

Learning to Play

Reinforcement Learning and Games



Aske Plaat

February 4, 2020

To my students

**This is a preprint. All your suggestions for improvement to
aske.plaat@gmail.com are greatly appreciated!**

© text Aske Plaat, 2019, 2020
Illustrations have been attributed to their authors as much as possible
This edition: 2019

Contents

1	Introduction	13
1.1	Tuesday March 15, 2016	13
1.2	Intended Audience	15
1.3	Outline	16
2	Intelligence and Games	21
2.1	Intelligence	22
2.2	Games of Strategy	31
2.3	Game Playing Programs	44
2.4	Summary	54
3	Reinforcement Learning	57
3.1	Markov Decision Processes	58
3.2	Policy Function and Value Function	60
3.3	Solution Methods	62
3.4	Reinforcement Learning and Supervised Learning	72
3.5	Practice	74
3.6	Summary	80
4	Heuristic Planning	83
4.1	The Search-Eval Architecture	84
4.2	State Space Complexity	93
4.3	Search Enhancements	96
4.4	Evaluation Enhancements	112
4.5	System 1 and System 2	117
4.6	Practice	119
4.7	Summary	122
5	Adaptive Sampling	127
5.1	Monte Carlo Evaluation	128
5.2	Monte Carlo Tree Search	131
5.3	*MCTS Enhancements	140
5.4	Practice	147
5.5	Summary	148

6 Function Approximation	151
6.1 Deep Supervised Learning	152
6.2 *Advanced Network Architectures	173
6.3 Deep Reinforcement Learning	182
6.4 *Advanced Deep Reinforcement Learning	193
6.5 Practice	198
6.6 Summary	204
7 Self-Play	211
7.1 Strong Play: AlphaGo	213
7.2 Self-Play: AlphaGo Zero	221
7.3 General Play: AlphaZero	231
7.4 Self-Play Techniques	234
7.5 *Enhancing Deep Reinforcement Learning	245
7.6 Practice	260
7.7 Summary	263
8 Conclusion	267
8.1 Artificial Intelligence in Games	268
8.2 Future Drosophila	274
8.3 A Computer's Look at Human Intelligence	279
8.4 System 3?	283
8.5 General Intelligence	284
8.6 Practice	287
8.7 Summary	288
I Appendices	291
A Deep Reinforcement Learning Environments	293
A.1 General Learning Environments	293
A.2 Deep Reinforcement Learning Environments	294
A.3 Open Reimplementations of Alpha Zero	295
B Running Python	297
C Tutorial for the Game of Go	299
D Matches of Fan Hui, Lee Sedol and Ke Jie	311
D.1 Fan Hui	311
D.2 Lee Sedol	311
D.3 Ke Jie	311
E *AlphaGo Technical Details	315
E.1 AlphaGo	315
E.2 AlphaGo Zero	317
E.3 AlphaZero	320

<i>CONTENTS</i>	5
References	320
Figures	357
Tables	362
Algorithms	363
Index	365

Preface

Amazing breakthroughs in reinforcement learning in games have taken place. Computers teach themselves to play Chess and Go at world champion level, and they do so faster than humans. There is talk about expanding application areas towards general artificial intelligence. The breakthroughs in computer game playing have been created by combining key technologies in planning and learning. If you want to know how the AI technologies behind AlphaGo, Atari, and Deep Blue work, this book covers them all, in depth.

The breakthroughs in Backgammon, Checkers, Chess, Atari, Go, Poker, and StarCraft have shown that we can build machines that exhibit intelligent game playing of the highest level. These successes have been widely publicized in the media, and inspire AI entrepreneurs and scientists alike. Reinforcement learning in games has become a mainstream AI research topic. It is a broad topic, and the successes build on a range of diverse techniques, from exact planning algorithms, to adaptive sampling, deep function approximation, and ingenious self-play methods.

Perhaps because of the breadth of these technologies, or because of the recency of the breakthroughs, there are few books that explain these methods in depth. This book covers them all in one comprehensive volume, explaining the latest research, bringing you as close as possible to working implementations, with many references to the original literature.

The programming examples in this book are in Python, the language in which most current reinforcement learning research is conducted. We help you get started with machine learning frameworks such as TensorFlow and Keras, and provide exercises to help understand how AI is learning to play.

This is not a typical reinforcement learning text book. Most books on reinforcement learning take the single agent perspective, of path finding and robot planning. We take as inspiration the amazing breakthroughs in game playing, and use two-agent games to explain the full power of deep reinforcement learning.

Games have always been associated with reasoning and intelligence. Our games perspective allows us to make connections with artificial intelligence and general intelligence, giving a philosophical flavor to an otherwise technical field.

Artificial Intelligence

Ever since my early days as a student I have been captivated by artificial intelligence, by machines that behave in seemingly intelligent ways. Initially I had been taught that computers were deterministic machines that can never do something new. Yet in AI, these machines do complicated things such as recognizing patterns, and playing Chess. Actions emerged from these machines, behavior that appeared not to have been programmed into them. The actions seemed new, and even creative, at times.

For my thesis I got to work on game playing programs for combinatorial games such as Chess, Checkers and Othello. The paradox became even more apparent. These game playing programs all followed an elegant architecture, consisting of a search function and an evaluation function.¹ These two simple functions together could find good moves all by themselves. The search-evaluation architecture has been around since the earliest days of computer Chess. Together with minimax it was proposed in a 1952 paper by Alan Turing, mathematician, code-breaking war-hero, and one of the fathers of computer science and artificial intelligence. The search-evaluation architecture is also used in Deep Blue, the Chess program that beat Kasparov in 1997 in New York.

After that historic moment, the attention of the AI community shifted to a new game with which to further develop ideas for intelligent game play. It was the East Asian game of Go that emerged as the new grand test of intelligence. Simple, elegant, and mind-bogglingly complex.

This new game spawned the creation of important new algorithms, and even two paradigm shifts. The first algorithm to upset the world view of games researchers is Monte Carlo Tree Search, in 2006. Since the 1950s generations of game playing researchers, myself included, were brought up with minimax. The essence of minimax is to look-ahead as far as you can, to then choose the best move, and to make sure that all moves are tried (since behind seemingly harmless moves deep attacks may hide that you can only uncover if you try all moves). And now Monte Carlo Tree Search introduced randomness into the search, sampling, deliberately missing moves. Yet it worked in Go, and much better than minimax.

Monte Carlo Tree Search caused a strong increase in playing strength, although not yet enough to beat world champions. For that, we had to wait another ten years.

In 2013 our world view was in for a new shock, because again a new paradigm shook up the conventional wisdom. Neural networks were widely viewed to be too slow and too inaccurate to be useful in games. Many Master's theses of stubborn students had sadly confirmed this to be the case. Yet in 2013 GPU power allowed the use of a simple neural net to learn to play Atari video games just from looking at the video pixels, using a method called deep reinforcement learning. Two years and much hard work later deep reinforcement learning was combined with Monte Carlo Tree Search in the program AlphaGo. The level of play was improved so much that a year later finally world champions in Go were beaten,

¹The search function performs moves to simulate the kind of look-ahead that many human game players do in their head, and the evaluation function assigns a numeric score to a board position.

many years before experts had expected that this would happen. And in other games, such as StarCraft and Poker, self-play reinforcement learning also worked well.

The AlphaGo wins were widely publicized. They have had a large impact, on science, on the public perception of AI, and on society. AI researchers everywhere were invited to give lectures. Audiences wanted to know what had happened, whether computers finally had become intelligent, what more could be expected from AI, and what all this would mean for the future of the human race. Many start-ups were created, and existing technology companies started researching what AI could do for them.

The modern history of computer games spans some 70 years. We have seen amazing achievements. Games research has witnessed multiple paradigm shifts, going from heuristic planning, to adaptive sampling, to deep learning, to self-play. The achievements are large, and so is the range of techniques that are used. We are now at a point where the techniques have matured somewhat, and achievements can be documented and put into perspective.

In explaining the technologies, I will tell the story how one kind of intelligence works, the intelligence needed to play two-person games of tactics and strategy. (As to knowing the future of the human race, surely more is needed than an understanding of heuristics, deep reinforcement learning and game playing programs.) It will be a story involving many scientists, programmers, and game enthusiasts, all fascinated by the same goal: creating artificial intelligence. I am sure you will find the ride as fascinating as we did.

Leiden, 2019

Acknowledgments

This book would not have been possible without the help of many friends. First of all, I want to thank everybody at LIACS, for creating such a wonderful and vibrant CS & AI department to work in. So many people make this place such a joy to do research, I love you all!

Many people have read and commented on drafts of this book. Thank you Vedran Dunjko, Mike Huisman, Walter Kosters, Wojtek Kowalczyk, Alfons Laarman, Thomas Moerland, and Mike Preuß for your very sharp eyes and criticism. Thank you to Hui Wang, Zhao Yang, Michael Emmerich, Mike Preuß, Thomas Moerland, Xuhan Lu, Joost Broekens, Matthias Müller-Brockhausen, my friends and fellow conspirators in the reinforcement learning group. In addition, thank you to Lise Stork and Wojtek Kowalczyk for discussions on deep learning and zero-shot learning. Thanks to Michael Emmerich for curriculum learning. Thank you to Walter Kosters, Frank Takes, Jan van Rijn, and Jonathan Vis for discussions on teaching, MCTS and combinatorial games. Thanks to Mike Preuß for countless discussions and for running the games research at Leiden. Thanks to Max van Duijn for inspiring discussions on theory of mind and creativity. Thank you Walter Kosters for a shared love of combinatorics and a sense of humor. A very special thank you to Thomas Bäck, with whom I have had so many discussions on science, the universe, and everything. Without you, this book would not have been possible.

I thank all students of the reinforcement learning course in Leiden, past, present and future, for their enthusiasm, feedback and suggestions.

I thank all my colleagues, teachers, fellow researchers and friends for giving so much and for selflessly sharing our love of science.

Finally, I thank Saskia, Isabel, Rosalin, and little Lily, for being who they are, for knowing no limits, for enduring me, and for their boundless love.

Aske

Chapter 1

Introduction

1.1 Tuesday March 15, 2016

Tuesday March 15, 2016 is an important day in the history of artificial intelligence, and not just in the history of artificial intelligence, but in the history of intelligence, period. That day was the final day of a five game match in which a computer challenged one of the strongest human Go players. Go is an ancient game of strategy, played by millions around the world, originating in Southeast Asia. Playing the game well requires years of training and a considerable amount of intelligence. Dedicated Go-schools exist in China, Korea, and Japan, and professional Go players achieve star-status in their countries and can live well from their prize money. Go is considered to be a hallmark of human intelligence and creativity, and the game was thought to be safe from machine intelligence for at least another decade.

Yet on March 15, at the luxurious Four Seasons hotel in Seoul, the computer program AlphaGo beat top Go-professional Lee Sedol (see Figure 1.1). AlphaGo was developed by a team of scientists of the AI company DeepMind. The prize money, 1 million dollars, was donated by DeepMind to charities. Lee Sedol received 170,000 dollars for his efforts. The match consisted of 5 games, the human champion was beaten 4–1. A subsequent match in 2017 against champion Ke Jie was also won by the computer. Commentators described AlphaGo's level of play as very strong, and the style of play as refreshing, with a few highly surprising moves, that some described as beautiful and creative. AlphaGo was awarded a 9 dan professional rank, the highest rank possible, by the Chinese Weiqi association, and the journal *Science* chose the event as one of the *breakthrough of the year* runners up. Today, human Go players are taking inspiration from the games played by AlphaGo, trying to learn from its deep insights and unusual moves [751].

The breakthrough performance of AlphaGo came as a surprise to the world and to the research community. The level of play of computer Go programs had been stagnant around the strong amateur level for years. Experts in computer games had expected grand master level play to be at least ten years away, let



Figure 1.1: AlphaGo versus Lee Sedol

alone beating the world's strongest players.

Beating the human Go champions has shaken the world of AI, indeed it has shaken the world. The AlphaGo work is truly sensational. The techniques used in AlphaGo are the result of many years of research, and cover a surprisingly wide range of topics. It is a culmination of decades of research by many researchers. A main motivation for writing this book is to provide a comprehensive treatment of the technologies that have led to the creation of AlphaGo.

This brings us to the problem statement of this book that describes the core of what it is about.

Problem Statement

The main question that we are concerned with in this book is the following:

What are the machine learning methods that are used in Chess and Go to achieve a level of play stronger than the strongest humans?

Among the technologies used are heuristic planning methods, adaptive sampling, deep reinforcement learning, and self-play. We provide an in depth introduction to all these methods. We will explain how they work with example code. We use the language of reinforcement learning as the common language to describe all technologies in this book, both for planning and learning. We describe the algorithms and architectures. The emphasis is on a hands-on style, providing

the intuition behind the algorithms to apply them in practice. References to the literature are provided to find more details and more on the relevant mathematics behind the methods.

1.2 Intended Audience

Now that we know what this book is about, we can see who this book is for.

This book is intended for you if you are excited to find out how computers play games, and what the techniques are that are used to achieve such amazing displays of intelligent behavior. We aim to be comprehensive, to provide all necessary details. Basic introductions to Markov Decision Processes, reinforcement learning and deep learning are provided. We also aim to make these details more accessible than the scientific papers on which the content is based. The level of difficulty of this book is targeted at a graduate level university course in AI. We explain the algorithms, and we provide opportunity for practice.

We take a hands on approach to artificial intelligence. After each new method is described, we will show how the concepts can be implemented in examples, exercises and small programs. Games-AI is an open field. Researchers have released their algorithms and codes allowing replication of results. This is a sharing-oriented research field. We point you to code bases at GitHub throughout the book. Even when you do not have access to Google-scale compute power, at the end of this book, you should be able to create a fully functioning self-learning program, and you can join the active community of reinforcement learning researchers. For that purpose, Appendix A contains pointers to software environments and open source code frameworks for machine learning, deep reinforcement learning, and games.

As you may notice, there is great excitement in the field of reinforcement learning, excitement that also prompted the writing of this book. The algorithms and the pointers to codes also serve this purpose: to make it easy for you and to stimulate you to join the excitement of the people already active in the field. For this purpose I have also included photos of some of the researchers, as an invitation to become part of the reinforcement learning community.

Graduate Course

This book is designed for a graduate course in Reinforcement Learning with Games. The material is organized in chapters that roughly correspond to the lectures of a single semester course. The availability of source code provides for practical assignments where you can get your hands dirty and immerse yourself in this fascinating topic. Pieces of working code can be extended to your liking for further study and research.

Each chapter starts with an introduction in which the core problems and core solution concepts of the chapter are summarized. Chapters end with questions to refresh your memory of the material that was covered, and with larger programming exercises. Chapters are concluded with a summary and with more pointers

Chapter	Algorithm	Select	Backup	Eval
Ch 4	alpha-beta	left-to-right	minimax	heuristic
Ch 5	MCTS	adaptive	average	sample
Ch 6	DQN	-	-	generalize
Ch 7	Self-Play	adaptive	average	generalize

Table 1.1: Heuristic-Sample-Generalize

to the literature.

Figure 1.3 shows the structure of the book. There is a logical progression in the chapters, they build on each other. It should be able to cover all chapters in a one-semester course, possibly skipping the starred sections. Normally the chapters would be taught in sequence. Chapters 2 and 8 are less technical, and provide some reflection on intelligence and artificial intelligence. They can be combined towards the end, putting the technical material first.

All links to web pages were correct at the time of writing and have been stable for some time. There is a companion website for this book <https://learningtoplay.net> that contains free updates, slides, and other course material that you are welcome to explore and use in your course.

Prerequisite Knowledge

In order to build up an understanding and to experiment with the algorithms a general computer science background at undergraduate level is necessary. Some proficiency in programming is needed, preferably in Python. Also some background in undergraduate AI is beneficial, although we strive to be comprehensive and self contained.

1.3 Outline

We have almost come to the end of the introduction. Let us have a closer look at what we can expect in the rest of this book.

This book covers the technologies behind the major AI breakthroughs in reinforcement learning in games. Chess, Atari, and Go required surprisingly diverse techniques. The methods in this book go from heuristic planning to self-play. First we describe methods that need domain specific heuristics, and we end with a method that is able to generalize features in many domains, and to teach itself to do so. We will describe in detail heuristic methods, sampling, and generalization. Together the chapters tell a story how AI progressed from Artificial Specific (or Narrow) Intelligence towards more General Intelligence. See Table 1.1.

Figure 1.2 gives a related overview of the topics of this book, in terms of mappings between problems and methods that go from specific to general. The first chapters are specific in the sense that the methods make direct use of the problem

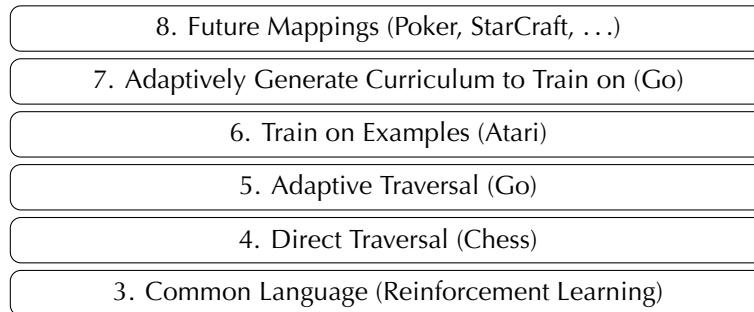


Figure 1.2: Stack of Chapters, with increasingly advanced Mappings from Problems to Methods (from bottom to top)

structure to directly traverse board positions. The later chapters are more abstract, and the mapping between method and problem becomes less direct. Methods are adaptive, and learn features from examples of the problem domain, instead of directly traversing the problem itself. Self-play methods are described, that teach themselves which examples to learn from.

The story of games research is the story of AI researchers that were learning how to make algorithms that can learn to learn.

*Starred Sections

The chapters are organized from planning to learning to self play. Figure 1.3 gives an overview of the chapters. The names of the four core technical chapters are printed in bold.

Sections and exercises whose name starts with a *star are advanced material, which is essential for high performing implementations, but may be skipped when in a hurry, without missing the main thread.

We will now describe the main topics of this book. Please refer to Table 1.1 and Figure 1.2, since the concepts heuristic-sample-generalize and (in)direct mappings between methods and problem spaces are important threads.

The Chapters

Let us now discuss the chapters in more detail. This book consists of four core technical chapters that look in detail at the methods that have been developed by researchers (heuristic search, adaptive sampling, function approximation, and self-play). They are preceded by a chapter on the reinforcement learning paradigm, as a common language for the four following chapters.

We will start in Chapter 2 with early thoughts on intelligence and intelligent machines. We then provide a brief historic overview of some of the important game games and game playing programs, starting with the early designs of Claude Shannon, Christopher Strachey and Alan Turing.

