{v_\pi : \pi \in P\}$.

Recall that the supremum here is the least vector in V such that it is greater than or equal to every v_π. Note that even though the set of policies is finite, we cannot guarantee that the supremum v^* is attained as V is only partially ordered. We *can* say that:

$$\forall s.\ \exists \pi.\ v^*(s) = v_\pi(s)$$

So we know that for each component s of v^*, there exists at least one v_π such that its component s attains the value $v^*(s)$. We just cannot yet say that there is one v_π that does so for every s. Indeed, *this* is what we are trying to prove.

If we can find a π^* such that $v_{\pi^*} = v^*$, then we have a universally optimal policy for all starting states, as we will know that for any other policy, the V-value of any state must be less than or equal to that of v_{π^*}.

We begin to formalise this by defining the optimum policy on state-action pairs via a function `optimal_policies` that returns these as a set for a given state `s` and action `a`. We also define the optimum policy on states using `optimal_policies_state`, a function that returns these for a given `s`. We demonstrate that these both exist for any choice of state or valid state-action pair. These will not form part of the definition of a universal optimal policy directly, but will demonstrate how our definition relates to the expected conditions for a universal optimal policy. We refer the reader back to our definitions of `Qexpected` and `Vexpected` in section 2.4.3.

```
definition optimal_policies :: "'s ⇒ 's pmf ⇒ ('s ⇒ 's pmf) set"
  where
"optimal_policies s a =
 {p∈policies.
   (∀p'∈policies. Qexpected p' s a ≤ Qexpected p s a)}"
```

```
definition optimal_policies_state :: "'s ⇒ ('s ⇒ 's pmf) set"
  where
"optimal_policies_state s =
  {p∈policies.
    (∀p'∈policies. Qexpected p' s (p' s) ≤ Qexpected p s (p s))}"
```

Listing 2.41: Sets of optimal policies fixed per state-action and per state

Next, we define `Vmax`, a function that returns the highest possible V-value for a state `s`. We prove that `p` is in `optimal_policies_state s` if and only if `Vexpected p s = Vmax s`:

```
definition Vmax :: "'s ⇒ real" where
"Vmax s = Max (⋃p∈policies.{Vexpected p s})"

lemma Vmax_optimal:
  fixes s :: 's and p :: "('s ⇒ 's pmf)"
  assumes p_in_pol:"p∈policies"
  shows "(p∈optimal_policies_state s) = (Vexpected p s = Vmax s)"
```

Listing 2.42: The `Vmax` definition and proof an optimal policy per state achieves it

We next define the set of universally optimal policies as the set of policies whose expected reward vector is greater than or equal to that of all other policies (recalling the ordering that we have on these vectors from section 2.5.2). We show that this definition is equivalent to the intersection of the sets of optimal policies over the individual states, as we would expect, and equivalent to defining it as the set of policies where `Vexpected p = Vmax`.

```
definition optimal_policies_universal :: "('s ⇒ 's pmf) set"
  where
"optimal_policies_universal =
 {p∈policies.
  (∀p'∈policies.
    vectify (Vexpected p) ≥ vectify (Vexpected p'))}"

theorem universal_equivalence:
  shows "optimal_policies_universal =
             (⋂(range optimal_policies_state))"
```

```
lemma universal_optimal_policy_Vmax_equivalence:
  fixes p :: "'s ⇒ 's pmf"
  assumes "p∈policies"
  shows "(Vexpected p = Vmax) = (p∈optimal_policies_universal)"
```

Listing 2.43: The definition of a universally optimal policy and proof of its relation to our prior definitions

2.5.8 The L_π operator and its supremum, $\mathscr{L}_{max}$

For each policy π, define $r_\pi(s)$ as the function which returns the expected immediate reward an agent earns for taking the action $\pi(s)$ from s. Recall that T^π is the transition matrix for any π. Using this, we derive a vector form of Bellman's equation (see section 2.4.4):

$$v_\pi = r_\pi + \gamma T^\pi v_\pi \tag{2.7}$$

We prove this in Isabelle by assuming that we have convergence of value, which we have previously proven when $\gamma < 1$ or when we have inevitable terminating states (see section 2.4.3):

```
theorem vec_expr_value:
  fixes p :: "'s ⇒ 's pmf"
  assumes "p∈policies"
    and "⋀s. ∀a∈K s. convergent (λn. Qexpectedn n p s a)"
  shows "vectify (Vexpected p) = vectify (single_reward_vec p)
    + γ *R (transition_matrix p) *M (vectify (Vexpected p))"
```

Listing 2.44: The vector form of Bellman's equations in Isabelle

Note the use of `vectify` here, needed so that we can have our expected values in vector form.

Next, we suppose that $u, v \in V$ and $\gamma < 1$ and show the following, which will be needed for subsequent proofs:

$$u \geq 0 \implies \forall \pi.\ (I - \gamma T^\pi)^{-1} u \geq u \tag{2.8}$$

$$u \geq v \implies \forall \pi.\ (I - \gamma T^\pi)^{-1} u \geq (I - \gamma T^\pi)^{-1} v \tag{2.9}$$

In Isabelle, they are formalised as follows:

```
lemma discounted_transition_inverse_incr_gezero:
  fixes p :: "'s ⇒ 's pmf" and u :: "('s, real) bcontfun"
  assumes "p∈policies" and "γ<1" and "u ≥ 0"
  shows "((1 - γ *R (transition_matrix p))^-1 *M u) ≥ u"

lemma discounted_transition_inverse_incr_gevec:
  fixes p :: "'s ⇒ 's pmf" and u v :: "('s, real) bcontfun"
  assumes "p∈policies" and "γ < 1" and "u ≥ v"
  shows "((1 - γ *R (transition_matrix p))^-1 *M u) ≥
    ((1 - γ *R (transition_matrix p))^-1 *M v)"
```

Listing 2.45: Isabelle lemmas for equations 2.8 and 2.9

The proofs here depend on the fact that our the vector space is a Banach space, so we can show that the limit of sequence of matrices converges using the completeness property, and on the invertibility properties of matrices with a spectral radius less than 1, which we proved previously (see section 2.5.5).

Next, define the operator L_π on any vector v as $L_\pi v = r_\pi + \gamma T_\pi v$. Intuitively, this returns the immediate reward vector for following policy π, then adds v proportionally based on the probability of transition from one state to another under π. Using (2.7), it is clear that v_π is a fixed point (but not necessarily a unique fixed point – we prove this below) for L_π as $L_\pi v_\pi = v_\pi$. In Isabelle, we formalise the `L` function as:

```
definition L :: "('s ⇒ 's pmf) ⇒ ('s, real) bcontfun
  ⇒ ('s, real) bcontfun" where
"L p v = vectify (single_reward_vec p) +
             γ *R ((transition_matrix p) *M v)"
```

Listing 2.46: The L operator

We then show that, using Gelfand's formula and (2.6), that v_π is the unique fixed point for L_π. We go on to derive the following equation assuming $\gamma < 1$, using (2.9) and the fact that the matrix space is complete:

$$v_\pi = (I - \gamma T^\pi)^{-1} r_\pi \tag{2.10}$$

This shows that provided we can find the inverse of $(I - \gamma T^\pi)$, which (2.6) guarantees when we have $\gamma < 1$ and thus a spectral radius less than one, we can calculate the vector of V-values for a given policy using only its transition matrix and expected rewards.

We formalise these results in Isabelle:

```
theorem find_expected_value:
  fixes p :: "'s ⇒ 's pmf"
  assumes "p∈policies" and "γ<1"
  shows "vectify (Vexpected p) =
    (1 - (γ *R (transition_matrix p)))^-1
      *M (vectify (single_reward_vec p))"
```

Listing 2.47: The `find_expected_value` theorem

Now, define the operator $\mathscr{L}_{max}$ over vectors in V as follows (note that in Puterman's text, this is denoted $\mathscr{L}$ – we add the subscript here for clarity):

$$\mathscr{L}_{max}v = \sup_{\pi}\{L_\pi v\} \tag{2.11}$$

Recall once more that V is partially ordered, so we cannot be sure that $\mathscr{L}_{max}v$ is attained by any one choice of policy π. We know, again given our finite choice of states, that for any single component s of $\mathscr{L}_{max}v$, there is a choice of π such that $(L_\pi v)(s) = (\mathscr{L}_{max}v)(s)$, but we do not yet know that there is always a policy π such that $L_\pi v = \mathscr{L}_{max}v$.

We now proceed to define an $\mathscr{L}_{max}$ function in Isabelle by taking the maximum values for `L` across the policies, and we prove that for any `v` it is attained by some policy `p`:

```
definition ℒmax ::
  "('s, real) bcontfun ⇒ ('s, real) bcontfun" where
"ℒmax b = Bcontfun (λs.
  Max(⋃p∈policies. {apply_bcontfun (L p b) s}))"

theorem ℒmax_attained:
  shows "∃p∈policies. L p v = ℒmax v"
```

Listing 2.48: The $\mathscr{L}_{max}$ definition

Note that Puterman does not prove that $\mathscr{L}_{max}$ is attained immediately in his proof – rather it is his final step, under a set of assumptions (such as compactness of the action space and continuous reward and transition functions) that we do not need to consider (as we deal with the finite MDP case only). We prove this now to fix an issue with one of Puterman's proofs as we will detail soon.

Next is the biggest step toward the proof of the existence of an optimal policy. We show using (2.9) that (assuming $0 \leq \gamma < 1$):

$$\mathscr{L}_{max}v = v \implies v = v^* \tag{2.12}$$

There is actually a minor error in Puterman's proof here that we uncovered during our formalisation – which is the reason why we show that $\mathscr{L}_{max}$ is attained earlier than Puterman does. During a particular part of his proof, Puterman asserts:

$$v \leq \mathscr{L}_{max} v \implies \forall \varepsilon > 0.\ \exists \pi.\ \forall s.\ v(s) \leq L_\pi v(s) + \varepsilon \tag{2.13}$$

He uses the ε here as he is also considering the infinite state case (where the supremum may not be attained even on a componentwise basis); it can be dismissed in the finite case, leaving:

$$v \leq \mathscr{L}_{max} v \implies \exists \pi.\ \forall s.\ v(s) \leq L_\pi v(s) \tag{2.14}$$

As we have previously discussed, it has not yet been shown that there exists a policy that attains $\mathscr{L}_{max}$. Puterman goes on to prove that such a policy exists much later as the final step in his proof of the existence of an optimal policy (albeit only under any one of several assumptions – one choice of which is that the MDP is finite, as in our case). Thankfully his delayed proof does not rely on any of his theorems where he seems to have implicitly assumed it. In any case, Puterman does not mention the limiting assumptions or the necessary proof that a policy exists that attains $\mathscr{L}_{max}$ in his proof of (2.14) here, and this proof is therefore incomplete.

We formally prove in Isabelle that if there is a fixed point for $\mathscr{L}_{max}$, that it must be `vectify (Vmax)`:

```
theorem v_eq_ℒmax_v:
  fixes v :: "('s, real) bcontfun"
  assumes "v = ℒmax v" "γ<1"
  shows "v = vectify (Vmax)"
```

Listing 2.49: A fixed point for $\mathscr{L}_{max}$ is `vectify (Vmax)`

2.5.9 Final steps

We then prove that (under the assumption $\gamma < 1$) both L_π and $\mathscr{L}_{max}$ are contraction mappings, in other words for any $u, v \in V$, $\|L_\pi u - L_\pi v\| \leq \|u - v\|$ and likewise for $\mathscr{L}_{max}$. This allows us to use the Banach fixed point theory (along with our proof that V is a Banach space) to show that there exists a unique fixed point of both operators. In Isabelle, we have:

```
theorem contraction_L:
  fixes u v :: "('s, real) bcontfun" and p :: "'s ⇒ 's pmf"
  assumes "γ<1" "p∈policies"
  shows "dist (L p u) (L p v) ≤ γ * dist u v"

theorem contraction_ℒmax:
  fixes u v :: "('s, real) bcontfun"
  assumes "γ<1"
  shows "dist (ℒmax u) (ℒmax v) ≤ γ * dist u v"

theorem unique_solution_L:
  fixes v :: "('s, real) bcontfun" and p :: "'s ⇒ 's pmf"
  assumes "γ<1" "p∈policies"
  shows "∃!v. L p v = v"

theorem unique_solution_ℒmax:
  fixes v :: "('s, real) bcontfun" and p :: "'s ⇒ 's pmf"
  assumes "γ<1" "p∈policies"
  shows "∃!v. ℒmax v = v"
```

Listing 2.50: The L operators are contraction mapping, with unique fixed points

All the pieces of the proof are in place now, and the only remaining significant step is to show that there exists a policy π such that for a choice of v, $L_\pi v = \mathscr{L}_{max} v$. We do this by noting that for each choice of state component, say s, we can maximise the value for $L_\pi v(s)$ just by assuming that π takes a particular choice of action from that state; we then note that if we make that assumption for π across every state, that $L_\pi v = \mathscr{L}_{max} v$. Note that we have already shown this in Isabelle with the $\mathscr{L}_{max}$`_attained` theorem in the previous section.

This done, we have found a policy π^* such that $L_{\pi^*} v = \mathscr{L}_{max} v$. $\mathscr{L}_{max}$ has a unique fixed point v^* (2.12), which by our vector version of Bellman's equation (2.7), we know must also be the unique fixed point for L_{π^*}, and hence the expected value function for policy π^*.

By our definition of v^*, we thus know that π^* attains the highest possible expected value for each state component, and is the universally optimal policy.

We demonstrate this in Isabelle by showing that we can select a policy `p` such that it attains $\mathscr{L}_{max}$ on the vector of its expected values on the states using the $\mathscr{L}_{max}$`_attained` theorem. We then know that $\mathscr{L}_{max}$ `(vectify (Vexpected p)) = (vectify (Vexpected p))` and therefore that `vectify (Vexpected p) = Vmax`. This all leads to a proof of the very satisfying statement of our sought-after theorem:

```
theorem universal_optimal_policy_exists:
  assumes "γ<1"
  shows "∃p. p∈optimal_policies_universal"
```

Listing 2.51: Proof under a discount that a universally optimal policy exists

2.6 Value and policy iteration

As the next step in our formalisation, we introduce the value iteration and policy iteration algorithms, and formally prove that they work as intended. Again, our work here focuses on formalising the work presented in Puterman's textbook [126].

2.6.1 Finding an optimal policy

In the previous section we have proved that a universal optimal policy always exists on any finite MDP with a discount. We know this is important because an agent following this policy should behave optimally over the MDP, attaining maximal reward.

But how do we construct the optimal policy for an arbitrary MDP? In the situation where we do not know the structure of the MDP and must discover it by exploration, reinforcement learning is the typical process used. However, we have simpler methods available in the case where we know the properties of the MDP, its states, actions, rewards and the transition probabilities associated with it. These methods include value iteration and policy iteration, two algorithms intended to compute the optimal policy, or an arbitrarily close approximation in the case of value iteration, in finite time.

In this section we will discuss each algorithm, before going on to show their representation in Isabelle/HOL, and our proofs of their correctness.

2.6.2 Value iteration

The first algorithm we will look at is value iteration. Before we examine it, we will define an ε-optimal policy as one where the distance between its expected value vector and that of an optimal policy is at most ε. This distance is defined using the vector norm, so we are saying that a policy p_ε is ε-optimal if and only if:

$$\forall s.\ |V_{p_\varepsilon}(s) - V_*(s)| \leq \varepsilon$$

The value iteration algorithm constructs an ε-optimal policy in finite time, via the following process:

1. Choose an arbitrary vector v_0 in V, the vector space of bounded functions on the states. Choose an ε arbitrarily close to 0. We begin with time step $n = 0$.

2. Find v_{n+1} using:
$$v_{n+1} = \mathscr{L}_{max} v_n$$

3. If
$$\|v_{n+1} - v_n\| \leq \varepsilon(1-\gamma)/2\gamma$$
then proceed, otherwise increment n and repeat step 2.

4. For each s, find $p_\varepsilon(s)$ using:
$$p_\varepsilon(s) = \underset{a \in A(s)}{\arg\max} \left\{ \sum_{s' \in a} P(s,a,s') \big(R(s,s',a) + \gamma v_{n+1}(s) \big) \right\}$$

The policy p_ε found this way is ε-optimal.

We represent this algorithm in Isabelle via four functions. `Value_Iteration` is the first of these, and gives us the vector v_n found in step 2 of the algorithm. It takes as parameters the time step of the algorithm `n` and the chosen initial vector `v`.

```
fun Value_Iteration ::
  "nat ⇒ ('s, real) bcontfun ⇒ ('s, real) bcontfun" where
 "Value_Iteration 0 v = v"
|"Value_Iteration (Suc n) v = ℒmax (Value_Iteration n v)"
```

Listing 2.52: The `Value_Iteration` definition

The function `Value_Iteration_Algo_n` gives us the first time step n when the conditions for step 3 of the algorithm are met, using Isabelle's `LEAST` operator. To find this, we must specify the parameter ε instead of a time step.

```
definition Value_Iteration_Algo_n ::
  "real ⇒ ('s, real) bcontfun ⇒ nat" where
"Value_Iteration_Algo_n ε v =
  (LEAST n. dist (Value_Iteration (Suc n) v) (Value_Iteration n v)
    < (ε * (1 - γ))/ (2 * γ))"
```

Listing 2.53: The `Value_Iteration_Algo_n` definition

`Value_Iteration_Algo_v` gives us the vector associated with the time step we need to calculate our policy.

```
definition Value_Iteration_Algo_v ::
  "real ⇒ ('s, real) bcontfun ⇒ ('s, real) bcontfun" where
"Value_Iteration_Algo_v ε v =
    Value_Iteration ((Value_Iteration_Algo_n ε v)+1) v"
```

Listing 2.54: The `Value_Iteration_Algo_v` definition

Finally, `Value_Iteration_Algo` constructs the policy associated with completion of the algorithm.

```
definition Value_Iteration_Algo ::
  "real ⇒ ('s, real) bcontfun ⇒ ('s ⇒ 's pmf)" where
"Value_Iteration_Algo ε v =
 (λs. arg_max
   (λa. (∑s'∈a. pmf a s' * (R(s, s', a) +
     γ * apply_bcontfun (Value_Iteration_Algo_v ε v) s')))
   (λa. a∈K s))"
```

Listing 2.55: The `Value_Iteration_Algo` definition

To show that our representations are accurate, we prove basic properties of the value iteration algorithm and conclude by formally verifying that it leads to an ε-optimal policy. We begin by proving that the vector produced by iteration tends towards the optimal value vector as $n \to \infty$. This proof falls out of the use of the $\mathscr{L}_{max}$ operator in our definitions and our previous proof of the existence of an optimal policy.

```
lemma Value_Iteration_Converges:
  fixes v :: "('s, real) bcontfun"
  assumes "γ<1"
  shows "lim (λn. Value_Iteration n v) = vectify Vmax"
```

Listing 2.56: Proof that value iteration converges to the maximum value vector

Next we show that provided $\varepsilon > 0$, we can always find a finite n that fulfils the conditions of step 3 of the value iteration algorithm. This falls easily out of the limit property we have just proven and the fact that our vectors form a Banach space. Note that we must specify the additional assumption that $\gamma > 0$, as otherwise division by zero would be possible on the value we test against.

```
lemma Value_Iteration_Algo_n_Exists:
  fixes v :: "('s, real) bcontfun" and ε :: real
  assumes "0 < γ" "γ<1"
  shows "∀ε>0. ∃n.
          dist (Value_Iteration (Suc n) v) (Value_Iteration n v) <
            (ε * (1 - γ))/ (2 * γ)"
```

Listing 2.57: The `Value_Iteration_Algo_n_Exists` lemma

We continue by proving that the final result of the algorithm is always a policy. This is important in our formalisation as policies are not a type – they are members of a set with certain properties. Even if the result is the correct type, we still need to be sure that it meets the conditions for being in the set of policies to know that it is well-behaved. This proof is not completely trivial – we show that the `arg_max` used in the definition always exists in the set of valid actions on a state, and then show that the resulting function always produces such a valid action.

```
lemma Value_Iteration_Algo_policy:
  fixes ε :: real and v :: "('s, real) bcontfun"
  shows "Value_Iteration_Algo ε v ∈ policies"
```

Listing 2.58: The `Value_Iteration_Algo_policy` lemma

This next proof is crucial to establishing ε-optimality: we show that the policy p_ε we produce with `Value_Iteration_Algo` and the vector v_ε we produce with `Value_Iteration_Algo_v` gives the equivalence $L_{p_\varepsilon}(v_\varepsilon) = \mathscr{L}_{max}(v_\varepsilon)$.

```
lemma Value_Iteration_Algo_Equiv:
fixes ε :: real and v :: "('s, real) bcontfun"
shows "L (Value_Iteration_Algo ε v) (Value_Iteration_Algo_v ε v) =
         ℒmax (Value_Iteration_Algo_v ε v)"
```

Listing 2.59: The `Value_Iteration_Algo_Equiv` lemma

Our final proof here makes use of two results – the equivalence we have established and the triangle inequality for norms – to give an upper bound on the distance between the value vector produced by our constructed policy and the optimal value vector. The equivalence is necessary to be able to use the contraction mapping properties of the $\mathscr{L}_{max}$ and L_{p_ε} operators to establish this bound. Once we have the bound, simple algebraic manipulation and repeated use of the triangle inequality leads to the result.

```
theorem Value_Iteration_Epsilon_Optimal:
  fixes ε :: real and v :: "('s, real) bcontfun"
  assumes "γ>0" "γ<1" "ε>0"
  shows "dist (vectify (Vexpected (Value_Iteration_Algo ε v)))
           (vectify Vmax) < ε"
```

Listing 2.60: The `Value_Iteration_Epsilon_Optimal` lemma

We conclude, then, by demonstrating that the policy we have produced has a value vector less than ε distant from the optimal value vector – in other words, that it is ε-optimal as expected.

2.6.3 Policy iteration

We move on to examine policy iteration. Unlike value iteration, it constructs a universally optimal policy in finite time, not simply an ε-optimal policy. We will formally prove that this is achieved by any algorithm which meets the specification.

Policy iteration follows this process:

1. Choose an arbitrary policy p_0. We begin with time step $n = 0$.

2. Find v_n using:

$$v_n = (I - \gamma T_{p_n})^{-1} r_{p_n}$$

   Recall that r_{p_n} is the vector of expected rewards earned by taking a single step from a state and T_{p_n} is the transition matrix associated with policy p_n.

3. Select a policy p_{n+1} such that:

$$p_{n+1}(s) = \arg\max_{a \in A(s)} \left\{ \sum_{s' \in a} P(s,a,s') \big(R(s,s',a) + \gamma v_n(s) \big) \right\}$$

   If p_n can fulfil this condition, choose it.

4. If $p_{n+1} = p_n$ then call it p^* and conclude the process. Otherwise, increment n, return to step 2 and continue.

The policy p^* found this way is universally optimal.

As we did for value iteration, we represent this algorithm in Isabelle with a number of recursive functions.

`Policy_Iteration` and `Policy_Iteration_v` are mutually recursive functions that give us the result of step 3 of the algorithm. `Policy_Iteration_v` represents the

vector v_n found in step 2. They take as parameters the time step `n` of the algorithm and the chosen initial policy `p`. Note that we use an `if` conditional to check if the policy constructed in the previous increment meets the conditions for the maximisation; if it does we select it, otherwise we choose an arbitrary policy that meets the conditions.

```
fun Policy_Iteration :: "nat ⇒ ('s ⇒ 's pmf) ⇒ ('s ⇒ 's pmf)"
  and Policy_Iteration_v ::
    "nat ⇒ ('s ⇒ 's pmf) ⇒ ('s, real) bcontfun" where
  "Policy_Iteration 0 p = p"
| "Policy_Iteration_v n p
  = (1 - γ *R (transition_matrix (Policy_Iteration n p)))^-1 *M
     (vectify (single_reward_vec (Policy_Iteration n p)))"
| "Policy_Iteration (Suc n) p =
    (if (∀s. is_arg_max (λa.
      (∑s'∈a. pmf a s' * (R(s, s', a) +
         γ * apply_bcontfun (Policy_Iteration_v n p) s')))
      (λa. a∈K s) ((Policy_Iteration n p) s))
     then (Policy_Iteration n p)
     else (λs. arg_max (λa.
            (∑s'∈a. pmf a s' * (R(s, s', a) +
               γ * apply_bcontfun (Policy_Iteration_v n p) s')))
            (λa. a∈K s)))"
```

Listing 2.61: The `Policy_Iteration` and `Policy_Iteration_v` definitions

`Policy_Iteration_Algo_n` gives us the first time step at which the conditions for step 4 are met. This is found in a similar way to the corresponding value iteration function, likewise using Isabelle's `LEAST` operator.

```
definition Policy_Iteration_Algo_n :: "('s ⇒ 's pmf) ⇒ nat"
  where
"Policy_Iteration_Algo_n p =
  (LEAST n. (Policy_Iteration (Suc n) p = Policy_Iteration n p))"
```

Listing 2.62: The `Policy_Iteration_Algo_n` definition

Finally, `Policy_Iteration_Algo` gives us the policy p^* which is the final outcome of the algorithm.

```
definition Policy_Iteration_Algo :: "('s ⇒ 's pmf)
  ⇒ ('s ⇒ 's pmf)" where
"Policy_Iteration_Algo p =
  Policy_Iteration (Policy_Iteration_Algo_n p) p"
```

Listing 2.63: The `Policy_Iteration_Algo` definition

As before, we need to verify some basic properties of the algorithm to show our representation is correct before moving on to our proof of optimality.

We begin by showing that for every step of the algorithm produces a new policy p_n, which meets the set-based definition of policy we are using in our model. Again, we simply have to show that the `arg_max` function can find a valid action and then show that these choices of actions define a policy.

```
lemma Policy_Iteration_policy:
  fixes p :: "'s ⇒ 's pmf" and n :: nat
  assumes "p∈policies"
  shows "Policy_Iteration n p ∈ policies"
```

Listing 2.64: Proof that policy iteration always results in a policy

Next we show that for subsequent policies $L_{p_n+1}v_{p_n} = \mathscr{L}_{max}v_{p_n+1}$. Again, this is crucial for subsequent proofs, but unlike value iteration, we do not use the contraction mapping properties of these functions. Instead we use their fixed points – we are not showing an approximation to optimality, but optimality itself.

```
lemma Policy_Iteration_Equiv:
  fixes p :: "'s ⇒ 's pmf" and n :: nat
  assumes "p∈policies" "γ<1"
  shows "L (Policy_Iteration (Suc n) p)
           (vectify (Vexpected (Policy_Iteration n p))) =
                𝓛max (vectify (Vexpected (Policy_Iteration n p)))"
```

Listing 2.65: The `Policy_Iteration_Equiv` lemma

A crucial part of our optimality proof is to show that every time step of the algorithm (unless there is no change) improves the policy – i.e. every policy constructed has a value vector that is strictly greater than that of the previous step. This proof relies on manipulation of vector calculations using the invertibility properties of the matrices that are used in the policy iteration definitions.

```
lemma Policy_Iteration_inc:
  fixes p :: "'s ⇒ 's pmf" and n :: nat
  assumes "p∈policies" "γ<1"
  shows "vectify (Vexpected (Policy_Iteration (Suc n) p)) ≥
            vectify (Vexpected (Policy_Iteration n p))"
```

Listing 2.66: The `Policy_Iteration_inc` lemma

Our next proof demonstrates that if we have a step of the algorithm where the policy does not change, the expected value vector of that policy must be optimal – and thus,

the policy is universally optimal. This makes use of the `Policy_Iteration_Equiv` lemma and the fixed points of the L_{p_n} and $\mathscr{L}_{max}$ operators.

```
lemma Policy_Iteration_Works:
  fixes p :: "'s ⇒ 's pmf" and n :: nat
  assumes "p∈policies" "γ<1"
          "Policy_Iteration (Suc n) p = Policy_Iteration n p"
  shows
  "vectify (Vexpected (Policy_Iteration n p)) = vectify (Vmax)"
```

Listing 2.67: The `Policy_Iteration_Works` lemma

We know that if the policy does not remain the same when the algorithm iterates, that it improves. And we know that if the policy does remains the same, then it must be universally optimal. We know that there are only finitely many valid policies. It is obvious from these results that in finite time we will produce a universally optimal policy. Unfortunately, despite the intuitive obviousness here, it remains to be shown formally that this will occur in finite time. We do this via some simple results about monotonicity and by incorporating our proof of finite policies.

```
lemma Policy_Iteration_Finite:
  fixes p :: "'s ⇒ 's pmf"
  assumes "p∈policies" "γ<1"
  shows "∃N. (Policy_Iteration N p) ∈ optimal_policies_universal"
```

Listing 2.68: The `Policy_Iteration_Finite` lemma

This proof complete, we have shown that policy iteration constructs a universally optimal policy in finite time.

2.7 Case study

We present here an illustration of the potential of this formalisation to benefit future research. We will describe a relatively simple game, but one where the optimal strategy is non-trivial. We will show how it is possible to formally model the game in Isabelle and prove that it is an instance of a finite Markov Decision Process. Thus, we know with certainty that at least one optimal strategy exists, and we can apply tools such as value iteration, policy iteration, or reinforcement learning to find that optimal strategy.