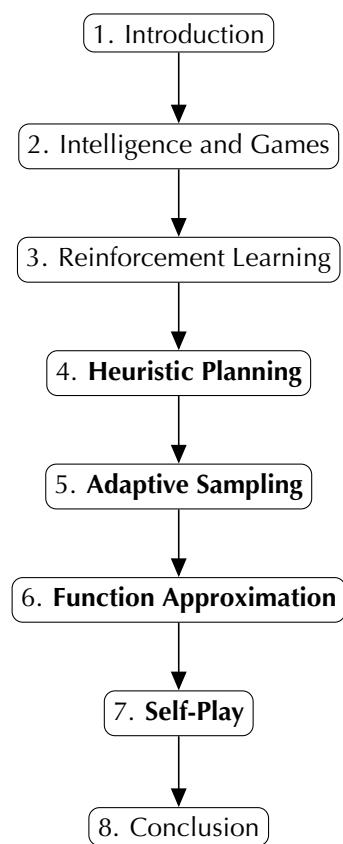


Figure 1.3: Overview of the Chapters of this Book

In the next chapter, Chapter 3, we introduce reinforcement learning. The concepts and formalizations of reinforcement learning are at the basis of the historic advances in artificial intelligence that we have recently witnessed. Concepts such as value function, policy function and Q-learning are discussed in this chapter.

In chapter 4, we introduce heuristic planning. Heuristic planning in games uses domain specific knowledge to reduce the size of the state space, and then uses minimax planning to traverse that reduced state space. In this chapter we discuss the principles of the search methods that are at the basis of AI breakthroughs in Checkers and Chess. The planning methods of heuristic planning are simple and rigid, and it turns out that many enhancements are needed to achieve good performance in real game playing programs. The following sections introduce enhancements that overcome some of the rigidity of the minimax approach. At the end of the chapter variable-depth and variable-width enhancements are therefore introduced.

In the next chapter, Chapter 5, we delve deeper into the principles of learning by trial and error. Major topics in this chapter are the exploration/exploitation trade-off and the concept of evaluation by random play-out, leading to the Monte Carlo Tree Search algorithm (MCTS). The algorithm in this chapter is variable-depth and variable-width by design, and great performance improvements, especially in 9×9 Go, are achieved with MCTS. The chapter ends with a discussion of enhancements of MCTS.

Then, we switch paradigms from planning to learning. Instead of *traversing* the states directly and finding good states, we now try to *generalize* features from the states. We start in Chapter 6 with a short review of deep learning methods for supervised learning. Among many other things, convolutional networks are discussed, and the problem of overfitting. In 2012 deep learning methods caused a breakthrough in image recognition performance. Much of the current interest in artificial intelligence is caused by this breakthrough for which Bengio, Hinton and LeCun received the 2018 Turing award, also known as the Nobel prize of computer science. Section 6.3 looks at how these methods can be used in a reinforcement learning setting: generalization of features in deep neural networks for learning by trial and error, of which Deep Q Networks (DQN) is the best known algorithm. Q-learning with function approximation and bootstrapping is in principle unstable, and important enhancements had to be developed to achieve stable training. With these enhancements, we find that deep reinforcement learning can be an effective way to improve on heuristic evaluation functions.

Chapter 7 combines planning and training to create self-learning systems. This is a long chapter in which AlphaGo is explained, and its variants. AlphaGo Zero teaches itself to play Go from scratch. The kind of learning resembles human learning, creating much interest in the research community, and many self-play and self-learning variants and other related research are discussed.

Finally, we conclude in Chapter 8 by reflecting on artificial intelligence progress, and on the relation to natural intelligence. We discuss possible future research directions.

The appendices contain an overview of open deep learning environments that are suitable for experimentation, a brief tutorial for the game of Go, details on the

AlphaGo implementations that were too detailed for the main text, and some pointers to the Python programming language.

After this overview, let us begin our journey.

Chapter 2

Intelligence and Games

As long as we have been around, humans have had dreams about artificial forms of intelligence, wondering if it would be possible to make intelligent machines. It is wonderful to live in a time where we are able see the first realizations of these dreams appearing.

This chapter introduces the field of study of this book, artificial intelligence and games.

We will briefly touch upon the nature of intelligence, human and artificial. We will see that most AI researchers take a pragmatic, behavioristic approach to intelligence. Board games such as Chess and Checkers have been used by the earliest computer science researchers to study reasoning processes. We will list the defining features of these games.

To give a broader understanding of the history of the field we will then review some of the better known game playing programs, both old and new.

Core Problem

- What is intelligence?
- Why are games used in artificial intelligence?

Core Concepts

- Intelligence: Embodied/Computational; AI: Symbolic/Numerical
- Early work: Shannon, Turing, Samuel
- Modern work: TD-Gammon, Chinook, Deep Blue, AlphaGo, Libratus

2.1 Intelligence

What is intelligence? Psychology provides various definitions of intelligence. Most definitions include recognition, reasoning, memory, learning, problem solving, adapting to your environment, and creativity. These elements are related to cognition.

In psychology most definitions implicitly assume that intelligence is human, and therefore embodied. Related to intelligence are intuition, emotion, self-awareness and volition, or *free will*. Humans have all of these capacities. However, artificial intelligence typically studies a limited list of capacities. Essential elements currently studied in AI are related to cognition, such as recognition, reasoning, memory and learning [554]. Intelligent decision making is at the core of current AI research.

An important difference between human intelligence and artificial intelligence is that humans are intelligent beings. We are, we have an identity, and we are self-aware. Artificial intelligence research mostly studies intelligent behavior. Machines that exhibit intelligent behavior are generally not considered to be intelligent, just to *behave* intelligent, even programs that jokingly suggest to have an identity by printing texts such as *Hello, World*. As Shakespeare wrote: “To be or not to be, that is the question.”

We will now discuss cognition and intelligence in more depth.

2.1.1 Schools of Thought on Intelligence

Psychology teaches us that intelligence includes cognitive abilities, the abilities to learn, form concepts, and reason. These abilities allow us to recognize patterns, comprehend ideas, solve problems, plan, and use language to communicate. Cognitive abilities enable intelligent behavior. They also enable other capacities, such as conscious thought.

Embodied and Computationalist Intelligence

Experience and conscious thought are elements of embodied intelligence. The theory of embodied intelligence stresses our ability to *perceive* information, to use it as knowledge in adaptive behaviors within an environment, in an embodied sense [222].

A rival theory is the theory of computationalism. Computationalism states that thoughts are a form of computation of a set of laws for the relations among representations. Computationalism regards the brain as a big computer [433]. Clearly, artificial intelligence, as part of computer science, studies intelligence through computations, and thus builds on computationalism.

Interestingly, the field of reinforcement learning has elements of both computationalism and embodiment. Reinforcement learning is all about learning in response to stimuli from an environment. It studies how agents act in an environment and learn from responses to alter their behavior, as in the embodied view of

intelligence. Reinforcement learning is a field both in psychology and in artificial intelligence. Chapter 3 provides an introduction to reinforcement learning.

Thinking, Fast and Slow

The philosophical implications of the AlphaGo achievement are large. The realization that we are now able to create a machine that can learn to play such a tremendously challenging and rich game of strategy, beating the smartest professionals, is baffling. Indeed, for many researchers the goal of artificial intelligence research is to understand intelligence, and here, we have made a tremendous step. It now certainly is the case, at least for the narrow domain of combinatorial board games, that we have succeeded in achieving learning and reasoning behavior beyond the human level, although it is not known how similar the artificial and human thought processes are. In fact, it has often been argued that humans and computers think in very different ways.

Daniel Kahneman is a psychologist and winner of the Nobel prize in Economics. In his works he combines psychology and economics. He has helped found the field of behavioral economics. In his book *Thinking Fast and Slow* [336] he presents topics of his most famous papers. Central in Kahneman's book are System 1 thinking, or fast, approximate, intuitive human thinking, and System 2 thinking, or slow, exact, reasoned human thinking. These two concepts fit well with our AI methods: learning/connectionist is related to System 1 thinking, and planning/symbolic is related to System 2 thinking.

It is interesting to note that Kahneman's System 1 and 2 suggests at least a crude similarity between artificial and human thought processes. At a few places we will highlight links between the methods described here and the concepts in Kahneman's book. Kahneman's book describes fast and slow thinking in human intelligence. This book does the same for artificial intelligence (see Chapter 8). Let us know have a closer look at the main topic of this book, artificial intelligence.

2.1.2 Artificial Intelligence

Artificial intelligence is old. There are indications that the first ideas on mechanical thought are as old as human thought, and are related to materialistic world views. The first recorded dreams of intelligent machines which we now call robots go back to Indian philosophies 1500 BC of Charvaka [433, 737]. In Greek the word automaton means *acting on one's own will*. In Greek mythology, Homer's Iliad describes automatons, self-operating machines, that open doors automatically. There are many more examples of automata in Greek mythology, such as an artificial person of bronze, or watchdogs of gold and silver. In Chinese texts mechanical engineers, or artificers, are described, as well as artificial wooden flying birds [472, 361].

By attempting to describe human thought as the mechanical manipulation of symbols, classical philosophers were the first to describe ideas of modern AI. This work on mathematical reasoning culminated in the invention of the digital computer in the 1940s. The advent of the first computers and the ideas behind

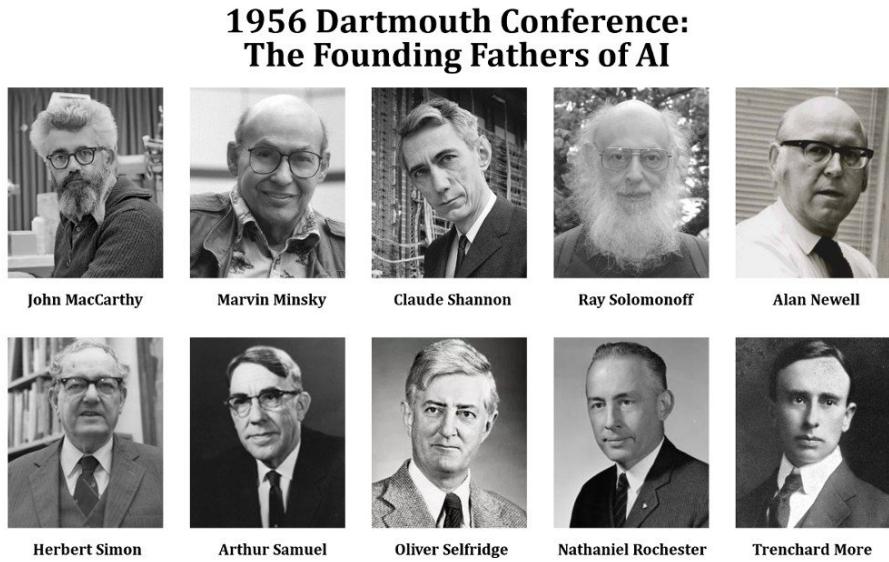


Figure 2.1: Dartmouth AI Workshop 1956

it prompted scientists to start serious discussions on the possibility of creating an electronic brain.

The modern field of AI research was subsequently founded in the summer of 1956 at a workshop at Dartmouth College, New Hampshire (see Figure 2.1). Among those present were John McCarthy, Marvin Minsky, Nathaniel Rochester, Claude Shannon, Allen Newell, Herbert Simon, Arthur Samuel, John Nash, and Warren McCulloch, some are shown in the pictures. These would all become leaders of AI research in the following decades.¹ There was much optimism then, and many predicted that human level machine intelligence would be achieved within 20 years (this was in 1956) [460, 433]. Eventually, however, it became clear that they had underestimated the difficulty of this project. An AI winter followed in which funding for AI research all but dried up (see Figure 2.2 [408]). Despite the funding drought, progress was made, and another period of inflated expectations followed, followed by another winter. Achievements in the 1990s heralded the current AI summer that we are still enjoying. After 2000 interest in AI has been high again, because of machine learning successes in search, vision, speech, and robotics. In fact, many other unexpected inventions did happen in industry, such as in search engines (Google and Baidu), in social networks (Facebook and Tencent), in recommender systems (Amazon, Alibaba and Netflix) and in multi-touchscreen communicators (Apple, Samsung, and Huawei).

In 2018 the highest recognition in computer science, the Turing award, was awarded to three key researchers in deep learning: Bengio, Hinton and LeCun

¹Some would become known for other fields, such as information theory and economics.

AI HAS A LONG HISTORY OF BEING “THE NEXT BIG THING”...

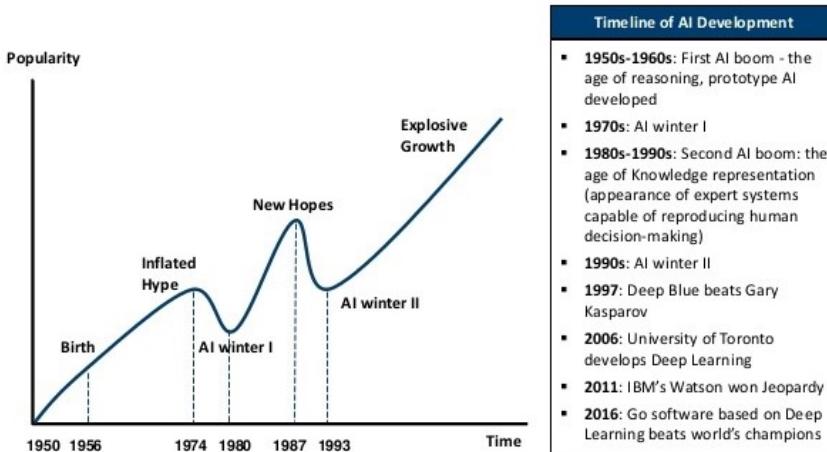


Figure 2.2: AI Winters [408]

(see Figure 2.3).

2.1.3 Turing Test and General Intelligence

In contrast to some parts of psychology, AI is traditionally less interested in defining the essence of intelligence (“being intelligent”) as it is in intelligent behavior, with a strong focus on creating intelligent systems. The AI approach is a pragmatic approach.

AI researchers are interested in machines that exhibit reasoning, knowledge, planning, learning, communication, perception, and the ability to interact with the physical environment. The focus is on behavior; by which methods the machines exhibit this behavior is less relevant. Consciousness and self-awareness are not (yet) on the mainstream AI-research agenda, and simulating the human brain is not part of typical AI research.

Note that the strict separation between behavior and being is becoming somewhat less strict. The focus on behavior of AI fits the school of symbolic AI especially well, where intelligence is based on human logic, and not on human hardware. The other school, connectionism, is emphatically inspired by biology and neurology, i.e., a simulation of human hardware to achieve intelligent behavior.

Behavior: Turing Test

In 1950 Alan Turing proposed a test on how to determine if a machine exhibits intelligence. The machine passes the test when its behavior is indistinguishable



Figure 2.3: Geoffrey Hinton, Yann LeCun and Yoshua Bengio

from that of a human. In the Turing test a human evaluator engages in a natural language conversation with a human and with a machine (see Figure 2.4). The evaluator is aware that one of the two participants is a machine, and both participants are separated from one another by a screen. Participants are restricted to text-only, such as through keyboard and screen (speech technology is deemed too hard). If the evaluator cannot tell which conversation is from the machine, the machine is considered to have passed the test. Note that the machine may make mistakes as long as they are mistakes that humans would also make. The Turing test is an imitation game. It might even be the case that making deliberate mistakes is a good idea, for example, when the domain would be a game of Chess, then a Chess machine could be identified by it being much stronger than a human would be.

The paper that introduced the Turing test was titled: “Computing Machinery and Intelligence” [678]. It opens with the words: “I propose to consider the question, ‘Can machines think?’ ” and is a classic in AI.

The Turing test has been highly influential, and also widely criticised.

One of the first criticisms was exemplified by the Eliza experiment. Eliza was a computer program developed by Joseph Weizenbaum in 1966 [727]. The program had a list of keywords for which it had responses in which it transformed the user’s input in a certain way. If no keyword was used, a generic “witty” reply was given, sometimes re-using part the user’s input. Eliza was developed to appear like a psychotherapist, allowing the program to have little knowledge of the real world. The program was able to fool some people into believing that they were talking to a real person. However, its simple rule-based structure, and its use of

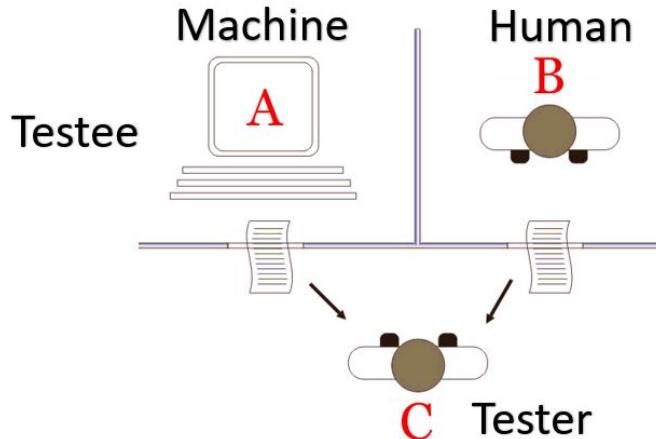


Figure 2.4: Turing Test

superficial tricks, only succeeded to suggest that intelligent behavior can be created with rather unintelligent means.² Eliza's success showed limitations of the Turing test as a mechanism for identifying true artificial intelligence.³

Another noteworthy approach to Turing's question became known as the Loebner Prize. The Loebner Prize provided an annual platform for practical Turing tests. It was named after Hugh Loebner, who provided the prize money.

The first Loebner Prize competition in 1991 was again won by a mindless program that misled interrogators into making the wrong identification by such means as imitating human typing errors. It generated much controversy in which the term *artificial stupidity* was used [601].

A problem of the Turing test is that it does not directly test if the computer behaves intelligently, but rather if humans think it behaves like another human being. Turing tests have since taught us that human behavior and intelligent behavior are not the same thing. The Turing test can fail to identify intelligent behavior correctly because some human behavior is unintelligent, and because some intelligent behavior is inhuman (such as performing complicated mathematical sums quickly and correctly). The Turing test requires deception on the part of the computer; it does not require highly intelligent behavior, in fact, it discourages it, to prevent the computer from appearing to be inhumanly smart.

For these reasons, other tests of intelligence have been devised, such as tests for the ability to play games.

Before we have a look at those games, we will look at what kinds of artificial

²Some would go so far to say that some unintelligent humans behave in a more human way than intelligent humans, confusing the discussion even more.

³The genealogy of Eliza and a list of programs that implement it can be found here <http://elizagen.org/index.html>.

intelligence have been created.

Specialized and General Intelligence

Humans are good at many different things. Our intelligence is a general kind of intelligence, it allows us to cook, read, argue, do mathematics and enjoy a movie. Studying artificial ways of being intelligent in all these domains is a daunting task. Machine intelligence researchers typically limit themselves to a single domain, and artificial intelligence is almost always narrow or special intelligence. Ever since the start of AI, games have played an important role. The rules of a game limit its scope, making it better suited as a first object of study than the full complexity of reality.

Artificial *general* intelligence (AGI) would be the intelligence of a machine that can perform any intellectual task that a human can. AGI is the holy grail of AI research, and a favorite topic in science fiction and future studies. AGI is sometimes called *strong AI* although that term is used by some other researchers to mean machines that experience consciousness [595]. As of 2017, one study found that over forty organizations worldwide are doing active research on AGI [40].

It should be noted, of course, that most humans also have special intelligence abilities. Some humans are better at playing Chess and weaker at Tennis, and very few humans are equally good in all kinds of intelligence (if such a thing could even be measured). In terms of definitions, AGI is even less well defined than AI.

Drosophila Melanogaster

In 1965, the Soviet mathematician Aleksandr Kronrod called Chess the Drosophila Melanogaster of artificial intelligence [387]. Drosophila Melanogaster is a type of fruit fly that is used by developmental biologists in genetics experiments as model organism, because of its fast reproduction cycle that allows quick experiments. Also, by using Drosophila, biologists hope to learn about biological processes in general, beyond the particular species. Similarly, by using Chess, AI researchers hope to learn about processes of intelligence in general, beyond the particular game.