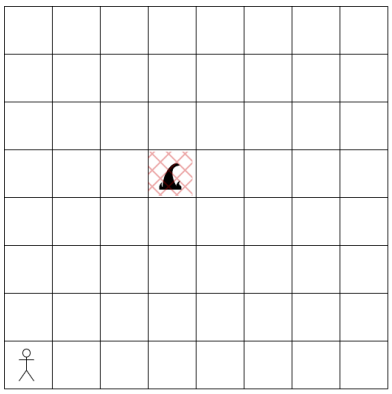

Figure 2.2: The initial conditions of the Lavaworld game. The stick man represents the player, the fire represents the lava elemental, and the red cross-hatching shows a square is lava.

2.7.1 The game - Lavaworld

The game of Lavaworld is played out on a 10x10 square grid, representing a crumbling and eroding floor, beneath which is deadly lava. The grid is indexed from 0 to 9 in both the horizontal and vertical directions, with the bottom left corner being $(0,0)$. The grid is joined at the left and right edges and the top and bottom edges, so if one were to travel left from $(0,0)$, you would end up in $(9,0)$ and vice versa. Similarly travelling south from $(0,0)$ takes you to $(0,9)$ and vice versa.

The player begins in the bottom left corner. An enemy, a lava-elemental creature that spreads deadly lava, begins in square $(4,5)$, which is initially filled with lava. This starting condition is shown in figure 2.2. Each game turn, the player chooses one of the four cardinal directions to travel. As they leave their current square, the floor collapses behind them exposing the lava. The lava-elemental can only travel into squares that are not already filled with lava; if squares clear of lava are adjacent to its current square, it chooses one at random and travels there. Any square it enters is then immediately filled with lava. An example of a first game turn is shown in figure 2.3.

At the start of every turn, if the player is in a square filled with lava, they lose 100 points, and the game restarts on the next turn. Otherwise, if they have managed to get the lava-elemental into a position where it is totally surrounded by lava filled squares (and hence, cannot move into any of them), they gain 100 points, and the game restarts. If neither of these conditions apply, they gain 1 point.

This is the entirety of the game. Despite being a simple game to describe, there are many thousands of possible game states as each consists of the player's position, the

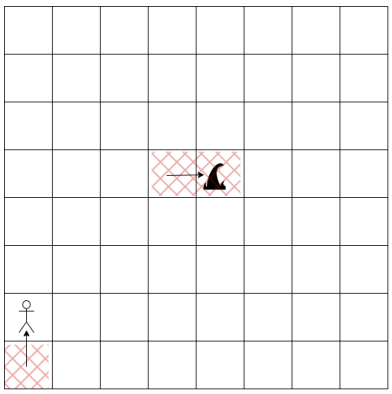

Figure 2.3: Game turn 1. The player has chosen to go north, and the lava elemental has gone east. It is clear that the lava is spreading.

lava-elemental's position, and the set of lava filled squares. It is also apparent that there are nuances of strategy. For example, the player should not go back onto any square they have been, always forging new ground; but that new ground should keep them distant from the lava-elemental and the potentially expanding lava area it is creating. However, trapping the lava-elemental is the only way to get a positive score at the end of the game; despite the risk associated with it, it is almost certainly a factor in any optimal strategy. Balancing risk against reward is obviously necessary in formulating a strategy, but it isn't clear how this should be done. It is not trivially obvious that there is an optimal strategy, or how to go about finding one. This is where the clarity of identifying that the game is an MDP can bring insight.

2.7.1.1 Formalising Lavaworld

We proceed to formalise the game in Isabelle. We begin by defining a base finite type that we can use to label the co-ordinates of our grid, essentially counting from 0 to 9 – the `coord` type. On this datatype we also define an increment and decrement function (`forcoord` and `backcoord` respectively):

```
datatype coord = Zero | One | Two | Three | Four
  | Five | Six | Seven | Eight | Nine

definition forcoord :: "coord ⇒ coord" where
"forcoord t =
  (case t of
    Zero ⇒ One | One ⇒ Two | Two ⇒ Three | Three ⇒ Four |
    Four ⇒ Five | Five ⇒ Six | Six ⇒ Seven | Seven ⇒ Eight |
    Eight ⇒ Nine | Nine ⇒ Zero)"

definition backcoord :: "coord ⇒ coord" where
"backcoord t =
  (case t of
    Zero ⇒ Nine | One ⇒ Zero | Two ⇒ One | Three ⇒ Two |
    Four ⇒ Three | Five ⇒ Four | Six ⇒ Five | Seven ⇒ Six |
    Eight ⇒ Seven | Nine ⇒ Eight)"
```

Listing 2.69: The `coord` datatype and basic functions on it.

Next, we define a type for the four directions that the player or the lava elemental may move in (the `dir` type). We then define the `directiongo` function describing the results of an object moving in a particular direction on the grid:

```
datatype dir = N | E | S | W

definition directiongo :: "dir ⇒ (coord×coord) ⇒ (coord×coord)"
   where
"directiongo d a =
  (case d of
    N ⇒ (fst a, forcoord (snd a)) |
    E ⇒ (forcoord (fst a), snd a) |
    S ⇒ (fst a, backcoord (snd a)) |
    W ⇒ (backcoord (fst a), snd a))"
```

Listing 2.70: The `dir` type and basic movement function.

What constitutes a game state? To know the exact conditions of the game at a given turn, we need to know the current location of the player, the current location of the lava elemental, and all the locations currently filled with lava, as discussed at the end of the previous section. We can formalise this as an object of type `((coord×coord)×(coord×coord)×(coord×coord) set)`. The first tuple holds the player location, the second holds the lava-elemental location, and the last set of tuples holds the set of locations that are currently filled with lava.

We define a type `gamestate` that has this structure, using the `typedef` command to define it by the set of all such objects. The use of this command also creates two functions, `Abs_gamestate` that changes an object of the underlying type into the `gamestate` type, and `Rep_gamestate` that does the opposite. We then define the starting conditions of the game with the `initialstate` function:

```
typedef gamestate =
  "UNIV::((coord×coord)×(coord×coord)×(coord×coord) set) set"
  by blast

definition initialstate :: "gamestate" where
  "initialstate = Abs_gamestate((Zero,Zero),(Four,Five),
    {(Four,Five)})"
```

Listing 2.71: The `gamestate` type and definition of initial conditions.

How do we determine if the lava-elemental has been trapped? A function `enemydirs` determines which directions are valid for the enemy to move in. If this set is empty, then the enemy is trapped. The function checks if the destination square in all four cardinal directions is not yet lava filled, and returns the set of directions where this is true.

```
definition enemydirs :: "gamestate ⇒ dir set" where
"enemydirs gs =
  (if directiongo N (fst (snd (Rep_gamestate gs)))
    ∈ (snd (snd (Rep_gamestate gs))) then {} else {N})
  ∪ (if directiongo E (fst (snd (Rep_gamestate gs)))
    ∈ (snd (snd (Rep_gamestate gs))) then {} else {E})
  ∪ (if directiongo S (fst (snd (Rep_gamestate gs)))
    ∈ (snd (snd (Rep_gamestate gs))) then {} else {S})
  ∪ (if directiongo W (fst (snd (Rep_gamestate gs)))
    ∈ (snd (snd (Rep_gamestate gs))) then {} else {W})"
```

Listing 2.72: The `enemydirs` function.

We next define the functions that define the game turn. `playermove` takes two direction parameters, the first for the player and the second for the lava elemental – we will show how this is randomly distributed next. If the player is currently in the lava, the game resets. If not, the player moves in their chosen direction, the lava elemental moves in its chosen direction, then both the space the player most recently occupied and the space the lava elemental has just entered are added to the set of lava filled squares. A reward is issued using the `reward` function: -100 if the player is in

lava, 100 if they are not in lave and have successfully trapped the lava-elemental, or 1 otherwise.

```
definition playermove :: "dir ⇒ dir ⇒ gamestate ⇒ gamestate" where
"playermove d e gs =
  (if fst (Rep_gamestate gs) ∈ (snd (snd (Rep_gamestate gs)))
    ∨ enemydirs gs = {} then initialstate
      else let enemymove =
         directiongo e (fst (snd (Rep_gamestate gs))) in
          Abs_gamestate (directiongo d (fst (Rep_gamestate gs)),
          enemymove,
          insert (fst (Rep_gamestate gs))
            (insert enemymove (snd (snd (Rep_gamestate gs))))))"

definition reward :: "gamestate ⇒ real" where
"reward gs =
  (if fst (Rep_gamestate gs) ∈ snd (snd (Rep_gamestate gs))
    then -100
    else (if enemydirs gs = {} then 100 else 1))"
```

Listing 2.73: The `playermove` and `reward` functions.

We next determine how the enemy moves using a probability mass function built using the `pmfgen` function. This has parameters for the direction that the player has chosen to travel and the current state. It outputs a function that takes a parameter of a possible state and outputs the probability of that state occurring (given the parameters passed to `pmfgen`). This distribution of possible result states defines the distribution of lava-elemental movement. Note that if the player is currently in lava or if they have trapped the lava-elemental, there is only one possible result state, so we must allow for this by measuring the number of possible state results using the `card` function.

The `Enemymove` function then builds a probability mass function using `embed_pmf` of all possible states that might result from a given player action – recall from section 2.3.2 that this is how actions are defined in our MDP formalisation:

```
definition pmfgen :: "dir ⇒ gamestate ⇒ gamestate ⇒ real" where
"pmfgen d gs = (if enemydirs gs = {}
  then (λx. if initialstate = x then 1 else 0)
  else
    (λx. (1/(real (card (⋃e∈enemydirs gs. {(playermove d e gs)})))
      * (indicat_real (⋃e∈enemydirs gs. {(playermove d e gs)}) x)))"

definition Enemymoves :: "dir ⇒ gamestate ⇒ gamestate pmf" where
```

```
"Enemymoves d gs = embed_pmf (pmfgen d gs)"
```

Listing 2.74: The `pmfgen` and `Enemymoves` functions.

Lastly, we use the `Permitted_game` and `Reward_game` functions to convert what we have into the form that we need to prove that this game is an instance of our MDP formalisation. `Permitted_game` takes a `gamestate` parameter and returns the set of all possible actions (defined as pmf distributions of subsequent states) in that state. This is the `K` function of our MDP formalisation. `Reward_game` returns the reward for moving into a specific state. This is the `R` function of our MDP formalisation. We also define a value for the discount γ of 0.95.

```
definition Permitted_game :: "gamestate ⇒ (gamestate pmf) set" where
"Permitted_game gs =
  {Enemymoves N gs}∪{Enemymoves E gs}
    ∪{Enemymoves S gs}∪{Enemymoves W gs}"

definition Reward_game ::
  "(gamestate × gamestate × gamestate pmf) ⇒ real" where
  "Reward_game x = reward (fst (snd x))"

definition γ :: real where "γ = 0.95"
```

Listing 2.75: The `Permitted_game` and `Reward_game` functions, and the γ `definition`.

It remains to show that our `gamestate` type is in the `finite_discrete_topology` class. We do this by assigning it the discrete metric $\mathscr{M}$ (where if $x = y$, $\mathscr{M}(x,y) = 0$, with $\mathscr{M}(x,y) = 0$ otherwise), the discrete topology (where every subset of the `gamestate` space is both open and closed), and proving that by doing so, it meets all the necessary criteria.

```
instance gamestate :: finite_discrete_topology ..
```

Listing 2.76: The instance proof to show `gamestate` is in the `finite_discrete_topology` class.

With all of these elements in place, the remaining proofs to show that we have constructed an MDP from our game specification are simple.

```
interpretation Finite_Markov_Reward_Process
  Permitted_game Reward_game γ
proof
  show "∀s. finite' (Permitted_game s)"
  proof -
    {fix gs :: gamestate
      have "∀d. card ({Enemymoves d gs}) = 1"
        by simp
      then have "finite' (Permitted_game gs)"
        using Permitted_game_def
        by simp}
    then show ?thesis
      by simp
  qed
  show "0≤γ∧γ≤1"
    using γ_def by simp
qed
```

Listing 2.77: The proof to show the Lavaworld game can be interpreted as an MDP.

This proven, all of the properties we have proven for MDPs over the course of this work are properties that are proven for the Lavaworld game, under this interpretation. So, we know that there is an optimal policy to maximise reward. We know that we might use policy iteration or value iteration to find or approximate that policy (although given the number of states involved, this would be extremely difficult). Perhaps more importantly, as we have established this is a discrete, finite Markov Decision Process, reinforcement learning methods such as Q Learning [158] (discussed more in chapter 3) can be applied to learn this optimal policy.

2.8 Related work

The most closely related prior work to our own is that of Hölzl [64], which we have discussed at length in section 2.3.2 as we use it as the starting point for the current mechanisation.

Shortly after the development on our formalisation had concluded, another formalisation in Isabelle of MDPs with rewards appeared in the Archive of Formal Proof, by Schäffeler and Abdulaziz [134, 135, 100], taken from Schäffeler's thesis [136]. This work covers the same area and uses the same textbook as its source [126]. However, there are some important differences between our work and that of Schäffeler and Ab-

dulaziz.

Their work is slightly more general in scope, covering infinite MDPs as well as finite MDPs. However, it uses the same mathematical approach as Hölzl, namely the category-theoretic and measure-theoretic approach of the Giry monad (as discussed in Section 2.3.2). The question of whether to opt to use the Giry monad or the linear algebra approach used by Puterman and ourselves for transition probability computation touches upon a fundamental difference in standpoints when it comes to framework usability. The intention of our work is to provide formal models and proofs that use the same mathematical structure as that widespread in the literature. By doing so, it renders our work more accessible and understandable. Our work is also extendable using existing informal proofs with a linear algebra focus.

The mathematical background for working with the Giry monad (category theory and measure theory over probability spaces) is substantially less common than the linear algebra background needed to understand and use stochastic matrices. So, our argument is that by basing our work upon linear algebra, our formalisation is more accessible to a wider section of the research community, especially those interested in reinforcement learning, who may not have as extensive a background in abstract (categorical) mathematics. While it would probably be possible to adapt the Giry monad theorems and lemmas to work in a linear algebraic fashion, this would both require work to translate between linear algebra concepts and the category theoretical concepts of the Giry monad, and additionally leave the proofs themselves still opaque to the majority of those who might find them useful. In the longer run, we believe that this may make our framework easier to adopt since it is anchored in more familiar textbook mathematics.

Also of interest was the error that we found in Puterman's proof (discussed under equation 2.12 in section 2.5.8), which is not discussed in Schäffeler's thesis [136] – it appears to have gone undiscovered using their approach. This illustrates the value when formalising a proof of keeping as close as possible to the source material if one wishes to uncover weaknesses within it. We believe that this brings additional value to our formalisation since the error, despite Puterman's proof being widely cited, seems to have gone undetected until now.

Another work which covers similar ground to our own is the CertRL work by Vajjha et al., which first appeared late in the development of our work [154]. This proves the existence of an optimal policy again using the same Giry monad approach taken by Hölzl and some properties related to value and policy iteration. It uses the Coq theorem

prover rather than Isabelle, and thus has a different proving style and audience.

CertRL only covers some of the work we have performed on value and policy iteration. It proves value iteration will find the optimal policy (the first section of our own set of value iteration proofs) and that policy iteration improves the policy with each iteration. We extend these results by showing that policy iteration completes in finite time with an optimal policy. Note that in the Vajjha et al.'s paper there is an assertion that this extended result was proven but we were unable to locate this proof in their Coq theories. We also prove that value iteration achieves ε-optimality in finite time, something which CertRL does not attempt.

The CertRL work proves optimality for value iteration in finite steps, but only against a finite-horizon MDP (in other words, a Markov Decision Process where the agent is limited to a known number of steps). Our own work assumes an infinite horizon for all proofs.

2.9 Conclusion

We have built a formal model of a finite MDP in Isabelle and have shown that it allows the mechanisation of the properties one might expect. In particular:

- We derived Bellman's equation, which describes the expected total value of being in a given state in an MDP, choosing a particular action, and then subsequently pursuing a given policy.
- We formalised the circumstances under which, given infinite time, our valuation of rewards earned by an agent converges.
- We formally proved that Gelfand's formula can be used to establish the invertibility of a square matrix and find its inverse.
- We developed the vector form of Bellman's equation and proved that its results match our scalar version.
- We mechanised a vector-based calculation that, providing $\gamma < 1$, gives the expected reward for any policy on any state without requiring calculation of an infinite series.
- We showed that, providing $\gamma < 1$, at least one optimal policy exists, that for any choice of start state or initial action, produces the maximal expected reward if it

is pursued.

- We demonstrated that the value iteration algorithm produces an ε-optimal policy in finite time.
- We showed that the policy iteration algorithm produces a universally optimal policy in finite time.
- We found a small gap in Puterman's proof of the existence of an optimal policy and explicitly detailed how it is eliminated in the finite case.
- We added a case study showing how a simple game can be interpreted in Isabelle as a MDP, thus possessing all of the properties established under that locale.

With regards to the mechanisation effort, the work is split across six Isabelle theories. Two of these are heavily reworked versions of existing theories, namely those associated with the square matrices from the "Matrices for ODEs" entry in the AFP. The remaining four theories establish a relationship between the `vec` and the `bcontfun` types and extend the matrix type with stochastic matrices and some spectral radius results. Altogether, there are almost 7,500 lines of proof script.

We believe that the work provides a framework that is general enough to use as a basis for formally verifying a number of different systems that use MDPs. We demonstrate that we can apply our work to examples of non-trivial processes using the Lava-world game, gaining in our understanding of these processes by doing so.Our primary interest is in reinforcement learning, but because we have used matrices to represent operators on functions over our MDPs, the vast majority of the existing mathematical literature on MDPs should be amenable to formalisation using our work as its foundations.

Isabelle was an effective tool for this purpose. The use of the Isar language breaks our proof into human-readable steps, and means that our mechanical proofs can be human-checked. This gives them greater persuasive weight than a more obtuse theorem proving system might have done.

Difficulties arose when working with types that were present in existing theories but for which we needed different norms or other properties. Those types needed to be rewritten to prove they could be interpreted as an instance of (for example) a normed vector space. We then had to introduce our new properties and rather laboriously reprove many of the existing properties.

It seems mathematically natural, for example, to assume that vectors might be measurable using the operator norm or might be measurable using the Euclidean norm. However, in Isabelle, one would need to define two types of vectors with each norm separately, then build functions to translate between each type. The alternative would be to define a norm as an entirely different function, but that would not inherit the large number of related lemmas that already exist for norms.

If it were possible to show that a type can be interpreted as a normed vector space using multiple norms, it would be easier to work with the same type for different purposes. Unfortunately, because this isn't currently possible, this made our work more difficult.

However, this is merely a matter of convenience, as our work has demonstrated that by type redefinition it is possible to establish different norms and inherit the proofs against an existing type with only a modicum of effort.

The decision to use matrices to represent probability transitions rather than using the Giry monad was fruitful. The most noticeable examples here were our proof that the expected value associated with a policy can be calculated using a matrix inverse multiplication over its immediate reward vector, and our discovery of a minor error in the flow of Puterman's existing proof (for both, see Section 2.5.8). In general this choice made it much easier to mechanise the many proofs in the literature that rely on the properties of matrices.

Chapter 3

Fundamentals of reinforcement learning in Isabelle

3.1 Introduction

We formalised MDPs with rewards as the first step in our formalisation of reinforcement learning systems – specifically Q learning. We discuss reinforcement learning in more detail here, how it works, and our approach in formalising it. In subsequent sections, we go on to formalise the Dvoretzky stochastic approximation theorem, vital to showing that the Q learning algorithm converges to the correct Q values given certain assumptions.

3.1.1 Q learning

Our goal is to formally prove that reinforcement learning processes can learn the true reward function of a general finite MPD and thus provide the optimal policy to navigate it. Q learning was chosen as the focus for the formalisation work, as it is discrete, with well-understood (if not widely discussed) mathematics.

3.1.1.1 A brief introduction to Q learning

Q learning [158] is an early modern approach to reinforcement learning that uses the principles of the Bellman Equation to estimate the true values of the Q reward function. The idea is that once the Q learning agent has converged its estimates to the actual reward values, then a greedy policy with respect to those estimates will be the optimal policy. Since it was first formulated, many variations of Q learning have been proposed

– deep Q learning and its own variations are in widespread use now [147].

Basic Q Learning is a model-free approach to reinforcement learning in the context of a finite MDP with (as expected) discrete time, action and state space. This means that the approach does not attempt to build a model of its environment as it interacts, instead directly estimating the Q value function. This is distinct from the model-based approach, which instead learns a model of the environment via interaction which can then be used to indirectly build a policy function.

$Q_n(s,a)$ is defined as the estimate of the Q values of state s with action a (see section 2.3.1), found after taking n steps of the Q learning process. At each step, n, of that process after action a_n is taken in state s_n, the agent goes to state s_{n+1} and receiving a reward r_n, the estimates for Q are revised as follows:

$$Q_n(s,a) = \begin{cases} (1-\alpha_n)Q_{n-1}(s,a) + \alpha_n(r_n + \gamma V_{n-1}(s_{n+1})) \text{ if } s = s_n \text{ and } a = a_n, \\ Q_{n-1}(s,a) \text{ otherwise} \end{cases} \tag{3.1}$$

Note here that V_n is the estimated value of a state assuming the optimal action is taken – $V_n(s) = \max_a Q_n(s,a)$. Note also that the learning factor α_n used to weigh the relative importance of information just discovered against information already learned at step n. The values $Q_0(s,a)$ are initialised arbitrarily before the agent commences.

3.1.1.2 Q learning formalisation

Formalising a Q learning process requires two main elements in addition to our work in Chapter 2. Firstly, the learning factor α needs to be introduced, and secondly, the update process that shows how the $Q_n(s,a)$ estimates are learned over time must be formalised. In Isabelle we formalise this as a locale:

```
locale Q_Learning_Process = Finite_Markov_Reward_Process +
fixes α :: "nat ⇒ real"
assumes alpha_range:"∀t. 0 < α t ∧ α t < 1"
  and alpha_div: "∀ε. ∃N. ∀n≥N. (∑i≤n. α i) > ε"
  and alpha2_conv:"summable (λt. (α t)²)"
```

Listing 3.1: The `Q_Learning_Process` locale

Note here the assumptions of the locale. There are needed on the learning factor to guarantee that the Q learning agent estimates convergence to the true Q values (as discussed in Section 3.1.1.4). Note that the `summable` predicate is true for some sequence x_i if and only if $\sum_i x_i$ converges.

The policy followed by a Q learning process can be arbitrary, but it must be following some path that is possible within a given MDP. We use the following function, `valid_MDP_paths`, to return the set of all such possible paths. A path here means a sequence of state-action pairs. A path is only valid if at time n, the state for time step $n+1$ is a possible result of the chosen action from time n, and that action is in the list of permissible actions from the state for time step n. This is formalised as follows:

```
definition valid_MDP_paths :: "(nat ⇒ ('a × 'a pmf)) set" where
"valid_MDP_paths = {f. (∀n. fst (f (n+1)) ∈ set_pmf (snd (f n))
                          ∧ snd (f n) ∈ K (fst (f n)))}"
```

Listing 3.2: The `valid_MDP_paths` function

An individual path's type is `nat ⇒ ('a × 'a pmf)`, which corresponds to the definition: the natural numbers index the sequence, which is of state and action pairs using the type variables of the MDP locales. Note that in Isabelle, `fst` returns the first element of a pair and `snd` returns the second.

The last key component is to formalise the process by which the Q_n values are updated. To do this, we formalise a function `Qestimate`. Note that the formalisation here matches the definition for the process given earlier in Section 3.1.1.1, with the values at time step 0 chosen as 0. In the parameters used below, `f` is some path through the MDP – a path which for the purposes of any proofs must be a member of `valid_MDP_paths` – `n` is the time step of the process, `s` is the state and `a` is the action:

```
fun Qestimate :: "(nat ⇒ ('a × 'a pmf)) ⇒ nat
  ⇒ 'a ⇒ 'a pmf ⇒ real" where
"Qestimate f 0 s a = 0" |
"Qestimate f (Suc n) s a = (if s = fst (f n) ∧ a = snd (f n)
  then ((1-(α (Suc n))) * Qestimate f n s a
    + (α (Suc n)) * (R(fst (f n), fst (f (Suc n)), snd (f n))
    + γ * Max (⋃a∈K (fst (f (Suc n))).
    {Qestimate f n (fst (f (Suc n))) a})))
  else Qestimate f n s a)"
```

Listing 3.3: The `Qestimate` function

Note that `Qestimate` recurses through the path sequence. The `if` clause verifies that the state and action are affected by the current time step. If not, the value is left with the value for the previous time step, as per the definition. If so, then they are updated accordingly. Note that the `Max` here finds the maximum estimated value for any action taken in the state being transitioned to; corresponding to the $V(s)$ value in the mathematical definition.

3.1.1.3 Proofs of Q learning convergence

We describe here the two main approaches taken mathematically to proving that Q learning estimates converge to the true values of the MDP:

1. The first is the proof provided by Watkins, who first conceived Q learning in 1989, as detailed in the technical note he wrote 3 years later [157]. If we have an MDP of interest, A, this proof works by constructing a second MDP, $ARP(A)$. It then shows that properties of $ARP(A)$ relate to the Q learning agent and its estimates, and using these properties proves convergence. Watkins's proof has a great deal of structure and little mathematical detail. It is also very specific to this proof.

2. The second is a more straightforward proof which does not rely on building any kind of structure to use as an intermediate step [72]. Instead, it works entirely using stochastic approximation techniques.

After looking at these both in detail, discovering the pre-requisites for each proof and their applicability, we decided to formalise the second. We will discuss our reasons below.

3.1.1.4 Watkins's proof

Watkins's approach uses an ancillary MDP called the action-replay-process (ARP), which is built using the episode sequence of our agent and the learning rate function α_n for which it is assumed $\sum_{n\in\mathbb{N}} \alpha_n = \infty$, $\sum_{n\in\mathbb{N}} \alpha_n^2 < \infty$, and $0 \leq \alpha_n < 1$, as formalised in the locale in Section 3.1.1.2. It is also assumed that there are a finite collection of states, which meets the assumptions of our MDP locale (as discussed in Chapter 2).

The ARP can be seen as an infinite deck of cards, with step 0 at the bottom and the step n increasing as one ascends through the deck. Each step the Q learning agent takes is written on the card corresponding to that step number, $(s_n, a_n, s_{n+1}, r_n, \alpha_n)$. The bottom card instead has the full $Q_0(s,a)$ values that were used to initialise the agent.

A state of the ARP is described as (s,n) where s is a state corresponding to the environment of the Q learning agent, and n is a number corresponding to a time step (and a card from our deck). Note that the s component of the ARP state does not have to be equal to the s_n on the card which corresponds to the n component.

There is some ARP agent that acts over the ARP, as the Q learning agent acts over the underlying MDP.

In a given state of the ARP (s,n), the ARP agent may take any action which would be allowed from s in our Q learning agent environment. Consider the effect such an action will have by using the following rules:

1. Remove all cards from the ARP deck later than n, leaving a finite deck.
2. Go through the remaining cards from the top down.
   (a) If a card corresponding to step m does not have s_m and a_m corresponding to our ARP state s and action a, it is thrown away and the next card processed.
   (b) If the above condition is not met, then with probability α_m the card is replayed. Replaying a card at step m means that the reward r_m is dispensed and the ARP moves to the state $(s_{m+1}, m-1)$. Otherwise it is treated as if the condition was not met and the card is thrown away.
3. If the bottom card is reached, the reward equal to $Q_0(s,a)$ is dispensed and the ARP agent stops.

It can be shown via a sequence of lemmas that the rewards earned from an agent following a policy in $ARP(A)$ tend to those of A, assuming that n is sufficiently large, and thus its optimum Q values do likewise. It can also be shown that the optimum Q values of $ARP(A)$ are the $Q_n(s,a)$ of A itself. Therefore, as $n \to \infty, Q_n(s,a) \to Q^*(s,a)$.

This proof relies on a result from Kushner and Clark (theorem 2.3.1 from their book [86]), which shows that if X_n are updated by $X_{n+1} = X_n + \beta_n(\xi_n - X_n)$ (with β_n constrained as per the learning factor, and ξ_n random variables with mean Ξ), then $\lim_{n\to\infty} X_n = \Xi$.

It also relies on the Arzelà-Ascoli Theorem and the Borel-Cantelli Lemma, both of which are thankfully already proven in Isabelle/HOL but which would need some work to get in the form required for a formalisation of this proof [113, 117].

In addition, to formalise Watkins's proof, the ARP itself and how it functions would have to be modelled – this would require much additional work. In particular, the ARP is not finite (even if the underlying MDP is). Thus, our existing MDP formalisation may need extended to MDPs of infinite states if we formalise Watkins's proof.

To follow this proof, we would need to define a function that outputs an ARP from an arbitrary MDP and prove that it meets the definitions given above, as well as extending our MDP work. The large amount of additional work this requires, which has no real application outside of this specific proof, meant that it was rejected as the basis for a proof of Q learning convergence.

3.1.1.5 Jaakkola's proof

A much simpler approach to proving that the Q learning algorithm converges to the true Q values is taken in the other proof by Jaakkola [72]. This approach has the same assumptions as in Watkins's proof in Section 3.1.1.4, namely that the set of states is finite, and also that (with regard to the learning rate α_n) $\sum_{n\in\mathbb{N}} \alpha_n = \infty$, $\sum_{n\in\mathbb{N}} \alpha_n^2 < \infty$, and $0 \leq \alpha_n < 1$, as we formalised in the locale in Section 3.1.1.2.