Such an experimenter's tool with quick turnaround times is useful in AI research. Chess was a convenient domain that was well suited for such experimentation. It is clearly a game for which humans need intelligence, requiring many years of dedication and study to achieve proficiency.

In addition to having a limited scope, a second advantage of games is the clarity of the rules. Possible actions are well defined by the rules and the end of a game is also well-defined. In this, games differ from real world activities, such preparing a meal, performing a dance, or falling in love, where goals and rules are often implicit and less well-defined. Combinatorial games are thought to abstract from non-essential elements. The games allow researchers to focus on reasoning processes, processes that are assumed to capture the essence of



Figure 2.5: Arpad Elo

intelligence. Thus, games are easier to perform than other real world tasks, a clear advantage for researchers.

Clear Performance

Games also allow for a clear and undisputed way to measure progress. Game rules define win and loss, and the intelligence of programs can be measured by letting them play against each other or against humans. Many games have rating systems, such as in tennis, where the best of the world are ranked based on how many wins have been scored against other players. An often used measure is the Elo-rating, a system devised by the Hungarian-American physicist Arpad Elo (see Figure 2.5). The basis of the Elo rating is pairwise comparison [189]. In Chess, a beginner is around 800, a mid-level player is around 1600, and a professional, around 2400.

Anthropomorphization

In the original Turing test *any* question can be asked of the machine. It is a test for General Intelligence, the kind of intelligence that humans have. As with the Drosophila, limiting the scope of our domain to a single game clearly cannot be used for the general Turing test, all we can hope for is to achieve special intelligence, the kind of intelligence needed to win a certain game. Still, achievements in specific intelligence, such as by Deep Blue, Watson and AlphaGo, attract large publicity. This may be in part because the human mind generalizes easily, and anthropomorphizes⁴ easily. When we see a device do something in a particular domain for which a human must be incredibly smart, we automatically assume that the device must *be* smart, and will do as well in another domain *because that is what a human would do*. It is easy for us to forget that the machine is not

⁴treat as if human.

a human, something that writers of science fiction books and Hollywood movies eagerly exploit. Current machine intelligence is almost always highly specialized, which is quite unhuman-like.

In Chapter 8 the issues of artificial and human intelligence will be revisited, including the topics of learning, and specialized and general intelligence. By then, we will have acquired a deep understanding of how to create artificial intelligence in games.

We will now have a look at two different approaches to artificial intelligence, a mathematical and a biological approach.

2.1.4 Mathematics and Biology

Artificial intelligence is the study of machines that exhibit behavior for which humans need intelligence. Main topics in the study of AI are cognitive functions such as learning and problem solving [554].

Artificial intelligence research is based on two fields of science, mathematical logic and biology [480]. One of the early ways of modeling thought processes is the top down deductive approach that has come to be known as *symbolic AI*. Symbolic AI is inspired by mathematical logic. In symbolic AI logic *reasoning* is used to draw inferences and conclusions from underlying facts and assumptions about the world. In this view, intelligence is considered to be equivalent to a top down reasoning process. This school has yielded progress in expert systems, reasoning systems, and theorem proving systems. Well-known outcomes of this line of research are the STRIPS planner [209], the Mathematica and MATLAB computer algebra systems [110], the programming language PROLOG [145], and also semantic (web) reasoning [71, 16]. The material covered in the chapter on heuristic planning has grown out of the symbolic approach.

A second school of thought is the *connectionist* approach to AI. It purports that intelligence emerges bottom up, out of interaction processes between many small elements. Connectionism is inspired by biology. Embodied intelligence (robotics) [101], nature inspired algorithms such as Ant Colony Optimization [180], swarm intelligence [347, 84], genetic algorithms and evolutionary strategies [216, 295, 26], and, last but not least, neural networks [272, 393] are examples of the successes of this approach. This approach is also called the numerical, data driven, approach, because of the many numerical parameters of artificial neural networks. The material covered in the chapter on training has grown out of the connectionist approach.

2.1.5 Machine Learning

Machine learning is the field of research that studies algorithms that learn relations that were not explicitly pre-programmed, such as training a classification algorithm with a training set to classify images. Machine learning encompasses both symbolic and connectionist methods. Machine learning algorithms find patterns, categorize data, or make decisions. Machine Learning consists of three

groups of learning problems: supervised learning, reinforcement learning and unsupervised learning [80].

In supervised learning a (typically large) database of labeled examples exists, from which the type of relation between example and label is inferred. The labels have to be provided beforehand, by humans, or by other means. Chapter 6 covers supervised learning in detail.

In reinforcement learning there is no database of labeled examples, interaction is the key. In reinforcement learning an agent can interact with its environment, which returns a new state and a reward-value after each action. The reward-value can be positive or negative. By probing the environment, the reinforcement learning agent can learn relationships that hold in the environment. Reinforcement learning problems are often modeled as Markov Decision Processes [638], see Chapter 3. Most games can be modeled as reinforcement learning problems. In addition to Chapter 3, Chapter 6 also covers reinforcement learning in detail.

Unsupervised learning algorithms find patterns in data without the need for human labeling. Typical unsupervised algorithms cluster data based on inherent properties such as distance.

Reinforcement Learning in Games

In this book we study reinforcement learning algorithms in games. Reinforcement learning problems are interesting in that some problems are best solved by symbolic methods, while for others connectionist methods work best. In the chapter on self-play, we see how the two methods in AI join to achieve great success.

Thus, the work in this book draws on both schools of thought. Planning is closely related to symbolic AI, and learning to connectionism. Reinforcement learning is related to both computationalism and embodied intelligence. The breakthroughs of Chess are related to symbolic AI. Breakthroughs in Atari are related to connectionism. Breakthroughs in AlphaGo (and other self-play programs) in Chapter 7 rely on both schools of thought, and their success is a success of an integrated approach to AI, expressed in a reinforcement learning framework. For other combined approaches, see [683, 228].

After having looked at artificial intelligence in some depth, it is time to look at games.

2.2 Games of Strategy

Games come in many shapes and sizes. Some are easy, some are hard. Before we will look at game playing programs, we will first describe the combinatorial games that are used actively to study artificial intelligence. In the literature on games the characteristics of games are described in a fairly standard taxonomy.

2.2.1 Characteristics of Games

Games are used in artificial intelligence because they provide a challenging environment to play them well. The complexity of games is determined by a number of characteristics. Important characteristics of games are: the number of players, whether the game is zero-sum or non-zero-sum, whether it is perfect or imperfect information, turn based or simultaneous action games, the decision complexity, and the state space complexity. We will discuss these characteristics in more detail.

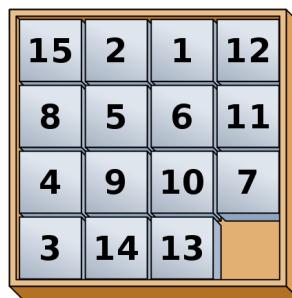


Figure 2.6: 15 Puzzle

Number of Players

One of the most important elements of a game is the number of players. One-player games are usually called puzzles. The goal of a puzzle is to find a solution while following the rules of the puzzle. Well-known examples are mazes, Sudoku, mathematical problems such as the Traveling Salesperson Problem, and the 15-puzzle (see Figure 2.6). Typical optimization problems are to find the shortest (lowest cost) solution of a problem instance.

Two-player games are “real” games. Most of the games in this book are two-player games; these have been studied the most in AI. The game-play in two-player games is often competitive, zero-sum. Keen reasoning and challenging calculation often play a large role in these games, providing satisfaction to the players. Quite a number of two-player games strike a balance between being too challenging for computer programmers, and not being too daunting. Many multi-player games are highly challenging for AI, especially in the early days of AI. Examples of two-player games that are popular in AI are Chess, Checkers, Go, Othello, and Shogi. These games will all be described in a little more detail shortly.

Multi-player games are played by three or more players. Psychology, collaboration, and hidden information (and bluffing) often play a large role in these games. Much of the fun of playing these games may be related to social aspects. The game play can be competitive and collaborative, since players may collude in

different phases to obtain their goals. Many card-games are multi-player games. Well-known examples of multi-player games are card games Bridge and Poker, and strategy games such as StarCraft and Defence of the Ancients (often abbreviated as DOTA).

Zero-Sum vs Non-Zero-Sum

An important aspect of a game is whether it is competitive or collaborative. Many two-player games are competitive: the win (+1) of player A is the loss (-1) of player B. These games are called *zero-sum* because the sum of the wins for the players remains a constant zero. Competition is an important element of the real world, and these games provide a useful model for the study of strategy and conflict, in practice, and especially in theory. The field of game theory has important links to economics and political science. The classic work is Von Neumann and Morgenstern's 1944 work "Theory of games and economic behavior" [707, 708], that laid the foundation for economic game theory. Many more modern and accessible works have been published since then, such as [160]. Classical game theory analyzes strictly rational behavior, the classical *homo economicus* is assumed to always act rationally. Many years later behavioral economics married psychological insight with economics. Kahneman and Thaler have written popular and accessible accounts [336, 658].

In contrast, in collaborative games the players win if they can create win/win situations. Again, the real world is full of opportunities for collaboration, and these games can be studied to hone our understanding of collaboration intelligence. Examples of collaboration games are Bridge, negotiation games such as Diplomacy [369, 165], management simulations such as Hexagon [186], or board games such as Magic Maze. Collaboration also occurs frequently in multi-player games such as Poker and Risk.

Perfect vs Imperfect Information

In perfect information games all relevant information is known to all players. This is the case in typical board games such as Chess and Checkers. In imperfect information games some information may be hidden for some players. This is the case in typical card games such as Bridge and Poker, where not all cards are known to all players.

A special form of (im)perfect information games are games of chance, such as Backgammon and Monopoly, in which dice play an important role. There is no hidden information in these games, and these games are therefore sometimes considered to be perfect information games, despite the uncertainty present at move time.

Turn-Based or Simultaneous-Action

In turn-based games players move in sequence. In simultaneous-action games, players make their moves at the same time. Simultaneous-action games are con-

sidered imperfect information, since players hold secret information that is relevant for a player at move time.

Chess, Checkers and Monopoly are examples of turn-based games. StarCraft and Diplomacy are simultaneous-action games.

Decision Complexity

The difficulty of playing a game depends on the complexity of the game. The decision complexity is the number of end positions that define the value (win, draw or loss) of the initial game position (see Chapter 4 for an explanation of *critical tree*). The larger the number of actions in a position, the larger the decision complexity. Games with small board sizes such as Tic-Tac-Toe (3×3) have a smaller complexity than games with larger boards, such as Gomoku (19×19). When the action space is very large it can be treated as a continuous action space. In Poker, for example, the monetary bets can be of any size, defining an action size that is practically continuous.

State Space Complexity

The state space complexity of a game is the number of legal positions reachable from the initial position of a game. State space and decision complexity are normally positively related, since in many games the structure of the game play that defines a winning position resembles the game play starting from the initial position. As we shall see in Chapter 4 determining the exact state space complexity of a game is a non-trivial task. For most games approximations of the state space have been calculated.

In general, games with a larger state space complexity are harder to play (“require more intelligence”) for humans and computers.

Now that we have seen several important aspects of games, let us look at concrete examples of zero sum perfect information games.

2.2.2 Zero Sum Perfect Information Games

Chess and Go are two player, zero sum, perfect information, turn based, discrete action games. If games are used in the study of intelligence because the domain allows concentration on the study of strategic reasoning, then two player zero sum perfect information games are a good choice. Strategies, or policies, determine the outcome of these kinds of games. This was the first class of games tried by computer programmers.

2.2.3 Examples of Games

We will now give examples of games that have played an important role in artificial intelligence research.

Table 2.1 summarizes the games and their characteristics.

Name	board	state space	zero sum	information	turn
Chess	8×8	10^{47}	zero sum	perfect	turn
Checkers	8×8	10^{18}	zero sum	perfect	turn
Othello	8×8	10^{28}	zero sum	perfect	turn
Backgammon	24	10^{20}	zero sum	chance	turn
Go	19×19	10^{170}	zero sum	perfect	turn
Shogi	9×9	10^{71}	zero sum	perfect	turn
Poker	card	10^{161}	non-zero	imperfect	turn
StarCraft	real time strategy	10^{1685}	non-zero	imperfect	simultaneous

Table 2.1: Games



Figure 2.7: Chess

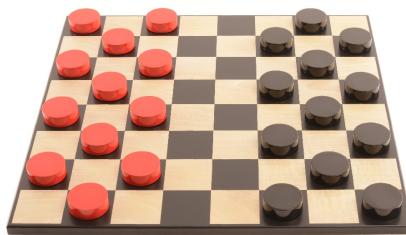


Figure 2.8: Checkers

Chess

Chess (see Figure 2.7) is a zero sum perfect information turn based game, played on an 8×8 board with 64 squares and 32 pieces, many with different move rules. The pieces move and can be captured. The goal of Chess is to capture the king. It has a state space complexity estimated to be 10^{47} . The number of actions in a typical board state is around 35. It is a game where material balance (the number and importance of the pieces that the sides have) is quite important for the outcome of the game. The ability to reason over deep tactical lines in order to capture a piece is important. Chess has been used since the start of AI research as a test bed for research.

Computer players are typically based on heuristic planning and consist of a search function (that looks ahead) and an evaluation function (that scores the board position using heuristics). A heuristic is a rule of thumb that captures domain knowledge. In Chess a useful heuristic is to count material balance (which side has more pieces) and mobility (which side has more legal moves and controls the center).

Chess is a tactical game in the sense that substantial changes in the static score of a position are frequent. Such sudden changes occur for example when an important piece, such as the Queen, is captured. Chess also has “sudden death:” the game can end in the middle when the King is captured (check-mate). Tactical situations (i.e., large changes between static evaluation between positions) can typically be dealt with by searching, or looking ahead.



Figure 2.9: Shogi

Checkers

Checkers (see Figure 2.8) is also a zero sum perfect information turn based game, played on an 8×8 board with 24 pieces. (North American and British Checkers is played on an 8×8 board. In most other countries the board size is 10×10 . In British English the game is called Draughts.) The pieces move diagonally (only on the dark squares) and can be captured. All pieces move the same. The number of actions in a typical board position is around 3. The goal of Checkers is to capture all the opponent's pieces. Ordinary pieces move only forward, until they reach the other side of the board, where they are promoted to kings, that can move backward as well. The game Draughts has similar rules, and is played on a 10×10 board. Checkers is mostly played in the UK and North-America, Draughts in other countries. As in Chess, material balance is a good indicator (heuristic) for the chance to win, with mobility a good second indicator.

The state space of Checkers is estimated to be 10^{18} . This state space is just small enough for the game to have been solved: it has been proven by traversing all relevant lines of play that perfect play by both sides yields a draw [570]. This computation took months of computer time by a state of the art research team.

Computer players for Checkers and Chess have a highly related design. Many of the techniques for search and evaluation functions that work on Checkers also work in Chess.

Shogi

Shogi (see Figure 2.9) is a zero sum perfect information turn based game, sometimes described as Japanese Chess. Shogi is played on a 9×9 board, with 40 pieces. Pieces can be captured, and when they are, they may be returned to the game as piece for the capturing side. The number of actions in a typical board position is around 92. The state space complexity of Shogi is significantly larger than Chess, 10^{71} , because of the reintroduction of pieces, and the larger board size, among others.

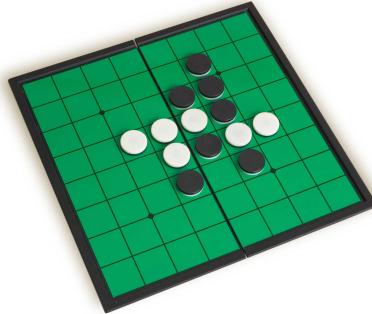


Figure 2.10: Othello

Important heuristics for Shogi are material balance and mobility. Techniques that work in Chess and Checkers playing programs often work in Shogi as well.

Othello

Othello (see Figure 2.10) is a zero sum perfect information turn based game, played on an 8×8 board. Othello is also known as Reversi. Othello is a disc flipping game. The number of actions in a typical board position is around 10. When a disc is placed adjacent to an opponent's disc, and there is a straight line to another friendly disc, the opponent's discs change color to the friendly side. In each move one disc is placed. Discs are never taken off the board. When the board is full the game stops, and the side with the most discs wins.

The state space complexity of Othello is estimated to be 10^{28} . Traditional heuristics such as material and mobility work to a limited extent. More advanced statistical and machine learning methods were needed for breakthrough performance [112].

Go

Go (see Figure 2.11) is a zero sum perfect information turn based game, played on a 19×19 board. It is also played on smaller boards, for quicker and easier games for teaching, 9×9 and 13×13 . Go originated in south-east Asia, and is popular in Korea, Japan and China. In Go stones are placed on intersections on the board. The number of actions in a typical board position is around 250. Isolated stones are captured, although this occurs infrequently among strong players. The board is large, and the state space complexity of Go is also large: 10^{170} .

In Go the objective in the game is to acquire territory, intersections that the opponent cannot claim. No effective heuristics have been devised for Go to calculate this territory efficiently for use in a real-time game playing program. Approaches as in Chess and Checkers failed to produce anything but amateur level



Figure 2.11: Go



Figure 2.12: Backgammon

play in Go. Sampling based methods in Monte Carlo Tree Search were needed for a breakthrough (see Chapter 5).

Capturing stones occurs less frequent in Go than in Chess. Go is a much more strategic game: stones, once placed, do not move, and typically radiate influence all the way to the end of the game. Strategically (long term) placement of stones to later work together is an important concept in Go. Sudden death (check-mate in Chess) does not exist in Go. Strategic implications of a move are long term, and are typically beyond the horizon of a search or look ahead. Strategic implications are typically dealt with by the evaluation function.

Backgammon

Backgammon (see Figure 2.12) is a zero sum turn based game of chance. Backgammon is one of the oldest and most popular board games, often played for money. It is played on a board as in the Figure. Dice are used to determine the number



Figure 2.13: Poker

of moves that a player can make. Each side has 15 pieces. The objective is to be the first to move all pieces off the board. The number of actions in a typical board position is around 250.

Backgammon is a game of skill and luck. The state space complexity of Backgammon is 10^{20} . Computer player BKG achieved some success in 1970–1980 with a heuristic planning approach [69, 70]. Later TD-Gammon used a small neural network achieving world champion level play [653].

Poker

Poker (see Figure 2.13) is a non-zero sum imperfect information turn based card game of chance. Betting is an integral part of the game play. It is used to signal the quality of a hand of cards (or to mislead). Poker is a multi-player game. The action space is large due to the betting, and varies greatly for different versions of the game. The difficulty of play is in estimating the quality of a hand with respect to others, and has psychological aspects, to ascertain if players are bluffing: placing high bets on weak hands, trying to scare other players.

The state space of No-Limit Texas Hold'em, a popular Poker variant, has been estimated between 10^{18} [77] and 10^{161} [103, 324]. There are many Poker variants, and this number can differ from variant to variant.

Poker is a multi-player, imperfect information game (the hidden hands 1 in the Figure) with elements of chance and psychology. Because of this it is closer to decision making situations that are common in the real world than games such as Chess and Checkers.

Most computer Poker players are based on a detailed analysis of the quality of the hands using a methods called counterfactual regret analysis [752, 104].