Note that the model of a Markov Decision Process used in this proof takes the "reward" for state transition and considers it as a cost, and thus the goal is taken to be to minimise the cost rather than maximise the reward. The reason this approach is taken is not clarified in the paper, but thankfully it is easily modified to fit our requirements with some simple adjustments.

It should also be noted that the paper containing the proof extends it to work for TD(λ) methods too (a different set of reinforcement learning methods related to Q learning) [146].

The proof is founded on two theorems; the first is more of a lemma used in the other, and is reliant on Dvoretzky's stochastic approximation theorem from his 1956 paper [38]. We discuss the details of this theorem in Section 3.3. This first lemma has a proof which makes use of probability spaces and integrations over subsets of the set of events within them.

The second theorem applies the first to the specific case of Q learning and follows easily by showing that the calculations from Q learning meet the criteria of the first theorem.

Overall, it seems that this proof of Q learning convergence is more substantial mathematically than Watkins's proof, but simpler in terms of the structure needed. Furthermore, the Dvoretzky stochastic approximation theorem is also much more generally applicable to a wide variety of problems.

Because of the relative simplicity of this approach, and its superior utility with other problems, we chose to formalise the Dvoretzky stochastic approximation theorem as the first step to a formalisation of Jaakkola's proof, instead of trying the more complicated ARP proof written by Watkins.

3.2 Measure theory and probability theory in Isabelle

To understand and formalise Dvoretzky's stochastic approximation theorem, a solid understanding and formalisation of measure-theoretic probability theory is needed as the theorem is based firmly in this field – as discussed in Section 3.3. There are existing measure theory and probability theory formalisations in Isabelle's libraries that we build upon to obtain what it is needed for our formalisation. We describe those here, beginning with an introduction to the necessary mathematical foundations (in section 3.2.1), and proceeding into a discussion of their existing formalisation in Isabelle and our work extending them (in section 3.2.2).

3.2.1 A very brief introduction to measure theory and probability theory

Measure theory is the field of mathematics that concerns itself with giving precise definitions to methods of measuring sets in some space. Probability theory is the extension of that to cover ways of measuring the probability of random events occurring. We give a brief introduction to the field here required for understanding our formalisations.

3.2.1.1 Measure spaces

A *measure space* is a 3-tuple (X, Σ, μ), where X is some non-empty space, Σ is a σ-algebra over X (a set of subsets of X with certain properties we will shortly discuss), and μ is a measure function from Σ to the real numbers [20].
As Σ is a σ-algebra over X, it possesses the following important properties:

- $X \in \Sigma$.
- $\forall x \in \Sigma.\, (X - x) \in \Sigma$. Note that, as $X \in \Sigma$, the empty set is always in Σ.
- If A is countable with $A \subseteq \Sigma$, then $(\bigcup_{x \in A} x) \in \Sigma$.
- If A is countable with $A \subseteq \Sigma$, then $(\bigcap_{x \in A} x) \in \Sigma$.

μ is a measure, which means it possesses these properties:

- $\forall x.\, \mu(x) \geq 0$.
- $\mu(\{\}) = 0$.

- (countable additivity) For any countable set A of *pairwise-disjoint* elements of Σ, $\mu(\bigcup_{x\in A} x) = (\Sigma_{x\in A}\mu(x))$.

In our work, we will often use the *Borel* measure over the real numbers with their standard topology. The Borel measure is a measure space where all open sets are in the σ-algebra used for the measure.

A *measurable* space is a tuple (X,Σ), essentially a measure space without a measure function.

3.2.1.2 Probability spaces

A *probability space* is a measure space (X,Σ,μ) where $\mu(X) = 1$. Note that as a consequence of this and the other measure properties, $\forall x \in \Sigma.\, 0 \leq \mu(x) \leq 1$. We use the following terminology when we are discussing a probability space: X is the *sample space*, Σ is referred to as the set of *events*, and $\mu(x)$ is the *probability* of event x occurring.

A very simple example is the probability space that describes the roll of a single dice: the sample space is $\{1,2,3,4,5,6\}$, the possible events are the σ-algebra formed by including each possible result, and μ the function that gives the probability of an event occurring on a single roll. So $\mu(\{1,2,3\}) = 0.5$ and $\mu(\{1\}) = 0.166...$ (the probability of rolling a 1 to 3 is 50% and the probability of rolling exactly 1 is 16.6...%).

It is possible to have a probability space in which not every possible result is measured – for example, with reference to the roll of a single dice, the set of events might be $\{\{\},\{1,2,3,4,5,6\},\{1,2,3\},\{4,5,6\}\}$. This describes a probability space where one only cares (or only knows) whether the dice rolls 1 to 3 or 4 to 6; the exact number rolled doesn't matter (or is not available) to us. This is especially important when it comes to *filtrations* as discussed in Section 3.2.4.

3.2.1.3 Almost everywhere

The notation *almost everywhere* or *almost surely* (usually abbreviated to *AE* or *wp 1* meaning *with probability one*) is used to say that the probability of some event not occurring is 0; in other words, that its probability of occurring is 1 [20]. The converse to this is to say that the event occurs *almost nowhere*. However, this does not mean such an event is guaranteed (or in the case of almost nowhere, impossible). An easy example (for almost nowhere) is to randomly pick any natural number (via some theoretical

perfectly random method). The probability of choosing any specific number is 0, but clearly a specific number is chosen.

3.2.1.4 Random variables, expectation, and conditional expectation

A *random variable* is a *measurable* function (see below) from a sample space to a measurable space (in this work, this will always be the real numbers using the Borel measure discussed in Section 3.2.1.1). A function $f : X \to X'$ is measurable from some measure $M = (X, \Sigma, \mu)$ to some measurable space $M' = (X', \Sigma')$ if it is a function from the space of M to the space of M', and for every y in Σ', the pre-image of y under f is in Σ.

This can be understood as "for every possible set of measurable results of the function, the sets from its domain that could have led to those results can be measured". As probabilities across the sample space can be measured, a probability can be associated with the potential results of a random variable.

The expectation of a random variable X can also be found by taking the Lebesgue integral of its results over the sample space – this is usually written as $E(X)$. One can think of this as the "average" result, defined precisely.

A *natural σ-algebra* arising from a random variable is the smallest σ-algebra found wherein that random variable is measurable to the Borel measure on its codomain. By definition, a random variable is always measurable from any measure including this natural σ-algebra to the Borel measure. This can be extended to sets of random variables by taking the union of the natural σ-algebra arising from each, then taking the smallest σ-algebra which includes that union.

The definition of *conditional expectation for the random variable X over a σ-algebra $\mathscr{Y}$*, denoted as $E(X|\mathscr{Y})$, is a function measurable between the probability space restricted to $\mathscr{Y}$ and the Borel measure on the reals, such that, for every event $y \in \mathscr{Y}$, it satisfies:

$$\int_y E(X|\mathscr{Y})dP = \int_y E(X)dP$$

Conditional expectation, intuitively, is a way that one can think about what one might expect as the result of a random variable, given certain conditions. In simple probability theory, this might be something like "given that I am rolling three dice, and one of the dice rolled a six, what is the chance that I roll a 12 in total?"

Note that the conditional expectation itself is a random variable, one that breaks

down the expectation of X across the sample space based on the events in the probability space (in the definition above, the σ-algebra $\mathscr{Y}$). For each event in that set, the conditional expectation returns the integral of X restricted to that event – which one can think of as the "average" result for X over that event.

The conditional expectation of a random variable X on another random variable Y can be defined using Y's natural σ-algebra. Thus, if a particular section of the sample space is selected and used to give a result for Y (which is determinable as Y is measurable using its natural σ-algebra), one can know using $E(X|Y)$ the expectation for X across that section of the sample space, matching our intuitive understanding of conditional expectation.

3.2.2 Measure spaces and probability spaces in Isabelle

We introduce here formalisations in Isabelle of the definitions given above in Section 3.2.1.

3.2.2.1 The `measure` type class

In Isabelle, the existing definition of measure uses a type class over the type variable `'a` [127]. Just as in the mathematical definition, there is a 3-tuple composed of:

1. Ω `::  'a set` corresponding with the space X as discussed in Section 3.2.1.1. For a measure `M` this is referred to as `space M`.

2. `A ::  'a set set` corresponding with the σ-algebra Σ (again, as in Section 3.2.1.1). For a measure `M` this is referred to as `sets M`.

3. μ `::  'a set` $\Rightarrow$ `ennreal` corresponding to the measure μ described in Section 3.2.1.1. The type `ennreal` here represents the extended real numbers in Isabelle, including positive and negative infinity. For a measure `M` this is referred to as `emeasure M`. As we are dealing with probability spaces, which have a finite measure, we will often refer to the measure using `measure M` which is identical in finite measure spaces.

In order to satisfy the properties we defined earlier, a predicate function checks each over each of the components of the 3-tuple to ensure that the properties are met for every member of the type class. As functions in Isabelle must be total (defined against an entire type), the measure of any `'a set` not in `A` is explicitly defined as 0. The definition is given below:

```
definition measure_space :: "'a set ⇒ 'a set set ⇒
  ('a set ⇒ ennreal) ⇒ bool" where
"measure_space Ω A μ ↔
  sigma_algebra Ω A ∧ positive A μ ∧ countably_additive A μ"

typedef 'a measure =
  "{(Ω::'a set, A, μ). (∀a∈-A. μ a = 0) ∧ measure_space Ω A μ }"
```

Listing 3.4: Measure spaces defined in Isabelle

Note that here `sigma_algebra`, `positive` and `countably_additive` are predicates that are true only if the properties defined in Section 3.2.1.1 are met by each component of the tuple.

3.2.2.2 The `prob_space` locale

The existing definition of probability space in Isabelle is a locale with the following specifications [63, 117]:

```
locale sigma_finite_measure =
  fixes M :: "'a measure"
  assumes sigma_finite_countable: "∃A::'a set set.
    countable A ∧
    A ⊆ sets M ∧
    (⋃A) = space M ∧
    (∀a∈A. emeasure M a ≠ ∞)"

locale finite_measure = sigma_finite_measure M for M +
  assumes finite_emeasure_space: "emeasure M (space M) ≠ top"

locale prob_space = finite_measure +
  assumes emeasure_space_1: "emeasure M (space M) = 1"
```

Listing 3.5: Locales used to define probability spaces in Isabelle

The `prob_space` locale assumes that the measure of the total space is 1; it inherits a number of lemmas from the `sigma_finite_measure` and `finite_measure` locales that apply more generally than to probability spaces.

The concept of *almost everywhere* is formalised in Isabelle using filters. It is usually written as `AE x in M. P x` which states that for almost every `x` in the measure `M`'s space, `P x` holds.

There are some abbreviations Isabelle introduces for use in the `prob_space` locale. We list them here as we will refer to them from time to time throughout this work:

```
abbreviation (in prob_space) "events ≡ sets M"
abbreviation (in prob_space) "prob ≡ measure M"
abbreviation (in prob_space)
   "random_variable M' X  ≡ X ∈ measurable M M'"
abbreviation (in prob_space) "expectation ≡ integral^L M"
abbreviation (in prob_space)
   "variance X ≡ integral^L M (λx. (X x - expectation X)^2)"
```

Listing 3.6: Abbreviations in prob_space

Note that `integral^L M f` is the Lebesgue integral of `f` over `M`, and that `measurable M M'` is the set of measurable functions from the space of `M` to the space of `M'` as defined above. This set is defined formally as shown below:

```
definition measurable :: "'a measure ⇒ 'b measure ⇒ ('a ⇒ 'b) set"
  where
"measurable A B = {f ∈ space A → space B.
                    ∀y ∈ sets B. f -` y ∩ space A ∈ sets A}"
```

Listing 3.7: `measurable` definition

Note that the notation `f -` y` refers to the pre-image of `y` under `f`.

In this thesis, we are only concerned with real random variables, so we define them as `random_variable borel X`. The term `borel` here is the Borel measure (the measure defined by taking the smallest σ-algebra which includes every open set, as discussed in Section 3.2.1.1). As the function `X` will be defined as a real-valued function, this Borel measure is implicitly over the reals.

There is an existing function `real_cond_exp` in Isabelle that formalises conditional expectation over a subalgebra [65]. This takes three parameters – `M`, the measure that the random variable is measurable from, `F`, the subalgebra of `M`, and `f`, the random variable whose conditional expectation is being evaluated. It returns the conditional expectation as a function from the space of `M` to the reals; this function is proven to be a random variable.

3.2.3 Integrability of random variables

Throughout our formalisation of theorems in probability theory, there is a repeated issue where the (Lebesgue) integrability of random-variables is ignored in the informal mathematical treatment.

A random variable can be non-integrable, with the result that it has no expectation, as its Lebesgue integral does not exist. In the pen-and-paper versions of the theorems

we are going to be working with, a random variable may be introduced and its expectation given, with no explicit mention of integrability. This is perfectly reasonable: if the expectation is given in the theorem's assumptions, there is an implicit assumption of integrability (as integrability can be defined as the existence of such an expectation).

However, it is not true that the product of two integrable random variables must necessarily be integrable. These products are often used in the pen-and-paper proofs in ways that demand integrability. There is nothing implicit in the presentation of the assumptions to suggest that the product need be integrable also.

For example, a proof might suggest that two random variables X and Y, both with a given expectation $E(X)$ and $E(Y)$ (and thus implicitly integrable), be multiplied together, and the expectation $E(XY)$ be used in some later part of the proof. Obviously, this implies their product must be integrable but does not explicitly say it. This must either be proven or given as an assumption in any formal treatment. However, given we are typically dealing with arbitrary integrable random variables (for which it is not true that their product must be integrable), we must proceed by assumption.

This assumption is unstated in the typical informal presentation of these theorems, as it is widely understood. The demands of a formalisation of these theorems require that the integrability of products of the random variables used as part of the proof be explicit.

In the course of the proofs, we are often dealing with sequences of random variables, f_n. These can be added or multiplied together, or a finite selection of them can be summed and producted together (with those operations occuring in any order). Our central assumption is that any combination of products of random variables taken from the sequence is integrable. From that, we can prove that the various differing combinations of operations, no matter how applied, result in an integrable random variable.

3.2.3.1 The `integrable_mults` lemma

We prove, then, under an assumption that the product of any random variables from a sequence are integrable, that different combinations of operations still result in an integrable random variable, using the `integrable_mults` lemma, as shown below:

```
lemma (in finite_measure) integrable_mults:
fixes f :: "nat ⇒ 'a ⇒ real"
assumes fpint:"∀I. length I > 0 ⟶
  integrable M (λx. (∏i∈{0..<length I}. f (I!i) x))"
shows fint:"∀n. integrable M (f n)"
  and fnint:"∀n m. integrable M (λx. (f n x)^m)"
  and f2int:"∀n. integrable M (λx. (f n x)^2)"
  and SIint:"∀I. length I > 0 ⟶
    integrable M (λx. (∑i∈{0..length I}. f (I!i) x))"
  and SpIint:"∀I. length I > 0 ⟶
    (∀i∈{0..<length I}. length (I!i) > 0) ⟶
      integrable M (λx. (∑i∈{0..<length I}.
        (∏j∈{0..<length (I!i)}. f ((I!i)!j) x)))"
  and SIpint:"∀I. length I > 0 ⟶
    (∀i∈{0..<length I}. length (I!i) > 0) ⟶
      integrable M (λx. (∏i∈{0..<length I}.
        (∑j∈{0..<length (I!i)}. f ((I!i)!j) x)))"
```

Listing 3.8: The `integrable_mults` lemma

Note that our assumption `fpint` takes a list of natural numbers to index random variables from the sequence `f`, asserting that any combination multiplied together will be integrable. We use the notation `∀I. length I > 0` to say that our list `I` is non-empty, and the notation `∏i∈{0..<length I}` to take the product for each element of the list.

It is more normal when seeing such proofs in Isabelle to see a proof over a set rather than a finite, ordered list, but as repetition is a possibility here, and sets do not allow for multiple identical elements, we use a list instead.

Each of the results gives a different set of circumstances under which we can expect integrability:

- `fint` states that each random variable in the sequence is integrable.
- `fnint` says that any of those random variables is integrable under any degree of exponentiation.
- `f2int` makes this explicit for squaring (saving some repetition in later proofs).
- `SIint` says that summing any random variables from the sequence results in an integrable random variable.
- `SpIint` shows that taking arbitrary sums of arbitrary products of random variables from the sequence results in an integrable random variable.

- Lastly, `SIpint` shows that taking arbitrary products of arbitrary sums still guarantees integrability under our assumption.

We have proven this lemma within the `finite_measure` locale, which `prob_space` inherits lemmas from (see Section 3.2.2.2). However, this restriction is only needed for proving `fnint` in the specific case when the exponent `m=0`. If we specify that `m>0`, the restriction to this locale is not needed. However, as our work is only concerned with probability spaces (which are by definition finite measures), the other, more general, version that was proven is not listed here.

The proof of the lemma proceeds through the sub-proof for each statement by instantiating a list that allows for the term in the proof to take the form of the assumption – products of random variables from the sequence over an indexed list. In a few of these sub-proofs, we rely on the fact that summed integrable random variables must also be integrable. However, for the proof of `SIpint`, showing that if one takes arbitrary summations from the sequence and then take the product of those summations, it must also be integrable, relies on a separate lemma.

This lemma, `prod_sum_sum_prod`, states that there is always a way to express a product of summations as a summation of products of the same terms (in some different order, possibly with repetitions). This means that `SIpint` above can be reduced to `SpIint`, which is more easily proven. The statement of the lemma is shown below:

```
lemma prod_sum_sum_prod:
fixes A :: "real list list"
assumes "length A > 0" "∀i<length A. length (A!i) > 0"
shows "∃A'. (∏i∈{0..length (rev A)-1}.
    (∑j∈{0..length ((rev A)!i)-1}. ((rev A)!i)!j)) =
  (∑i∈{0..length A'-1}.
    (∏j∈{0..length (A'!i)-1}. (A'!i)!j))
∧ (⋃i∈{0..<length A}. set (A!i)) = (⋃i∈{0..<length A'}. set (A'!i))
∧ (length A' > 0) ∧ (∀i<length A'. length (A'!i) > 0)"
```

Listing 3.9: The `prod_sum_sum_prod` lemma

3.2.4 A brief introduction to filtrations of sequences of random variables

A *filtration* F in the context of a probability space, is a sequence of σ-algebras over the sample space such that $i \leq j \implies F_i \subseteq F_j$. The intention behind a filtration is to

represent probability spaces where the probability of more events can be measured as the sequence number increases. Filtrations can be used to form a sequence of probability spaces (X, F_i, μ), each step of which represents a space where more events are known (or more are relevant). These are commonly used to represent time advancing, and more events occurring.

Another common case for this is with reference to a sequence of random variables Y_i. A filtration F_i can be formed where each subalgebra is the smallest σ-algebra which contains the union of the sets needed for all Y_j where $j \leq i$ to be measurable from F_i to the real Borel measure (which as mentioned before, are the smallest σ-algebras containing sets of the pre-images in X under Y_i of all the open sets in the reals that are in the range of Y_i). If this is done, then $\forall Y_j.\ j \leq i,\ Y_j$ is measurable using F_i by definition. Such a filtration is called the *natural filtration* associated with the sequence of random variables Y_i.

Note that if a different random variable, Z for example, is measurable using (X, F_n, μ) then it is dependent on the random variables Y_i that are used to construct the filtration. One can measure the probability of events in Z using the probability of events in $\{Y_1 ... Y_n\}$, all of which are measurable under F_n by definition. It is precisely this way of characterising dependence that make filtrations so useful later in our proof of Dvoretzky's stochastic approximation theorem (see Section 3.3.1).

The conditional expectation of a random variable X on a sequence $\{Y_1 ... Y_n\}$ associated with a natural filtration F, is $E(X|F_n)$.

3.2.4.1 Filtrations in Isabelle

The notion of a filtration is formalised already in Isabelle as a locale, but there is no concept of natural filtration over a sequence of random variables [61]. The definition is given below:

```
locale filtration =
  fixes Ω :: "'a set" and F :: "'t::{linorder_topology,
     second_countable_topology} ⇒ 'a measure"
  assumes space_F: "∧i. space (F i) = Ω"
  assumes sets_F_mono: "∧i j. i ≤ j ⟹ sets (F i) ≤ sets (F j)"
```

Listing 3.10: The `filtration` locale

Note an important distinction here from the definition given before. Here, the filtration is defined as a sequence of measures rather than a sequence of σ-algebras.

This difference preserves the σ-algebras and if we were to restrict it to the sets of each measure, the definition would be identical.

To build a notion of a natural filtration in Isabelle, the natural σ-algebra associated with a sequence of random variables up to an index `n` is defined as follows:

```
definition sigma_sequence  :: "nat ⇒ 'a set ⇒ (nat ⇒ 'a ⇒ 'b)
  ⇒ 'b measure ⇒ 'a set set" where
"sigma_sequence n X f M =
  sigma_sets X (⋃i∈{0..n}. sigma_sets X
    {(f i) -` A ∩ X | A. A ∈ sets M})"
```

Listing 3.11: The `sigma_sequence` function

where `⋃i∈{0..n}` is the union for all natural numbers up to `n`. `(f i) -` A` is the pre-image of `A` under `f i`.

The `sigma_sets X A` function returns the smallest σ-algebra in the space `X` that contains all the sets in `A`. The `sigma_sequence` function takes a natural number `n`, a space `X`, a sequence of random variables `f` and a measure over their codomain type `M`. It generates the smallest σ-algebra for each random variable in the sequence up to and including `n`, then returns the smallest σ-algebra over the union of all these sets.

To show that this returns the sequence of σ-algebras needed to build the natural filtration, a sequence of associated properties must be proven. These are necessary to show that the function generates a σ-algebra that matches the usual mathematical definition, and also for proving later that the natural filtration for a sequence of random variables can be made using it.

We begin by showing that using `sigma_sets` over a random variable always returns a subset of the sets for the measure that the random variable is measurable from. We also show that this is a subset of the power set of the measure's space:

```
lemma sigma_variable_subalgebra:
fixes X :: "'a measure" and f :: "'a ⇒ 'b"
  and M :: "'b measure"
assumes "f ∈ measurable X M"
shows
  "sigma_sets (space X) {f -` A ∩ (space X) | A. A∈sets M} ⊆ sets X"
  "sigma_sets (space X) {f -` A ∩ (space X) | A. A∈sets M}
    ⊆ Pow (space X)"
```

Listing 3.12: The `sigma_variable_subalgebra` lemma

Note that `Pow (space X)` is the power set of the space of `X`.

We continue by proving that the sigma_sequence lemma with a natural number parameter of 0 generates the natural σ-algebra associated with the first random variable in the sequence:

```
lemma sigma_sequence_0:
fixes X :: "'a measure" and f :: "nat ⇒ 'a ⇒ 'b"
  and M :: "'b measure"
assumes "f 0 ∈ measurable X M"
shows "sigma_sequence 0 (space X) f M = sigma_sets (space X)
  {f 0 -` A ∩ (space X) | A. A ∈ sets M}"
```

Listing 3.13: The sigma_sequence_0 lemma

Next, we show that for any n, the sigma_sequence function for n is a subset of that for Suc n, or n+1:

```
lemma sigma_sequence_subsets:
fixes X :: "'a set" and f :: "nat ⇒ 'a ⇒ 'b"
  and M :: "'b measure" and n :: nat
shows "sigma_sequence n X f M ⊆ sigma_sequence (Suc n) X f M"
```

Listing 3.14: The sigma_sequence_subsets lemma

And that for a sequence of random variables, the sigma_sequence function for some measure X returns a subset of the sets of X:

```
lemma sigma_sequence_subalgebra:
fixes X :: "'a measure" and f :: "nat ⇒ 'a ⇒ 'b"
  and M :: "'b measure" and n :: nat
assumes "∀n. f n ∈ measurable X M"
shows "sigma_sequence n (space X) f M ⊆ sets X"
```

Listing 3.15: The sigma_sequence_subalgebra lemma

Lastly, we show that the sigma_sets function over the sigma_sequence function is the identity:

```
lemma sigma_sets_sigma_sequence:
fixes X :: "'a measure" and f :: "nat ⇒ 'a ⇒ 'b"
  and M :: "'b measure" and n :: nat
assumes "∀n. f n ∈ measurable X M"
shows "sigma_sets (space X) (sigma_sequence n (space X) f M)
  = sigma_sequence n (space X) f M"
```

Listing 3.16: The sigma_sets_sigma_sequence lemma

We now describe the filtration_sequence function which creates a sequence of measure spaces where each σ-algebra is that returned by the appropriate sigma_sequence.

This takes as parameters `n`, the index of the measure in the sequence; `X`, the measure whose space is the domain; `f`, the sequence of functions measurable from `X` to `M`; and `M`, the measure that whose space is the range for each `f`.

```
definition filtration_sequence ::
  "nat ⇒ 'a measure ⇒ (nat ⇒ 'a ⇒ 'b) ⇒ 'b measure ⇒ 'a measure"
    where
"filtration_sequence n X f M
  = measure_of (space X) (sigma_sequence n (space X) f M)
    (emeasure X)"
```

Listing 3.17: The `filtration_sequence` function

The `measure_of` function is used, which forms a measure tuple from its three parameters. With this defined, we need to show that it has the properties expected of it and several properties that will be useful in our later proofs (see Section 3.3). We begin by showing that the sequence formed via this function is in fact a filtration using the existing Isabelle definition given in Section 3.2.4.1:

```
lemma filtration_sequence_filtration:
fixes X :: "'a measure" and f :: "nat ⇒ 'a ⇒ 'b"
  and M :: "'b measure"
assumes "∀n. f n ∈ measurable X M"
shows "filtration (space X) (λn. filtration_sequence n X f M)"
  "∀n. subalgebra X (filtration_sequence n X f M)"
  "∀i. sets (filtration_sequence i X f M)
    = sigma_sequence i (space X) f M"
```

Listing 3.18: The `filtration_sequence_filtration` lemma

The `subalgebra` function here is a predicate that verifies that its second parameter is a subalgebra of the first. The `emeasure` functions of these two measures most likely will not be exactly the same – recall that for the measure function `emeasure` in Isabelle, `(x ∉ sets X) ⟹ (emeasure X x = 0)`. So, if there are sets with positive measure in `X` that are not in the sets of `filtration_sequence n X f M`, these measure functions must return different values on those sets.

This lemma shows three important properties: that the function creates a filtration sequence as expected, that the set of events for each measure in the filtration sequence thus formed is a subalgebra of that of the measure `X`, and that these sets for each measure in the sequence are those defined by the `sigma_sequence` function. This last property may seem trivial, but it must be verified that the sets passed into the `measure_of` function will not be altered in the measure it produces.

Next we show that if the measure function is restricted to sets in the measures from the filtration sequence, it produces identical results to those sets in the underlying measure `X`:

```
lemma filtration_sequence_measure:
fixes X :: "'a measure" and f :: "nat ⇒ 'a ⇒ 'b"
  and A :: "'a set" and M :: "'b measure" and n :: nat
assumes "∀n. f n ∈ measurable X M"
  "A∈sets (filtration_sequence n X f M)"
shows "emeasure (filtration_sequence n X f M) A = emeasure X A"
```

Listing 3.19: The `filtration_sequence_measure` lemma

We go on to show that any random variable from the sequence is measurable under a given measure from the filtration sequence if its index is lower than that of the measure's:

```
lemma filtration_sequence_measurable:
fixes X :: "'a measure" and f :: "nat ⇒ 'a ⇒ 'b"
  and M :: "'b measure" and n m :: nat
assumes "∀n. f n ∈ measurable X M"
  "n ≤ m"
shows "f n ∈ measurable (filtration_sequence m X f M) M"
```

Listing 3.20: The `filtration_sequence_measure` lemma

Lastly, we show that the set of events for the measures formed by the filtration sequence are subalgebras of each other, as required by the `filtration` predicate. This is a trivial corollary of the `filtration_sequence_filtration` lemma, but spelling it out here prevents much repetition in later proofs: the specific result is often required.

```
lemma filtration_sequence_nested:
fixes X :: "'a measure" and f :: "nat ⇒ 'a ⇒ 'b"
  and M :: "'b measure" and n m :: nat
assumes "∀n. f n ∈ measurable X M"
  "n ≤ m"
shows "subalgebra (filtration_sequence m X f M)
  (filtration_sequence n X f M)"
```

Listing 3.21: The `filtration_sequence_nested` lemma

3.3 Dvoretzky's stochastic approximation in Isabelle

Dvoretzky's stochastic approximation theorem is fundamental to our proof of Q learning convergence and an important result in probability theory. We discuss it and its formalisation in this section.

3.3.1 A brief introduction to Dvoretzky's stochastic approximation

Dvoretzky's stochastic approximation theorem [38] gives a set of criteria under which a sequence of random variables will almost surely converge to 0. These seem fairly complicated at first glance, but actually represent a fairly common set of conditions for sequences of random variables. It is the evolution of a set of other more specific stochastic approximation algorithms to which it is the general case [128, 76]. It also applies well to the convergence of Q learning estimates to the underlying reward values in a Markov decision process, which is our interest here.

The theorem was stated at first in 1956 with an elaborate proof that spanned five pages [38]. After its publication, several further papers simplified the proof, in general by referring to prior results (in particular, results dependent on Kolmogorov's inequality for dependent random variables, formalised in Section 3.3.3) and clarifying the approach [166, 36]. We use the proof by Derman and Sacks as the basis of our formalisation here, as it is the most straightforward proof [36].

We describe the theorem below, and in subsequent sections will proceed through each of the required lemmas and theorems in turn, describing them and their formalisations, before concluding this chapter with the formalisation of the Dvoretzky stochastic approximation theorem itself in Section 3.3.6.

The theorem itself states that if there are sequences of random variables $\{X_n\}, \{T_n\}$ and $\{Y_n\}$, and sequences of numbers $\{\alpha_n\}, \{\beta_n\}$ and $\{\gamma_n\}$, with the following assumptions:

1. $\{\alpha_n\}, \{\beta_n\}$ and $\{\gamma_n\}$ are positive for all n.
2. $\lim_{n\to\infty} \alpha_n = 0$.
3. $(\Sigma_i \beta_i)$ converges.
4. $(\Sigma_i \gamma_i)$ diverges to positive infinity.
5. $\forall n.\, T_n$ and Y_n are both dependent on $\{X_1 \ldots X_n\}$.
6. $\forall n.\, X_{n+1} = T_n + Y_n$.
7. $\forall n.\, E(Y_n | X_1 \ldots X_n) = 0$ almost surely.

8. $(\Sigma_i E(Y_i^2))$ converges.
9. $T_n \leq \max(\alpha_n, (1+\beta_n)|X_n| - \gamma_n)$.

Then almost surely, $X_n \to 0$ as $n \to \infty$. An outline of the proof follows:

Find a sequence α'_n which satisfies the properties given in the assumptions for α_n, but additionally makes

$$\sum_{n=0}^{\infty} \frac{E(Y_n^2)}{\alpha_n'^2} \tag{3.2}$$

convergent. We do this using a variant of the Du Bois-Reymond convergence test for non-negative sequences (which we discuss and formalise in Section 3.3.5).

Define a sequence of random variables $Z_n = Y_n \operatorname{sgn}(T_n)$. We can substitute Z_n in place of Y_n in assumptions 7 and 8, and in the sum 3.2, preserving its convergence. Using the first two of these properties, and a set of criteria for showing convergence of sequences of random variables (which we discuss and formalise in Section 3.3.4), we then show that $\sum_n Z_n$ is convergent almost everywhere. This last property, along with the Borel-Cantelli lemma and the Chebyshev inequality [20] shows that

$$(\exists N.\, \forall n \geq N.\, \|Z_n\| \leq \alpha'_n) \text{ almost everywhere} \tag{3.3}$$

From assumption 6 and equation 3.3, we have that

$$\begin{aligned} \text{if } |T_n| \leq \alpha'_n,\ |X_{n+1}| &\leq 2\alpha'_n \\ \text{if } |T_n| > \alpha'_n, |X_{n+1}| &= |T_n| + Z_n \end{aligned} \tag{3.4}$$

Therefore,

$$\begin{aligned} |X_{n+1}| &\leq \max(2\alpha'_n, |T_n| + Z_n) \\ &\leq \max(2\alpha'_n, (1+\beta_n)|X_n| + Z_n - \gamma_n \end{aligned} \tag{3.5}$$

for sufficiently large n, almost everywhere. We then use this with a preparatory lemma regarding convergence of real sequences (which we discuss and formalise in Section 3.3.2) to finish the proof. □

Throughout the proof, Derman and Sacks are not completely explicit about the theorems and lemmas they rely on in each step (with the exception of the Borel-Cantelli lemma, the Chebyshev inequality and the preparatory lemma). In particular, the use of the Du Bois-Reymond convergence test and the convergence criteria for random variables is not made clear.

3.3.2 Preparatory lemma

Derman and Sacks begin their proof by outlining a preparatory lemma necessary for the later proof. As in the main theorem, this involves a large set of assumptions.

Let $\{a_n\}, \{b_n\}, \{c_n\}, \{\delta_n\}$ and $\{\xi_n\}$ be sequences of real numbers such that:

1. $\{a_n\}, \{b_n\}, \{c_n\}$ and $\{\xi_n\}$ are non-negative for all n.
2. $\lim_{n\to\infty} a_n = 0$.
3. $(\Sigma_i b_i)$ converges.
4. $(\Sigma_i c_i)$ diverges to positive infinity.
5. $(\Sigma_i \delta_i)$ converges.
6. $\exists N.\, \forall n > N.\, \xi_{n+1} \leq \max(a_n, (1+b_n)\xi_n + \delta_n - c_n)$

Then $\lim_{n\to\infty} \xi_n = 0$. The proof given by Derman and Sacks is as follows:

Obtain N_0 such that $\forall n > N_0.\, \xi_{n+1} \leq \max(a_n, (1+b_n)\xi_n + \delta_n - c_n)$ holds, as per assumption 6. Then fix some $N > N_0$. Have $B_n = \prod_{i=0}^{n}(1+b_i)$. From some $n > N$, work backwards using $\forall n > N_0.\, \xi_{n+1} \leq \max(a_n, (1+b_n)\xi_n + \delta_n - c_n)$ to N, giving

$$\xi_{n+1} \leq \max\Big(\frac{B_n}{B_{N-1}}\xi_N + B_n \sum_{j=N}^{n} \frac{\delta_j - c_j}{B_j}, \max_{N\leq k\leq n}\Big[\frac{B_n}{B_k}a_k + B_n \sum_{j=k+1}^{n} \frac{\delta_j - c_j}{B_j}\Big]\Big) \tag{3.6}$$

The assumptions imply that B_n increases to some finite B. We can then show (using the Dirichlet test [22], although that is not expressed in the Derman and Sacks proof) that $\sum_{j=1}^{\infty} \frac{\delta_j}{B_j}$ converges and $\sum_{j=1}^{\infty} \frac{c_j}{B_j}$ diverges to positive infinity. Because $(B_n/B_{N-1})\xi_N$ is finite, $\frac{B_n}{B_{N-1}}\xi_N + B_n \sum_{j=N}^{n} \frac{\delta_j - c_j}{B_j}$ must eventually be negative as n increases. Thus it can be ignored in Equation 3.6 for large n, giving

$$\begin{aligned} \xi_{n+1} &\leq \max_{N\leq k\leq n}\Big[\frac{B_n}{B_k}a_k + B_n \sum_{j=k+1}^{n} \frac{\delta_j - c_j}{B_j}\Big] \\ &\leq B\Big(\max_{k\geq N} a_k + \max_{N\leq k\leq n}\Big|\sum_{j=k+1}^{n} \frac{\delta_j}{B_j}\Big|\Big) \end{aligned} \tag{3.7}$$

From the convergence of $\sum_{j=1}^{\infty} \frac{\delta_j}{B_j}$ and assumption 2, the total in Equation 3.7 tends to 0 as $N \to \infty$, proving the lemma. □

To make the proof slightly easier to formalise, several properties of the B_i sequence described above were moved into a separate lemma, `B_props`. The reasoning for this was to separate out those proofs into a single lemma where any reusable properties of B could be established under the minimal assumptions (assumptions 3 and 1 from the list above) needed to define the b sequence. That lemma is stated here:

```
lemma B_props:
fixes b :: "nat ⇒ real"
defines "B ≡ (λn. (∏i∈{0..n}. 1 + b i))"
assumes "∀n. b n ≥ 0" "summable b"
shows "∀n. B n > 0"
  and "∀n. B n ≥ 1"
  and "incseq B"
  and "decseq (λn. 1/(B n))"
  and "∀n m. B (m+n)/B m = (∏i∈{m+1..n+m}. (1 + (b i)))"
  and "convergent B"
  and "∃L≥1. B → L"
  and "∃L>0. L≤1 ∧ (λn. 1 / (B n)) → L"
```

Listing 3.22: The `B_props` lemma

The assumptions here are limited to the assumptions of the preparatory lemma which involve the b sequence, as these are the only ones relevant to the B sequence. The properties shown include some of the assertions made in the pen-and-paper proof for B, which are reused later in the formalisation for the preparatory lemma.

The Dirichlet test is vital to the lemma. This test states that if there are two sequences a and b, with a decreasing to 0 and $\sum_{i=0}^{n} b_i$ bounded above for all n, then $\sum_i a_i b_i$ converges. There was no existing treatment for the Dirichlet test in Isabelle, so we formalised it as shown below:

```
lemma dirichlet_test:
fixes a b :: "nat ⇒ real"
assumes a_mono_d:"antimono a" and a_lim:"a → 0"
  and b_sum_bounded:"∃M. ∀n. |sum b {0..n}| ≤ M"
shows "summable (λn. a n * b n)"
```

Listing 3.23: The `dirichlet_test` lemma

This was a relatively straight-forward proof using the sandwich lemma, but required an additional lemma proving that summation by parts can be used to rewrite sequences:

```
lemma summation_by_parts:
fixes f g F :: "nat ⇒ 'b::linordered_idom" and m n :: nat
assumes F_defn:"F = (λk. sum f {..k})"
shows "sum (λk. f k * g k) {m..m+Suc n} =
       F (m+Suc n) * g (m+Suc n) -
       (if m=0 then 0 else 1)*(F (m-1) * g m) +
       (sum (λk. F k * (g k - g (Suc k))) {m..m+n})"
```

Listing 3.24: The `summation_by_parts` lemma

The proof here was separated into cases where `m=0` and where it is not, but otherwise simply involved the rewriting of the terms involved. Note that the `if` clause allows explicit specification of the behaviour when `m=0`.

With all the components in place, we can proceed with the formalisation of the preparatory lemma. In Isabelle, the lemma is stated as shown below:

```
lemma prep_for_Dvoretzky:
fixes a b c δ ξ :: "nat ⇒ real"
assumes "∀n. a n ≥ 0"
  and "∀n. b n ≥ 0"
  and "∀n. c n ≥ 0"
  and "∀n. ξ n ≥ 0"
  and "a → 0"
  and "summable b"
  and "∀ε. ∃N. ∀n≥N. (Σi≤n. c i) > ε"
  and "summable δ"
  and "∃N. ∀n>N. ξ (Suc n)
    ≤ max (a n) (((1 + (b n)) * (ξ n)) + (δ n) - (c n))"
shows "ξ → 0"
```

Listing 3.25: The `prep_for_Dvoretzky` lemma

The formal statement of the lemma fairly clearly corresponds to the informal version. Recall that the `summable` predicate is true for some sequence `x` if and only if its sum over its index converges.

The proof involves extensive rewriting of the sequence terms, proceeding by making substitutions (as outlined in the pen-and-paper proof) after proving they are correct. The results of the `B_props` lemma are used throughout, and the `dirichlet_test` is used as in the pen-and-paper proof.

3.3.3 Kolmogorov's inequality

Kolmogorov's inequality, although not specifically addressed or named, is a key component of the Derman and Sacks proof [20]. In particular, the result is vital in proving that the set of convergence criteria discussed in Section 3.3.4 are valid. The inequality is a result in probability theory which gives a way to evaluate the probability of the partial sums of a sequence of independent random variables being greater than or equal to some value λ in terms of that value and its variance.

In mathematical terms, if there is a probability space (P, Σ, μ), and a sequence of independent random variables $\{X_n\}$, each with $E(X_n) = 0$ and finite variance, and $S_n = \sum_{i=0}^{n} X_i$, then for every $\varepsilon > 0$:

$$\mu\Big(\max_{1 \leq k \leq n} |S_k| \geq \varepsilon\Big) \leq \frac{1}{\varepsilon^2} \sum_{k=1}^{n} E(X_k^2) \tag{3.8}$$

There is one major problem with this version of the inequality that make it unsuitable for our needs with regard to our later proofs. The sequences of random variables we will be working with, as mentioned in the earlier description of Dvoretzky's theorem (in section 3.3.1), are not independent, but usually depend on other sequences of random variables. So the assumptions of independence and expectation made here are too strict for our work.

A version of the inequality where the random variables can be dependent is needed, meaning that the expectation assumption also must change so that $E(X_{n+1}|F_n) = 0$, where F is the natural filtration of the sequence of random variables that X is dependent on (we discuss natural filtrations in Section 3.2.4). There is an existing extension of the Kolmogorov inequality proof where each random variable X_{n+1} is dependent on $\{X_1 ... X_n\}$, not some potentially different sequence of random variables [98]. Thankfully, it is straightforward to adjust this version of the inequality (and its proof) so that the assumption of conditional expectation is against other sequences of random variables as necessary.

This means we can prove a more general version of the Kolmogorov inequality that does not require independent random variables or that the conditional expectation of those variables be against their own natural filtration, but against an arbitrary natural filtration:

```
lemma (in prob_space) kolmogorov_inequality:
fixes f g :: "nat ⇒ 'a ⇒ real"
  and m :: nat and ε :: real
assumes fpint:"∀I. length I > 0 ⟶
    integrable M (λx. (∏i∈{0..<length I}. f (I!i) x))"
  and gint:"∀n. integrable M (g n)"
  and fexp:"∀n. AE x in M. real_cond_exp M
    (filtration_sequence n M g borel) (f (Suc n)) x = 0"
  and epos:"ε>0"
  and fadg:"∀n. f n ∈ borel_measurable (filtration_sequence n M g
     borel)"
shows
  "prob {x∈space M. Max ((λk. |(∑i∈{0..k}. f i x)|) ` {0..m}) ≥ ε}
    ≤ (∑i∈{0..m}. expectation (λx. (f i x)^2)) / (ε^2)"
```

Listing 3.26: The `kolmogorov_inequality` lemma

We add a sequence of random variables `g` here, which, apart from being integrable, is entirely arbitrary. This is used to construct an arbitrary natural filtration that the sequence `f` (corresponding to X_n in the informal description above) is dependent on, removing the dependence condition of the assumptions given there.

The assumptions here relate to the statement of the inequality in the following ways: `fpint` is our assumption that the arbitrary products of integrable random variables from the `f` sequence are also integrable, as discussed in Section 3.2.3.1. `gint` asserts that the random variables in the `g` sequence are integrable. `fexp` gives the conditional expectation assumption for `f` over the natural filtration of `g`. `epos` asserts that ε is positive, and `fadg` says that `f` is measurable against the natural filtration for `g`.

The proof here gets around the independence condition by using disjoint events to measure the probabilities over; to do this we use the `disjointed` function in Isabelle's libraries [62]. This function takes a sequence of sets `A`, and returns a sequence of pairwise disjoint sets such that the statement

```
(⋃i∈{0..n}.  A i) = (⋃i∈{0..n}.  disjointed A i)
```

holds. We are required to prove that the disjoint events are still measurable in the same probability space, which is done using the σ-algebra properties. The `disjointed` function definition is below:

```
definition disjointed :: "(nat ⇒ 'a set) ⇒ nat ⇒ 'a set" where
  "disjointed A n = A n - (⋃i∈{0..<n}. A i)"
```

Listing 3.27: The `disjointed` function

Using this approach is useful with some of the other proofs we work on – in particular, accessing the countable additive property of measures requires that the sets being measured are pointwise disjoint: this is not something we can assume about arbitrary events of interest but which can be induced on them using the `disjointed` function. This was important to the proofs of `prob_sets_limits`, `union_from_n_finite`, and `union_from_n`, as discussed in Section 3.3.4.

To prove Kolmogorov's inequality, a few more additions to Isabelle's existing probability theory are needed – in particular, we show that if $E(X_{n+1}|F_n) = x$, then $\forall k > n. E(X_k|F_n) = x$, using the following lemma:

```
lemma (in finite_measure) real_cond_exp_Suc_r_imp_AE_geq:
fixes f g :: "nat ⇒ 'a ⇒ real" and r :: real
assumes fint:"∀n. integrable M (f n)" and
  gint:"∀n. integrable M (g n)" and
  fexp:"∀n. AE x in M.
    real_cond_exp M (filtration_sequence n M g borel)
      (f (Suc n)) x = r"
shows "∀n. ∀k>n. AE x in M.
  (real_cond_exp M (filtration_sequence n M g borel) (f k)) x = r"
```

Listing 3.28: The `real_cond_exp_Suc_r_imp_AE_geq` lemma

The sequence `g` here is the arbitrary sequence of random variables whose natural filtration is represented by F_n in the statement above. `f` is the sequence of random variables whose conditional expectation we are interested in. The assumption `fexp` expresses that the condition expectation for `f` at an index one higher than the (arbitrary) natural filtration it is being evaluated over is some fixed value – and the lemma shows that this holds for all higher indices of `f` as well.

The proof of the lemma hinges on showing that if one fixes the value `n` then the measure `filtration_sequence n M g borel` is a subalgebra of `filtration_sequence k M g borel`. Once this has been done, then `real_cond_exp_nested_subalg`, an existing lemma, can be used to show that the conditional expectation over both is the same [65].

3.3.4 A set of convergence criteria for sequences of random variables

During a particular section of the Derman and Sacks proof, they assert that since $E(Y_n|X_1...X_n) = 0$ and $\sum_{i=0}^{\infty} E(Y^2)$ converges, then it must be true that $\sum_{i=0}^{\infty} E(Y)$ con-

verges too. This is not at all trivial, although nothing is mentioned in the proof to support the assertion. A version of the proof required to support the claim can be found in section 6.2.1 of Ash's *Probability and Measure Theory* [7].

However, as with the proof of the Kolmogorov inequality, this assumes independence of the random variables and requires some reworking. Unfortunately, no version of the proof rewritten for dependent random variables was available, but thankfully the adjustments required were not too onerous: our modified version of the Kolmogorov inequality handles them, so much of the work required has been done already.

```
theorem (in prob_space) rv_convergence_crit:
fixes X :: "nat ⇒ 'a ⇒ real"
assumes X2sum: "summable (λn. expectation (λx. (X n x)^2))"
  and Xexp:"∀n. AE x in M. real_cond_exp M
    (filtration_sequence n M X borel) (X (Suc n)) x = 0"
  and Xexp0:"expectation (X 0) = 0"
  and Xpint:"∀I. length I > 0 ⟶
    integrable M (λx. (∏i∈{0..<length I}. X (I!i) x))"
shows "AE x in M. summable (λn. X n x)"
```

Listing 3.29: The `rv_convergence_crit` theorem

This proof depends on a number of other results in measure theory that needed to be proven for this work. We discuss these results and show their formalisations in the remainder of this section. The following lemma, `LIMSEQ_meas_conv_Un`, shows that if a countably infinite union of measurable and increasing sets equals some set B, then their measure tends to that of B as the union increases to infinity:

```
lemma (in finite_measure) LIMSEQ_meas_conv_Un:
fixes A :: "nat ⇒ 'a set"
defines "B≡(⋃n∈{0::nat..}. A n)"
assumes  "∀n. A n ∈ sets M"
  and "∀n. A n ⊆ A (Suc n)"
shows "(λn. measure M (A n)) → measure M B"
```

Listing 3.30: The `LIMSEQ_meas_conv_Un` lemma

The `LIMSEQ_meas_conv_Un` lemma is used to prove the following lemma, showing that if a countably infinite intersection of measurable decreasing sets equals some set B, the same result holds. The result over unions relates to that over intersections by set negation, making the proof, given the lemma above, trivial:

```
lemma (in finite_measure) LIMSEQ_meas_conv_In:
fixes A :: "nat ⇒ 'a set"
defines "B≡(⋂n∈{0::nat..}. A n)"
assumes "∀n. A n ∈ sets M"
  and "∀n. A (Suc n) ⊆ A n"
shows "(λn. measure M (A n)) → measure M B"
```

Listing 3.31: The `LIMSEQ_meas_conv_In` lemma

The next lemma, `LIMSEQ_meas_seq`, states that

$$\lim_{n\to\infty} X_n = X' \text{ almost everywhere}$$

$$\iff (\forall \delta > 0, \lim_{n\to\infty} \mu\big(\bigcup_{i=n}^{\infty} |X_i - X'| \geq \delta\big) = 0 \text{ almost everywhere}$$

Both of the lemmas above are used in the proof:

```
lemma (in prob_space) LIMSEQ_meas_seq:
fixes X :: "nat ⇒ 'a ⇒ real" and X' :: "'a ⇒ real"
assumes "∀n. X n ∈ borel_measurable M"
  "X' ∈ borel_measurable M"
shows "(AE x in M. (λn. X n x) → (X' x))
  = (∀δ>0. (λn. measure M (⋃i∈{n..}.
    {x∈space M. |X i x - X' x| ≥ δ})) → 0)"
```

Listing 3.32: The `LIMSEQ_meas_seq` lemma

And this related lemma, `Cauchy_meas_seq`, says that if a sequence of random variables X_n is Cauchy almost everywhere, this is equivalent to the statement that $\forall \delta > 0.\ \lim_{n\to\infty} \mu\big(\bigcup_{j,k\geq n}\{|X_j - X_k| \geq \delta\}\big) = 0$. This uses all of the preceding results, and gives us a useful way to express the Cauchy property for sequences of random variables:

```
lemma (in prob_space) Cauchy_meas_seq:
fixes f :: "nat ⇒ 'a ⇒ real"
assumes "∀n. f n ∈ borel_measurable M"
shows "(AE x in M. Cauchy (λn. f n x))
  =  (∀δ>0. (λn. measure M (⋃j∈{n..}. (⋃k∈{n..}.
    {x∈space M. |f j x - f k x| ≥ δ}))) → 0)"
```

Listing 3.33: The `Cauchy_meas_seq` lemma

The next lemma says that if some random variable $Y_n = X_{n+m}$, then its natural filtration G_n is a subalgebra of X's natural filtration F_{n+m}. Once this is known, we can use `real_cond_exp_nested_subalgebra` to relate conditional expectations across them in the next lemma:

```
lemma filtration_shift_subalgebra:
assumes "∀n. X n ∈ measurable M borel"
shows "subalgebra (filtration_sequence (n+m) M X borel)
  (filtration_sequence n M (λi. X (i+m)) borel)"
```

Listing 3.34: The `filtration_shift_subalgebra` theorem

The above is used to prove the next lemma, which says that if, almost everywhere, $E(X_{n+1}|G_n) = 0$ (where G is the natural filtration for Y), then for some random variable sequences X', Y' such that $X'_n = X_{n+m}, Y'_n = Y_{n+m}$, (with G' the natural filtration for Y'), then almost everywhere, $E(X'_{n+1}|G'_n) = 0$. This is used with the Kolmogorov inequality in the proof of `rv_convergence_crit`, as the indices of the random variables being used in the inequality are arbitrarily higher than a fixed value used earlier in the proof:

```
lemma (in prob_space) real_cond_exp_filtration_shift:
fixes X Y :: "nat ⇒ 'a ⇒ real"
assumes Xexp:"∀n. AE x in M. real_cond_exp M (filtration_sequence n
    M Y borel) (X (Suc n)) x = 0"
  and Yint:"∀n. integrable M (Y n)"
  and Xint:"∀n. integrable M (X n)"
shows "∀n m. AE x in M. real_cond_exp M
  (filtration_sequence n M (λi. Y (m+i)) borel)
    ((λi. X (m+i)) (Suc n)) x = 0"
```

Listing 3.35: The `real_cond_exp_filtration_shift` lemma

There are also several lemmas needed that show how measuring probability over a union (possibly infinite) of sets works. The next one shows how to express the probability of a finite union of events as a sum of the probabilities over the disjoined events. The use of the `disjointed` function introduced in Section 3.3.3 is vital to the proof. It is fundamental to showing that the probability of a union of events can be related to a summation of the probabilities of pairwise disjointed events, needed in particular to use the properties of infinite summations for the next lemma:

```
lemma (in prob_space) union_from_n_finite:
  fixes f :: "nat ⇒ 'a set" and n m :: nat
  assumes "range f ⊆ events"
  shows "prob (⋃j∈{n..n+m}. f j)
    = (∑j∈{0..m}. prob (disjointed (λj. f (j+n)) j))"
```

Listing 3.36: The `union_from_n_finite` lemma

The next result expresses the probability of an infinite union of events as an infinite sum of the probability over the disjoined events. This uses the previous lemma for its

proof, which is then straightforward:

```
lemma (in prob_space) union_from_n:
  fixes f :: "nat ⇒ 'a set" and n :: nat
  assumes "range f ⊆ events"
  shows "prob (⋃j∈{n..}. f j)
    = (∑j. prob (disjointed (λj. f (j+n)) j))"
```

Listing 3.37: The `union_from_n` lemma

The `union_from_n_finite` and `union_from_n` lemmas taken together give us the following important result. This shows that we can express the probability of an infinite union of events as the limit of the probability of a sequence of finite unions. This makes reasoning over infinite unions, as required throughout the proof of `rv_convergence_crit`, much easier:

```
lemma (in prob_space) prob_UNION:
  fixes f :: "nat ⇒ 'a set"
  assumes "range f ⊆ events"
  shows "prob (⋃k∈{0..}. f k) = lim (λm. prob (⋃k∈{0..m}. f k))"
```

Listing 3.38: The `prob_UNION` lemma

3.3.5 The Du Bois-Reymond test for convergence

Near the start of the Derman and Sacks proof, there is an assertion that given some sequence of random variables Y_n whose series $\sum_{n=1}^{\infty} E(Y_n^2)$ converges, we can always find a sequence α'_n, that converges to zero, and $\sum_{n=0}^{\infty} \frac{E(Y_n^2)}{\alpha_n'^2}$ is a convergent series.

This is a corollary of a theorem from Du Bois-Reymond [21]. This states that for a positive sequence a_n, its summability is equivalent to the existence of some sequence b_n such that b_n diverges to positive infinity, $\forall n.\, b_n > 0$ and $a_n b_n$ is summable. An intuition for this is that there is no such thing as the largest convergent series – we can always find one such that the terms are eventually larger.

There are some immediate issues with this theorem's application to Dvoretzky's stochastic approximation theorem, as we cannot guarantee that our sequence of interest, $E(Y_n^2)$ is always positive. It could be zero for any n. The best we can say is that it is non-negative. We could not find a proof for non-negative sequences, so we build our own – the intuition for the proof is straight-forward but the formalisation less so.

In any case, the basis for the formalisation comes from an existing proof for positive sequences, by Ash [6]. The Du Bois-Reymond test for positive sequences is stated as follows in Isabelle:

```
lemma duboisreymond_test_real_pos:
fixes a :: "nat ⇒ real"
assumes "∀n. a n > 0"
shows "summable a =
  (∃b.
    (∀ε. eventually (λn. b n > ε) sequentially)
      ∧ (∀n. b n > 0)
      ∧ summable (λn. a n * b n))"
```

Listing 3.39: The `duboisreymond_test_real_pos` lemma

The result includes the text "∀ε `eventually (λn. b n > ε) sequentially`". This is telling us, in mathematical terms, that $\forall\varepsilon.\,\exists N.\,\forall n \geq N.\,b_n > \varepsilon$.

The proof here is a relatively straight-forward geometric proof, relying at a certain point on the comparison test for divergent series, which was not available in Isabelle's libraries. We show our formalisation of this test below:

```
lemma divergence_comparison_test:
fixes a b :: "nat ⇒ real"
assumes "∀n. a n ≥ 0" "¬summable a" "∀n. b n ≥ a n"
shows "¬summable b"
```

Listing 3.40: The `divergence_comparison_test` lemma

The proof of this lemma was simple and proceeded by showing that the partial sums for `b` could not possibly be convergent.

Our proof for the Du Bois-Reymond test for non-negative sequences relies on taking the summable, non-negative sequence a_n, and constructing the sequence a'_n by *de-zeroing the sequence*. By this we mean that if we produce a strictly increasing sequence of natural numbers c_n where $\forall n.\,a_{c_n} > 0$ and $\forall n.\,a_n > 0 \implies (\exists m.\,c_m = n)$, then $a'_n = a_{c_n}$. We then use the Du Bois-Reymond test for positive sequences with a'_n (which is positive in every term) to find b'_n (which diverges to positive infinity, is always positive, and $a'_n b'_n$ is summable) . From b'_n, we then construct b_n which is a sequence where for any n where $a_n = 0$, we simply repeat the preceding value in b. When we reach an n thereafter such that $a_n \neq 0$, we resume the sequence for the next term in b' – we call this *padding out for zeros*. For an illustration of this, please see figure 3.1.

Obviously, any term in b_n is also a term in b'_n, and equally obviously, $\forall n.\,b_n > 0$. As b'_n is divergent to positive infinity, we know the same is true for b_n. And we know that as $a'_n b'_n$ is summable, so must $a_n b_n$ be summable, as whenever a_n is zero, that index for

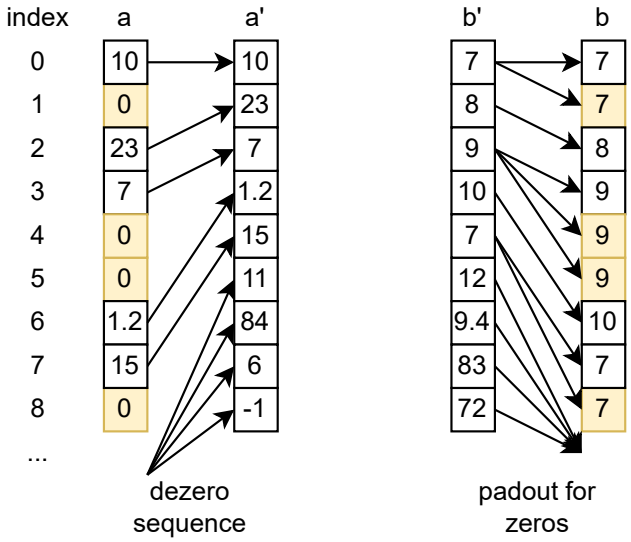

Figure 3.1: An illustration of dezeroing a sequence and padding one out for zeros.