Figure 2.14: StarCraft

StarCraft

StarCraft (see Figure 2.14) is a non-zero sum imperfect information simultaneous action real time strategy game. It is a multi-player game. The goal of StarCraft is to gather resources, create buildings, develop new technologies, and train attack units, which then must prepare for and fight battles. StarCraft combines imperfect information with multi-player aspects, long term strategy and short term tactics. The imperfect information aspect is that the game map is only partially visible, and the actions of the other players are not all known. Armies have to be built up (strategy) and battles have to be played (tactics).

It is a highly popular, highly complex game. The number of actions in StarCraft positions has been estimated to be between 10^{50} and 10^{200} [487]. The state space complexity of StarCraft has been conservatively estimated by these authors at 10^{1685} , which is a very, very large number.

The complexity of the actions StarCraft comes even closer to real world decision making. As AI methods are improving, AI researchers have shown a great interest in creating computer StarCraft players.

2.2.4 History of Games Playing Programs

Now that we have studied the characteristics of some combinatorial games, it is time to have a look at the research field of computer game play.

In order to get a good understanding of the field, let us start with a short overview of history with some of the better known game playing programs in the history of AI. First we discuss two hypothetical paper design game players, and then we continue with the first real game playing programs.

Even before computers were powerful enough to run game playing programs, there were paper designs of game playing programs. There is a common approach in these attempts. Look ahead was formalized with a minimax search function and used a heuristic evaluation function to score board positions. Together the two functions formed the search-eval architecture, which we will study in detail in Chapter 4.



Photo: © Stanley Rowin

Figure 2.15: Claude Shannon

Claude Shannon 1949

Claude Elwood Shannon (1916–2001, see Figure 2.15) was an American mathematician, electrical engineer, and cryptographer. Shannon is primarily known as the father of information theory (but he also knew how to juggle).

In 1949 Shannon presented a paper titled “Programming a Computer for playing Chess.” The paper describes a design of how to program a computer to play Chess based on position evaluation and move selection. The paper describes strategies for restricting the number of possibilities to be considered. Shannon’s paper is one of the first articles published on the topic [600].

Alan Turing 1950

In 1948 Alan Turing (1912–1954, see Figure 2.17) worked with David Champernowne on a Chess program for a computer that did not yet exist. They named their program Turbochamp. In 1950, the program was finished. In 1952, Turing tried to implement it on a Ferranti Mark 1, see Figure 2.16, but the computer was unable to execute the program, so he ended up executing the program by hand, flipping through the pages of the algorithm and carrying out its instructions on a Chessboard. Reportedly, this took about half an hour per move. The program is the first to have played a full Chess game, even though it was executed on pa-

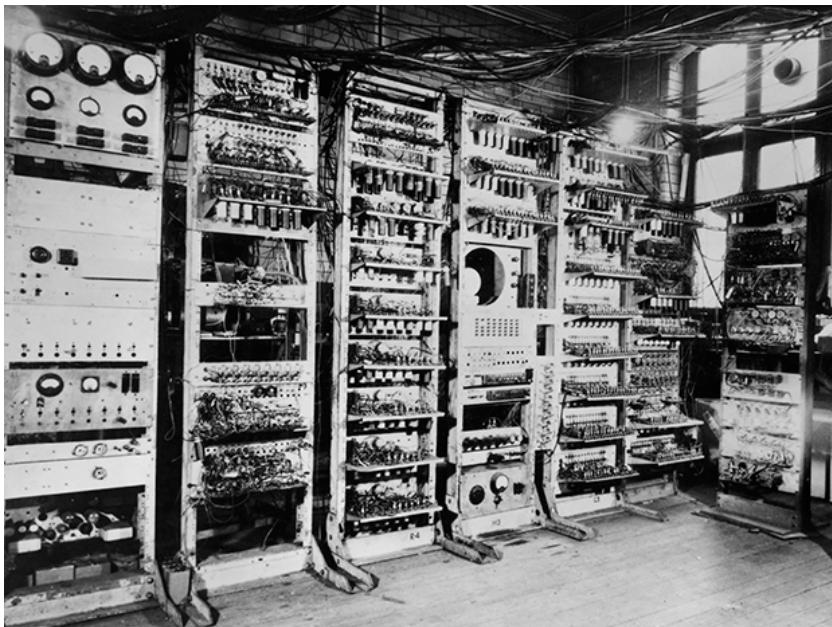


Figure 2.16: Ferranti Mark 1

per and by hand [679]. Turbochamp had an evaluation function and a search function, a design that is still used in today's programs.

Christopher Strachey 1951

Christopher Strachey (1916–1975, see Figure 2.18) was a preeminent computer scientist at Oxford, well-known for his work on denotational semantics and programming language design. In 1951, Christopher Strachey wrote a Checkers program for the Ferranti Mark 1 computer of the University of Manchester. It could play a complete game of Draughts (Checkers) at reasonable speed [629].

Arthur Samuel 1959

Arthur Samuel (1901–1990, see Figure 2.19) started in 1949 to write a Checkers program for the IBM 701 computer. It was among the world's first successful self-learning programs, adapting coefficients in the evaluation function based on the outcome of the search [560].

The main driver of the program was a search tree of the board positions reachable from the current state. To reduce memory consumption Samuel implemented a version of what is now called alpha-beta pruning. He used a heuristic evaluation function based on the position of the board to prevent having to search to the end of the game. He also used a method to memorize search results, and later versions

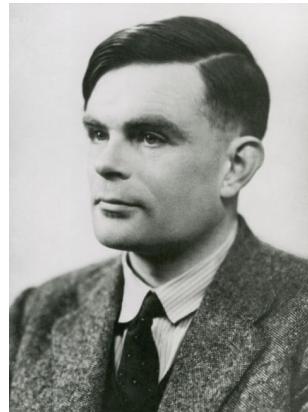


Figure 2.17: Alan Turing

used supervised learning from professional games and from self-play. Around the 1970s the program had progressed to amateur level [569].

2.3 Game Playing Programs

The playing strength of game playing programs has increased steadily since the early days. Computer hardware has become much more powerful, and better algorithms have been found. Most of these algorithms are described in this book.

After the pioneers we will now skip a few years. We will pick up around 1990, where some programs started playing at world champion level. We will discuss these world champion programs next.

2.3.1 TD-Gammon 1992

At the end of the 1980s Gerald Tesauro (see Figure 2.20) worked on Backgammon programs. Figure 2.12 showed a Backgammon board.

His programs were based on neural networks that learned good patterns of play. His first program, Neurogammon, was trained using supervised learning. It achieved an intermediate level of play [651]. His second program, TD-Gammon, was based on temporal difference learning and self-play, a form of reinforcement learning (see Chapters 3 and 6 and [637, 638]). Combined with hand-crafted heuristics, in 1992 it played at world-class human championship level, becoming the first computer program to do so in a game of skill [655].

TD-Gammon is named after temporal difference because it updates its neural net after each move, reducing the difference between the evaluation of previous and current positions.



Figure 2.18: Christopher Strachey



Figure 2.19: Arthur Samuel

In Tesauro's previous program Neurogammon an expert trained the program by supplying the "correct" evaluation of each position. In contrast, TD-Gammon initially learned knowledge-free, or *tabula rasa*. Tesauro describes TD-Gammon's tabula rasa self-play as follows: "The move that is selected is the move with maximum expected outcome for the side making the move. In other words, the neural network is learning from the results of playing against itself. This self-play training paradigm is used even at the start of learning, when the network's weights are random, and hence its initial strategy is a random strategy" [654].

The self-play version used only a raw board encoding, without any hand-crafted heuristic features. It reached a level of play comparable to Neurogammon: that of an intermediate-level human Backgammon player. After adding a simple heuristic search it reached world-class level.

TD-Gammon's success inspired many other researchers to try neural networks and self-play approaches, culminating eventually in recent high profile results in

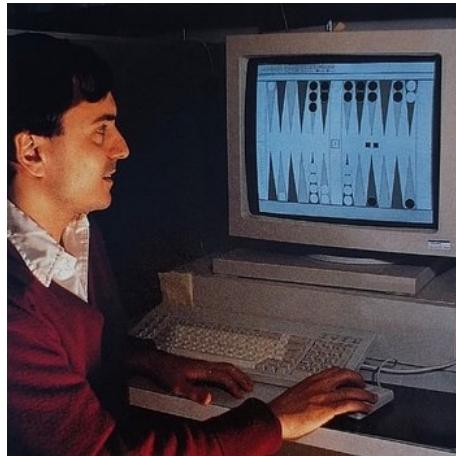


Figure 2.20: Gerald Tesauro



Figure 2.21: Jonathan Schaeffer

Atari [453] and AlphaGo [604, 607] (see Chapters 6 and 7).

Interestingly, a modern re-implementation of TD-Gammon in TensorFlow is available on GitHub TD-Gammon.⁵

2.3.2 Chinook 1994

The Checkers program Chinook is the first computer program to win the world champion title in a competition against humans. It was developed by Canadian computer scientist and Chess player Jonathan Schaeffer, see Figure 2.21. Based on his experience as a Chess player and programmer Schaeffer decided to write a Checkers program, believing that the technology existed to achieve world championship level.

Chinook's architecture follows the standard search-eval approach of computer Chess, based on alpha-beta search and a hand-crafted heuristic evaluation func-

⁵<https://github.com/fomorians/td-gammon>



Figure 2.22: Marion Tinsley

tion. Both functions were highly enhanced, well tested, and highly tuned (see Chapter 4) [571, 572]. Apart from meticulous testing and tuning, Chinook derived much of its strength from its endgame databases. These endgame databases contained perfect knowledge for any board position with 8 or fewer pieces on the board.⁶

Schaeffer started development in 1989. At first progress came swift. In 1990 Chinook won the right to play in the human World Championship by being second to Marion Tinsley in the US Nationals (see Figure 2.22). Tinsley was an exceptionally strong player. He had reigned supreme for such a long time in Checkers, that he enjoyed having a strong and fresh opponent again.

It took until 1994 before Chinook was declared the Man-Machine World Champion. In 1995, Chinook defended its Man-Machine title against Don Lafferty in a 32-game match. The final score was 1–0 with 31 draws for Chinook over Lafferty. At the time Chinook was rated at a very high 2814 Elo. After the match, Chinook retired from competitive playing. Later, Schaeffer and his team used Chinook to solve Checkers, proving in 2007 that with perfect play by both sides, Checkers is a draw [570].

Chinook's win is made all the more impressive because of the opponent it faced: Dr. Marion Tinsley. Tinsley, a professor of mathematics, played Checkers at a level that was unheard of. He was world champion from 1955–1958 and 1975–1991 and never lost a world championship match, and lost only seven games in his 45-year career, two of which to Chinook [569]. He withdrew from championship play during the years 1958–1975, relinquishing the title during that time. It was said that Tinsley was to Checkers what Leonardo da Vinci was to

⁶This endgame database technology allowed Schaeffer and his team to go on to solve the game and create a computational proof that Checkers, with perfect play, is a draw [570].

science, what Michelangelo was to art and what Beethoven was to music [426]. Schaeffer wrote a book about his experience, in which he recalls one particular event [569]. In one game from their match in 1990, Chinook made a mistake on the tenth move. Tinsley instantly remarked, “You’re going to regret that.” And, yes, after they played on, Chinook resigned after move 36, fully 26 moves later, when it saw its loss. Schaeffer looked back into the database and discovered that Tinsley picked the only strategy that could have defeated Chinook from that point and that Tinsley was able to see his win an implausible 64 moves into the future [569].



Figure 2.23: Feng-hsiung Hsu not playing Chess

2.3.3 Deep Blue 1997

If the Checkers events caused excitement, then Chess did even more so. In 1985 Feng-hsiung Hsu (see Figure 2.23), a PhD student at Carnegie Mellon University, started work on a Chess program that, after much development, showed enough promise for IBM to become involved. After quite some more years of development, in 1996 performance had improved enough for IBM to sponsor a match against the human world Chess champion Garry Kasparov (Figure 2.24). The match took place at the ACM Computer Science Conference in Philadelphia. Figure 2.25 shows a team photo from those days. On February 10, 1996, Deep Blue won its first game against human champion Kasparov, making Chess history. However, Kasparov went on to win three games and draw two, winning the six-game match by 4–2.

A year later, in May 1997, the rematch was played in New York at the 35th floor of the Equitable Center (see Figure 2.26). An upgraded version of Deep Blue played Kasparov again, this time capable of evaluating 200 million positions per second, twice as fast as the 1996 version. Kasparov was defeated $3\frac{1}{2} - 2\frac{1}{2}$ in a



Figure 2.24: Garry Kasparov

highly publicized match. Deep Blue became the first computer system to defeat a reigning World Chess Champion in a match under standard Chess tournament time controls. Several accounts of the match, its historical significance, and the technology behind it, have been written [574, 118, 303, 302, 405, 474].

Deep Blue started out as a hardware design project, when Feng-hsiung Hsu wanted to see if he could create a special purpose VLSI Chess move generator chip. This project evolved into a full-fledged Chess program, which was named Deep Thought, after the fictional computer in Douglas Adam's *The Hitchhiker's Guide to the Galaxy*, that computed the answer to Life, the Universe, and Everything.⁷ After their graduation from Carnegie Mellon, Hsu, Thomas Anantharaman, and Murray Campbell were hired by IBM Research to continue their quest to build a Chess machine that could defeat the world champion. IBM named the machine Deep Blue, after IBM's nickname, Big Blue. Anantharaman subsequently left IBM and Joe Hoane and Jerry Brody joined the team in 1990. Figure 2.25 shows the Deep Blue team. Grandmaster Joel Benjamin was signed up by IBM Research to assist with the preparations for Deep Blue's matches against Garry Kasparov.

The design of Deep Blue followed the standard search-eval architecture, with quite a number of special enhancements. To start with, the evaluation function used 480 specially developed hardware chips, allowing it to evaluate 200 million positions per second in parallel. The search algorithm was also heavily parallelized, running on a 30 node IBM RS/6000 system. Deep Blue would typically search to a depth of 6–8 moves and to a maximum of more than 20 moves in some situations. The opening library was provided by grand-masters Miguel Illescas, John Fedorowicz, and Nick de Firmian. It had an endgame database of positions with 5–6 pieces.

Deep Blue's evaluation chips were highly customizable, containing 8000 in-

⁷which, as you may know, is 42.



Figure 2.25: 1996 Deep Blue Team: Chung-Jen Tan, Murray Campbell, Joe Hoane, Feng-hsiung Hsu, and Jerry Brody

dividual parts, and a database of master games was used to tune their weights. The success of Deep Blue was rightfully billed as the success of brute force, due to the impressive number of 200 million evaluations per second. However, the fact that the evaluation weights were initially tuned by supervised learning, and not by hand, would equally allow a claim to a victory due to machine learning (although it did not use neural nets, nor did *Deep Blue* use any deep learning).

Before the matches, Kasparov was not allowed to practice against Deep Blue to familiarize himself with the play style of the computer. However, Kasparov did study many popular PC games to become familiar with computer game play in general.

Kasparov was quite upset during the match. After the loss in game 2, he said that he sometimes saw "deep intelligence and creativity" in the machine's moves, implying that during the second game, humans had intervened on behalf of the machine. IBM denied, saying the only human intervention occurred *between* games, but not *during* games. The rules allowed the developers to fix errors between rounds, an opportunity they used, and which may have caused apparent differences in play style between games.

In its time, Deep Blue, with its capability of evaluating 200 million positions per second, was the fastest computer to face a world chess champion. Since then, the focus has shifted to software, rather than using dedicated chess hardware. Chess algorithms have progressed, and modern chess programs such as Stockfish, Houdini, and Deep Fritz are more efficient than the programs of Deep Blue's era.



Figure 2.26: Excitement in 1997. Game 6, Caro Kahn. Kasparov resigned after 19 moves

Today less hardware is needed to achieve the same performance. In a November 2006 match between Deep Fritz and world chess Champion Vladimir Kramnik, the program ran on a computer system containing one dual-core Intel Xeon 5160 CPU, capable of evaluating only 8 million positions per second, but searching to an average depth of 17 to 18 plies in the middle game, thanks to fast heuristics and aggressive search extensions (see Chapter 4). Stockfish is currently one of the strongest Chess programs. It is open source, available on GitHub here.⁸

2.3.4 Logistello 1997

After the massive team-effort for Chess, a year later a one-person effort achieved impressive results in the game of Othello. The program is named Logistello. It was written by Michael Buro (see Figure 2.27) and was a very strong player, having beaten the human world champion Takeshi Murakami with a score of 6–0 in 1997. The best Othello programs are now much stronger than any human player.

Logistello's evaluation function is based on patterns of discs, and has over a million numerical parameters which were tuned using advanced logistic linear regression [112, 115, 116].

⁸<https://github.com/official-stockfish/Stockfish>



Figure 2.27: Michael Buro



Figure 2.28: David Silver

2.3.5 AlphaGo 2016

After the 1997 defeat of the Chess World Champion, the game of Go (Figure 2.11) became the next benchmark game, the Drosophila of AI, and research interest in Go intensified significantly.

Traditionally, computer Go programs followed the conventional Chess setup of a minimax search with an influence-based heuristic evaluation function. The GNU Go program is a good example of this approach [441]. This Chess approach, however, did not work for Go. The level of play was stuck at mid-amateur level for quite some time.

In a sense, the game of Go worked very well as Drosophila, in that new AI approaches were developed, and many researchers produced interesting findings. New algorithms have been developed, such as Monte Carlo tree search, and impressive progress has been made in deep supervised learning and deep reinforce-



Figure 2.29: Michael Bowling



Figure 2.30: Tuomas Sandholm

ment learning.

In 2015–2017 the DeepMind AlphaGo team, headed by David Silver (see Figure 2.28) played three matches in which it beat all human champions that it played. Books are being written analyzing the theoretical innovations that AlphaGo has unlocked for human Go masters.

We will cover all these developments in quite some depth in this book. Chapter 5 covers the search algorithm, Chapters 6 covers machine learning, and Chapter 7 goes in depth into self-play.

Appendix C contains a brief tutorial on the rules of Go.

2.3.6 Poker 2018

In contrast to games of perfect information such as Chess, Checkers, and Go, Poker is a game of imperfect information. It is a card game in which some of the cards are hidden from the player. The set of possible actions is large, posing a challenge for search-based AI approaches. This has held back the development of strong Poker programs for some time.

Recently, however, impressive progress has been made by groups from the

University of Alberta headed by Bowling [93, 94, 461] (Figure 2.29) and Carnegie Mellon University headed by Sandholm [562, 103] (Figure 2.30) in heads-up (two-person) variants of Poker. Programs DeepStack and Libratus have defeated some of the strongest human Poker players in one-on-one play [105]. Tuomas Sandholm is the 2003 recipient of the IJCAI Computers and Thought award. A year later they published equally strong results for multi-player Poker, with the Pluribus program [104], defeating top players in six-player Poker.

Research in computer Poker is an active field, with versions of research programs being licensed for commercial entertainment purposes. Section 8.2.4 provides more information on the methods for Poker.

2.3.7 StarCraft 2018

StarCraft is a popular real time strategy game with strong human champions. Researchers have become interested in using the game as a test bed for AI. Since 2010 academic competitions have been organized, that stimulated research [671, 487]. In 2017, the companies DeepMind and Blizzard together have created a Python interface to the game [704].

In 2018 DeepMind's player AlphaStar played two-person test matches against some of the world's strongest players, which it won [701]. StarCraft is becoming quite an active field of research, and more research can be expected. Section 8.2.5 provides more information on StarCraft methods.

2.3.8 Conclusion

We have covered a lot of ground in this chapter. We have discussed elements of intelligence, and mentioned that most AI researchers take a behavioristic approach to intelligence. “To be or not to be,” is not the question, most AI researchers take the pragmatic behavioral approach of duck-typing: “If it walks like a duck and quacks like a duck, then it must be a duck.”

We have looked at elements of games, and discussed the concepts *zero sum* and *perfect information*. We have also had a look at historic approaches to game playing programs, and we have listed modern successes, where computer programs have achieved a level of play beyond that of the strongest human players.

It is now time to prepare ourselves for the four core chapters that look in detail at the methods that have been developed by researchers. We will do so by first discussing the reinforcement learning paradigm, as a common language for the four following chapters.

2.4 Summary

This chapter discussed key aspects of AI: recognition, reasoning, memory and learning. AI takes a behavioristic approach to intelligence, which is operationalized with the Turing Test, introduced by Alan Turing in 1950. We then proceeded with a brief history of AI, starting with the 1956 Dartmouth College workshop.

A list of ambitious goals for AI were formulated, which turned out to be overly ambitious, causing a AI funding winters. However, after research picked up by the turn of the millennium, a number of AI technologies (such as search, social networks, deep learning, and recommender systems) had deeply affected society, and AI summer arrived.

Games have been a favorite test-bed for AI research. Games such as Chess and Go have been called the Drosophila of AI, after the fruit fly from genetics research. An important difference between human intelligence and artificial intelligence is that human intelligence is general, and artificial intelligence is special. Human intelligence can solve many problems, an AI program can solve only one, such as playing good chess. Research is under way into artificial general intelligence.

The chapter provided a short overview of important designs and programs. We listed the early designs by Shannon, Strachey, Turing, and Samuel. The programs featured were TD-Gammon (Backgammon), Chinook (Checkers), Deep Blue (Chess), Logistello (Othello), AlphaGo (Go), Libratus and DeepStack (Poker) and AlphaStar (StarCraft).

Games are a favorite experimenter's tool for AI. Many breakthroughs in intelligence and learning have taken place in a variety of games, each with their own characteristic.

The early designs established some of the basic algorithms and approaches, such as the search-eval architecture, minimax, alpha-beta, heuristics, and learning. Computers were not yet powerful enough to reach strong levels of play. The later programs all did, using highly enhanced versions of the original blueprints, as well as new techniques, such as deep learning.

Historical and Bibliographical Notes

We conclude each chapter with a brief summary of entry points to the scientific literature. The works mentioned here may be more accessible than some of the more technical papers.

In this chapter we now list three such works. An excellent popular and all-encompassing standard text book of AI is by Russell and Norvig [554]. If you have taken an undergraduate course in AI, chances are that you already own a copy.

Artificial intelligence and search are also used to generate content for computer games [672]. An excellent text on how to use artificial intelligence for creating good computer games is Yannakakis and Togelius [740], which also covers procedural content generation and game design.

OpenSpiel is a library for reinforcement learning in games. It provides a wealth of high quality implementations of algorithms [385]. A link to OpenSpiel is OpenSpiel.⁹ Appendix A contains more environments relevant to research in reinforcement learning and games.

⁹<https://deepmind.com/research/open-source/openspiel>

A popular psychology book is Thinking Fast and Slow by Kahneman [336]. Kahneman is one of the founders of the field of behavioral economics, and his approach to intelligence fits well with the search-eval architecture that is prevalent in artificial intelligence in games.

Chapter 3

Reinforcement Learning

In order to study our games in a way that is precise enough to program solution methods for a computer, we need a suitable paradigm and formal language. The paradigm that we will use is reinforcement learning. Reinforcement learning studies how agents interact with their environment, and allows solution methods for games to be formalized concisely and precisely.

Reinforcement learning is a general paradigm, with links to behavior studies and trial and error conditioning. Our language will be quite mathematical in nature, but the implicit links to psychology are never far away.

To start, let us summarize the core problem and concepts that will be covered in this chapter.

Core Problem

- How can an agent learn from interaction with an environment?

Core Concepts

- Agent and environment
- Markov decision process, state, action, policy, reward, value function
- Credit assignment, exploration/exploitation, exact/approximate

In order to study our board games, we need a suitable paradigm and formal language. The language should be precise enough to allow us to program our ideas into a computer. Reinforcement learning agents attempt to learn which actions to take by trial and error. The reinforcement learning paradigm consists of an agent and an environment (see Figure 3.1). The environment is in a certain state. At time t , the agent performs an action, resulting in a new state in the environment. Along with this new state comes a reward value (which may be

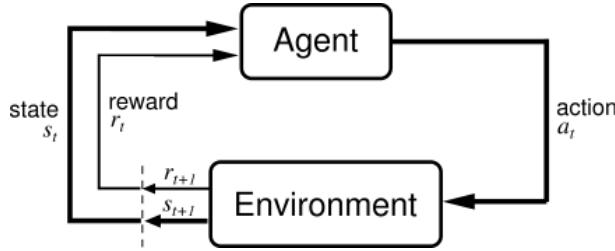


Figure 3.1: Reinforcement Learning [638]

positive or negative). The goal of the agent is to learn a how to maximize the rewards that the environment returns. The function that returns an action for each possible state is called a policy. The optimal policy is learned (reinforced) by repeated interaction with the environment.

3.1 Markov Decision Processes

Reinforcement learning problems can be modelled as Markov Decision Processes (MDP). Markov Decision problems are an important class of problems that contain the Markov Property: the next state depends only on the current state and the actions available in it (no memory of previous states or other information necessary) [299]. The no-memory property is important because it makes reasoning about the state possible using only the information present in the current state. If previous histories would all influence the current state then reasoning about the current state would be much harder or impossible. MDPs are named after Russian mathematician Andrey Markov (1856–1922) best known for his work on stochastic processes.

Where board games use the terms *board position* and *move*, the mathematical formalisms MDP and reinforcement learning talk about *state* and *action*.

A Markov Decision Process is a 5-tuple (S, A, P, R, γ) :

- S is a finite set of legal states of the environment; the initial state is denoted as s_0
- A is a finite set of actions (if the set of actions differs per state, then A_s is the finite set of actions at state s)
- $P_a(s, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$ in the environment
- $R_a(s, s')$ is the reward received after action a transitions state s to state s'
- $\gamma \in [0, 1]$ is the discount factor representing the difference between future and present rewards.

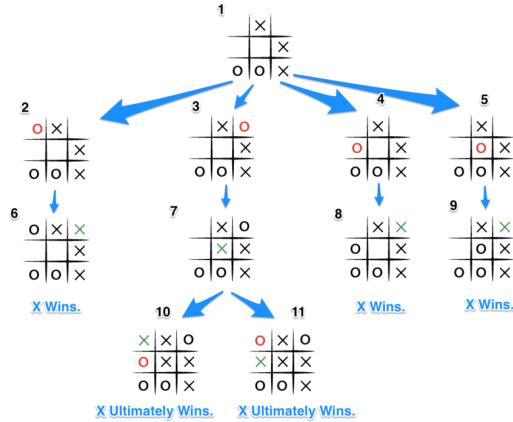


Figure 3.2: Tic Tac Toe Game Tree

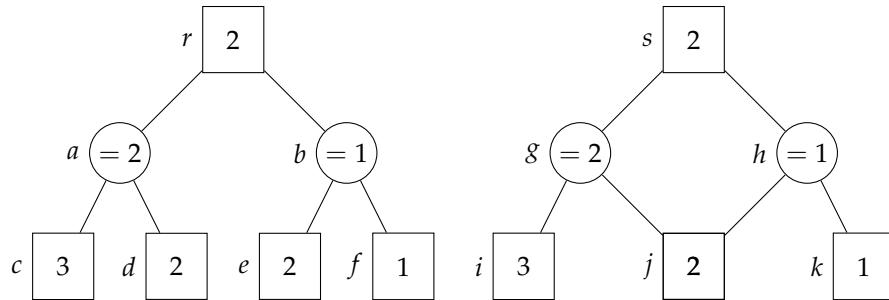
Deterministic board games such as Chess and Go can easily be formulated as MDPs with $\gamma = 1$ (future outcomes are as important as present outcomes) and $P = 1$ (actions in a state predictably lead to future states).¹

3.1.1 Reinforcement Learning, MDP and Graph Terminology

Reinforcement learning introduces a new formal terminology for board games. The board configuration is called state s , a move is action a , and the score of an end position is the reward R (typically win, loss or draw). In reinforcement learning a strategy for a player (all the moves from positions leading to a certain end position and outcome) is called a *policy*. In games such as Chess and Go, actions lead deterministically to one state (in Poker and Backgammon they can stochastically lead to different states, see [638]). The state space of a game is the set of all states that are reachable from the initial board or state.

The state space is easily visualized with a directed graph. Fig. 3.2 shows a partial graph of a position in the well-known game of Tic Tac Toe (also known as Noughts and Crosses, or Three in a Row). Graphs introduce yet another terminology. States/boards are nodes in the graph, actions/moves are links, and terminal states are leaves. Note that the graph in the figure does not have cycles, and is a tree (artificial intelligence is full of “trees,” all growing upside down, with the root at the top). Such a graph of the state space of a game is often called a game tree. In general, MDPs and game graphs may contain cycles, although most games have rules to prevent never-ending game situations. Another difference between graphs and trees are transpositions, nodes that have more than one parent. Transpositions are identical board positions that can be created by different move

¹To be precise, only versions of Chess and Go that do not use rules to prevent repetition, such as the 50-move rule and the ko-rule. The repetition-prevention rules require memory and thus violate the Markov property.

Figure 3.3: A Tree and a Graph; Node j is a Transposition

sequences. Section 4.3.3 discusses transpositions in more detail. A tree does not contain transpositions, graphs can. In a tree only one path exists between any two nodes, all children have exactly one parent. In a graph child nodes may be shared between parents, there are more ways than one to reach a certain position. See Figure 3.3 for an example of a tree and a graph.

A score function is defined for the leaves of the game tree to tell whether the game has been won, lost, or drawn. In reinforcement learning terminology we say that the reward is known. If we look at the graph in Figure 3.2 again, and we want to find the value of a non-terminal board position, all we have to do is work upwards from the leaves of the tree, and determine the values of the parent nodes, until finally the value of all nodes in the tree is known. At that point we know if the initial position of the game is won or lost if both players play perfectly.

Now that we have discussed basic terminology we are ready to look at two central elements in reinforcement learning: the policy function and the value function.

3.2 Policy Function and Value Function

In reinforcement learning two functions are usually associated with the states: the policy function and the value function.

3.2.1 Policy Function

Of central importance in reinforcement learning is the concept of *policy*, or π . The policy function π associates states with actions. A deterministic policy $\pi: s \mapsto a$ returns one action for each state. A stochastic policy $\pi(a|s)$ gives the probability of taking action a in state s .

For two person games the goal of reinforcement learning and MDPs is for the agent to find optimal policy π^* , that gives an action $a = \pi(s)$ that should be chosen in each state s that leads to a win [638]. Finding the optimal policy function is also known in engineering as solving the *optimal control problem* [638].

3.2.2 Value Function

The value function $V(s)$ returns the expected value of state s . In leaves, the “expected” value is equal to the reward $V(s) = R(s)$. The expected value $V(s)$ is determined by the current policy, and should therefore be written $V^\pi(s)$. Many solution methods exist to determine efficient value functions and policy functions. Finding the value function solves the *estimation problem*.

In the next chapter, Chapter 4, we will discuss methods that efficiently determine this root value, and, by extension, the optimal policy π^* , the tree of actions that leads to it.

3.2.3 Q(s,a) Function

The value of taking action a in state s under policy π is the expected return starting in state s , taking action a and then taking policy π for the remainder. The value of the Q function is equal to the value function when taking action a in state s .

$$Q^\pi(s, a) = V_a^\pi(s)$$

The final value of state s is given by the value function V^π , after the fact, when all computations have been done. The $Q^\pi(s, a)$ function gives the value of a single action a , and is important during the calculation. The Q -value is computed for intermediate single action values. Important algorithms such as Q-learning and DQN are named after this function.

Functions and Mappings as Tables and Parameterized Networks

It is common to discuss value and policy problems in terms of functions. Functions map input values to output values, and are sometimes called mappings.

At this point it is useful to consider that Value and Policy functions can be implemented in different ways.

The most basic and easy to implement situation is when for a function $f(x) \mapsto y$ a closed-form mathematical formula such as $f(x) = x + 5$ is known. The formula can be implemented straightforwardly as a procedure that takes its input arguments and runs some code to calculate the return value: `def f(x): return x+5`. This is the *code* approach.

A slightly more complicated situation is the following. Some non-linear functions $f(x) \mapsto y$ can not be expressed as a closed form mathematical formula, or no such formula is known. However, when a database of input/output pairs is available, then we can implement the mapping as an array or lookup-table in most programming languages: `def f(x): return table[x]`. This is typically the case in machine learning problems, where the resulting function is to be learned by looking at examples. When the number of examples is manageable, they can be stored in memory. The function is then implemented as a lookup table. This is the *database* approach.

For most real world machine learning situation an even more complicated situation holds, where the database is too large to fit in memory, or the training

set must generalize to cover unseen examples from the test set. In this case the function can be learned with machine learning methods. The function values will necessarily be approximated. The function mapping can be implemented as a parameterized (neural) network that takes as input a state, and produces as output a value. This is what is used in many real world machine learning situations. Since the function is approximated by a set of parameters θ , it is often written as $f_\theta(x) \mapsto y$. This is the *function approximation* approach.

3.3 Solution Methods

Now that we have introduced the basic elements and terminology of reinforcement learning, it is time to discuss solution methods, how to find the optimal values and policies.

In this section we will introduce the many building blocks of reinforcement learning: the Bellman equation to recursively compute root values, the exploration/exploitation trade-off to find optimal paths, temporal-difference learning to find the policy by sampling, the value iteration and policy iteration solution methods, and on- and off-policy methods for learning the optimal policy.

In this section the methods will be introduced one by one. At the end of the chapter full algorithms are described that combine all these theoretical concepts.

We will now start with two essential elements of reinforcement learning: Bellman recursion and the exploration/exploitation trade off.

3.3.1 Bellman Recursion

Games consist of actions by the players. For each action that the computer player makes, the optimal policy of the board position must be found by the program. (Or, equivalently, the optimal value must be found.) In the previous section we discussed Tic Tac Toe and how the optimal policy and the value of a position depend on the rewards, or leaf values. We mentioned how the value of the root can be found by recursively traversing the tree upwards, working backwards, computing the values of inner nodes. This recursive solution method described informally in the previous section can be described formally with the Bellman equation. Figure 3.5 gives an artistic impression of the concept of recursion, known as the Droste-effect. Richard Bellman (Figure 3.4) showed that discrete optimization problems can be described as a recursive backward induction problem [56], using what he called dynamic programming. All that is needed is the relationship between the value function in one state and the next state. This relationship is called the *Bellman equation*.

The Bellman equation for the expected value for being in state s and following policy π is:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^\pi(s')$$

where R is the reward function and γ is the discount rate. Before a node has been searched, it is not yet known which child is the best successor. The Bellman



Figure 3.4: Richard Bellman

equation defines a recursive relation on how to compute the expected value of the initial state S_0 . Note that for common two person zero sum perfect information games the search space is defined implicitly by the rules of the game and the initial board position, of which we desire to know the optimal policy and the expected value (win, draw or loss). For common two person zero sum perfect information games the Bellman equation simplifies greatly: $\gamma = 1$, and R is commonly chosen to be 1 for win, -1 for loss, and 0 for draw, and also for intermediate nodes. The expected value therefore depends recursively on the rewards of leaf nodes. The probability of transitioning from state s to s' under perfect information in the optimal policy is 1. In perfect information game playing algorithms the Bellman equation reduces to a simple backup function.

Dynamic programming

Dynamic programming was introduced by Bellman in the 1950s. It recursively breaks down problems into smaller sub-problems that are then solved. Dynamic



Figure 3.5: Recursion: Droste-effect

programming can be used to solve problems whose structure follow the Bellman equation. The key idea of dynamic programming is to use the value function to guide the search for good policies [638]. We will discuss two dynamic programming algorithms for solving reinforcement learning problems in Section 3.5. In the next chapter, we will see more dynamic programming methods.

Approximate Solutions

An important choice in implementing reinforcement learning algorithms is which storage concept to use for the states, the actions, and the policy and value functions. For small state spaces, all states can be stored individually in a table or a tree. Small state spaces can be enumerated and solutions can be exact.

Many reinforcement learning problems have state spaces that are arbitrarily large. For example, the number of possible images of even a small black and white picture is very large (the number of gray scale values raised to the power of the number of pixels). The number of instances encountered during training will be small in relation to the possible number of instances, the trained policy should work well on unseen problems. Employing some kind of generalization is crucial for achieving meaningful performance in learning a good model of the domain observations. The number of possible states is too large to be stored,

so some form of generalization has to be used. Where exact methods store the Q-function values in a table with an entry for each state/action pair, for large state spaces function approximation must be used. Artificial neural networks are popular methods.

Neural networks, or function approximators, are parameterized by a vector of weights. The length of the vector is much less than the number of possible states. States are represented by the weight vector. This has two advantages. First, the examples from the very large number of possible states can be stored efficiently in limited memory. Second, the example states are generalized according to some method, recording the generalization into the weight parameters, allowing unseen states to be recognized as belonging to some (hopefully appropriate) value or class.

In classification or other statistical tasks where unseen states must be classified or evaluated, generalization is a goal by itself. Through generalization the “essence” of the states is found. Generalization transforms a high dimensional space to a lower dimensional space.

In Chapter 6 approximation methods will be introduced. Deep Q-networks (DQN) will be discussed. DQN consists of a deep convolutional network for function approximation, which is trained by reinforcement learning using Q-learning.

3.3.2 Exploration and Exploitation

Now that we know how to calculate the expected value of a state, we should look at how to do so efficiently. For each action to be taken from a state, let us maintain a table of numbers that represent the probability of winning for that action. As actions are performed, the value in the table begins to reflect more reliably the expectation of winning when taking that action.

Now the question is: Which action should we take? One approach is to always choose the action with the highest winning probability. This approach is called a *greedy* approach. Another approach is to sometimes try another action, temporarily ignoring the one with the highest winning probability, exploring a new successor state, in the hope to find an untried action with an even higher winning probability. The ϵ -greedy approach is to mostly try the best (greedy) action except to explore an ϵ fraction of times a randomly selected other action. If $\epsilon = 0.1$ then 90% of the times the best action is taken, and 10% of the times a random other action. The best action is exploited 90% of the times, a random exploration action is taken 10% of the times. The ϵ -greedy policy is an example of a *soft* policy: a policy with a finite probability of selecting any of the possible actions.

This choice between greedily exploiting known information and exploring unknown nodes to gain new information is called the exploration/exploitation trade off. It is a central concept in reinforcement learning.