$a_n b_n$ will also be zero; whenever a_n is non-zero, there is a matching index n' such that $a_n b_n = a'_{n'} b'_{n'}$.

This argument is fairly simple and easy to intuit, but more difficult to formalise. Throughout the description of the formalisation of this proof, we will refer to `a` as the original non-negative sequence, `a'` as its dezeroed sequence, `b'` as the sequence found using the positive version of the Du Bois-Reymond test, and `b` as its equivalent once padded out for the zeros in `a` (as discussed briefly above).

The `nonzeros_set` function takes a real sequence, and gives the set of the indices where that sequence is not zero:

```
definition nonzeros_set :: "(nat ⇒ real) ⇒ nat set" where
"nonzeros_set a = {n. a n ≠ 0}"
```

Listing 3.41: The `nonzeros_set` definition

The `nth_seq` function takes in a natural number `n` and a set of natural numbers and returns the `n`th smallest number from the set:

```
fun nth_seq :: "nat ⇒ nat set ⇒ nat" where
"nth_seq 0 A = (if A={} then 0 else (LEAST n. n∈A))" |
"nth_seq (Suc m) A = (if {n∈A. n > nth_seq m A}={} then 0
  else (LEAST n. n∈A ∧ n > nth_seq m A))"
```

Listing 3.42: The `nth_seq` definition

The seq_nth function takes a natural number n and a set of natural numbers A, and finds the ranking of the highest number n′∈ A. n′≤ n if we were to order the set:

```
definition seq_nth :: "nat ⇒ nat set ⇒ nat" where
"seq_nth n A = card {n'∈A. n'<n}"
```

Listing 3.43: The seq_nth definition

The dezero_seq function is the function that takes the sequence a above and removes the zeros from it to make a′ (see Figure 3.1 above):

```
definition dezero_seq :: "(nat ⇒ real) ⇒ nat ⇒ real" where
"dezero_seq a n = a (nth_seq n (nonzeros_set a))"
```

Listing 3.44: The dezero_seq definition

The padout_for_zeros function takes the a sequence and the b′ sequence as inputs and produces b, the equivalent to b′ once it is "padded out" for the zero in a (again, see Figure 3.1 above):

```
fun padout_for_zeros :: "(nat ⇒ real) ⇒ (nat ⇒ real) ⇒ nat ⇒ real
    " where
"padout_for_zeros b a 0 = b 0" |
"padout_for_zeros b a (Suc m) = (if (Suc m) ∉ nonzeros_set a
  then padout_for_zeros b a m
  else b (seq_nth (Suc m) (nonzeros_set a)))"
```

Listing 3.45: The padout_for_zeros definition

These functions are the tools that can be used to replicate the argument above formally. However, these are not simple definitions, and they require some basic properties to be proven for them before we can be certain that they will do the trick.

We show that if such an a is summable, then so is dezero_seq a:

```
lemma summable_dezero:
fixes a :: "nat ⇒ real" and x :: real
assumes "infinite (nonzeros_set a)"
shows "summable a = summable (dezero_seq a)"
```

Listing 3.46: The summable_dezero lemma

Key to the final proof, we show that for a sequence with infinite non-zero terms, its summation and that of its dezeroed sequence are identical:

```
lemma sums_dezero:
fixes a :: "nat ⇒ real" and x :: real
assumes "infinite (nonzeros_set a)"
shows "a sums x = (dezero_seq a) sums x"
```

Listing 3.47: The `dezero_seq_val` lemma

This proof relies on a number of less important properties for the various functions we have defined above, primarily those showing how we can relate a dezeroed sequence to the original sequence.

Finally, there are two proofs showing that the `padout_for_zeros` function behaves as expected. The first relates it to the pre-transformation sequence using `seq_nth` and `nonzeros_set`, assuming the index `n` being used is such that `a n` $\neq$ `0`:

```
lemma padout_for_zeros_works_1:
fixes a b :: "nat ⇒ real" and n :: nat
assumes "n∈nonzeros_set a"
shows "padout_for_zeros b a n = b (seq_nth n (nonzeros_set a))"
```

Listing 3.48: The `padout_for_zeros_works_1` lemma

We then show that if `padout_for_zeros` is being used, the order of the indices is preserved from the original sequence:

```
lemma padout_for_zeros_works_2:
fixes a b :: "nat ⇒ real" and n :: nat
assumes "n∈nonzeros_set a"
shows "∀m. padout_for_zeros b a (m+n)
  ∈ b ` {seq_nth n (nonzeros_set a)..}"
```

Listing 3.49: The `padout_for_zeros_works_2` lemma

These proofs are all essentially showing that the definitions given match the intuitive understanding given earlier for the various transformations being applied to the sequences which we need to prove the Du Bois-Reymond test for non-negative sequences. With each in place, we can take the final step and prove the lemma itself:

```
lemma duboisreymond_test_real_nn:
fixes a :: "nat ⇒ real"
assumes "∀n. a n ≥ 0"
shows "summable a =
  (∃b.
    (∀ε. eventually (λn. b n > ε) sequentially)
      ∧ (∀n. b n > 0)
      ∧ summable (λn. a n * b n))"
```

Listing 3.50: The `duboisreymond_test_real_nn` lemma

The proof here is split into two cases: where the sequence `a` has infinite non-zero valued terms, and where it does not. The latter case is trivial, and does not need much of the machinery we have built. The former case, though, requires all of the lemmas above be used, as well as the comparison test for divergence (see Listing 3.40). We follow the process outlined in the intuitive proof above. The final step for the trickiest property to prove is to show that the summable sequence found by using the Du Bois-Reymond test for positive sequences, `λn. a′ n * b′ n`, which is summable, is in fact equal to `dezero_seq (λn. a n * b n)`, at which point the summability of the latter is a consequence of the `summable_dezero` lemma.

3.3.6 Dvoretsky's stochastic approximation theorem

Finally, we are in a position to prove Dvoretzky's stochastic approximation theorem.

In the statement of the formalisation below, our definition and assumptions match those given in the pen-and-paper description earlier, in Section 3.3.1. We will explain each in mathematical terms that can be related to that section after the formalisation:

```
theorem (in prob_space) Dvoretzky_stochastic_approximation:
  fixes X T Y :: "nat ⇒ 'a ⇒ real"
    and α β γ :: "nat ⇒ real"
  assumes αnn:"∀n. α n ≥ 0"
    and βnn:"∀n. β n ≥ 0"
    and γnn:"∀n. γ n ≥ 0"
    and αtt0:"α → 0"
    and βsum:"summable β"
    and γdiv:"∀ε. ∃N. ∀n≥N. (∑i≤n. γ i) > ε"
    and TadX:"∀n. T n ∈ borel_measurable
      (filtration_sequence n M X borel)"
    and YadX:"∀n. Y n ∈ borel_measurable
      (filtration_sequence n M X borel)"
    and Xmes:"∀n. X n ∈ measurable M borel"
    and Ypint:"∀I. length I > 0 ⟹
      integrable M (λx. (∏i∈{0..<length I}. Y (I!i) x))"
    and Xsucdep:"∀n. (λx. X (Suc n) x) = (λx. T n x + Y n x)"
    and Yexp:"∀n. AE x in M. real_cond_exp M
      (filtration_sequence n M X borel) (Y n) x = 0"
    and Yexp0:"expectation (Y 0) = 0"
    and Y2sum:"summable (λn. expectation (λx. (Y n x) ^ 2))"
    and T:"∀n x. |T n x| ≤ max (α n) ((1 + β n) * |X n x| - γ n)"
  shows "AE x in M. (λn. X n x) → 0"
```

Listing 3.51: The `Dvoretzky_stochastic_approximation` theorem

1. X T Y are our sequences of random variables. Note their type `nat ⇒ 'a ⇒ real`: this shows they are functions from the natural numbers (their index) to the `'a ⇒ real` type. The `'a` is the type variable of the probability space from the probability locale, so any function of this type is a random variable if measurable.
2. α β γ are our sequences of real numbers. Note their type is a function from the natural numbers (their index) to the reals.
3. α`nn`, β`nn` and γ`nn` establish that our sequences are non-negative.
4. α`tt0` shows that $\lim_{n\to\infty} \alpha_n = 0$.
5. β`sum` establishes that $\sum_i \beta_i$ is finite.
6. γ`div` shows that the $\sum_i \gamma_i$ diverges to positive infinity.
7. `TadX` shows that the random variable sequence T is measurable over the natural filtration for X; in other words, T_{n+1} is dependent on $\{X_1, X_2, ..., X_n\}$.
8. `YadX` shows the same for the random variable sequence `Y`.

9. `Xmes` shows that X_i is measurable between `M` (the probability space) and the Borel measure on the real numbers. In other words, it is a random variable.
10. `Ypint` shows that the product of any arbitrary random variables from the sequence Y is integrable. Recall from Section 3.2.3.1 that this assumptions allows us to prove the integrability of random variables resulting from a wide variety of operations over Y.
11. `Xsucdep` shows that $X_{n+1} = T_n + Y_n$.
12. `Yexp` shows that almost everywhere, $E(Y_n|X_1, X_2 ... X_n) = 0$.
13. `Yexp0` shows that $E(Y_0) = 0$.
14. `Y2sum` shows that $\sum_i Y_i^2$ is convergent.
15. `T` shows that $|T_n| \leq \max(\alpha_n, (1+\beta_n)|X_n| - \gamma_n)$.

There is a key, albeit small, additional lemma which is needed here. The `advar_subalg` lemma shows that if X_n is Borel measurable against some F_n, then the set of events from its equivalent natural filtration must be a subalgebra of that of F_n. The existing `real_cond_exp_nested_subalg` lemma from Isabelle's libraries can then be used to treat the conditional expectation against that F_n as if it were that against the natural filtration for X. This is used with the `rv_convergence_crit` result as at that point in the proof, the Z sequence has a conditional expectation against the natural filtration for X, not itself.

```
lemma advar_subalg:
fixes X :: "nat ⇒ 'a ⇒ real" and F :: "nat ⇒ 'a measure"
  and M :: "'a measure"
assumes XmesF:"∀n. X n ∈ borel_measurable (F n)"
  and Xmes:"∀n. X n ∈ borel_measurable M"
  and Ffilt:"filtration (space M) F"
shows "∀n. subalgebra (F n) (filtration_sequence n M X borel)"
```

Listing 3.52: The `advar_subalg` lemma

With the exception of this minor change, the formal proof proceeds exactly like the pen-and-paper proof, using each of the important results given already in this chapter.

3.3.6.1 Dvoretzky's stochastic approximation theorem extension

Jaakkola's proof uses an extension of Dvoretzky's stochastic approximation theorem, in which the sequences of real numbers, α, β and γ, are treated as bounded random variables instead. We formalise a proof for this extension by building some necessary

precursor results that allow us to work more easily with these sequences of random variables.

First we build a lemma, `seq_bounded_integrable_real`, that shows that if we assume a sequence of random variables is bounded, then we can show that any arbitrarily chosen elements of the sequence are integrable when multiplied together. This allows us to use our integrability results from section 3.2.3.1.

```
lemma (in finite_measure) seq_bounded_integrable_real:
  fixes X :: "nat ⇒ 'a ⇒ real"
  assumes "∀n. bounded (range (X n))"
    "∀n. (X n) ∈ borel_measurable M"
  shows "∀I. length I > 0
    ⟶ integrable M (λx. (Πi∈{0..<length I}. X (I!i) x))"
```

Listing 3.53: The `seq_bounded_integrable_real` lemma

We also show that if $\forall n$, P_n holds almost everywhere, then it is also true that almost everywhere, $\forall n\, P_n$ holds. In other words, that the universal quantifier and the almost everywhere quantifier are commutable for natural numbers.

```
lemma AE_seq:
  fixes P :: "nat ⇒ 'a ⇒ bool"
  assumes "∀n. AE x in M. P n x"
  shows "AE x in M. ∀n. P n x"
```

Listing 3.54: The `AE_seq` lemma

These preliminary results allow us to prove the extension. The proof proceeds almost exactly as for the original theorem, with the addition of showing that for each of the properties which are true almost everywhere, their conjunction is also true almost everywhere. This allows us to select an element within the probability space where all these properties are true, and use that to reason about our random variables more concretely.

```
theorem (in prob_space)
  Dvoretzky_stochastic_approximation_extension:
  fixes X T Y α β γ :: "nat ⇒ 'a ⇒ real"
  assumes αnn:"∀n x. α n x ≥ 0"
    and βnn:"∀n x. β n x ≥ 0"
    and γnn:"∀n x. γ n x ≥ 0"
    and αatt:"∀U∈sets M. prob U = 1
      ⟶ (∃x∈U. ∀n. α n x = Sup (α n ` U))"
    and αbound:"∀n. bounded (range (α n))"
    and βbound:"∀n. bounded (range (β n))"
    and γbound:"∀n. bounded (range (γ n))"
    and αtt0:"AE x in M. (λn. α n x) → 0"
    and βsum:"AE x in M. summable (λn. β n x)"
    and γdiv:"AE x in M. ∀ε. ∃N. ∀n≥N. (Σi≤n. γ i x) > ε"
    and TadX:"∀n. T n ∈ borel_measurable
      (filtration_sequence n M X borel)"
    and YadX:"∀n. Y n ∈ borel_measurable
      (filtration_sequence n M X borel)"
    and αadX:"∀n. α n ∈ borel_measurable
      (filtration_sequence n M X borel)"
    and βadX:"∀n. β n ∈ borel_measurable
      (filtration_sequence n M X borel)"
    and γadX:"∀n. γ n ∈ borel_measurable
      (filtration_sequence n M X borel)"
    and Xmes:"∀n. X n ∈ measurable M borel"
    and Ypint:"∀I. length I > 0
      ⟶ integrable M (λx. (Πi∈{0..<length I}. Y (I!i) x))"
    and Xsucdep:"∀n. (λx. X (Suc n) x) = (λx. T n x + Y n x)"
    and Yexp:"∀n. AE x in M. real_cond_exp M
      (filtration_sequence n M X borel) (Y n) x = 0"
    and Yexp0:"expectation (Y 0) = 0"
    and Y2sum:"summable (λn. expectation (λx. (Y n x) ^ 2))"
    and T:"AE x in M. ∀n. |T n x|
      ≤ max (α n x) ((1 + β n x) * |X n x| - γ n x)"
  shows "AE x in M. (λn. X n x) → 0"
```

Listing 3.55: The `Dvoretzky_stochastic_approximation_extension` theorem

3.4 Related work

During our work, a formalisation of the Dvoretzky stochastic approximation theorem was completed in Coq [155]. This also uses the proof by Derman and Sacks as the basis for its formalisation. Interestingly, it does not (for its main proof) formalise T_n as a dependent random variable, but as a measurable function on $\{X_1 ... X_n\}$. This removes the notion of stochasticity from T_n, meaning that this proof is not quite of the theorem as presented by Dvoretzky.

The paper does mention issues of integrability: the expectation of a random variable may be set to `None` if it is not defined, and an `IsFiniteExpectation` function tests for integrability. However, it does not explain how this is treated for lemmas and theorems that use operations over sequences of integrable random variables in ways that demand integrability. The Du Bois-Reymond test is introduced in the paper but with no clear explanation on how it was proven for non-negative rather than positive sequences.

It does prove an extension of the Dvoretzky stochastic approximation theorem whereby T_n is treated as a random variable dependent on $\{X_1 ... X_n\}$ and α_n, β_n and γ_n are treated as real functions instead of sequences. The usual version of the theorem can be seen as a special case of this, where these real functions are constant.

3.5 Conclusion

In this chapter we have made significant contributions to probability theory in Isabelle. In particular:

- We formalised a locale within which the convergence of the Q learning algorithm would be provable when it acts on a valid path, including a formalisation of the algorithm update process.
- We formally proved that under the assumption that arbitrary products of random variables from a sequence are integrable, a number of other operations will also result in an integrable random variable.
- We formalised the Dirichlet convergence test and summation by parts.
- We formalised the notion of a natural filtration of a random variable sequence.

- Using the formal model of natural filtrations we developed, we completed a formalisation of a more general version of the Kolmogorov inequality, where random variables could be dependent upon an arbitrary sequence of random variables.

- We showed a number of results related to sequences of random variables, including in particular, a set of convergence criteria critical to the Dvoretzky stochastic approximation theorem.

- We formalised the divergence comparison test preparatory to formalising the Du Bois-Reymond test. We then formalised a novel proof of a version of the test across non-negative sequences rather than positive sequences.

- We formalised the Dvoretzky stochastic approximation theoremand an extension of the theorem where the sequences of real numbers used are replaced with sequences of real random variables..

With the Dvoretzky stochastic approximation theorem formally proven, we are very close to a formal proof that a Q learning agent (as formally modelled in Section 3.1.1.2) converges to the correct Q values for the states and actions it experiences moving through an MDP.

Jaakkolo's proof (see Section 3.1.1.5) is a relatively simple application of Dvoretzky's result, and what we have is sufficient to proceed to formally verify it. By formalising the necessary precursor results we have demonstrated that formal methods can be used to build a model of the fundamental mathematics of modern artificial intelligence algorithms in frequent use – in this case reinforcement learning – and to verify their correctness.

One thing that seems apparent from the results that both were and were not in Isabelle's libraries, is that the existing probability theory formalisations therein had not yet considered sequences of random variables or of events. Many of the results that we needed to develop for this work involved such sequences. The work we have done on these can be used for further development in this area, beyond that required for the Dvoretzky stochastic approximation result.

This result, in itself, has a wide applicability in other areas of statistical learning [89]. Its formal verification may therefore be of value to statistical learning and approximation in those areas, beyond the current contribution to the understanding of sequences in probability.

Chapter 4

Formalised logical constraint satisfaction implemented into a neural network

4.1 Introduction

The usefulness and general applicability of the neural networks used in deep learning (introduced in Section 1.2) is well understood, but there are several ways in which their behaviour could be improved. In particular, it can be hard to understand the behaviour of a neural network, as they take a black box approach in which it is usually difficult to establish the reason why a given result was achieved.

A lot of training is often needed before a neural network behaves as desired, and additionally, uncertainty in its behaviour can make it entirely unsuited to certain safety-critical tasks, such as robotic movement [118].

If one could use some form of logical specifications or constraints as part of the training of a neural network then this could have benefits such as:

- The network's output can be interpreted in light of the specifications passed to it. A logical specification is clear and well-defined, and if it is violated by the output, this can provide a way of assessing the network more directly than by measuring its error against training data.

- The volume of data required to train the network to ensure that it does not breach the constraints could be reduced e.g. where a specification may be simple to express but need many sets of training data for a neural network to learn through

the usual deep learning process [119].

Of particular relevance to the current work is the approach by Innes and Ramamoorthy [71], which builds on work by Fischer [41] and aims to improve the training speed of learning robotic movement by mimicking a demonstrator. The suggestion is that given some logical criterion specified in linear temporal logic (e.g. "don't tip the cup until you are above the bowl"), the network learning the movements will need less demonstrator data to have confidence that it will not breach the rule, even on unseen inputs. We extend this work by taking a fully-rigorous, theorem-proving based approach to the logical underpinnings of the loss function and its derivative.

In particular, by formalising a deep embedding of linear temporal logic over finite traces (LTL_f) and its semantics in the higher-order logic (HOL) of the proof-assistant Isabelle, we formulate a loss function $\mathscr{L}$ that measures whether a statement is satisfied. Moreover, it is then formally proven that $\mathscr{L}$ is differentiable, with an explicit derivative $d\mathscr{L}$ that can be used as part of the gradient-descent minimisation of the loss.

We use Isabelle to automatically generate OCaml code from our provably correct formal specifications, and integrate it into a PyTorch neural network via a library that provides OCaml bindings for Python. Isabelle cannot generate Python code directly, however, the OCaml language allows for functional and object oriented programming and Isabelle has existing code generation methods for it. Both the loss function and its derivative are used in combination with the usual training process so that the neural network can learn from both training data and any specified constraints. The loss function generated could in general be applied to any neural network with a notion of time-sequences – for the purposes of this work, this is demonstrated using a specific network that predicts paths of motion.

More specifically, we demonstrate experimentally that the neural network learns constrained behaviour when given a wide variety of logical constraints in LTL_f. Because the code for the logical implementations of the specifications, the loss function and its derivative was generated automatically by Isabelle, the approach provides enhanced guarantees about the correctness of the whole pipeline.

This chapter is organised as follows: we briefly review some related work to provide some background. Then in Section 4.2, we discuss LTL_f and its formalisation. In Section 4.3 we show how the loss function and its derivative are built, proving both are correct. Section 4.4 discusses how the OCaml code in PyTorch was generated and implemented. In Section 4.5 we demonstrate that the code works as expected. We conclude with some final thoughts about the work in Section 4.7.

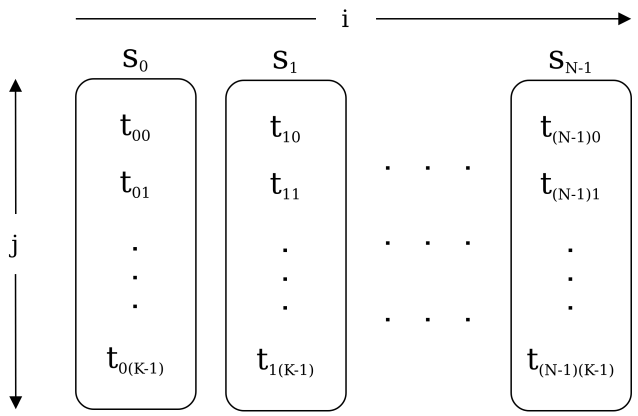

Figure 4.1: A path of states with values: i indexes time proceeding from left to right and j indexes the values each state can measure

4.2 Formalising linear temporal logic

As already mentioned, we mechanise linear temporal logic over finite paths in Isabelle and use it to formulate a loss function for training a neural network under rigorously specified logical constraints. We review some of the salient aspects of the formalisation next.

4.2.1 States and paths

We provide the types for states and paths in Isabelle and discuss their meaning here:

1. `state :: "int ⇒ real"` tells us that a `state` is a function from the `int` type, the integers, to the real numbers.

2. `path :: "state list"` tells us that a `path` is a finite list of such states i.e. a list of functions.

Note that in Isabelle, a list can be written as `s#ss` denoting the element `s` being consed (concatenated) onto the (possibly empty) list `ss` and that the lambda abstraction, or anonymous function, is denoted by λ`x. M` so e.g. λ`x. x`^{2} denotes the function that squares its argument `x`.

Consider a state function as a function indexed on the integers, which returns a real number dependent on the index. These real numbers taken together, describe the "state" of interest. For example, if we are measuring the motion of a robot on a two

dimensional plane, a state function might return its x and y co-ordinate through indices 1 and 2, and the components of its velocity along either axis through indices 3 and 4.

In the context of this work, the state function at a particular time step i tracks several values in a learning problem that might be compared. In almost all cases, a state function only needs to represent some finite number K of values. By only considering indices j over this range, the state function for a given time step can be considered as a vector, as can be seen in Fig. 4.1.[1]

With each state function encoding information about a system at a specific time step, a `path` of length N encodes the evolution of a system over N time steps. In the case of Fig. 4.1, increasing i represents the forward temporal progression of the system. It should also be clear from this figure that the values encoded by a `path` of `states` can equivalently be represented in matrix form, which will be of vital importance to the PyTorch integration (see Section 4.4.2).

4.2.2 LTL and finite traces of states

Our formalisation uses a variant of LTL [120] known as LTL$_f$ [34], which is interpreted over finite traces (of states) and is often viewed as a more natural choice for applications in AI (for example, planning [11]), where processes are usually of finite duration. Note that in what follows, any reference to LTL means LTL$_f$ unless otherwise stated.

As we are using LTL to evaluate conditions during the training of a neural network, we take comparisons of real-valued terms (each term corresponding to measurements in the environment or constants), namely $t < t', t \leq t', t = t', t \neq t'$ as our atomic propositions ρ [41]. Thus we divide our LTL constraints into these comparisons (`comp`) and those constraints (`constraint`) arising solely from LTL's operators (see further down for their Isabelle implementations).

The LTL$_f$ formulae are thus:

1. ρ
2. $\rho_1 \wedge \rho_2$ (conjunction, logical and)
3. $\rho_1 \vee \rho_2$ (disjunction, logical or)
4. $\mathscr{N}\rho$ (Next)

[1] Strictly speaking, natural numbers could have been used as the indices. However, integers are used because OCaml, the target language for these specifications, has no native natural number type.

5. $\Box\rho$ (Always)

6. $\Diamond\rho$ (Eventually)

7. $\rho_1 \mathscr{U} \rho_2$ (Weak Until)

8. $\rho_1 \mathscr{R} \rho_2$ (Strong Release)

The usual logical operators behave as expected. Note that we model a negation operator via a function that transforms any logical constraint c into an equivalence of its negation expressed in terms of the logical operators listed above. This is discussed in more detail later in this section. The temporal operators are evaluated at a given time step t, and behave as follows:

- $\mathscr{N}\rho$ (Next ρ) is true if at the *next* step along the path, ρ holds. Note that as we are evaluating along a finite path, we must consider how to evaluate this at its termination – we discuss this below.

- $\Box\rho$ (Always ρ) holds if at the current step and all subsequent steps along the path, ρ holds.

- $\Diamond\rho$ (Eventually ρ) holds if at the current or at least one subsequent step along the path, ρ holds.

- $\rho_1 \mathscr{U} \rho_2$ (ρ_1 `Until` ρ_2) means that ρ_1 holds at least for all steps until ρ_2 holds. ρ_2 need not hold at any future point – this is a Weak `Until`.

- $\rho_1 \mathscr{R} \rho_2$ (ρ_1 `Release` ρ_2) means that ρ_2 holds at least until and including the point when ρ_1 holds. ρ_1 must hold at some point in the path – this is a Strong `Release`.

In LTL$_f$ these operators have the usual semantics of LTL, except at the end of a path. In particular, for a sequence of length i, $\neg(\mathscr{N}\rho)$ holds at the final step i [34]. As a consequence, when dealing with a finite path of length i, it is important to recognise that $\neg(\mathscr{N}\rho) \iff \mathscr{N}(\neg\rho)$ is not true. $\neg(\mathscr{N}\rho)$ is true at the last step in a path and $\mathscr{N}(\neg\rho)$ is false at that step.

We proceed by defining Isabelle datatypes `comp` and `constraint` for the comparison and constraint operators respectively. This approach to specifying the language in (higher-order) logic is known as a deep embedding [23]. Using a deep embedding in this way allows us to reason directly about formulas written in the logic as typed

objects; for example, this makes inductive proofs over these formulas possible.Doing so will enable us to prove that the loss function is sound with respect to the usual LTL_f semantics and, importantly, generate fully self-contained code for the specification language that will be used as part of the training of our neural net. Note also our formulation of `path` which, as per the previous section, gives a type that represents finite paths over which LTL_f can be evaluated.

```
type_synonym state = "int ⇒ real"
type_synonym path = "state list"

datatype comp = Less int int | Lequal int int | Equal int int
    | Nequal int int

datatype constraint = Comp comp | And constraint constraint
    | Or constraint constraint | Next constraint | Always constraint
    | Eventually constraint | Until constraint constraint
    | Release constraint constraint
```

Listing 4.1: Datatypes for our LTL_f implementation

Lists (such as those we use for paths) in Isabelle can be empty, usually shown as `[]`, which is of little interest to us in our proofs, except for checking if the end of the path has been reached. So when we seek to prove something over a path, we usually notate it as `s # ss`. `s` here is a single state, the `#` operator joins that to the (potentially empty) list `ss`. By using this notation in a statement, we ensure that it is over a non-empty path.

Next, we formalise the `eval` function, which characterises the semantics of LTL_f by recursively evaluating the truth-value of a constraint over a path. The function recurses through the constraint, from the outside in, and when it reaches a temporal operator, it recurses down the path as required. Note too that when given an empty path, `eval` is always false:

```
function eval :: "constraint ⇒ path ⇒ bool" where
  "eval c [] = False"
| "eval (Comp (Lequal v1 v2)) (s#ss) = ((s v1) ≤ (s v2))"
| "eval (Comp (Nequal v1 v2)) (s#ss) = ((s v1) ≠ (s v2))"
| "eval (Comp (Less v1 v2)) (s#ss) = ((s v1) < (s v2))"
| "eval (Comp (Equal v1 v2)) (s#ss) = ((s v1) = (s v2))"
| "eval (And c1 c2) (s#ss) = ((eval c1 (s#ss)) ∧ (eval c2 (s#ss)))"
| "eval (Or c1 c2) (s#ss) = ((eval c1 (s#ss)) ∨ (eval c2 (s#ss)))"
| "eval (Next c) (s#ss) = eval c ss"
| "eval (Always c) (s#ss) = ((eval c (s#ss)) ∧ (if ss = [] then True
    else (eval (Always c) ss)))"
| "eval (Eventually c) (s#ss) = ((eval c (s#ss))
    ∨ (eval (Eventually c) ss))"
| "eval (Until c1 c2) (s#ss) = ((((eval c1 (s#ss))
    ∧ (if ss = [] then True else (eval (Until c1 c2) ss))))
      ∨ eval c2 (s#ss))"
 | "eval (Release c1 c2) (s#ss) = (((((eval c2 (s#ss))
    ∧ (eval (Release c1 c2) ss)))
      ∨ eval (And c1 c2) (s#ss))
        ∧ eval (Eventually c1) (s#ss))"
```

Listing 4.2: The LTL$_f$ evaluation function

If we look in detail at the listing above we can use the `Eventually` clause to give an example of how the function works. `eval (Eventually c) (s#ss)` is true if either `(eval c (s#ss))` is true (in other words, `c` is true at the current time step), or if `(eval (Eventually c) ss)` is true, in other words, if `Eventually c` is true at the next time-step. If the end of the path is reached without `c` ever being true, then the clause `eval c []` is used to return false.