Let us look at the exploration/exploitation trade off in more concrete terms. Assume you have moved into a new neighbourhood, which you do not yet know, and you want to find the route to the supermarket. After some trial and error, you have found a route, which you meticulously memorize. The greedy approach is

then, when you need to go to the supermarket again, to always follow this same route. You exploit the knowledge that you have to the fullest, without ever trying out a possible new route (exploring). Trying out a different route will take some extra time and effort, but might pay off in finding a shorter route, which you can then use many times for as long as you live in the neighbourhood.

Clearly, always exploiting might be the safe way, but is likely to not find an optimal solution. How much to explore, and in what situations, is a fundamental topic in reinforcement learning that has been studied in depth in the optimization literature [294, 733] and in multi armed bandit problems [21, 381] (see Chapter 5).

Smart use of exploitation and exploration is at the basis of the breakthroughs in Go. In Chapter 5 we will discuss in depth the Monte Carlo Tree Search algorithm, MCTS, whose selection rule UCT makes extensive use of bandit theory to find a good exploration/exploitation trade off.

Temporal Difference

We now have discussed how to calculate the values of states when the policy function is known and how to then find the optimal policy for states. We are building up our tools to construct solution methods with which we can estimate the optimal policy and value of a game state, allowing us to construct a game playing program that can compute the best moves so that it can play a full game.

The Bellman equation calculates the value of states using the policy function. When the policy function is implicit in interactions with the environment, and the policy is the function we want to estimate, we need another solution method.

Perhaps the best known solution method for estimating policy functions is temporal difference learning [637]. This method is at the basis of TD-Gammon [654]. The temporal difference in the name refers to the difference that it calculates between two time steps, which it uses to calculate the value of the new time step.

The Bellman equation describes how $V(s)$ can be calculated based on the reward function and the transition function P . TD is different, under the policy π TD samples from the environment to adjust the current $V(s)$ estimate using a learning rate. TD does not need to know the transition function P and estimates the policy function π . TD finds $V(s)$ by sampling.

TD works by updating the current estimate of the state $V(s)$ with an error value based on the estimate of the next state. It is a bootstrapping method: the update takes into account previous estimates. Estimates are backed up through the state space.

$$V(s) \leftarrow V(s) + \alpha[R' + \gamma V(s') - V(s)]$$

Here s is the current state, s' the new state, and R' the reward of the new state. Note the introduction of α , the learning rate. The γ parameter is the discount rate. The last term $-V(s)$ subtracts the value of the current state, so that TD computes the temporal difference. Another way to write the update rule is $V(s) \leftarrow$

```
def value_iteration():
    initialize(V)
    while not convergence(V):
        for s in range(S):
            for a in range(A):
                Q[s, a] = next_q(s, a)
            V[s] = max_a(Q[s])
    return V
```

Listing 3.1: Value Iteration pseudo code (based on [7])

$\alpha[R' + \gamma V(s')] + (1 - \alpha)V(s)$ as the difference between the new TD-target and the old value.

Now, in order to arrive at a full reinforcement learning algorithm we also need a control policy. TD gives us a learning rule for value updates, the error-part of trial and error. For a full reinforcement learning algorithm we also need a control policy, which tells us which nodes to choose for expansion, the trial part of trial and error. Two such control policy are SARSA, yielding on-policy TD control, and Q-learning, yielding off-policy TD control. SARSA and Q-learning are discussed in Section 3.5.

3.3.3 Value Iteration and Policy Iteration

We are now ready to describe full solution methods.

Value Iteration

Value iteration methods directly improve the estimate of the optimal value function by traversing the state space. Pseudo code for a basic version of value iteration is shown in Listing 3.1. Value iteration converges to the optimal value function by iteratively improving the estimate of $V(s)$. Value function $V(s)$ is first initialized to random values. Value iteration repeatedly updates $Q(s, a)$ and $V(s)$ values until convergence occurs (when values of $V(s)$ stop changing much). Value iteration has been proven to converge to the optimal values, but, as we can see in the pseudo code in Listing 3.1, it does so quite inefficiently by essentially enumerating the entire state space.

Value iteration methods help find the best action for each state by following the action that yields the optimal value function. This can work with a finite set of actions. However, convergence in large spaces is slow [7, 638].

Policy Iteration

Value iteration algorithms keep improving the value function at each iteration until the value function converges. However, most agents only care about finding the optimal control policy (the best move) and not its value.

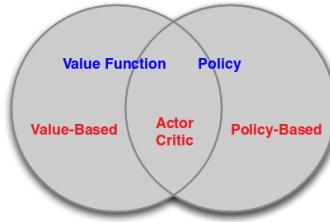


Figure 3.6: Value vs Policy in Actor-Critic

In games with a low branching factor, such as Checkers, there are only few actions in a state. When there are also many different state values (which is the case in most game playing programs) then the optimal policy may converge before the value function. Directly updating the policy function will converge faster than taking the route via the value function. Policy iteration methods directly improve the policy function. A well known example is the REINFORCE method that directly optimizes the policy without using a value function [731]. This approach is useful when the action space is continuous (or stochastic).

Both value iteration and policy iteration can be used when there is prior knowledge about the model/environment. For both methods the MDP 5-tuple, especially the transition and reward function, must be known. When that is not the case, then other methods are needed, such as Q-learning (to be discussed shortly).

Actor-Critic

As you may have guessed, value iteration and policy iteration steps can be combined. Generalized Policy Iteration (GPI) interleaves value and policy steps extending value prediction to include control.

Generalized Policy Iteration (GPI) is a general framework that applies to any interaction of value iteration and policy iteration. Actor-critic is one of the better known examples of GPI. It interleaves a policy function (actor) and a value function (critic) [363, 350, 638, 684, 364, 483].

The actor's role is to perform exploitation: it should escape from low-reward regions as fast as possible. It is aimed at maximizing the reward. It is greedy and works well with deterministic actions.

The critic's role is to do exploration: it should find examples where learning is optimal. It aims at minimizing the error. It works best in non-deterministic settings. This actor/critic value/policy idea is shown in Figure 3.6.

Actor-critic alternates policy evaluation with policy improvement steps. Actor-critic methods combine the best of both worlds: better convergence than value methods, and better policies due to learning and reduced variance. In Section 6.4.1 we will discuss A3C, an advanced actor-critic method in the context of function

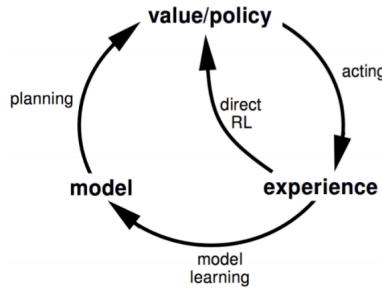


Figure 3.7: Model-free and Model-based methods

approximation that is an improvement over the popular Deep Q-Network (DQN) algorithm.

Now that we have described two important solution methods that require the full MDP model to be known, let us look in more detail at these models.

3.3.4 Model-free and Model-based

A basic distinction between reinforcement learning methods is between model-free and model-based methods. Model-free methods learn a value and policy function directly from the observations (see Figure 3.7, the inner loop). Model-based methods, on the other hand, first learn an intermediate model, often an MDP, which is then used by a planning algorithm to find the value and policy function. Model-based methods follows the outer loop in the Figure. Model-free methods are also known as direct methods.

Some reinforcement learning methods are model-based methods [338], and others are model-free [638, 223]. Model-based methods can be used when a model of the environment exists, for example, transition and reward functions are known. This can be a statistical model, a simulator, or the rules for the dynamics of the environment, such as we have in games.

When a model is available, it can be used to do look-ahead planning, as we shall do in the next chapters on planning. When no model is available, methods must be used to first learn a model of the environment. The chapters on training discuss deep reinforcement learning methods, that start with no model, and create a model: this book is therefore also about model-free reinforcement learning methods in games, in the chapter on function approximation.

Finally, combinations between model-based and model-free, between planning and training, have been successful. AlphaGo is perhaps the best known example of this combined approach [604], see Section 7.4.3 and 7.5.5 for many more works.

Let us now look at another central concept in reinforcement learning: that of on-policy and off-policy learning, methods for model-free learning.

3.3.5 On-Policy Learning and Off-Policy Learning

We will now discuss model-free solution methods, methods that also work when the reward function and the transition function of the MDP are not known.

All learning methods face a trade off: they try to learn the best action from current behavior that is known so far (exploitation), and they choose new (most likely non-optimal) moves in order to explore and be able to find moves that are better than the current best moves. In model-free reinforcement learning there are two important main approaches for exploration: on-policy learning and off-policy learning.

To discuss on-policy learning and off-policy learning we introduce a new value function. So far we have seen the value function $V(s)$ that returns the expected value of a state, the policy function $\pi(s)$ that returns an action for a state. The $Q(s, a)$ function returns the expected value of taking action a in state s .

On-policy learning and off-policy learning update the $Q(s, a)$ function. On-policy learning and off-policy learning focus on the way the policy is updated. Note that the policy is a mapping from states to actions. Policies are typically implemented as an array, or lookup table.

On-Policy

In on-policy learning a single policy function is used. On-policy learning updates actions directly on this single policy. The policy is used for exploration behavior and for the incumbent optimal policy.

The update formula is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

On-policy learning chooses an action, evaluates it, and moves on to better actions. A well-known on-policy approach is SARSA. On-policy learning starts with a starting policy, samples the state space with this policy, and improves the policy. Note that the term $Q(s_{t+1}, a_{t+1})$ can also be written as $Q(s_{t+1}, \pi(s_{t+1}))$, highlighting the difference with off-policy learning. In a short while we will look at SARSA example code, and compare its behavior to off-policy learning.

The primary advantage of on-policy learning is that it directly optimizes the quantity of interest, and achieves stable learning. The biggest drawback is sample inefficiency: since policies are estimated from roll outs themselves the variance is often extreme—if it goes in the wrong direction, there is no stabilizing other information to get it out—and the need for variance reduction remains large. It must either use on-policy data, or updates must be sufficiently slowed to avoid significant bias [466].

Off-Policy

Off-policy learning is more complicated. It uses two separate policy-arrays: one for exploratory behavior, and one to update as the current best policy. Learning is

from data *off* the best policy, and the whole method is therefore called off-policy learning.

The update formula is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

The difference with on-policy learning is that the $\gamma Q(s_{t+1}, a_{t+1})$ term has been replaced by $\gamma \max_a Q(s_{t+1}, a)$.

Off-policy learning collects all available information and uses it simultaneously to construct a good policy. The best known off-policy algorithm is Q-learning [721]. It gathers information from (partially) random moves, it evaluates states as if a greedy policy was used, and it slowly reduces randomness.

Differentiating On-Policy and Off-Policy Learning

To understand the difference, let us look at the update formulas. A way to differentiate on-policy and off-policy learning is to look how the update formula depends on the next action a_{t+1} , which can be written as $\pi(s_{t+1})$. Q-learning is off-policy learning since it does not depend on the next action a_{t+1} , i.e., it does not depend on the policy π . No matter what a_{t+1} is, the $\max_a Q(s_{t+1}, a)$ part does not depend on a_{t+1} . $Q(s_t, a_t)$ is updated regardless of the future policy π starting from s_{t+1} . On the other hand, SARSA is on-policy learning because the term $Q(s_{t+1}, a_{t+1})$ in $Q(s_t, a_t)$ depends on the next actual action a_{t+1} , the one taken by following the policy $\pi(s_{t+1})$. SARSA thus depends on the behavior policy.

To recapitulate, off-policy approaches can learn the value of the optimal policy regardless of the behaviour policy, while in on-policy methods the agent learns the value of the policy whose actions it is following.

Off-policy learning is especially important when there is a database of previously stored trajectories (i.e., data in the form of tuples (s, a, r', s')). This data has been collected by previously applying some policy, and cannot be changed. This is a common case, for example for medical problems. To use this kind of data only off-policy methods can be used.

Off-policy methods are more flexible in the type of problems they can be used for. Their theoretical properties, however, are different. If we compare Q-learning to SARSA, the difference is in the max-operator used in the Q-learning update rule. The max-operator is non-linear, which can make it more difficult to combine the algorithm with function approximators. With function approximators (neural nets) on-policy methods are usually more stable. In Chapter 6 we will cover learning stability of off-policy methods in more depth.

We will now look at an underlying topic of all machine learning: sample efficiency.

3.3.6 Sample Efficiency

An algorithm is sample efficient if it gets the most out of every sample and efficiently learns a function with only few samples. Model-based algorithms are

often sample efficient, using the model to converge quickly to an optimum with few samples, if the model is good.

Model-free methods lack such guidance and often require more samples. SARSA and Q-learning are effectively performing random search in the environment, not using information from previous samples to guide the search. A drawback of on-policy learning is that it samples from its own target policy, which may lead to myopia, tunnel vision, sampling only a part of the search space. When an algorithm learns only a part of the search space learning can be unstable.

In off-policy learning old samples may be used, not related to the target policy. This may prevent myopia, but not all samples may be useful in that they are not part of the distribution that we are interested in, also leading to low sample efficiency.

Several methods have been proposed to increase sample efficiency. One such method is importance sampling [242, 465, 720], which samples from a distribution that over-weights the important region. In this way the target is sampled heavily (as in on-policy) but the rest is not neglected completely (as in off-policy, preventing myopia).

3.4 Reinforcement Learning and Supervised Learning

We have discussed in depth many elements of the reinforcement learning paradigm, one of the major paradigms of machine learning. To conclude this chapter, let us now discuss some differences with supervised learning.

3.4.1 Supervised Learning

Machine learning is the part of artificial intelligence that studies mechanical learning principles. Machine learning algorithms are algorithms that use techniques to learn new patterns from data [80, 554].

In its most basic classification form, machine learning works by looking at pairs of examples and labels (E, L). The task of the learning algorithm is then to associate the correct label with the correct class of examples, so that when shown a new example, it classifies it with the correct label. A well-known machine learning problem is to learn the classification of a large sequence of pictures (examples E) as either dogs or cats (labels L). Such a problem is an example of supervised learning where the labels play the role of supervisor in the learning process. If not all examples have labels, then supervised learning is only possible for the examples with labels. Supervised learning will be treated in Chapter 6.

3.4.2 Interaction

Note that trial and error (or *interaction*) is an essential aspect of learning by reinforcement. Like supervised learning, reinforcement learning is a kind of machine learning. But unlike supervised learning, in reinforcement learning there is no

Chapter	Name	MDP-Tuple	Reinforcement Learning
Ch 4	alpha-beta	$(S, A, 1, R, 1)$	policy, backup
Ch 5	MCTS	$(S, A, P, R, 1)$	pol, b/u, exploration/exploitation
Ch 6	DQN	(S, A, P, R, γ)	pol, b/u, expl/expl, discount, off-policy
Ch 7	Self-Play	(S, A, P, R, γ)	pol, b/u, expl/expl, discount, off-pol, self-play

Table 3.1: MDP Tuple and Reinforcement Learning in the Chapters

pre-existing database with labeled states. All states to learn from will be generated during the learning process. By smartly choosing the agent's actions, the sequence of states may be generated to learn efficiently, without generating a large database of examples.

3.4.3 Credit Assignment

Now, in supervised learning all examples are labeled. However, in reinforcement learning some actions may not immediately return a reward, but later rewards must be propagated to previous states. In other words, state spaces in reinforcement learning may have a *sparse* reward structure. This is the case in most games, where only terminal states have a reward $\{-1, 0, +1\}$, and for all other states the reward value must be propagated backward from the terminal states. This is known as the credit assignment problem. Long range credit assignment is a challenging problem. This too will be discussed in Chapter 6.

3.4.4 Dependency

There are more differences between supervised learning and reinforcement learning. Supervised learning typically learns a classifier that maximizes matching, reinforcement learning learns a policy that maximizes reward. Supervised learning has a database from which it draws examples to learn from, reinforcement learning generates its own learning examples through interaction with the environment. In reinforcement learning there is therefore a dependence since the learning algorithm influences its own learning examples, which may cause learning anomalies, cycles, and local maxima. Special care must be taken in reinforcement learning to not get stuck in local maxima, as we will see in Chapter 6.

We are approaching the end of this chapter. Let us look at how the principles that were covered apply to the different chapters in this book.

3.4.5 MDP tuple and Reinforcement Learning

Reinforcement learning can formally be described by the 5-tuple of Markov decision processes. The remainder of this book will discuss many different reinforcement learning methods in depth, from heuristic planning, adaptive sampling, to generalization (as we saw in Table 1.1). These methods will start simple, with

some of the MDP tuples being constant at 1, in the early chapters. As the methods become more elaborate, the full tuple is used, and more reinforcement learning elements apply. Table 3.1 gives an overview of how the full generality of reinforcement learning develops throughout the chapters, as the methods progress from simple search to advanced generalization methods with adaptive self-learning.

3.5 Practice

Below are some questions to check your understanding of this chapter. Each question is a closed question where a simple, one sentence answer is possible.

Questions

1. Name three important elements of intelligence.
2. What element of intelligence does computer game playing focus on?
3. Describe the symbolic approach to AI. Describe the connectionist approach to AI.
4. Who wrote the first Chess program?
5. When did Deep Blue beat Kasparov, in 1996 in Philadelphia or in 1997 in New York?
6. Describe the difference between strategy and tactics in a board game. Give an example game of each.
7. Describe the reinforcement learning model.
8. Give the 5-tuple of a Markov decision process.
9. What is a value function?
10. What is a policy function?
11. Describe in words the intuition of Bellman's equation.
12. Describe the exploration/exploitation dilemma. Can you give one simple algorithm for trading off exploration and exploitation?
13. Describe in words the intuition behind temporal difference learning.
14. Why is it sometimes necessary to approximate a solution?
15. What is the difference between on-policy learning and off-policy learning? Can you give one example algorithm for each?
16. Why is sample efficiency important?

We will now have a deeper look into how on-policy and off-policy algorithms behave. First we look at the algorithms, then we will use the Taxi world for practice.

3.5.1 Taxi

We will now look at on-policy and off-policy algorithms and at their behavior.

How do on-policy learning and off-policy learning learn optimal policies for the Taxi environment? We will use SARSA as the on-policy learner, and Q-learning as the off-policy learner.

As we saw in the previous chapter, benchmarks are of great importance for progress in AI. To support AI progress, the organization OpenAI has provided an easy to use suite of benchmarks for scientists and students to use. It is called Gym, and has a Python interface. One of the classic examples in reinforcement learning is the Taxi problem, introduced by Tom Dietterich [173]. We will use the Taxi example from OpenAI Gym.

Python is a programming language that is popular in artificial intelligence. It supports quick development, rich and flexible data structures, and many third party packages ranging from numerical simulation, graphics to machine learning. If Python is not present on your computer, please go to Appendix B to learn more about Python and how to install it on your computer.

Now please go to the webpage and the Github page of OpenAI Gym and to have a look around to see what is there. OpenAI Gym can be found here.² The Gym Github page can be found here.³ You will see different sets of environments, from easy to advanced. There are the classics, such a Cartpole and Mountain Car. There are also small text environments. Taxi is there, and there is the Atari Learning Environment [55], that was used in the paper that introduced DQN [453]. MuJoCo is also available, an environment for experimentation with simulated robotics. (Our Taxi environment can be found in `gym/envs/toy_text`, in case you would like to have a look at how the environment is written.) Installing Gym on your computer is easy. Type `pip3 install gym` (or `pip install gym`) at a command prompt.