We formalise negation via a `Not` function which, given any constraint, returns a constraint that is provably logically equivalent to its negation [2]. This reduces the number of primitive operators that one needs to specify for LTL$_f$, thereby simplifying reasoning about its properties.

[2]Unless evaluated at the end of the path, where, for example, $\mathcal{N}\rho$ is always false.

```
fun Not :: "constraint ⇒ constraint" where
  "Not (Comp (Equal v1 v2)) = Comp (Nequal v1 v2)"
| "Not (Comp (Nequal v1 v2)) = Comp (Equal v1 v2)"
| "Not (Comp (Less v1 v2)) = Comp (Lequal v2 v1)"
| "Not (Comp (Lequal v1 v2)) = Comp (Less v2 v1)"
| "Not (And c1 c2) = Or (Not c1) (Not c2)"
| "Not (Or c1 c2) = And (Not c1) (Not c2)"
| "Not (Next c) = Next (Not c)"
| "Not (Always c) = Eventually (Not c)"
| "Not (Eventually c) = Always (Not c)"
| "Not (Until c1 c2) = Release (Not c1) (Not c2)"
| "Not (Release c1 c2) = Until (Not c1) (Not c2)"
```

Listing 4.3: The `Not` function

We show that, as expected, $\neg\neg\rho \equiv \rho$ using the `Not_Not` lemma:

```
lemma Not_Not:"Not (Not c) = c"
```

Listing 4.4: The `Not_Not` lemma

Given the complexity of LTL_f compared to propositional logic, we also prove a number of LTL_f equivalences, which confirms that `eval` behaves as expected. There are a series of lemmas suffixed `_dist`, each of which shows that several of the temporal operators are distributive over the conjunction and disjunction operators. We next show that $\Box\Box\rho \equiv \Box\rho$, and likewise $\Diamond\Diamond\rho \equiv \Diamond\rho$. Lastly, we show some more intuitive ways to rewrite three of the temporal operators recursively, with `AlwaysNext`, `EventuallyNext`, and `UntilNext`.

```
lemma NextOr_dist: "eval (Next (Or c1 c2)) ss
  = eval (Or (Next c1) (Next c2)) ss"

lemma NextAnd_dist: "eval (Next (And c1 c2)) ss
  = eval (And (Next c1) (Next c2)) ss"

lemma AlwaysAnd_dist: "eval (Always (And c1 c2)) ss
  = eval (And (Always c1) (Always c2)) ss"

lemma EventuallyOr_dist: "eval (Eventually (Or c1 c2)) ss
  = eval (Or (Eventually c1) (Eventually c2)) ss"

lemma EventuallyEventually: "eval (Eventually (Eventually c)) ss
  = eval (Eventually c) ss"
```

```
lemma AlwaysAlways: "eval (Always (Always c)) ss
  = eval (Always c) ss"

lemma AlwaysNext:
  fixes ss :: path and c :: constraint
  assumes "length ss ≥ 2"
  shows "eval (Always c) ss = eval (And c (Next (Always c))) ss"

lemma EventuallyNext: "eval (Eventually c) ss
  = eval (Or c (Next (Eventually c))) ss"

lemma UntilNext:
  fixes ss :: path and c1 :: constraint and c2 :: constraint
  assumes "length ss ≥ 2"
  shows "eval (Until c1 c2) ss
    = eval (Or c2 (And c1 (Next (Until c1 c2)))) ss"
```

Listing 4.5: Some simple LTL$_f$ equivalencies using `eval`

There are two remaining important lemmas which prove that our definition for $\Box\rho$ and $\Diamond\rho$ match the normal mathematical definitions. In the definition here, the `drop` function works to shorten the path by some value `n`, where n is smaller than the length of the path:

```
lemma Always_works:
  fixes ss :: path and c :: constraint
  assumes "length ss ≥ 1"
  shows "(∀n < length ss. eval c (drop n ss)) = eval (Always c) ss"

lemma Eventually_works: "(∃n < length ss.
  eval c (drop n ss)) = eval (Eventually c) ss"
```

Listing 4.6: Showing that Always and Eventually work

4.3 A LTL-based loss function and its derivative

The loss function $\mathscr{L}$ – which takes a constraint ρ, a path P and a relaxation factor γ, and returns a real value – needs to satisfy several important properties:

1. $\mathscr{L}(\rho, P, 0) \geq 0$;
2. $\mathscr{L}$ is differentiable wrt any of the terms on the path;

3. (Soundness) $\lim_{\gamma \to 0} \mathscr{L}(\rho, P, \gamma) = 0 \iff \rho(P)$, where $\rho(P)$ is the truth value of ρ on P. In other words, our loss function returns a loss of 0 if and only if the constraint is true on the path (as γ approaches 0).

4.3.1 Soft functions and their derivatives

When formalising $\mathscr{L}$, in order for it to be differentiable, it is necessary to use *soft* (i.e. differentiable) versions of various functions. Thus, based on the work by Cuturi and Blondel [33], we define binary softmax and softmin functions, $\max_\gamma$ and $\min_\gamma$, respectively, as:

$$\max_\gamma\ a_1\ a_2 = \begin{cases} \max a_1\ a_2 & \gamma \leq 0 \\ \gamma \log(e^{a_1/\gamma} + e^{a_2/\gamma}) & \text{otherwise} \end{cases} \tag{4.1}$$

$$\min_\gamma\ a_1\ a_2 = \begin{cases} \min a_1\ a_2 & \gamma \leq 0 \\ -\gamma \log(e^{-a_1/\gamma} + e^{-a_2/\gamma}) & \text{otherwise} \end{cases} \tag{4.2}$$

Each of these soft functions takes an additional parameter γ. The intention behind this is that as $\gamma \to 0$, $\max_\gamma \to \max$, $\min_\gamma \to \min$, and that they are differentiable for $\gamma > 0$. In Isabelle, these are easily formalised:

```
fun Max_gamma :: "real ⇒ real ⇒ real ⇒ real" where
  "Max_gamma γ a b = (if γ ≤ 0 then max a b
     else γ * ln (exp (a/γ) + exp (b/γ)))"

fun Min_gamma :: "real ⇒ real ⇒ real ⇒ real" where
  "Min_gamma γ a b = (if γ ≤ 0 then min a b
     else -γ * ln (exp (-a/γ) + exp (-b/γ)))"
```

Listing 4.7: The soft gamma Max and Min functions

and proven to be correct:

```
lemma Max_gamma_lim: "(λγ. Max_gamma γ a b) −0→ max a b"
lemma Min_gamma_lim: "(λγ. Min_gamma γ a b) −0→ min a b"
```

Listing 4.8: The correctness lemmas for the soft gamma Max and Min functions

where $(\lambda x.f\,x)\ -0\to\ \mathtt{L}$ denotes $\lim_{x\to 0} f(x) = L$ in Isabelle [42].

It should be noted that our specific formalisation here caused some problems when exported as code (see Section 4.4). This is because the order of operations can easily cause a floating point error. If γ is very small (as we will typically want it to be), and a or b are large, then computing $e^{\frac{a}{\gamma}}$ can easily produce a floating point overflow.

The solution was to redefine a more practical version of these functions:

```
function Max_gamma_comp :: "real ⇒ real ⇒ real ⇒ real" where
  "Max_gamma_comp γ a b = (if γ ≤ (0::real) then max a b
    else γ * (if (a/γ) < (b/γ)
      then (a/γ) + ln ((1::real) + exp ((b/γ) - (a/γ)))
      else (if (b/γ) < (a/γ)
        then (b/γ) + ln ((1::real) + exp ((a/γ) - (b/γ)))
        else (a/γ) + ln ((1::real)+(1::real)))))"

function Min_gamma_comp :: "real ⇒ real ⇒ real ⇒ real" where
  "Min_gamma_comp γ a b = (if γ ≤ (0::real) then min a b
    else -γ * (if (-a/γ) < (-b/γ)
      then (-a/γ) + ln ((1::real) + exp ((-b/γ) - (-a/γ)))
      else (if (-b/γ) < (-a/γ)
        then (-b/γ) + ln ((1::real) + exp ((-a/γ) - (-b/γ)))
        else (-a/γ) + ln ((1::real)+(1::real)))))"
```

Listing 4.9: More practical versions of `Max_gamma` and `Min_gamma`

and prove that they are equivalent to the straight-forward versions given above:

```
lemma Max_gamma_comp_eq:"Max_gamma_comp γ a b = Max_gamma γ a b"
lemma Min_gamma_comp_eq:"Min_gamma_comp γ a b = Min_gamma γ a b"
```

Listing 4.10: Correctness lemmas of the practical versions of `Max_gamma` and `Min_gamma`

These functions minimise the chance of any calls to the `exp` function which could produce floating point overflow errors.

For `Max_gamma`, the derivatives with respect to a and b are built separately before being combined to give the `dMax_gamma_ds` function. This is shown to be the expected derivative using Isabelle's `has_real_derivative` relation by proving the following theorem (for $\gamma > 0$):

```
theorem dMax_gamma_chain:
assumes "γ > 0" and "(f has_real_derivative Df) (at x)"
  and "(g has_real_derivative Dg) (at x)"
shows "((λy. Max_gamma γ (f y) (g y)) has_real_derivative
  (dMax_gamma_ds γ (f x) Df (g x) Dg)) (at x)"
```

Listing 4.11: The derivative's correctness stated in Isabelle

Likewise, other soft functions are used to capture losses from the `Nequal` comparison, again using γ as a parameter. There is an important distinction between our

approach and that of Fischer [41] and Innes and Ramamoorthy [71]: in their respective work, the `Nequal` and `Lequal` comparisons were not defined using soft-functions and were not differentiable. They relied on the implicit auto-differentiation machinery of PyTorch to handle these for backpropagation purposes. In this case, though, an explicit derivative of the loss function is provided to PyTorch thereby giving guarantees that backpropagation is using the desired function for gradient descent (see Sections 4.3.3 and 4.4.2). It is therefore required that the loss functions for all our comparisons to be provably differentiable.

4.3.2 Formalising the loss function

We proceed to define $\mathscr{L}$ recursively over the constraint, given a path and a relaxation factor γ:

```
function L :: "constraint ⇒ path ⇒ real ⇒ real" where
  "L c [] γ = 1"
| "L (Comp (Lequal v1 v2)) (s#ss) γ = Max_gamma γ (s v1 - s v2) 0"
| "L (Comp (Nequal v1 v2)) (s#ss) γ = Nequal_gamma γ (s v1) (s v2)"
| "L (Comp (Less v1 v2)) (s#ss) γ = Max_gamma γ
     (L (Comp (Lequal v1 v2)) (s#ss) γ)
     (L (Comp (Nequal v1 v2)) (s#ss) γ)"
| "L (Comp (Equal v1 v2)) (s#ss) γ = Max_gamma γ
     (L (Comp (Lequal v1 v2)) (s#ss) γ)
     (L (Comp (Lequal v2 v1)) (s#ss) γ)"
| "L (And c1 c2) (s#ss) γ = Max_gamma γ
     (L c1 (s#ss) γ)
     (L c2 (s#ss) γ)"
| "L (Or c1 c2) (s#ss) γ = Min_gamma γ
     (L c1 (s#ss) γ)
     (L c2 (s#ss) γ)"
| "L (Next c) (s#ss) γ = L c ss γ"
| "L (Always c) (s#ss) γ = Max_gamma γ (L c (s#ss) γ)
     (if ss = [] then 0 else (L (Always c) ss) γ)"
| "L (Eventually c) (s#ss) γ = Min_gamma γ (L c (s#ss) γ)
     (L (Eventually c) ss γ)"
| "L (Until c1 c2) (s#ss) γ = Min_gamma γ (L c2 (s#ss) γ)
     (Max_gamma γ (L c1 (s#ss) γ) (if ss = [] then 0
       else (L (Until c1 c2) ss γ)))"
| "L (Release c1 c2) (s#ss) γ = Max_gamma γ
     (L (Eventually c1) (s#ss) γ)
     (Min_gamma γ
```

```
(Max_gamma γ (L c1 (s#ss) γ) (L c2 (s#ss) γ))
(Max_gamma γ (L c2 (s#ss) γ) (L (Release c1 c2) ss γ)))"
```

Listing 4.12: The $\mathscr{L}$ function

We examine some of the components of the definition and explain them to illustrate how the function works. In the above formulation, the definition for `Lequal` shows that if the first state-value is smaller or equal to the first, $\mathscr{L}$ produces 0, equivalent to logical truth. If this is not true, a positive value will be returned equal to their difference. This works in a similar way to a soft rectification function – its limit as γ tends to zero is identical and proven in Isabelle. As another remark, our comparison operators for $\mathscr{L}$ are defined in terms of $\leq$ and $\neq$, with the other two comparisons, $<$ and $=$ defined using them.

For all the LTL operators, the $\mathscr{L}$ function, in common with the `eval` function, recurses over the constraint from the outside in, and recurses down the path as required for temporal operators. Innes and Ramamoorthy do not use a recursive definition [71], which though fine for relatively simple LTL operators such as `Always`, leads to a much more involved formulation for more complicated ones such as `Until`.

In particular, our recursive definition of $\mathscr{L}$ against the `Until` operator is significantly simpler than that given in their paper because `Until` is logically equivalent to the recursive version shown below (as proven as part of Listing 4.5):

$$\rho_1 \mathscr{U} \rho_2 = (\rho_2 \vee (\rho_1 \wedge (\mathscr{N}(\rho_1 \mathscr{U} \rho_2)) \tag{4.3}$$

Note also that when taken over an empty path, $\mathscr{L}$ takes value 1, which is equivalent to any constraint on the empty path evaluating as false. This means that the `Next` constraint matches expectations (as discussed in Section 4.2.2) at the end of a finite trace. However, this understanding of how $\mathscr{L}$ treats the empty path means we must specify a slightly different behaviour for how $\mathscr{L}$ handles finite paths for the `Always` and `Until` constraints. As $\mathscr{L}$ recurses down the path for these two, if it reaches the end of the path, it should return a 0 value (equivalent to true) if every state it has recursed through meets the specified constraints.

Once we formulate $\mathscr{L}$, via a series of lemmas and an inductive proof on the `constraint` datatype, we show that it has the expected properties, i.e. that it is always non-negative, differentiable, and sound with respect to the `eval` function (as stated at the start of Section 4.3).

We begin, though, by showing that the formalisation of $\mathscr{L}$ is always non-negative, under a γ value of 0:

```
lemma L_nn:
  fixes ss :: path and γ :: real and c :: constraint
  shows "L c ss 0 ≥ 0"
```

Listing 4.13: The `L_nn` lemma

Next, we prove the critical soundness property holds. We begin by proving that if $\gamma = 0$, then $\mathscr{L}$ is zero over some constraint and path if and only if the constraint is true on that path:

```
lemma L_eval: "((L c ss 0) = 0) = (eval c ss)"
```

Listing 4.14: Equivalence of $\mathscr{L}$ to `eval`, if $\gamma = 0$

This proof proceeds by induction along the path `ss` and then the constraint `c`. It is relatively straight-forward for each possible operator (proceeding by induction along the path for temporal operators), except for `Until` and `Release`, both of which are rather complicated. This is because they are temporal and operate over two constraints; thus the induction hypothesis must be defined against both constraints in the proof and the final proof completed against both.

From this, we show that $\mathscr{L}$ is continuous, from which the soundness property must hold as required:

```
lemma L_cont_0:
  fixes c :: constraint and ss
  shows "isCont (λγ. L c ss γ) 0"
```

Listing 4.15: Continuity of $\mathscr{L}$ at $\gamma = 0$

The function `IsCont` here is an existing Isabelle function which checks for continuity of a function at a given value. The proof splits into different lemmas, checking continuity when $\gamma < 0$ (trivially, as it is constant), then for other value of γ. We check for continuity when $\mathscr{L}$ is applied against an empty path (which was trivial). Lastly, we check against a non-empty path, with the added assumption we know that it is true for all constraints against a path one time step shorter. This is used like an induction hypothesis when temporal operators would normally proceed down the path. This proof against a non-empty path proceeds by induction on the constraint, using our induction hypothesis against the path as required. Once complete, we combine it with the empty path proof with an argument by induction down the path, thus covering both induction by constraint and by path, leading to our final proof. We needed to show as part of this proof continuity of our various soft gamma functions when $\gamma = 0$.

With this continuity shown, though, the final proof of soundness is straightforward:

```
theorem L_sound: "((λγ. L c ss γ) −0→ 0) ⟷ (eval c ss)"
```

Listing 4.16: Soundness of $\mathscr{L}$ stated in Isabelle

We have now formally defined an LTL-based, soft loss function $\mathscr{L}$ that, for any constraint and finite trace, tends to 0 as its gamma parameter tends to 0, if and only if the constraint is satisfied over that trace.

4.3.3 Derivative of the loss function

A derivative $d\mathscr{L}$ for the $\mathscr{L}$ function is constructed, to be used for gradient-based methods in PyTorch (see Section 4.4.2). The derivative must be defined with respect to each time step *i* and state-value index *j* at that time step along the finite trace.

```
function dL :: "constraint ⇒ path ⇒ real ⇒ int ⇒ int ⇒ real"
   where
  "dL c [] γ i j = 0"
| "dL (Comp (Lequal v1 v2)) (s#ss) γ i j = (if i ≠ 0
     then 0 else (dLequal_gamma_ds γ (s v1)
       (if eqint j v1 then (1::real) else 0) (s v2)
       (if eqint j v2 then (1::real) else 0)))"
| "dL (Comp (Nequal v1 v2)) (s#ss) γ i j = (if (i ≠ 0) then 0
     else (dNequal_gamma_ds γ (s v1)
       (if j = v1 then 1 else 0) (s v2)
       (if j = v2 then 1 else 0)))"
| "dL (Comp (Less v1 v2)) (s#ss) γ i j = dMax_gamma_ds γ
  (L (Comp (Lequal v1 v2)) (s#ss) γ)
    (dL (Comp (Lequal v1 v2)) (s#ss) γ i j)
  (L (Comp (Nequal v1 v2)) (s#ss) γ)
    (dL (Comp (Nequal v1 v2)) (s#ss) γ i j)"
| "dL (Comp (Equal v1 v2)) (s#ss) γ i j = dMax_gamma_ds γ
  (L (Comp (Lequal v1 v2)) (s#ss) γ)
    (dL (Comp (Lequal v1 v2)) (s#ss) γ i j)
  (L (Comp (Lequal v2 v1)) (s#ss) γ)
    (dL (Comp (Lequal v2 v1)) (s#ss) γ i j)"
| "dL (And c1 c2) (s#ss) γ i j = dMax_gamma_ds γ
  (L c1 (s#ss) γ)
    (dL c1 (s#ss) γ i j)
  (L c2 (s#ss) γ)
    (dL c2 (s#ss) γ i j)"
| "dL (Or c1 c2) (s#ss) γ i j = dMin_gamma_ds γ
  (L c1 (s#ss) γ)
    (dL c1 (s#ss) γ i j)
```

```
  (L c2 (s#ss) γ)
    (dL c2 (s#ss) γ i j)"
| "dL (Next c) (s#ss) γ i j = dL c ss γ (i-1) j"
| "dL (Always c) (s#ss) γ i j = dMax_gamma_ds γ
  (L c (s#ss) γ)
    (dL c (s#ss) γ i j)
  (if ss = [] then 0 else (L (Always c) ss) γ)
    (if ss = [] then 0 else (dL (Always c) ss) γ (i-1) j)"
| "dL (Eventually c) (s#ss) γ i j = dMin_gamma_ds γ
  (L c (s#ss) γ)
    (dL c (s#ss) γ i j)
  (L (Eventually c) ss γ)
    (dL (Eventually c) ss γ (i-1) j)"
| "dL (Until c1 c2) (s#ss) γ i j = dMin_gamma_ds γ (L c2 (s#ss) γ)
     (dL c2 (s#ss) γ i j) (Max_gamma γ (L c1 (s#ss) γ)
     (if (ss = []) then 0 else (L (Until c1 c2) ss γ)))
     (dMax_gamma_ds γ (L c1 (s#ss) γ) (dL c1 (s#ss) γ i j)
       (if (ss = []) then 0 else (L (Until c1 c2) ss γ))
       (if (ss = []) then 0 else (dL (Until c1 c2) ss γ (i-1) j)))"
| "dL (Release c1 c2) (s#ss) γ i j = dMax_gamma_ds γ
  (L (Eventually c1) (s#ss) γ)
    (dL (Eventually c1) (s#ss) γ i j)
  (Min_gamma γ (Max_gamma γ (L c1 (s#ss) γ) (L c2 (s#ss) γ))
  (Max_gamma γ (L c2 (s#ss) γ) (L (Release c1 c2) ss γ)))
    (dMin_gamma_ds γ (Max_gamma γ (L c1 (s#ss) γ) (L c2 (s#ss) γ))
      (dMax_gamma_ds γ
      (L c1 (s#ss) γ) (dL c1 (s#ss) γ i j)
      (L c2 (s#ss) γ) (dL c2 (s#ss) γ i j))
    (Max_gamma γ (L c2 (s#ss) γ) (L (Release c1 c2) ss γ))
      (dMax_gamma_ds γ
      (L c2 (s#ss) γ) (dL c2 (s#ss) γ i j)
      (L (Release c1 c2) ss γ) (dL (Release c1 c2) ss γ (i-1) j)))"
```

Listing 4.17: The $d\mathscr{L}$ function

The $d\mathscr{L}$ function formalisation is structured similarly to the formalisation of the $\mathscr{L}$ function, defined recursively over the components of the LTL$_f$ constraint passed to it and essentially follows from repeated applications of the chain rule. In defining it, extensive use is made of the derivatives of the soft-functions described in Section 4.3.1.

4.3.3.1 Correctness of $d\mathscr{L}$

While the formulation of $d\mathscr{L}$ may look intricate, the fact that it is indeed the correct derivative for our loss function (with respect to any of the components of the path) can be formally proven and thus guaranteeing that when used for backpropagation it will achieve the desired results.

In order to differentiate the `L` function with respect to a given state-value at a specific time, we must turn a `path` into a function of a single variable representing that state-value at that time. This is done by formalising the state-update function `update_state` and the recursive function `update_path`, which allows one to specify the value at a particular i and j:

```
fun update_state :: "state ⇒ int ⇒ real ⇒ state" where
  "update_state s j x = (λn. if n=j then x else (s n))"

primrec update_path :: "path ⇒ int ⇒ int ⇒ real ⇒ path" where
  "update_path [] i j = (λx. [])"
| "update_path (s#ss) i j = (if i = 0
     then (λx. (update_state s j x) # ss)
     else (if i < 0 then (λx. (s#ss))
       else (λx. s # (update_path ss (i-1) j x))))"
```

Listing 4.18: The update functions

Using this mechanism, $d\mathscr{L}$ is proven to be the derivative of the $\mathscr{L}$ function, with respect to the value at any i and j. In Isabelle, the theorem formalising this is as follows:

```
theorem L_has_derivative:
  assumes "γ > 0"
  shows "((λy. L c (update_path pth i j y) γ) has_field_derivative
     (dL c (update_path pth i j x) γ i j)) (at x)"
```

Listing 4.19: Correctness of $d\mathscr{L}$ stated in Isabelle

4.4 A PyTorch-compatible LTL loss function

With $\mathscr{L}$ and $d\mathscr{L}$ formalised in Isabelle, what remains is to integrate them into the PyTorch environment. Unfortunately, there does not exist a mechanism for generating Isabelle functions as Python code. Instead, we chose to generate intermediate representations of $\mathscr{L}$ and $d\mathscr{L}$ in OCaml since the recursive Isabelle functions can

be straightforwardly translated to type-safe OCaml ones and, moreover, there exists a Python library that can be used to call OCaml functions from within Python code [102, 91].

4.4.1 OCaml code generation

In order to produce computable code, we need to map the real numbers of Isabelle to floating points. As this is an approximation, it naturally has some scope for machine arithmetic errors, even though the code generated for the various functions is fully faithful to their definitions in Isabelle. So, assuming that the floating point computations are well-behaved, the OCaml functions will satisfy the properties that were proven for their Isabelle counterparts (e.g. that the OCaml-generated $d\mathscr{L}$ is the derivative of the OCaml-generated $\mathscr{L}$).

We make use of the code generation machinery of Isabelle, which provides code printing instructions for translating between real numbers in Isabelle to floating point numbers in OCaml [53] to generate an OCaml module `LTL_Loss`.

To translate from real types in Isabelle to floating point types in OCaml, an existing Isabelle theory called "Code Real Approx to Float" was used. However, it appeared to be out of date with respect to the version of OCaml being used and no longer produced working code. Thankfully, this was fairly easily adjusted for by a few additions to the code printing instructions contained therein (for example, referring to the current module used for floating point computation). As well as adjusting the theory for obsolete OCaml, we also had to write a few code printing statements for very simple constants (such as the real values 0 and 1) to generate code that relied upon them – again, this was easily done.

One potentially thornier issue is that the code printing did not interpret the Isabelle function `ln` as the correct function in OCaml – this was called against the real type in our work but defined against a type class in Isabelle's libraries. The code generator was unable to translate the function to a function against floating points in OCaml as required, as it was attempting to translate the function against a type class. To work around this issue, we defined a version of the function restricted to the real type, proved it was equal to the type class version of the function under those restrictions, and then defined code printing using that function.

4.4.2 PyTorch integration

PyTorch is a Python library for deep learning [112]. It is used here with our experiments, some of which tests whether the generated code from Isabelle enables the intended training within the framework developed by Innes and Ramamoorthy [71], and some of which demonstrate that we can extend those tests (see Section 4.5). Crucial to PyTorch (and other similar libraries) is the tensor datatype that is used to represent inputs and outputs over a network of mathematical operations in a typically multi-dimensional array-like form.

A sequence of tensors and operations performed thereon are recorded in a directed acyclic graph known as a computational graph [112]. A tensor passed to this graph as an input traverses through it, changing as per the operations carried out by the graph. PyTorch's automatic differentiation engine `torch.autograd` computes the gradients of the tensor with respect to each of the elements of the tensors in the graph, via successive application of the chain rule of differentiation. This algorithm enables a fundamental approach to neural network training, backpropagation [130], where a network's weights are iteratively adjusted according to the gradient of a computed loss with respect to themselves.

In order for the LTL loss function to form part of a computational graph in PyTorch, it must be implemented as a subclass of `autograd.Function`. This class, `LTL_Loss` as per our OCaml module, is parameterised by an LTL constraint, represented as an OCaml expression, the constants for comparison, and a value for γ.

As noted in Section 4.2.1, an instantiation of a path can be represented by a matrix:

$$P = \begin{bmatrix} \mathbf{s}_0 \\ \vdots \\ \mathbf{s}_{N-1} \end{bmatrix} = \begin{bmatrix} t_{00} & \cdots & t_{0(K-1)} \\ \vdots & \ddots & \vdots \\ t_{(N-1)0} & \cdots & t_{(N-1)(K-1)} \end{bmatrix}$$

Here, each *row* is a representation of a state along the path. This matrix is useful as it can be implemented as a PyTorch tensor.

To interface the OCaml code for $\mathscr{L}$ and $d\mathscr{L}$ with `autograd`, two methods are defined:

1. `forward`: this applies the loss function to a 2-dimensional input tensor representing a path, using the OCaml bindings to apply the OCaml representation of $\mathscr{L}$. If a 3-dimensional input tensor is given, with the leading dimension indexing separate paths, an average loss is computed.

2. `backward`: this computes the derivative of the scalar loss output with respect to the input tensor. Iterating over each element of the tensor, we call the OCaml representation of $d\mathscr{L}$ to compute the derivative of the loss with respect to it.

With this, the class is functionally identical to a differentiable PyTorch operation on tensors, as discussed in Section 4.4.2.

4.5 Experiments

With the Isabelle-formalised loss function $\mathscr{L}$ and its explicit derivative $d\mathscr{L}$ fully available as a generic `autograd` function in PyTorch, we then verify that it can achieve experimental results that include and extend those of Innes and Ramamoorthy [71].

We take the experimental models from their work and extend them to show the improvements provided by this approach. Specifically, after replicating some of the main results from their work using the method, we also demonstrate an application of the `Until` constraint whose loss evaluation was not implemented in their code.

Despite our formal proofs of the properties of our loss function and our consequent confidence in its behaviour, the purpose of our work is to allow a neural network to learn from logical constraints. As this involves work outside the scope of the theorem proving we have done – software engineering in OCaml and Python – we must demonstrate the success of our approach experimentally.

4.5.1 Domain setup

Each of the tests takes place in a 2-dimensional planar environment with Cartesian co-ordinates. The training data follow a spline-shaped curve consisting of $N=100$ sequenced points in the plane following the curve with small random perturbations, simulating a demonstrator moving via some trajectory from the origin to some destination in the plane. A feed-forward neural network is trained to learn a dynamic movement primitive (DMP) [133, 132] to follow this trajectory. A dynamic movement primitive is a way to model complicated motion in some space by planning a trajectory that is relatively easy to adjust.

4.5.2 Unconstrained training

Let D denote the trajectory of the demonstrator along the spline and Q denote the trajectory of the DMP learned by the neural network. Both Q and D are $N \times 2$ matrices. Moreover, let Q_i denote the row vector at index i of Q (likewise for D_i and D). The per-sample imitation loss, L_d, for this sample pair is given by $L_d(Q,D) = \frac{1}{N}\sum_{i=0}^{N-1} \|Q_i - D_i\|^2$.

Intuitively, L_d penalises the learned trajectory for deviating from the demonstrator. For a batch of samples, we compute the average imitation loss. L_d is used as the sole loss in the training of the neural network over 200 epochs using the Adam optimizer [77] with a learning rate of 10^{-3}.

4.5.3 Constrained training with LTL$_f$

First, it is important to distinguish that for a given sample, its *trajectory* Q is not necessarily the same as the corresponding *path* P to be reasoned over by LTL. The trajectory Q encodes the x and y co-ordinates of each point, but the path P encodes the entire state of the environment over time, as well as these co-ordinates. The state may include intermediate functions of x and y as well as any constants for comparison (for example, distance from a certain point), that are reasoned over by the LTL constraint.

With this in mind, consider a differentiable function g which maps Q to a path P, an LTL constraint ρ which reasons over that path, and a relaxation factor $\gamma > 0$. We can incorporate this constraint into the learning process by augmenting the per-sample loss function to be minimised to give the full loss L_{full}. $L_{full} = L_d(Q,D) + \eta\mathscr{L}(\rho, g(Q), \gamma)$, where η is a positive constant representing the weighting of the constraint violation against the imitation loss. This weighting can be adjusted to reflect the priority in satisfying the constraints relative to following the demonstrator accurately.

The same training procedure is repeated as for the unconstrained case, but with this new loss function and with $\gamma = 0.005$. In PyTorch, the loss $\mathscr{L}$ is implemented by an instantiation of `LTL_Loss` (mentioned in Section 4.4.2) with ρ and γ as arguments.

There are 4 different problems. Tests 1, 2, and 4 are similar to those of Innes and Ramamoorthy [71], while Test 3 evaluates the `Until` constraint, which is a non-trivial extension to their available constraints:

1. **Avoid**: The trajectory (always) avoids an open ball of radius 0.1 around the point $o = (0.4, 0.4)$. At each time step, the Euclidean distance between the trajectory

	Unconstrained		Constrained	
Test	L_d	$\mathscr{L}$	L_d	$\mathscr{L}$
Avoid	0.0047	0.0789	0.0146	0.0239
Patrol	0.0052	0.1238	0.0312	0.0049
Until	0.0036	0.0970	0.0158	0.0207
Compound, $\eta = 1$	0.0044	0.1612	0.0318	0.0291
Compound, $\eta = 4$	0.0055	0.1611	0.0358	0.0269

Table 4.1: The losses associated with each test, averaged over 5 iterations

and o is computed, and denoted as p_{do}. The LTL constraint becomes: $\Box(0.1 \leq p_{do})$.

2. **Patrol**: The trajectory eventually reaches $o_1 = (0.2, 0.4)$ and $o_2 = (0.85, 0.6)$ in the plane. With Euclidean distances p_{do_1}, p_{do_2}, this constraint becomes: $(\Diamond(p_{do_1} \leq 0)) \wedge (\Diamond(p_{do_2} \leq 0))$. Note that the comparison $=$ is not used as both p_{do_1} and p_{do_2} are non-negative by construction and this formulation has a lower computational cost.

3. **Until**: The y co-ordinate, p_y, of the trajectory cannot exceed 0.4 until its x co-ordinate, p_x, is at least 0.6: $(p_y \leq 0.4)\ \mathscr{U}\ (0.6 \leq p_x)$.

4. **Compound**: A more complicated test combining several conditions. The trajectory should avoid an open ball of radius 0.1 around the point $o_1 = (0.5, 0.5)$, while eventually touching the point $o_2 = (0.7, 0.5)$. Further, the y co-ordinate of the trajectory should not exceed 0.8. With p_{do_1} and p_{do_2} defined in the same way as before, this compound constraint is represented as: $(\Box(0.1 \leq p_{do_1})) \wedge (\Diamond(p_{do_2} \leq 0)) \wedge (\Box(p_y \leq 0.8))$.

For a more concrete explanation of the role of function g, consider the Avoid test. Here, g is defined to act row-wise on a trajectory Q, producing new row vectors whose elements are the Euclidean distance p_{do} and the constant 0.1, as these are the only quantities reasoned over by the LTL constraint.

4.5.4 Results

When the tests are run, first the neural network is trained ignoring the logical constraint, then the constraint is included as part of its loss calculations and its training. The

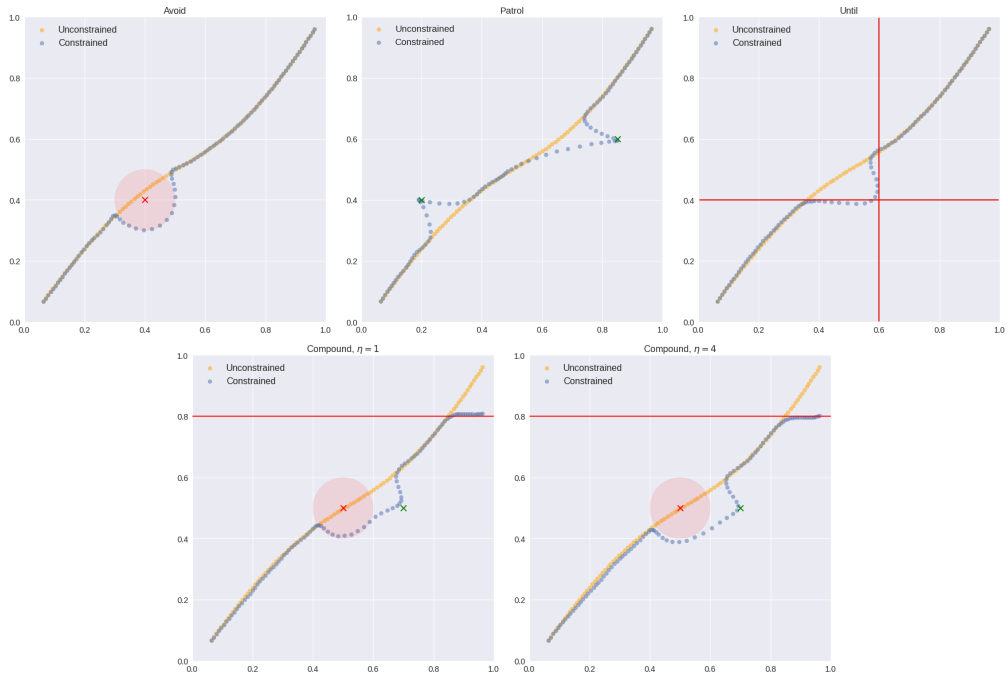

Figure 4.2: Avoid, Patrol and Until tests; and the Compound test, with $\eta = 1$ and $\eta = 4$

differences between the two sets of results demonstrates the effectiveness of the logical constraint as used in the loss function for the training. The loss figures are shown in Table 4.1.

The losses in the table show that when unconstrained training is performed, understandably there are high constraint losses. These are substantially improved when the training is constrained, although the imitation losses increase as the trajectory outputted by the neural network no longer follows the training trajectory as closely.

Different constraints respond with different loss values based on the specific definitions used, so care should be taken when making comparisons between the different constrained tests (save for the two differently weighted compound tests). Even though in most of the tests the constraints are satisfied, the constraint losses are not zero. Given that the soft $\mathscr{L}$ function has a positive γ parameter, this is expected. By the soundness theorem (Listing 4.16), the constraint losses would be reduced further with smaller values of γ.

Visually, one can confirm that the constrained training is actually satisfying the specifications of the Avoid, Patrol, and Compound (for $\eta = 4$) tests in Fig. 4.2. The Until and Compound (for $\eta = 1$) tests are not fully satisfied (Until starts crossing the line $y = 0.4$ slightly early, and Compound briefly enters the open ball and doesn't quite reach its goal at o_2), but clearly exhibit altered behaviour towards meeting the expected

Test	Avoid		Until		Patrol	
	L_d	$\mathscr{L}$	L_d	$\mathscr{L}$	L_d	$\mathscr{L}$
Unconstrained	0.0037	0.0780	0.0071	0.0991	0.0048	0.1237
$\eta = 0.1$	0.0072	0.0503	0.0026	0.0852	0.0027	0.1134
$\eta = 0.5$	0.0102	**0.0242**	0.0143	**0.0219**	0.0244	0.0071
$\eta = 1.0$	0.0108	0.0238	0.0161	0.0210	0.0307	0.0048
$\eta = 3.0$	0.0165	0.0233	0.0186	0.0192	0.0481	**0.0003**
$\eta = 5.0$	0.0118	0.0233	0.0248	0.0174	0.0691	0.0011

Table 4.2: The losses associated with each test with varying η figures. Results in **bold** represent the greatest percentage improvement in $\mathscr{L}$ from the previous η value for a given constraint test.

constraints. The slight violations happen because, in addition to the constraint loss, the trajectory loss plays a part in forming the neural network's training. These losses work against each other in the tests. Overall, though, these results demonstrate the clear effectiveness of the logical constraint in changing the learned behaviour of the DMP.

Examining the Compound test (for $\eta = 1$) more closely, we note that the trajectory slightly wanders within the open ball and does not quite reach the point desired. To verify how weighting affects this, the test can be run again with an increased weighting value, namely $\eta = 4$, applied to the constraint loss and the effects become obvious – the trajectory now properly avoids the ball and gets closer to the point required. Altering the η value clearly impacts the results, and we will examine this effect in more detail in section 4.5.5.

At the conclusion of the experiments it is clear that the certainty of the approach comes at a cost in performance: running a full test can take anywhere from a few minutes to a few hours for the more complicated constraints, and the performance cannot be enhanced by running on a GPU as the OCaml functions used cannot take advantage of this. There are several possible approaches to resolving this and improving performance as discussed in section 4.7.

4.5.5 Altering the η weighting value

The hyperparameter η is used to weight the loss due to LTL_f constraint violation ($\mathscr{L}$) against the loss due to comparison with training data (L_d), as briefly discussed in

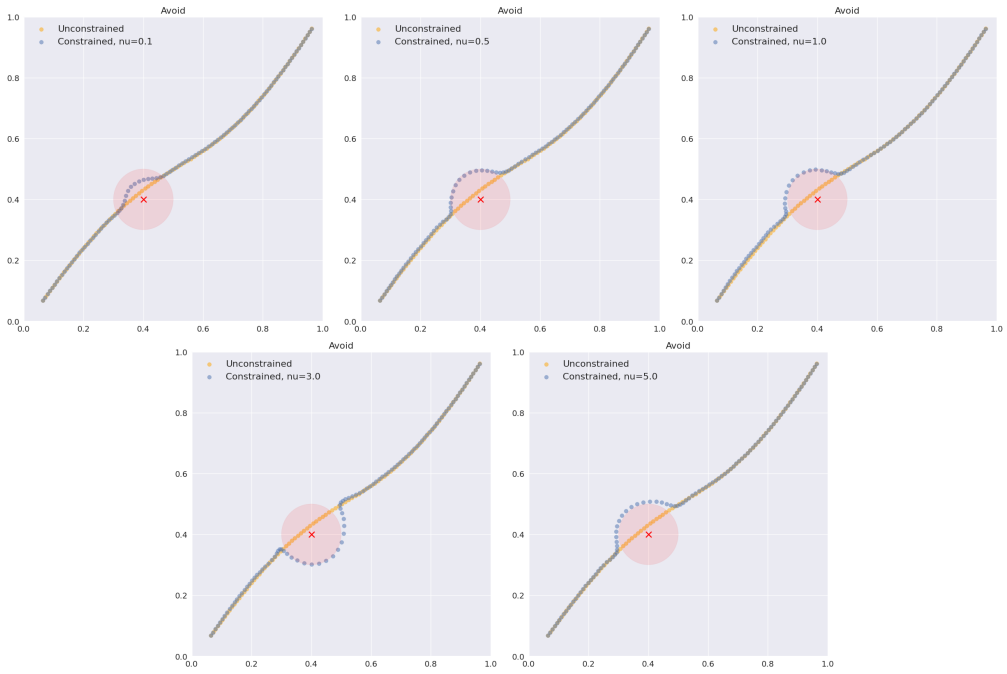

Figure 4.3: Testing the Avoid constraint with varying η, (0.1, 0.5, 1.0, 3.0, and 5.0)

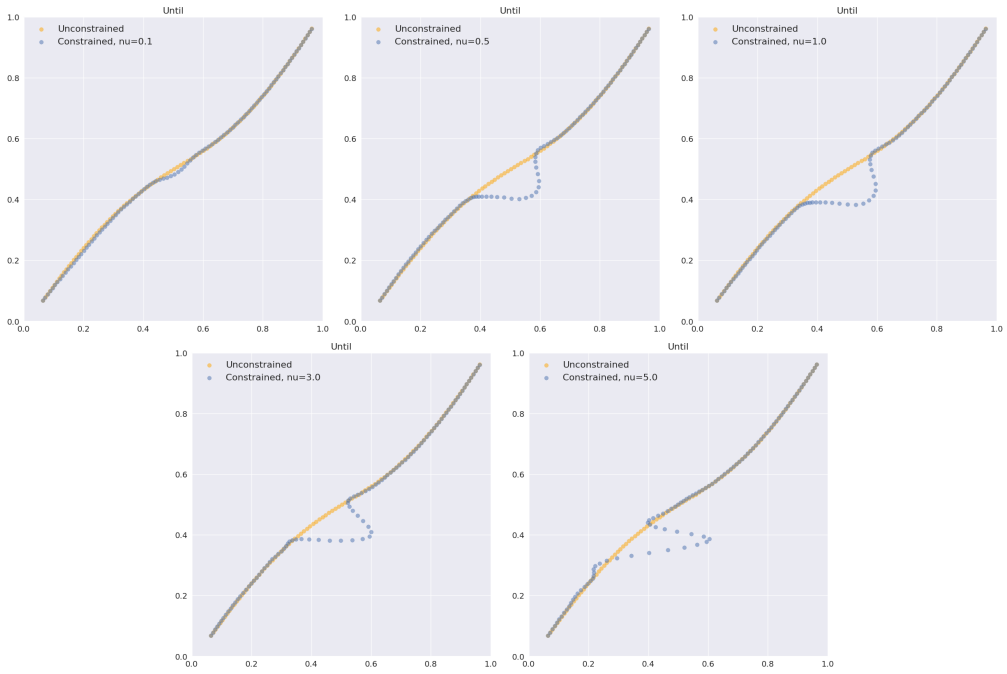

Figure 4.4: Testing the Until constraint with varying η, (0.1, 0.5, 1.0, 3.0, and 5.0)

section 4.5.3. The total loss (L_{full}) used by the neural network to train is calculated as $L_{full} = L_d + \eta\mathscr{L}$. The higher the value of η, the greater the contribution towards the neural networks training is provided by constraint violation. We might expect to see, then, that the higher η is, the more pronounced the focus on constraint violation

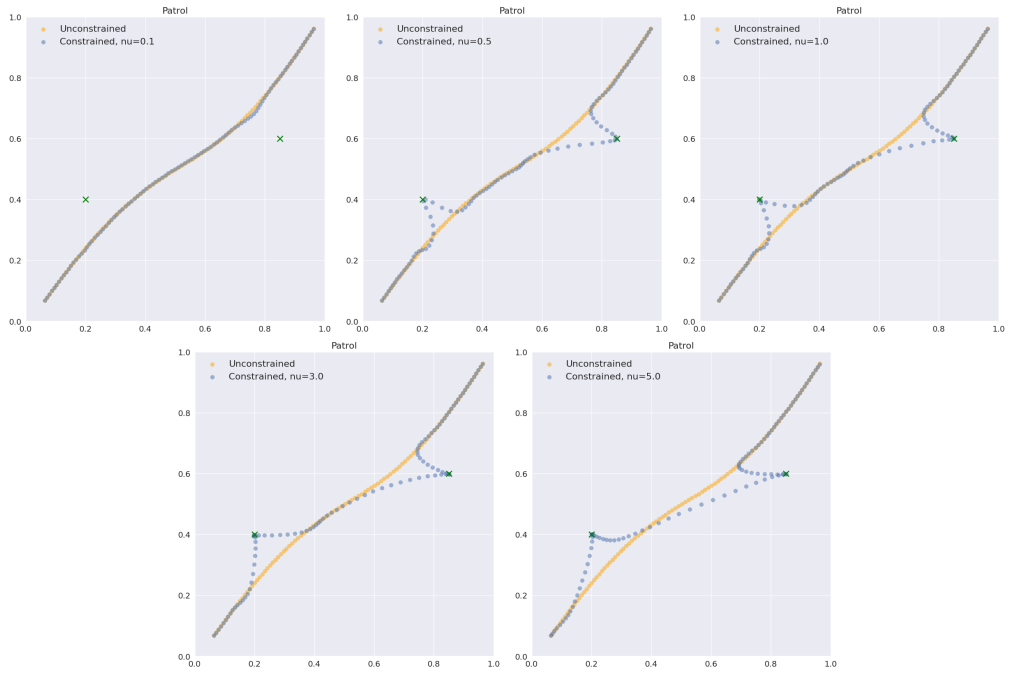

Figure 4.5: Testing the Patrol constraint with varying η, (0.1, 0.5, 1.0, 3.0, and 5.0)

will be, and consequently, the lower $\mathscr{L}$ will be after the neural network has trained. However, we also might expect there to be some unnnecessary distortion to the training path being learned if η is too high. For this reason, it is important to test the effect that varying η values might have.

To demonstrate the effect of varying the η weighting value, we repeated the Avoid, Until and Patrol experiments, varying η each time. We chose to experiment with η values of 0.1, 0.5, 1.0, 3.0 and 5.0. These values were chosen as they cover a difference of an order of magnitude and allow us to examine the difference η might make if it weights $\mathscr{L}$ to contribute both much less and much more than L_d.

The loss results are displayed in Table 4.2. The graphical results for the Avoid experiments are shown in figure 4.3, for the Until experiments in figure 4.4, and for the Patrol experiments in figure 4.5. From these results, there are several conclusions that are clear:

1. An η value of 0.1 produces a very small difference in behaviour from having no loss due to constraint violation. Only a minor deviation from the training path is observable, and in all of the constraints, the constraint is clearly not satisfied with this value of η.

2. Increasing η values above this leads to an improvement in constraint satisfaction, observable both visibly in the result graphs and also in the table of losses. All

tests appear to be satisfiable by sufficiently increasing η. The η-weightings with greatest percentage reduction (in $\mathscr{L}$) from the next lower η value have been highlighted in bold in Table 4.2. Note that for the Patrol result, the improvement at $\eta = 0.5$ (from $\eta = 0.1$) is 93.74%, almost as good as at $\eta = 3.0$ (from $\eta = 1.0$, 93.75%). This does seem to indicate that the relatively low η value of 0.5 is sufficient for substantial improvement in loss values measuring constraint satisfaction.

3. Varying the η figure appears to have a different impact depending on the constraint being measured. For example, observe that with the Avoid experiment, increasing the η result from 0.1 to 0.5 leads to a reduction in $\mathscr{L}$ from 0.05 to 0.02, and subsequent increases do not produce a much greater reduction. Whereas, for the Patrol experiment, further significant reductions are seen increasing η from 0.5 to 1.0. This may be a consequence of the relative simplicity of the Avoid constraint, involving only a small number of operations. By simplicity here, we mean the number of mathematical operations involved in performing a single check, which depends both on the number of recursions down the path of the `L` function for each operator, the length of the path, and the subconstraints being checked against.

4. Increasing η appears to have a moderate negative impact on the training loss L_d. This is to be expected, as to meet the LTL_f constraint of the test, we are necessarily varying from the training path. However, the distortion from the training path is only strongly visible in the graphs when η is increased to 5.0, and even then, the distortion does not appear to be great – the path is still followed.

5. The path taken around the area to be avoided in the Avoid test is almost always to the left of the avoided area, travelling above it. However, when η was 3.0, a path to the right, below the area was taken. It's unclear why this difference occurs, as the typical path is resumed in the $\eta = 5.0$ test. It may be simply down to randomness in the early stages of the neural network's learning process.

6. Each experiment involving a single constraint and a specific η-weighting took several hours to run, and this meant that replication of the experiments was costly in terms of time. For that reason, the results here are taken from a single experiment for each value of η and each constraint rather than being averaged across several experiments. Given the relative consistency of the results across those

values and constraints, they appear to be representative.

Overall, we can say that there appears to be some tolerance to variable η values in these experiments. Clearly, it needs to be sufficiently large (certainly above 0.1 in each of these experiments), but once it is large enough, subsequent increases do not lead to a distortion of training behaviour until it is around an order of magnitude higher. The η hyperparameter is clearly significant but thankfully need not be tuned to too fine a degree for the method we are using to produce results that avoid constraint violation.

4.6 Related work

There has been some work aimed at unifying propositional logical constraints with neural networks that have probabilistic outputs [167]. Hu et al. trained a network to follow a rules-based "teacher" [69] while work by Li et al. incorporates first-order logic rules directly in the network design, with the aim of guiding training and prediction [93]. Other authors modify the loss function based on the satisfaction of logical constraints to train a neural network [10, 41, 101]. On the reinforcement learning front, there has been recent work, e.g. on specifying policy learning via LTL instructions [84, 153]. None of the above approaches involves any theorem proving or formal verification like the current work– we have a new approach to defining loss functions used for this kind of neurosymbolic learning.

Of note is existing work using Signal Temporal Logic (STL) to enforce logical constraints upon deep learning [92]. This work can be viewed as a development of the work performed by Innes and Ramamoorthy that our approach is inspired by, and thus indicates a potential direction for future work. This introduces a Python module, `stlcg`, which generates a loss for a neural network based on the robustness trace of an STL constraint applied against a signal (a continuous time analogue to a path). Again, there is no formal verification of correctness, but there are a number of points of interest: the use of soft functions is made explicit here as it is in our work, the logic itself is similar (although extended to continuous time), and the loss function itself is explicitly conceived as a computational graph, with the autodifferentiation of PyTorch used for those graphs. The paper uses a rectified linear (ReLU) function to account for the fact that robustness in STL can be positive (meaning a constraint was well satisfied) – this prevents a well satisfied constraint being pursued at the expense of imitation training (as the robustness value, if used directly, could offset any imitation

losses in an instance of training – see Section 4.5.3 for more discussion on how these values interact). Our theorem proving approach using code generation from Isabelle (as discussed in Section 4.4) means we cannot directly work on PyTorch tensors in the way that their Python module can, but this could indicate a direction for future work.

A distinct but related strand of work centres around the formal verification of neural networks, as discussed in Chapter 1. This typically involves using SMT solvers or MILP programs to formally verify properties of neural networks [39, 70, 74, 137]. These SMT solvers typically verify simple propositional logic constraints over boolean variables, and in any case are not used to train the network but to check it. This is distinct from our objective, where we are formally verifying the logical system used to train a neural network and automatically generating code for the actual training.

Affeldt et al. [2] have formalised a general differentiable logic (of which the form employed in our work in this chapter is an example) in Coq, to provide a uniform description of these logics. They also are able to verify important properties such as soundness and that derivative functions against a differentiable logic can be used in gradient descent based learning processes.

4.6.1 Comparison with Innes and Ramamoorthy's work

As mentioned already, the work is motivated by that of Innes and Ramamoorthy [71]. However, on examination of their approach, and when comparing the fidelity of the code with the paper's descriptions, we uncovered several weaknesses that we believe support the need for a more formal neurosymbolic pipeline.

In particular, the Python code for their logical formulation does not always match that given in their paper: the "$\neq$" comparison, for example, is given in the paper as an indicator function, but in the Python script, it is defined as $a \neq b \iff (a < b) \vee (b < a)$. Moreover, as already mentioned, although presented in their paper, they do not encode a component of the loss function for the LTL `Until` operator, meaning they were not able to cover any tests involving it – something which we can do in our work (see Section 4.2.2).

There are also some aspects of LTL in Innes and Ramamoorthy's paper that remain implicit and are not discussed – notably its use of LTL over finite traces, which has a distinct semantics from the more usual formulation of LTL over infinite traces (as discussed in Section 4.2).

The primary advantage of our work in comparison to theirs resides in the fully rig-

orous specifications and proofs, with the training constraints guaranteed via systematic code generation from the specifications. This guarantee means one can know that the code generated will have the properties that were established for it during the theorem proving stage of our pipeline, and that the implementation will *fully* match the specification. Our work demonstrates that using this approach with any system of logic, and implemented into any system of code, we can enhance trust that the results will match specifications..

4.7 Conclusion

We have demonstrated that using a theorem proving approach, one can formulate a deep embedding of LTL_f in higher-order logic and use this to fully formalise logical constraints for training, as well as the loss function and its derivative. Code can then be generated for the whole logical framework and integrated with PyTorch. Furthermore, the experimental results show that these constraints successfully changed the training process to match the desired behaviour. Thus, we believe that this work provides much stronger guarantees of correctness than one could expect from an ad-hoc implementation of logical operators made directly in Python.

Our specific contributions here are:

1. We formalised a deep embedding of linear temporal logic over finite traces and proved a variety of results over it.

2. We formalised differentiable versions of a maximum and minimum function and their derivatives, proving each was correct in the limit of their softening parameter γ.

3. We formalised a function that can measure the loss associated with a breach of an LTL_f constraint over some finite path of states, proving its soundness with respect to the semantics of LTL_f.

4. We used Isabelle to generate OCaml code for our LTL_f loss function and embed it in a PyTorch neural network.

5. We demonstrated that the generated code worked with this neural network to allow for a logically constrained learning process, verifying this experimentally, and testing it with varying hyperparameter choices.

There are some practical limitations to the current work. In particular, training the neural network can be slow because OCaml functions are used to compute the loss and its derivative with respect to each possible input, and these functions run outside the highly optimised PyTorch infrastructure. There are some potential ways of addressing this shortcoming, which we discuss further in Section 5.1.

The approach of this work is generic, so in principle a different formalism, e.g. a continuous-time logic, could be used instead of LTL_f. Moreover, by formalising the derivative of the loss function, the potential to reason formally about the traversal of the loss surface during gradient descent is unlocked. Given the results, we believe this work opens the way to a tighter integration between fully-formal symbolic reasoning in a theorem prover and machine learning.

Chapter 5

Conclusion

We believe that our work has demonstrated that formal mathematics has benefits to offer the artificial intelligence field. We have shown that it can contribute to an increased certainty in the fundamental mathematics of reinforcement learning, and also that it can provide a way to formally verify the properties of complex systems that can be interpreted as MDPs. Lastly, we demonstrated formal mathematics can be used to practically implement a new method in deep learning with greatly increased guarantees of correctness compared to a manual implementation.

Throughout this work, we have used Isabelle as an interactive theorem prover to gain various guarantees of correctness. In the reinforcement learning portion of the work, this involved formalisations of existing mathematical work, extending the existing libraries of Isabelle to cover the new ground. In the deep learning portion of the work, we formalised a deep embedding of LTL_f and a novel process to implement code based on it in a deep learning network.

Our work began with an extension of Hölzl's mechanisation of Markov decision processes (MDPs) to consider rewards (in section 2.4.1). From this formalisation, we derived Bellman's equation in both scalar and vector form (sections 2.4.4 and 2.5.8). By doing this, we showed that our formalisation matches the usual mathematical understanding of an MDP with rewards.

In order to prove the existence of an optimal policy for negotiating a MDP, several results were required along the way. We showed that the reward an agent navigating a MDP earns will converge to a finite number if either future reward is discounted or a terminal state is guaranteed (section 2.4.3). Under these circumstances, having established that future reward can be evaluated as a finite real number, we were able to look further at proving formally whether an optimal policy exists.

To do this, we proved that Gelfand's formula can establish the invertibility of a square matrix and find its inverse (section 2.5.5). We also proved that we can produce a formal vector based calculation that gives the expected reward for a discounted MDP without requiring the use of an infinite series or recursion (section 2.5.8). Using these tools to formalise Puterman's proof of the existence of an optimal policy on an MDP, we found a flaw in the proof and resolved it (section 2.5.8). Finally, we formally proved the existence of a universally optimal policy (section 2.5.9).

With this knowledge gained, we formalised value and policy iteration, two dynamic programming techniques, and proved that they find ε-optimal and optimal policies respectively (section 2.6).

To demonstrate the utility of our approach, we formalised a game with potentially complicated strategies. We were able to prove that it was possible to formally interpret the game as an MDP, thereby carrying the proofs for general MDPs into the specific case of the game. Thus we have proven that an optimal strategy exists and we have tools to find it (section 2.7).

In order to examine reinforcement learning against MDPs, we investigated stochastic processes. We formalised the Q learning process and described the tools we would need to formalise the processes that underlie it (section 3.1.1.2).

These stochastic processes depend on the behaviour of sequences of random variables. To show that it was possible to discuss these, we needed to demonstrate that the elements of the sequence were integrable under a number of different combinations given a basic assumption (section 3.2.3). We also formalised the natural filtration of a sequence of random variables (section 3.2.4.1), and thus we were able to prove facts about conditional expectations dependent on parts of those sequences.