The Taxi example (see Figure 3.8) is an environment where taxis move up, down, left, and right, and can pickup and dropoff passengers. The Gym documentation describes the Taxi world as follows. There are four designated locations in the grid world indicated by R(ed), B(lue), G(reen), and Y(ellow). When the episode starts, the taxi starts off at a random square and the passenger is at a random location. The taxi drives to the passenger's location, picks up the passenger, drives to the passenger's destination (another one of the four specified locations) and then drops off the passenger. Once the passenger is dropped off, the episode ends.

There are 500 discrete states since there are 25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is in the taxi), and 4 destination locations $25 \times 5 \times 4$.

The environment returns a new result-tuple at each step. There are 6 discrete deterministic actions for the Taxi driver:

0: move south

²<https://gym.openai.com>

³<https://github.com/openai/gym>

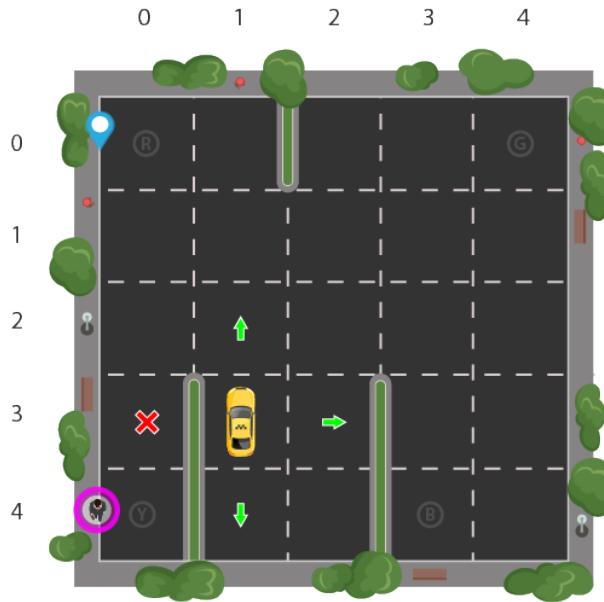


Figure 3.8: Taxi World [339]

- 1: move north
- 2: move east
- 3: move west
- 4: pickup passenger
- 5: dropoff passenger

There is a reward of -1 for each action and an additional reward of +20 for delivering the passenger. There is a reward of -10 for executing actions *pickup* and *dropoff* illegally.

To use the Gym environment, start the Python interpreter `python3` and import `gym` and `numpy`. Please refer to Listing 3.3.

SARSA

SARSA is the on-policy learning algorithm that we will use first. Its name comes from the tuple (s, a, r', s', a') that determines the next value of the $Q(s, a)$ -function. We recall the on-policy update formula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

```

# Q learning for OpenAI Gym Taxi environment
import gym
import numpy as np
import random
#Environment Setup
env = gym.make("Taxi-v2")
env.reset()
env.render()
# Q[state,action] table implementation
Q = np.zeros([env.observation_space.n, env.action_space.n])
gamma = 0.7 # discount factor
alpha = 0.2 # learning rate
epsilon = 0.1 # epsilon greedy
for episode in range(1000):
    done = False
    total_reward = 0
    current_state = env.reset()
    if random.uniform(0, 1) < epsilon:
        current_action = env.action_space.sample() # Explore state space
    else:
        current_action = np.argmax(Q[current_state]) # Exploit learned values
    while not done:
        next_state, reward, done, info = env.step(current_action) # invoke Gym
        if random.uniform(0, 1) < epsilon:
            next_action = env.action_space.sample() # Explore state space
        else:
            next_action = np.argmax(Q[next_state]) # Exploit learned values
        sarsa_value = Q[next_state, next_action]
        old_value = Q[current_state, current_action]

        new_value = old_value + alpha * (reward + gamma * sarsa_value - old_value)

        Q[current_state, current_action] = new_value
        total_reward += reward
        current_state = next_state
        current_action = next_action
    if episode % 100 == 0:
        print("Episode {} Total Reward: {}".format(episode, total_reward))

```

Listing 3.2: SARSA Taxi Example, after [339]

```

total_epochs, total_penalties = 0, 0
ep = 100
for _ in range(ep):
    state = env.reset()
    epochs, penalties, reward = 0, 0, 0
    done = False
    while not done:
        action = np.argmax(Q[state])
        state, reward, done, info = env.step(action)
        if reward == -10:
            penalties += 1
        epochs += 1
        total_penalties += penalties
        total_epochs += epochs
print(f"Results after {ep} episodes:")
print(f"Average timesteps per episode: {total_epochs / ep}")
print(f"Average penalties per episode: {total_penalties / ep}")

```

Listing 3.3: Evaluate the Optimal SARSA Taxi Result (based on [339])

Listing 3.2 shows Python code for improving a policy in the Taxi world with the SARSA algorithm, adapted from [339]. The best policy found by SARSA can now be used by following the best action values in the Q-table. We now evaluate the performance of our SARSA agent. We do not need to explore any more, since the best action is right there, in the Q-table.

Listing 3.3 shows the simple code to evaluate the SARSA Taxi policy. The number of illegal pickups/dropoffs is shown as penalty.

Q-learning

Q-learning performs off-policy learning. We recall its update formula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Listing 3.4 shows Python code for finding a policy in the Taxi world with the Q-learning algorithm.

Again, Listing 3.3 can be used to evaluate the Q-learning Taxi policy. It should be just as optimal as with SARSA. On average, Q-learning tends to find better policies than SARSA [638].

Exercises

1. Experiment with Taxi World. Download Gym, and perform the SARSA and Q-learning steps described before. Running ready-made SARSA and Q-learning code is nice, but you get a better idea how Taxi World works by tracing a few steps of the simulator by yourself. Do so and add code to allow you to print (render) each step, step by step. Watch the taxi move in its world.

```

# Q learning for OpenAI Gym Taxi environment
import gym
import numpy as np
import random
#Environment Setup
env = gym.make("Taxi-v2")
env.reset()
env.render()
# Q[state,action] table implementation
Q = np.zeros([env.observation_space.n, env.action_space.n])
gamma = 0.7 # discount factor
alpha = 0.2 # learning rate
epsilon = 0.1 # epsilon greedy
for episode in range(1000):
    done = False
    total_reward = 0
    state = env.reset()
    while not done:
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample() # Explore state space
        else:
            action = np.argmax(Q[state]) # Exploit learned values
        next_state, reward, done, info = env.step(action) # invoke Gym
        next_max = np.max(Q[next_state])
        old_value = Q[state, action]
        new_value = old_value + alpha * (reward + gamma * next_max - old_value)
        Q[state, action] = new_value
        total_reward += reward
        state = next_state
    if episode % 100 == 0:
        print("Episode {} Total Reward: {}".format(episode, total_reward))

```

Listing 3.4: Q-learning Taxi Example, after [339]

2. In Taxi World, which is better, SARSA or Q-learning? How do you know? On which criteria do you measure algorithm quality?
3. The environment is initialized randomly each time. This makes it hard to compare the two algorithms if each run is different, and it complicates debugging. Make a deterministic version of the algorithm comparison, where each starts at the same configuration.
4. How many runs do you need to do to get statistically significant results for $p < 0.05$ (if necessary, see a text on experiment design such as [36] or statistics)?
5. Experiment with different values for ϵ , α and γ . Which are better? Experiment with decaying values for ϵ , α and γ .
6. Go to Gym, and look for the Mountain Car example. Compare SARSA and Q-learning. Which learns faster, and what about the solution quality?

3.6 Summary

This has been a challenging chapter. We have introduced many abstract formal concepts from reinforcement learning, to lay the ground work and introduce the language for understanding the algorithms in the following chapters.

We started with the introduction of the concepts of agent and environment, and introduced the states, actions and rewards that are the basis of the agent-environment interaction. In reinforcement the agent learns through interaction with an environment, by trial and error. Markov Decision Processes are a powerful and popular formalism. In processes with the Markov property, the distribution of future states depends solely on the current state, not on the states preceding it, i.e., there is no memory, which simplifies the mathematical analysis of Markov processes.

Two related central elements in reinforcement learning are the value of a state and the policy: the list of actions to be taken when in a state, defining a future behavior (strategy). Policies can be implemented as arrays, functions, mappings, or lists, and all these terms are used in the literature. The Bellman equation is a central concept in reinforcement learning. It recursively defines the value of a state based on the reward of its successor states. Other important algorithms that we covered are Temporal Difference learning, SARSA and Q-learning.

Reinforcement learning algorithms typically gradually build up information about the best actions to be taken in a state. Greedy algorithms always exploit the available information by following the action with the highest expected reward. Other algorithms may explore new or under-explored actions, with a larger uncertainty, to possibly find better answers. A well-known simplistic approach is the ϵ -greedy approach, that chooses to exploit in $1 - \epsilon$ occurrences and chooses to explore in ϵ times.

On-policy learning updates state values for the current behavior policy, off-policy learning updates state values for the best policy independent of the behavior policy it follows. Q-learning is a model-free off-policy method.

We discussed exact and approximate methods. Exact methods are suitable for small state spaces, approximate methods are used when the state space is very large. In fact, the state space may be so large that almost all of the states that we see are new, and can not have been trained on. Approximation methods must be able to generalize well. Lookup tables are well-known exact data structures, neural networks are well-known approximation data structures.

In the next chapter, Chapter 4, we will discuss exact search methods for determining the value of the state.

In Chapter 6 DQN, or deep Q-networks, are discussed. DQN is a reinforcement learning method for function approximation.

Historical and Bibliographical Notes

Game playing programs solve reinforcement learning problems. Reinforcement Learning is a large and rapidly expanding field. This chapter has only introduced the bare minimum. Two excellent comprehensive treatments of the field are the books by Sutton and Barto [638] (see Figure 3.9) and Bertsekas and Tsitsiklis [74]. Sutton and Barto divide reinforcement learning into exact tabular methods and approximate methods. This distinction maps nicely to Kahneman’s categories of thinking Slow and thinking Fast, and Shannon and Turing’s search-eval architecture.

Insightful and entertaining accounts have been written about the human and scientific side of writing world champion level game playing programs. Schaeffer’s account of how he and his team created Chinook, is memorable, not just because of his side of the story, but also because of the picture that it paints of Marion Tinsley, the amazing Checkers champion [569].

Much has been written about the Chess match between Deep Blue and Garry Kasparov. Feng-hsiung Hsu’s account gives the story behind their “quest to build the mother of all Chess machines.” It is an awe-inspiring story of this moment in the history of Chess-playing intelligence versus computer programming intelligence [302].

A good reference on how computers play Chess is [405], and Chapter 4 covers many of the techniques that are used in Chess programs.

Q-learning was introduced by Watkins [721]. Kaelbling provides a widely cited review of reinforcement learning [334]. Kaelbling has also contributed greatly to robotics, planning, and hierarchical reinforcement learning (see Section 7.5.4). She received the 1997 IJCAI Computers and Thought award. See Figure 3.10. For a review of reinforcement learning in robotics see [357].

A popular lecture series on reinforcement learning is by David Silver here.⁴

⁴<http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html>



Figure 3.9: Richard Sutton and Andrew Barto



Figure 3.10: Leslie Kaelbling

Chapter 4

Heuristic Planning

Combinatorial games are used in AI to study reasoning and decision making. The main challenge is how to search very large state spaces efficiently. The earliest method for efficient decision making is heuristic planning. It is a model-based method. Heuristic planning uses a human-inspired approach: it is believed that when playing a game, most human players try (1) to look ahead a few moves, and then (2) to see if they like the situation that they arrive at. Heuristic planning formalizes this concept, using a search function and a heuristic evaluation function.

A heuristic is a domain specific rule of thumb. As a rule of thumb, it works most of the time, but not always. Being domain specific, it is not general, it works only in certain games, often exploiting a game-specific feature. Scientists, who strive for general methods, have a love-hate relationship with heuristics. They are often too successful to ignore for a specific problem, but the domain specificity limits their general applicability. Much of the efforts for generalization and feature discovery that have been so successful in deep learning (see Chapter 6) are driven by the desire to transcend beyond heuristics.

A first implementation of the look ahead idea is the minimax algorithm. Minimax has been highly successful, especially since many more or less general enhancements have been developed over the years. We will cover some of these enhancements in depth, such as alpha-beta, iterative deepening, and transposition tables.

Heuristic planning has been quite successful in highly tactical games, such as Chess, Checkers and Othello. Planning and heuristics are basic concepts of AI. Future chapters will introduce different paradigms, but often elements of good old planning and heuristics will appear.

Core Problem

- How to search a large state space efficiently?

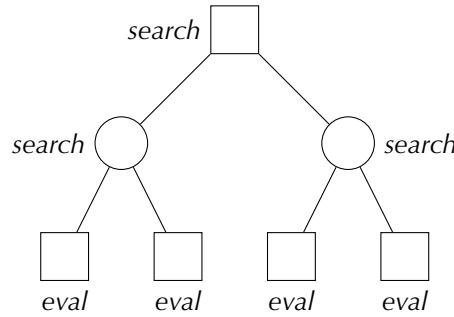


Figure 4.1: Search-Eval Architecture in a Tree

- How to spend search effort in promising areas of the state space?

Core Concepts

- Search function
- Heuristic evaluation function
- Critical tree, alpha-beta pruning, move ordering

First we will discuss the basis of heuristic planning: the search-eval architecture. Then we delve deeper into computing the size of the state space, and discuss basic search and evaluation functions. We continue with an analysis of the minimal part of the state space that must be traversed to find the optimal policy. Finally, we discuss search enhancements, such as alpha-beta and transposition tables, and evaluation function enhancements, such as end game databases.

4.1 The Search-Eval Architecture

Let us start with the basic architecture of heuristic planning. Traditional Chess-style game playing programs consist of both a search function and an evaluation function. Together these two functions form the search-eval architecture. The transition function P is known in these games, all successor states can be generated in each board position by following the game rules. Finding the optimal policy is done by generating all successor states and performing a look ahead search. Searching the state space exhaustively is infeasible.¹ Therefore, after the

¹Full-width look ahead search takes time exponential in the search depth. In Chess there are on average 35 actions in each state, and a typical game lasts 40 moves, 2 moves by each side. Searching to depth 10 would already mean searching a tree with $35^{10} = 2758547353515625$ leaves. If each leaf evaluation would take an impossibly quick nanosecond, then evaluating this tree would take more than 31 days, which would be slow even for correspondence Chess.

initial state has been searched to a certain search depth, the evaluation function is called. See Figure 4.1 for an illustration. The Figure shows the root node at the top. The root represents the initial state s_0 , the initial board position for which we wish to find the optimal policy and the value. One level deeper we see two inner nodes, the children of the root, and at depth-2 we see four leaf nodes, that are not searched further, but where the evaluation function is called to determine a heuristic leaf value. In the square nodes it is the player's turn to move, and in the circle nodes the opponent's.

The search-eval architecture is a simple architecture. It consists of two functions: a search function and an evaluation function. The search function traverses the states one by one. It is an exact function. The evaluation function returns an approximation value of the state on which it is called.

Note that the introduction of a heuristic evaluation function changes the range of values of the value function. The values for the full state space are determined by the terminal states, whose value can be $\{-1, 0, +1\}$ for win, draw, and loss (viewed from the first player). A heuristic returns more values, typically in a more fine grained range such as $[-16000, +16000]$. Positive values indicate an advantage for the first player, negative values indicate a disadvantage. Section 4.1.3 provides more details.

Exact Search and Approximate Eval

Finding the optimal policy in a game of skill requires searching through state spaces. For small state spaces methods that traverse the states one by one are suitable. They give precise answers, but are unsuitable for large state spaces, since traversal would take too long. The alternative, approximation methods, do not traverse the state space at all. Instead, they approximate the value of a single state directly. Approximation methods use combinations of *features* of the state. Examples of features are heuristics such as material balance—which side has more and better pieces—or mobility (see Section 4.1.3). In later chapters we will encounter other approximation methods, such as sampling in Chapter 5 and generalization in neural nets in Chapter 6.

Exact methods are precise but slow. Approximation methods are fast but imprecise. The search-eval architecture provides a framework for the two approaches to work together and achieve the best of both worlds: exact search methods traverse a part of the state space, calling the approximation methods to evaluate states at their search horizon. Together they find a quality approximation of the value function without taking too long.

The goal of exact and approximate methods is the same: to determine the value function (and the optimal policy). Researchers have developed search and eval functions that together are strong enough to achieve a level of play that beats the best humans in Chess and Checkers. The first ideas for the search-eval architecture go back to Shannon [600] and Turing [679].

One could ask the question why search has to be used, what it is that the heuristics miss that search has to be used at all? The reason that the search function is necessary is that the evaluation function misses game dynamics. An evaluation

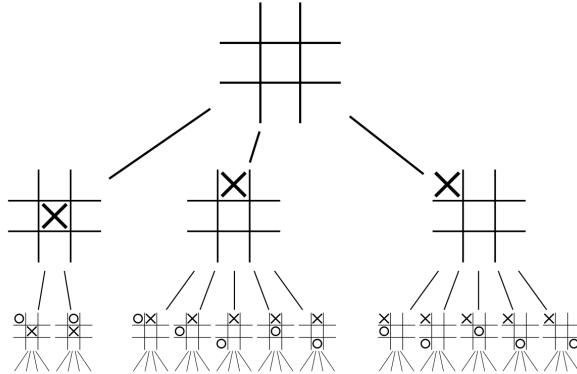


Figure 4.2: Tic Tac Toe Game Tree (part)

function provides a static assessment of a state, but cannot foresee the effects of dynamics caused by captures and other tactical play. Evaluation functions only work well in quiet (stable) positions (see also quiescence search in Section 4.4.2). The exercises at the end of this chapter will provide an opportunity to experiment with the search and the evaluation function to develop an understanding of the search-eval interplay.

On the other hand, the evaluation function is necessary because the search space is too large to search completely. Let us now look in more detail at the size of the state space.

4.1.1 State Space

Searching for the best action of a state is hard because the state space of possible successors (and successors of successors of ...) is so large. Let us try to see how large it is. An often used measure to bound the size of the state space is to calculate how many legal states would be traversed if all successors are evaluated. In general, finding the exact state space size of games is a surprisingly challenging problem, that has attracted many interesting research papers [600, 570, 85, 5, 221, 317, 301, 314, 676].

For small games, such as Tic Tac Toe, the state space is small enough to enumerate with a computer. See Fig. 4.2 for an impression of a depth-2 tree of how the possible states are enumerated. To compute the size of the Tic Tac Toe state space, we note that there are 9 squares on the board that can be either empty, cross, or circle. Thus there are $3 \times 3 \times \dots \times 3 = 3^9 = 19,683$ possible board configurations. However, this is an over-estimation of the reachable state space, since it contains unreachable and illegal positions. When we remove illegal positions we come to 5,478 states. When we remove symmetric positions (rotations and reflections) there are only 765 positions left [566].

```
# a small tree, to be searched by the minimax algorithm
leaf_node1 = {'type': 'LEAF', 'value': 1, 'material': 3}
leaf_node2 = {'type': 'LEAF', 'value': 2, 'material': 4}
leaf_node3 = {'type': 'LEAF', 'value': 3, 'material': 2}
leaf_node4 = {'type': 'LEAF', 'value': 4, 'material': 9}
leaf_node5 = {'type': 'LEAF', 'value': 5, 'material': 1}
leaf_node6 = {'type': 'LEAF', 'value': 6, 'material': 7}
min_node1 = {'type': 'MIN', 'children': [leaf_node6, leaf_node1, leaf_node3], 'material': 2}
min_node2 = {'type': 'MIN', 'children': [leaf_node3, leaf_node4, leaf_node2], 'material': 7}
min_node3 = {'type': 'MIN', 'children': [leaf_node1, leaf_node6, leaf_node5], 'material': 1}
max_node1 = {'type': 'MAX', 'children': [min_node1, min_node2, min_node3], 'material': 3}

root = max_node1
```

Listing 4.1: Small Tree code

Beyond Tic Tac Toe, for more challenging games such as Chess or Go, calculating the precise number of legal and reachable game positions is harder. For Chess the size of the state space is often approximated as 10^{47} and for Go on the standard 19×19 board this number is 10^{170} [5, 711, 676].²

Note that the state space size is different from the game tree size. The former is the game *graph*, the latter is the game *tree*. The same position from the graph can occur in many different lines of play that constitute the tree. These shared positions are called transpositions. The difference between the game graph and the game tree is large [600, 5, 519, 519]. For Tic Tac Toe an upper bound for the former is $3^9 = 19,683$ (pieces on squares), for the latter it is $9! = 362,880$ (different full game sequences of moves).