We continued our formalisation of stochastic processes by formally proving that the Dirichlet test and summation by parts produce the expected results (section 3.3.2). We also formally verified the divergence comparison test for sequences, the Du Bois-Reymond test and an extension of it usable with non-negative sequences (section 3.3.5). All of these tools were vital to proving our results.

We finished our examination of stochastic processes by formally proving Kolmogorov's inequality (section 3.3.3), and to conclude, the Dvoretzky stochastic approximation theorem and its extension (section 3.3.6).

To examine the benefit of using formal mathematics in neurosymbolic work, we looked at ways to use formal theorem proving to contribute to a practical machine learning process. Our focus was on building a differentiable loss function that a neural

network could use to learn to satisfy constraints in linear temporal logic.

We began by formalising a deep embedding of linear temporal logic over finite traces (LTL_f, in section 4.2). As our loss function needed to be differentiable, we formalised a differentiable version of a maximum and minimum function that we would use in it (section 4.3.1).

We used these to formalise a differentiable loss function that we proved was sound with respect to the semantics of LTL_f (section 4.3.2). We used Isabelle to generate OCaml code for our LTL_f loss function and embed it in a PyTorch neural network (sections 4.4.1 and 4.4.2).

We were then able to demonstrate that the generated code works with this neural network to allow for a logically constrained learning process (section 4.5), by balancing the logical loss against the imitation loss normally used by the neural network. We examined the effect of changing the weighting constant η used in our experiments, adjusting the balance between the two losses, gaining insight into how it might affect similar experiments (section 4.5.5).

Our initial goal was to demonstrate that formal mathematics and logic could be beneficial to research in deep learning and reinforcement learning. This has been achieved in two ways: by showing that a formalisation of MDPs can be used to interpret a non-trivial example and provide proofs against it, and by demonstrating a formally verified loss function can be used to train a neurosymbolic network. We believe there is scope for further work to enhance and realise those benefits. There are two further steps that could be taken to this end with our reinforcement learning formalisation.

Firstly, our work on the mathematics of discrete reinforcement learning is very fundamental and would need extended to be able to reflect the kind of research ongoing in recent years. In particular, the mathematics of deep learning are an important component of deep reinforcement learning, focus of much contemporary research, and further work is required if it were to be formalised. This work would require further understanding of deep learning from the mathematical community.

Secondly, the idea that a reinforcement learning researcher could adapt their ideas to our formal model depends on their having skills in formal theorem proving that are not widespread. The possibility is definitely there, but it may not be realistic to expect this idea to be adopted quickly, and by doing so increase confidence in research results. Increased awareness of the enhanced trust formalisation can bring would make this more likely.

The work succeeds in providing a formal verification of the proofs underlying dis-

crete reinforcement learning, and by doing so advances the goal of formalising mathematics.

The work on logic in deep learning demonstrates clearly that a theorem proving approach can allow features of deep learning systems to be practically implemented in such a way as to provide guarantees of correctness. The experiments here make clear that not only does the formally proven loss function behave as expected, but that the the implementation process functions.

A repeated problem throughout the reinforcement learning work was uncovering similar work being carried out simultaneously by other parties: Vajjha and the team from IBM Research worked on MDPs and the Dvoretzky stochastic approximation theorem in the theorem prover Coq [154, 155]; and work on MDPs with rewards was carried out in Isabelle by Schäffeler [135, 134]. This certainly shows current interest in the area.

We believe our work has much to offer alongside these other endeavours. With reference to the MDP work, we uniquely chose to take a linear algebra approach. As outlined in Chapter 2, our reasons were to have a system of proof that matched the most widely used extant literature, enabling extensions using the same linear algebra approach. This resulted in us finding a small error in Puterman's proof which is widely cited currently (please see Section 2.5.8 for the details). We also believe that this makes our work more accessible and understandable to the reinforcement learning research community.

With reference to our work in probability theory, our work in integrability is unique to our approach, and demonstrates that a simple assumption can lead to a wide proof of integrability under a number of circumstances. Additionally, of course, our work is in Isabelle, and thus has a differing audience and approach to formalisation than that in Coq.

5.1 Future work

With reference to our work on MDPs, the most obvious way to extend this work would be to adapt the model to MDPs of infinite states and actions. The results would still follow, although they would require different proof methods, notably a different formalisation of the types used in Isabelle. In particular, our use of the vector and matrix types in Isabelle would need to be redefined as both are only valid with finite dimensionality.

However, it should be noted that Jaakkola's proof of Q learning convergence (discussed in Section 3.1.1.5) assumes a finite state MDP. By extending our work to cover infinite state MDPs, we would not be focusing on our primary goal – to formalise the mathematics of reinforcement learning. Nevertheless, it remains a potential avenue for future progress.

As regarding our proof of the Dvoretzky stochastic approximation theorem, the next step is to use it to finish a formalisation of Jaakkola's proof [72], for both Q learning and TD-λ learning convergence. This would certainly provide a more complete demonstration of the formalisation of the mathematics underlying reinforcement learning.

Our work on logic as a loss function can be extended in numerous ways; the work done is a proof that this concept works. It could be applied to different logical systems (signal temporal logic being an obvious choice [125]), or to different domains other than movement over some path. The principle of implementing code for incorporation into a deep learning process via formal specification and proof is demonstrated and admits many possibilities.

A secondary focus on the work here, though, would be in making the process more efficient. Currently, the OCaml code generated is not optimised for deep learning – it does not use the automatic differentiation of PyTorch (although as discussed in Chapter 4, this can be a benefit as we can then reason about its differentiation formally), and it does not gain the benefit of a GPU as native PyTorch tensor operations can.

There is a potential solution here: an implementation of PyTorch in OCaml and Haskell [103, 59]. With a formalisation of tensor operations, and an extension of the code generation, it should be possible to generate suitable code directly. Obviously, this would require substantial effort, but the potential gains, certainly in terms of practical and speedy learning, would be substantial also.

Merging the two main strands of the work we present here, potentially, it should be possible to formalise the process of inverse reinforcement learning [131]. Inverse reinforcement learning is a method of reinforcement learning where the agent learns the rewards by modelling the behavior of an agent negotiating the same environment, much as a neural net might learn by imitation. Recently, there has been some work on adapting inverse reinforcement learning using logical constraints [88]. It is possible that with such work, one could develop work based on both our formalisation of reinforcement learning and adapting logical constraints via a loss function.

References

[1] Andrew Aberdein. Deep Disagreement in Mathematics. *Global Philosophy*, 33(1):17, 2023.

[2] Reynald Affeldt, Alessandro Bruni, Ekaterina Komendantskaya, Natalia Ślusarz, and Kathrin Stark. Taming differentiable logics with Coq formalisation. *arXiv preprint arXiv:2403.13700*, 2024.

[3] Oguzhan Alagoz, Heather Hsu, Andrew J Schaefer, and Mark S Roberts. Markov decision processes: a tool for sequential decision making under uncertainty. *Medical Decision Making*, 30(4):474–483, 2010.

[4] Andrei Aleksandrov and Kim Völlinger. Formalizing piecewise affine activation functions of neural networks in Coq. In *NASA Formal Methods Symposium*, pages 62–78. Springer, 2023.

[5] Kenneth Appel and Wolfgang Haken. *The four-color problem*. Springer, 1978.

[6] J Marshall Ash. Neither a worst convergent series nor a best divergent series exists. *The College Mathematics Journal*, 28(4):296–297, 1997.

[7] Robert B Ash, B Robert, Catherine A Doleans-Dade, and A Catherine. *Probability and measure theory*. Academic press, 2000.

[8] Frank Ayres Jr and Elliott Mendelson. *Schaum's Outline of Calculus*. McGraw-Hill Education, 2013.

[9] Akhil Raj Azhikodan, Anvitha G. K. Bhat, and Mamatha V. Jadhav. Stock Trading Bot Using Deep Reinforcement Learning. In *Innovations in Computer Science and Engineering*, pages 41–49, Singapore, 2019. Springer Singapore.

[10] Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini, and Michael Spranger. Logic Tensor Networks. *Artificial Intelligence*, 303:103649, 2022. doi:https://doi.org/10.1016/j.artint.2021.103649.

[11] Jorge Baier and Sheila Mcilraith. Planning with First-Order Temporally Extended Goals using Heuristic Search. In *Proceedings of the National Conference on Artificial Intelligence*, volume 1, 01 2006.

[12] Mislav Balunovic and Martin Vechev. Adversarial training and provable defenses: Bridging the gap. In *International Conference on Learning Representations*, 2019.

[13] EN Barron and H Ishii. The Bellman equation for minimizing the maximum cost. *Nonlinear Analysis: Theory, Methods & Applications*, 13(9):1067–1090, 1989.

[14] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. *Advances in neural information processing systems*, 29, 2016.

[15] Nicole Bäuerle and Ulrich Rieder. *Markov decision processes with applications to finance*. Springer Science & Business Media, 2011.

[16] Bernard Beauzamy. *Introduction to operator theory and invariant subspaces*. Elsevier, 1988.

[17] Richard Bellman. A Markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[18] Alexander Bentkamp, Jasmin Christian Blanchette, and Dietrich Klakow. A formal proof of the expressiveness of deep learning. *Journal of Automated Reasoning*, 63(2):347–368, 2019.

[19] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[20] Patrick Billingsley. *Probability and Measure*. John Wiley & Sons, 2008.

[21] Paul du Bois-Reymond. Eine neue Theorie der Convergenz und Divergenz von Reihen mit positiven Gliedern. 1873.

[22] Daniel D Bonar and Michael J Khoury Jr. *Real infinite series*, volume 56. American Mathematical Soc., 2018.

[23] Richard J Boulton, Andrew D Gordon, Michael JC Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *TPCD*, volume 10, pages 129–156. Citeseer, 1992.

[24] Stephen Boyd and Jacob Mattingley. Branch and bound methods. *Notes for EE364b, Stanford University*, 2006:07, 2007.

[25] Achim D Brucker and Amy Stell. Verifying feedforward neural networks for classification in Isabelle/HOL. In *International Symposium on Formal Methods*, pages 427–444. Springer, 2023.

[26] Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. A unified view of piecewise linear neural network verification. *Advances in Neural Information Processing Systems*, 31, 2018.

[27] Efe Camci and Erdal Kayacan. Game of drones: UAV pursuit-evasion game with type-2 fuzzy logic controllers tuned by reinforcement learning. In *2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 618–625. IEEE, 2016.

[28] Mu-Fa Chen and Yong-Hua Mao. *Introduction to stochastic processes*, volume 2. World Scientific, 2021.

[29] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. In *Automated Technology for Verification and Analysis: 15th International Symposium*, pages 251–268. Springer, 2017.

[30] Mark Chevallier, Matthew Whyte, and Jacques D. Fleuriot. Constrained training of neural networks via theorem proving. In *Short Paper Proceedings of the 4th Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis*, volume 3311, pages 7–12. CEUR-WS.org, 2022.

[31] Davide Corsi, Enrico Marchesini, and Alessandro Farinelli. Formal verification of neural networks for safety-critical tasks in deep reinforcement learning. volume 161 of *Proceedings of Machine Learning Research*, pages 333–343. PMLR, 27–30 Jul 2021.

[32] Davide Corsi, Enrico Marchesini, Alessandro Farinelli, and Paolo Fiorini. Formal verification for safe deep reinforcement learning in trajectory generation. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 352–359. IEEE, 2020.

[33] Marco Cuturi and Mathieu Blondel. Soft-DTW: a differentiable loss function for time-series. *arXiv preprint arXiv:1703.01541*, 2017.

[34] Giuseppe De Giacomo and Moshe Y Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 854–860. Association for Computing Machinery, 2013.

[35] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[36] C Derman and J Sacks. On Dvoretzky's stochastic approximation theorem. *The Annals of Mathematical Statistics*, 30(2):601–606, 1959.

[37] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods Symposium*, pages 121–138. Springer, 2018.

[38] Aryeh Dvoretzky. On Stochastic Approximation. In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*, pages 39–55, Berkeley, Calif., 1956. University of California Press.

[39] Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer, 2017.

[40] Ethan Fast and Eric Horvitz. Long-term trends in the public perception of artificial intelligence. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[41] Marc Fischer, Mislav Balunovic, Dana Drachsler-Cohen, Timon Gehr, Ce Zhang, and Martin Vechev. Dl2: Training and querying neural networks

with logic. In *International Conference on Machine Learning*, pages 1931–1941, 2019.

[42] Jacques D. Fleuriot. On the Mechanization of Real Analysis in Isabelle/HOL. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 145–161. Springer, 2000.

[43] Martin Fränzle and Christian Herde. Hysat: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30:179–198, 2007.

[44] Rudolf Fritsch, Rudolf Fritsch, G Fritsch, and Gerda Fritsch. *Four-Color Theorem*. Springer, 1998.

[45] Nathan Fulton and André Platzer. Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[46] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. AI^2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE symposium on security and privacy (SP)*, pages 3–18. IEEE, 2018.

[47] Izrail Gelfand. Normierte ringe. *Matematicheskiy sbornik*, 9(1):3–24, 1941.

[48] Michele Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.

[49] Georges Gonthier et al. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[50] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[51] Sebastien Gouezel. Ergodic Theory in Isabelle. *Archive of Formal Proofs*. URL: `https://isa-afp.org/entries/Ergodic_Theory.html`.

[52] Florian Haftmann and Lukas Bulwahn. Code generation from Isabelle/HOL theories. *Isabelle documentation*, 2021. URL: `https://isabelle.in.tum.de/doc/codegen.pdf`.

[53] Florian Haftmann, Johannes Hölzl, and Tobias Nipkow. Code_Real_Approx_By_Float. URL: https://isabelle.in.tum.de/library/HOL/HOL-Library/Code_Real_Approx_By_Float.html.

[54] Benjamin Haibe-Kains, George Alexandru Adam, Ahmed Hosny, Farnoosh Khodakarami, et al. Transparency and reproducibility in artificial intelligence. *Nature*, 586(7829):E14–E16, 2020.

[55] Thomas C Hales. Formal proof. *Notices of the AMS*, 55(11):1370–1380, 2008.

[56] John Harrison. A HOL theory of Euclidean space. volume 3603 of *Lecture Notes in Computer Science*, pages 114–129, Oxford, UK, 2005. Springer-Verlag.

[57] John Harrison. HOL light: An overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer, 2009.

[58] John Harrison et al. *Formalized mathematics*. Turku Centre for Computer Science Turku, 1996.

[59] Hasktorch. URL: http://hasktorch.org/.

[60] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[61] J Hölzl. Stopping_Time. URL: https://isabelle.in.tum.de/dist/library/HOL/HOL-Probability/Stopping_Time.html.

[62] J Hölzl and TU München. Disjoint_Sets. URL: https://isabelle.in.tum.de/dist/library/HOL/HOL-Library/Disjoint_Sets.html.

[63] J Hölzl, TU München, and A Heller. Probability_Measure. URL: https://isabelle.in.tum.de/dist/library/HOL/HOL-Probability/Probability_Measure.html.

[64] Johannes Hölzl. Markov chains and Markov decision processes in Isabelle/HOL. *Journal of Automated Reasoning*, 59(3):345–387, 2017.

[65] Johannes Hölzl and Armin Heller. Three chapters of measure theory in Isabelle/HOL. In *International Conference on Interactive Theorem Proving*, pages 135–151. Springer, 2011.

[66] Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in Isabelle/HOL. In *Interactive Theorem Proving: 4th International Conference*, pages 279–294. Springer, 2013.

[67] Johannes Hölzl, Andreas Lochbihler, and Dmitriy Traytel. A formalized hierarchy of probabilistic system types. In *International Conference on Interactive Theorem Proving*, pages 203–220. Springer, 2015.

[68] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[69] Zhiting Hu, Xuezhe Ma, Zhengzhong Liu, Eduard Hovy, and Eric Xing. Harnessing deep neural networks with logic rules. *arXiv preprint arXiv:1603.06318*, 2016.

[70] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International conference on computer aided verification*, pages 3–29. Springer, 2017.

[71] Craig Innes and Ram Ramamoorthy. Elaborating on Learned Demonstrations with Temporal Logic Specifications. In *Proceedings of Robotics: Science and System XVI*, 2020. doi:10.15607/RSS.2020.XVI.004.

[72] Tommi S. Jaakkola, Michael I. Jordan, and Satinder P. Singh. On the Convergence of Stochastic Iterative Dynamic Programming Algorithms. *Neural Computation*, 6:1185–1201, 1993.

[73] Florian Kammüller, Markus Wenzel, and Lawrence C Paulson. Locales a sectioning concept for Isabelle. In *International Conference on Theorem Proving in Higher Order Logics*, pages 149–165. Springer, 1999.

[74] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[75] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al.

The Marabou framework for verification and analysis of deep neural networks. In *Computer Aided Verification: 31st International Conference*, pages 443–452. Springer, 2019.

[76] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, pages 462–466, 1952.

[77] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations*, 2015.

[78] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[79] Nate Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *IEEE International Conference on Robotics and Automation, 2004*, volume 3, pages 2619–2624. IEEE, 2004.

[80] Panagiotis Kouvaros and Alessio Lomuscio. Formal verification of CNN-based perception systems. *arXiv preprint arXiv:1811.11373*, 2018.

[81] Elena Krasheninnikova, Javier García, Roberto Maestre, and Fernando Fernández. Reinforcement learning for pricing strategy optimization in the insurance industry. *Engineering Applications of Artificial Intelligence*, 80:8 – 19, 2019. doi:https://doi.org/10.1016/j.engappai.2019.01.010.

[82] Erwin Kreyszig. *Introductory functional analysis with applications*, volume 1. John Wiley & Sons, 1978.

[83] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[84] Yen-Ling Kuo, Boris Katz, and Andrei Barbu. Encoding formulas as deep networks: Reinforcement learning for zero-shot execution of LTL formulas. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5604–5610. IEEE, 2020.

[85] Zeshan Kurd and Tim Kelly. Establishing safety criteria for artificial neural networks. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 163–169. Springer Berlin Heidelberg, 2003.

[86] Harold Joseph Kushner and Dean S Clark. Stochastic approximation methods for constrained and unconstrained systems. 1978.

[87] Gitta Kutyniok. The mathematics of artificial intelligence. In *Proceedings of the International Congress of Mathematicians*, 2022.

[88] Panagiotis Kyriakis, Jyotirmoy V. Deshmukh, and Paul Bogdan. Learning from Demonstrations under Stochastic Temporal Logic Constraints. In *2022 American Control Conference (ACC)*, pages 2598–2603, 2022. doi:10.23919/ACC53348.2022.9867861.

[89] Tze Leung Lai. Stochastic Approximation. *The Annals of Statistics*, 31(2):391–406, 2003.

[90] Guillaume Lample and Devendra Singh Chaplot. Playing FPS games with deep reinforcement learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[91] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 54, 2014.

[92] Karen Leung, Nikos Aréchiga, and Marco Pavone. Backpropagation through signal temporal logic specifications: Infusing logical structure into gradient-based methods. *The International Journal of Robotics Research*, page 02783649221082115, 2020.

[93] Tao Li and Vivek Srikumar. Augmenting Neural Networks with First-order Logic. In *Proceedings of the 57th Conference of the Association for Computational Linguistics*, pages 292–302, 2019. doi:10.18653/v1/p19-1028.

[94] Zhipeng Liang, H Chen, J Zhu, K Jiang, and Y Li. Adversarial Deep Reinforcement Learning in Portfolio Management. *arXiv preprint arXiv:1808.09940*, 2018.

[95] Jeffrey T Linderoth, Andrea Lodi, et al. Milp software. *Wiley encyclopedia of operations research and management science*, 5:3239–3248, 2010.

[96] Seymour Lipschutz and Marc Lars Lipson. *Schaum's Outline of Linear Algebra*. McGraw-Hill Education, 2018.

[97] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. On the Reproducibility and Replicability of Deep Learning in Software Engineering. *ACM Trans. Softw. Eng. Methodol.*, 31(1), 2021. doi:10.1145/3477535.

[98] Michel Loève. On almost sure convergence. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, volume 2, pages 279–304. University of California Press, 1951.

[99] Carlin MacKenzie, Jacques Fleuriot, and James Vaughan. An Evaluation of the Archive of Formal Proofs. *arXiv preprint arXiv:2104.01052*, 2021.

[100] Mohammad Abdulaziz Mansour and Maximilian Schäffeler. Formally Verified Solution Methods for Markov Decision Processes. In *37th AAAI Conference on Artificial Intelligence*, 2022.

[101] Giuseppe Marra, Francesco Giannini, Michelangelo Diligenti, and Marco Gori. LYRICS: A General Interface Layer to Integrate Logic Inference and Deep Learning. In *Machine Learning and Knowledge Discovery in Databases*, pages 283–298, Cham, 2020. Springer International Publishing.

[102] Laurent Mazare. Using Python and OCaml in the same Jupyter notebook. *Jane Street Tech Blog*, December 2019. https://blog.janestreet.com/using-python-and-ocaml-in-the-same-jupyter-notebook/.

[103] Laurent Mazare. OCaml-Torch. *GitHub*, 2021. URL: https://github.com/LaurentMazare/ocaml-torch.

[104] Warren Mcculloch and Walter Pitts. A Logical Calculus of Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[105] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. A survey on bias and fairness in machine learning. *ACM Computing Surveys (CSUR)*, 54(6):1–35, 2021.

[106] Robin Milner. Logic for computable functions description of a machine implementation. Technical report, Stanford University, 1972.

[107] Hédi Nabli. An overview on the simplex algorithm. *Applied Mathematics and Computation*, 210(2):479–489, 2009.

[108] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[109] James R Norris. *Markov Chains*. Number 2. Cambridge University Press, 1998.

[110] Albert B Novikoff. On convergence proofs for perceptrons. Technical report, Stanford Research Inst., Menlo Park CA, 1963.

[111] Bo Pang, Erik Nijkamp, and Ying Nian Wu. Deep learning with tensorflow: A review. *Journal of Educational and Behavioral Statistics*, 45(2):227–248, 2020.

[112] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[113] Lawrence C Paulson. Great_Picard. URL: `https://isabelle.in.tum.de/dist/library/HOL/HOL-Complex_Analysis/Great_Picard.html`.

[114] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[115] Lawrence C Paulson. A generic tableau prover and its integration with Isabelle. Technical report, University of Cambridge, Computer Laboratory, 1998.

[116] Lawrence C Paulson. Isabelle's logics, 2013.

[117] Lawrence C Paulson, J Hölzl, TU München, and A Heller. Measure_Space. URL: `https://isabelle.in.tum.de/dist/library/HOL/HOL-Analysis/Measure_Space.html`.

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

[119] Lerrel Pinto and Abhinav Gupta. Supersizing self-supervision: Learning to grasp from 50K tries and 700 robot hours. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3406–3413, 2016. doi:10.1109/ICRA.2016.7487517.

[120] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977. doi:10.1109/SFCS.1977.32.

[121] Marius-Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588, 2009.

[122] Ameya Pore, Davide Corsi, Enrico Marchesini, Diego Dall'Alba, Alicia Casals, Alessandro Farinelli, and Paolo Fiorini. Safe reinforcement learning using formal verification for tissue retraction in autonomous robotic-assisted surgery. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4025–4031. IEEE, 2021.

[123] Luca Pulina and Armando Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *Computer Aided Verification: 22nd International Conference*, pages 243–257. Springer, 2010.

[124] Luca Pulina and Armando Tacchella. Challenging SMT solvers to verify neural networks. *AI Communications*, 25(2):117–135, 2012.

[125] Aniruddh G. Puranic, Jyotirmoy V. Deshmukh, and Stefanos Nikolaidis. Learning From Demonstrations Using Signal Temporal Logic in Stochastic and Continuous Domains. *IEEE Robotics and Automation Letters*, 6(4):6250–6257, 2021. doi:10.1109/LRA.2021.3092676.

[126] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[127] S Richter, M Wenzel, TU München, J Hölzl, and LC Paulson. Sigma_Algebra. URL: https://isabelle.in.tum.de/dist/library/HOL/HOL-Analysis/Sigma_Algebra.html.

[128] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[129] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[130] David E Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, pages 1–34, 1995.

[131] Stuart Russell. Learning agents for uncertain environments. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 101–103, 1998.

[132] Matteo Saveriano, Fares J Abu-Dakka, Aljaz Kramberger, and Luka Peternel. Dynamic movement primitives in robotics: A tutorial survey. *arXiv preprint arXiv:2102.03861*, 2021.

[133] Stefan Schaal, Jan Peters, Jun Nakanishi, and Auke Ijspeert. Learning movement primitives. In *Robotics research. the eleventh international symposium*, pages 561–572. Springer, 2005.

[134] Maximilian Schäffeler and Mohammad Abdulaziz. Markov Decision Processes with Rewards. *Archive of Formal Proofs*, December 2021. URL: `https://isa-afp.org/entries/MDP-Rewards.html`.

[135] Maximilian Schäffeler and Mohammad Abdulaziz. Verified Algorithms for Solving Markov Decision Processes. *Archive of Formal Proofs*, December 2021. URL: `https://isa-afp.org/entries/MDP-Algorithms.html`.

[136] Maximillian Schäffeler. Verified Solution Methods for Markov Decision Processes. Master's thesis, 2021.

[137] Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. Towards Verification of Artificial Neural Networks. In *MBMV*, pages 30–40, 2015.

[138] Olivier Sigaud and Olivier Buffet. *Markov decision processes in artificial intelligence*. John Wiley & Sons, 2013.

[139] David Silver, Alex Graves, Ioannis Antonoglou, Martin Riedmiller, Volodymyr Mnih, D Wierstra, and K Kavukcuoglu. Playing Atari with deep reinforcement learning. *DeepMind Lab. arXiv*, 1312, 2013.

[140] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[141] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[142] Herbert A. Simon. Artificial intelligence: an empirical science. *Artificial Intelligence*, 77(1):95–127, 1995. doi:https://doi.org/10.1016/0004-3702(95)00039-H.

[143] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the single neuron convex barrier for neural network certification. *Advances in Neural Information Processing Systems*, 32, 2019.

[144] William D Smart and L Pack Kaelbling. Effective reinforcement learning for mobile robots. In *Proceedings 2002 IEEE International Conference on Robotics and Automation*, volume 4, pages 3404–3410. IEEE, 2002.

[145] Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. Formal verification of neural network controlled autonomous systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 147–156, 2019.

[146] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3:9–44, 1988.

[147] Richard S. Sutton. *Reinforcement learning : an introduction*. Adaptive computation and machine learning. MIT Press, Cambridge, Massachusetts, 1998.

[148] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[149] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[150] Eline Tetteroo, Wilco Van den Heuvel, and Ayse Sena Eruguz. C2C Return Logistics: a large Markov Decision Process analysis of an innovative concept, 2019.

[151] Edward L Thorndike. The law of effect. *The American journal of psychology*, 39(1/4):212–222, 1927.

[152] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.

[153] Pashootan Vaezipoor, Andrew C Li, Rodrigo A Toro Icarte, and Sheila A. Mcilraith. LTL2Action: Generalizing LTL instructions for multi-task RL. volume 139 of *Proceedings of Machine Learning Research*, pages 10497–10508. PMLR, 18–24 Jul 2021.

[154] Koundinya Vajjha, Avraham Shinnar, Barry Trager, Vasily Pestun, and Nathan Fulton. CertRL: formalizing convergence proofs for value and policy iteration in Coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 18–31, 2021.

[155] Koundinya Vajjha, Barry Trager, Avraham Shinnar, and Vasily Pestun. Formalization of a Stochastic Approximation Theorem. *arXiv preprint arXiv:2202.05959*, 2022.

[156] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. *Advances in neural information processing systems*, 31, 2018.

[157] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992. doi:10.1007/BF00992698.

[158] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[159] Keith Weber and Jennifer Czocher. On mathematicians' disagreements on what constitutes a proof. *Research in Mathematics Education*, 21(3):251–270, 2019. doi:10.1080/14794802.2019.1585936.

[160] Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. Towards fast computation of certified robustness for ReLU networks. In *International Conference on Machine Learning*, pages 5276–5285. PMLR, 2018.

[161] Markus Wenzel. Isar—a generic interpretative approach to readable formal proof documents. In *International Conference on Theorem Proving in Higher Order Logics*, pages 167–183. Springer, 1999.

[162] Markus Wenzel. Using axiomatic type classes in Isabelle. *Tutorial, TU Muenchen*, 2005.

[163] Douglas J White. Real applications of Markov decision processes. *Interfaces*, 15(6):73–83, 1985.

[164] Chelsea C White III and Douglas J White. Markov decision processes. *European Journal of Operational Research*, 39(1):1–16, 1989.

[165] Alfred North Whitehead and Bertrand Russell. *Principia mathematica to* 56*, volume 2. Cambridge University Press, 1997.

[166] Jacob Wolfowitz. On stochastic approximation methods. *The Annals of Mathematical Statistics*, 27(4):1151–1156, 1956.

[167] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Broeck. A Semantic Loss Function for Deep Learning with Symbolic Knowledge. volume 80 of *Proceedings of Machine Learning Research*, pages 5502–5511, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[168] Jonathan Julián Huerta y Munive. Matrices for ODEs. *Archive of Formal Proofs*, 2020.

[169] Artem Yushkovskiy. Comparison of two theorem provers: Isabelle/hol and coq. *arXiv preprint arXiv:1808.09701*, 2018.