An essential element of Chess and Checkers programs is the transposition table, a cache of positions, preventing the generation of states that have already been searched before. In Section 4.3.3 we will go deeper into transposition tables and their advantages.

Code

Listing 4.1 gives an example in Python code of a very small tree, with one max node (the root), three min nodes, and 9 leaves. This code is meant to be easy to understand, clearly more efficient ways of coding a tree are possible. In a short while we will provide an algorithm to compute the value of the tree by looking at the leaf values, and later on by looking at the material. In Figure 4.4 this tree is shown. The values at the leaves (6,1,3,3,...) stand for the heuristic value of the leaves, 6 is better than 3. The values themselves have no special meaning, they were chosen for explanatory purposes.

²John Tromp has published in 2016 that the exact number of legal Go states is 208168199 381979984 69947863 33448627 7028652 24538845 3054842 56394568 209274196 127380153 78525648 45169851 9643907 25991601 5628128 54608988 831442712 971531931 75577366 20397247 064840935. Tromp used advanced combinatorial methods in his calculations [675].

```

INF = 99999

def eval(n):
    if n['type'] == 'LEAF':
        return n['value']
    else:
        error("Calling eval not on LEAF")

def minimax(n):
    if n['type'] == 'LEAF':
        return eval(n)
    elif n['type'] == 'MAX':
        g = -INF
        for c in n['children']:
            g = max(g, minimax(c))
    elif n['type'] == 'MIN':
        g = INF
        for c in n['children']:
            g = min(g, minimax(c))
    else:
        error("Wrong node type")
    return g

print("Minimax value: ", minimax(root))

```

Listing 4.2: Minimax code

4.1.2 Search: Depth-First

How can we determine if a position is won, lost, or drawn, and how can we find the best move that leads to this outcome? In a two player zero sum game players are always assumed to choose moves that lead to the best successor positions. Let us assume that all values of states in the tree are viewed from the first player. If node n represents the position of which we wish to determine the value, then the value of node n is the maximum of the value of its children C_n . The value of these child nodes is determined analogously, although here the moves are made by our opponent, who chooses positions that will minimize the outcome for us. Next it is our move again, and we choose the position that maximizes our winning probability. This alternating of maximizing and minimizing is the reason the procedure is called *minimax*.³

In Figure 4.3 a Tic Tac Toe tree is drawn with values. Figure 4.4 shows a more abstract minimax tree, where nodes are drawn as squares and circles. Maximizing nodes (where it is our turn) are drawn as squares, minimizing nodes (where it is our opponent's turn) as circles. By following the values we can see how the values of the nodes are determined.

Listing 4.2 gives example code for the minimax algorithm. The minimax func-

³Note that minimaxing is a kind of self-play. Since our opponent uses the same algorithm as we do, in minimax we play against ourselves, our opponent is just as smart as we are. This kind of minimax self-play is different from Chapter 7, where a self-play loop is used to train a neural network evaluation function to learn to play a game from scratch.

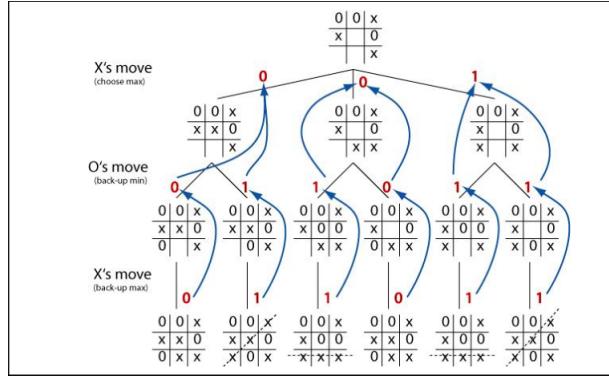


Figure 4.3: Tic Tac Toe Game Tree with values

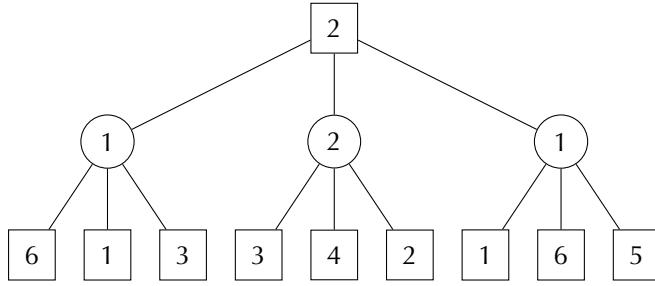


Figure 4.4: Minimax Tree of Listing 4.2

tion will determine the value of a node by taking the minimum of the value of the children of the min nodes, and the maximum of the values of the children of the max nodes. In the case of the tree in Listing 4.1, the value of nodes `min_node1` and `min_node3` is both 1, and the value of `min_node2` is 2. The maximum of 1 and 2 is 2, so the minimax value of the root is 2.

Search Tree

The minimax procedure recursively traverses the entire game tree (Listing 4.2). Minimax is a *backtracking* procedure. It starts at the root, and does a depth-first traversal of the tree by calling itself recursively for each child node. When a child has been searched, it backtracks up (returning its value) and searches down the next child, going down-up-down-up through the tree. Minimax is a trial and error procedure, as all reinforcement learning procedures are. The trial element is trivial: try all successors. The error part is the backup rule.

The nodes that are actually expanded during a search procedure are referred to as the *search tree*. The tree version of the full state space is also known as the

```

INF = 99999

def heuristic_eval(n):
    return n['material']

def minimax(n, d):
    if d <= 0:
        return heuristic_eval(n)
    elif n['type'] == 'MAX':
        g = -INF
        for c in n['children']:
            g = max(g, minimax(c, d-1))
    elif n['type'] == 'MIN':
        g = INF
        for c in n['children']:
            g = min(g, minimax(c, d-1))
    else:
        error("Wrong node type")
    return g

print("Minimax value:", minimax(root, 2))

```

Listing 4.3: Depth Limited Minimax

game tree. For the minimax procedure the search tree is identical to the game tree. In the rest of this book we will see many examples of procedures whose search tree is significantly smaller than the game tree. Minimax does a full-depth full-width expansion of the game tree. For regular trees, where all nodes have the same number of children, the number of leaves of the game tree is w^d , where w is the width of the nodes (the number of children of a node) and d is the depth of the tree. For the tree in Figure 4.4 the width is 3 and the depth is 2, so the number of leaves is $3^2 = 9$. The size of the game tree is exponential in the depth parameter, and is dominated by the number of leaves. A useful approximation of the size therefore is w^d , and so is the running time of the minimax procedure, since it visits every node.

Now that we have looked at the search function, it is time to look at the heuristic evaluation function.

4.1.3 Eval: Heuristic Features

Calling the minimax function on a Chess position, whose game tree is of size 10^{47} would take too long to compute its value.⁴ In order to reduce the size of the search tree, the search depth can be reduced.⁵ To do so, even the earliest Chess and Checkers programs (Turing, Samuel) used a heuristic evaluation function as an approximation of the true value of a position. The heuristic is based on domain

⁴Even if node evaluation would take a very fast one nanosecond, it would still take 10^{38} seconds, or 3×10^{30} years, which is about 10^{20} times the estimated age of the known universe.

⁵Also the width of the tree can be reduced. With forward pruning fewer children are expanded at inner nodes. See Section 4.3.5.

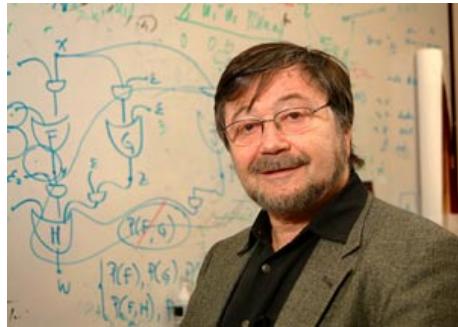


Figure 4.5: Judea Pearl

knowledge, provided by domain experts. In Chess and Checkers, the most important heuristic is material balance (the number and the importance of the pieces on the board).

Instead of traversing the tree to the full depth where the game ends due to an end game position, the heuristics create artificial leaves, at a much shallower search depth. The deeper search is artificially limited, and a heuristic is called to statically evaluate whatever position happens to occur at that depth.

Listing 4.3 adds depth-limiting to our Python code of minimax. New is parameter `d` which is decremented in the recursive call of the function, and new is that the `eval` function no longer uses the score at the leaves, but takes a heuristic value of the node whenever the depth parameter happens to be 0. The heuristic `eval` in the example code is quite straightforward, it uses the `material` value of the node (see Listing 4.1).

A heuristic is a function that encodes domain specific knowledge, and can be computed efficiently. For example, in Chess, having two pawns is better than having one pawn, and the Queen is more valuable than the Rook. For many years heuristic evaluation functions have been a central piece of combinatorial search. In fact, heuristics have traditionally been so important that the name of the field has been called heuristic planning. (See, for example, Judea Pearl's seminal work [504]. Judea Pearl (Figure 4.5) would win the 2011 Turing award for his work on probabilistic reasoning.)

A heuristic evaluation function in general takes the form:

$$h(s) = c_1 \times f_1(s) + c_2 \times f_2(s) + c_3 \times f_3(s) + \dots + c_n \times f_n(s)$$

where $h(s)$ is the heuristic function of the state, $f_i(s)$ are the feature terms such as material balance, mobility or center control, and c_i are the coefficients, the weights that signal the importance that a feature has in the total evaluation function.

In Chess and Checkers material balance is the dominant factor in the strength of a position. For this reason, the heuristic evaluation function of Listing 4.3 uses the `material` field of the nodes defined in Listing 4.1.

Heuristic evaluation functions are linear combinations of features. A typical Chess evaluation function consists of the following features: material, mobility, king safety, center control. The evaluation function is static (it applies to a single position). Tactical aspects such as captures are taken care of by the search. To search as many positions as possible speed of evaluation is important. For that reason heuristics are often optimized for speed, not for accuracy.

Heuristic evaluations are, in almost all games, crude approximations of the value of the states. They capture static elements of a position, and lack insight into the dynamics. Achieving good play requires either extra work on heuristics or on capturing dynamics with search algorithms. In the reinforcement learning theory the two approaches (exact planning and function approximation) are often treated as mutually exclusive approaches. In most board games, however, they are studied together out of necessity, due to the low quality of play of heuristic evaluation functions without search, and conversely the low quality of play of search without heuristic evaluation functions.

In tournament Chess and Checkers programs the heuristic evaluation function is a highly advanced piece of code, using many features. The features are typically manually chosen, and manually optimized and tuned. This feature engineering is quite labor intensive, and making a complicated piece of code do exactly the right thing is hard. The search space is large and finding the precise state at which unwanted behavior occurs may be difficult. It may even be unclear if solving a problem in one part of the play does not introduce a weakness in another part. To achieve world championship level, the teams behind Chinook and Deep Blue spent years of refining their heuristics.

A further downside of a heuristic is, of course, that it is domain specific, e.g., a piece of Chess code that assigns the queen 900 points is of no use in Checkers, which does not have queens.

Today, many Chess programs are open source, including some of the top programs, such as Stockfish [542].⁶ To get an idea of just how extensive current state of the art heuristic evaluation function have become, you may want to have a look at the Stockfish source at GitHub, which can be found here.⁷

Stockfish has a conventional evaluation function with manually designed features and manually tuned weights. In other games evaluation functions based on machine learning have been successful, such as Othello (Logistello [113]) and Backgammon (Neurogammon [651]) and of course Go. Despite some effort to automate tuning of weights by Samuel [560] and the Deep Blue team [302], in Chess and Checkers most programs use manual evaluation functions. Some papers describe success in evolving coefficients of manual features [158], and in AlphaGo both the features themselves and the coefficients are learned [604, 607, 606]. (Table 7.4 in Section 7.4.4 lists game-playing programs that use machine learning.)

Machine learning evaluation functions have the advantage of being more general and easier to debug. At the end of this chapter we will revisit machine learn-

⁶The name Stockfish may seem strange for a Chess program. The authors Romstad and Costalba state that is reflects their two countries : “produced in Norway and cooked in Italy.”

⁷<https://github.com/official-stockfish/Stockfish>

ing evaluation enhancements. But first, we will explore in more depth how the minimax function works.

4.2 State Space Complexity

The state space of two-agent zero sum perfect information games has a minimax structure. This structure follows directly from the fact that the value of a state is determined by the value of the best successor state. The minimax state space is redundant; only a small part of the state space (the optimal policy) defines the value of the initial state. We will now do a theoretical analysis to find the optimal policy. This analysis will show us that there is an exponential amount of redundancy in the tree, which allows us to create a much more efficient search algorithm.

Width and Depth

Let us start with a regular fixed-width fixed-depth tree. The size of this tree is dominated by the number of leaf nodes, which is w^d for width w and depth d . In Figure 4.4 we see a tree with $w = 3$ and $d = 2$ (the root node is depth 0).

Critical Path

Figure 4.6 shows a different minimax tree, $w = 2$, $d = 3$, with leaf values, and values at the inner nodes. Max nodes are squares, min nodes are circles. Recall that the leaf values are typically integer values from a heuristic score, for example in range [-16000, +16000]. As usual, the root is a max node, its value is the maximum of its child values, and the children of the root are min nodes, whose value is the minimum of their child values, as can be seen in the example tree. The path of bold, italic number 6s is called the *critical path*, the path of nodes whose value is equal to the value of the root.

solution tree	minimax value	# children at MAX node	# children at MIN node
Max	upper bound	all	≥ 1
Min	lower bound	≥ 1	all

Table 4.1: Solution Trees and Bounds

Critical / Solution Trees

$$\text{Critical Tree} = \text{Max solution tree} \cup \text{Min solution tree}$$

$$\text{Critical Path} = \text{Max solution tree} \cap \text{Min solution tree}$$

Table 4.2: Critical Tree and Path

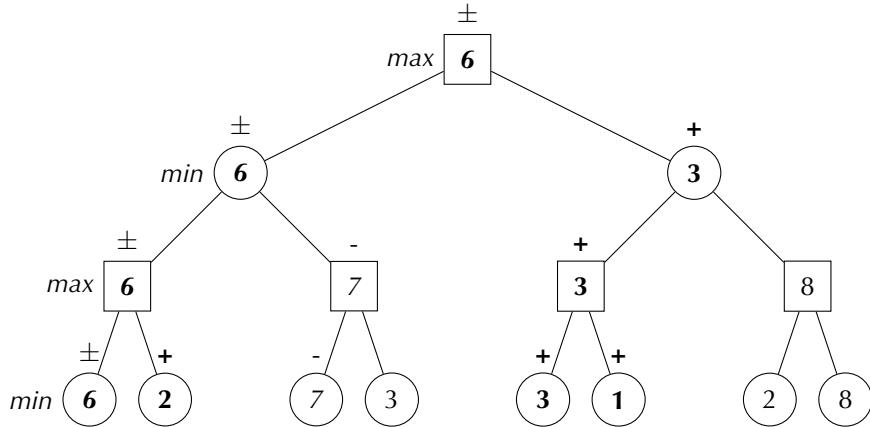


Figure 4.6: \pm **Critical Path**, $+$ **Max Solution Tree**, $-$ **Min Solution Tree**

Critical Tree

We will now analyse the lower bound of the size of the tree that needs to be expanded.

As we shall see, not all nodes in the minimax tree play a role in determining the value at the root. In fact, only a small fraction of the tree nodes do so. Let us try to construct the smallest possible sub-tree that defines the minimax value. By the definition of the maximum function that takes the highest of its inputs, at max nodes, the highest child determines the value of the parent node. All other children are unnecessary, they are non-critical nodes that are “dominated” by their critical sibling. Likewise, at min nodes, only the lowest-valued child determines the value of the node. So, it would seem that we only need to expand a single child to know the value of a node. There is a complication, however. If we only expand one child, then we do not know for certain what the value of a node is. Even if it is the critical child, since we have only seen one child so far we do not yet know if there is another, better child. At a max parent, the single expanded child creates a *lower bound* on the minimax value, since expansions of subsequent children can only increase the value upwards. Likewise, a min node whose children have been partly expanded has an upper bound on its final value, since subsequent expansion can only lower the value further.

A partly expanded max node defines a lower bound on the minimax value. Lower bounds of *min* nodes, however, require all children of that *min* node to be expanded (since, if it were partly expanded, then expansion of another child could lower the value further, and the lower bound evidently was not a lower bound).

Likewise, a partly expanded min node is an upper bound on the min node value. Upper bounds of *max* nodes, however, require all children of that *max* node to be expanded.

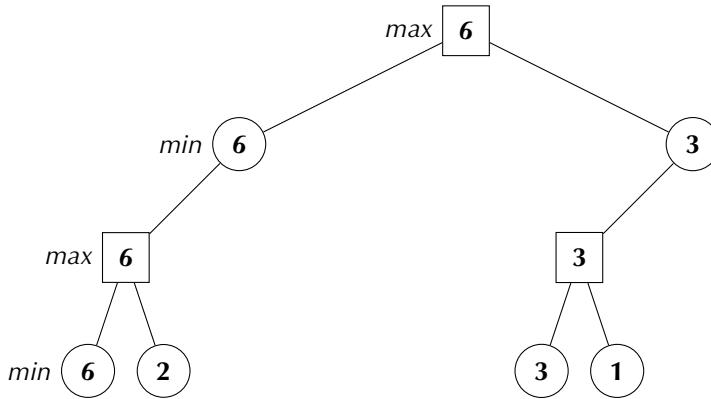


Figure 4.7: Max Solution Tree: Upper bound determined by all children at max nodes, a single child at min nodes

Now we have enough information to construct the smallest tree that defines the minimax value. If we wish to have an upper bound at the root node, then we need a tree where at least *one* child of the min nodes is present, and *all* children of all max nodes are present. Figure 4.7 shows such a tree. This tree is called a max solution tree in the literature, the tree determining an upper bound [162, 515].

Conversely, if we wish to have a lower bound at the root node, then we have to construct a tree where at least one child of all max nodes is present, and all children of all min nodes. Figure 4.8 shows such a tree, which is called a min solution tree, a tree determining a lower bound in a minimax tree. (See Table 4.1.)

The minimax value of a position is, of course, defined by two solution trees of equal value, one min solution tree defining a lower bound of the same value as the upper bound defined by its max solution tree. The intersection of the min solution tree and the max solution tree is the critical path. Figure 4.6 contains such an intersection of a min solution tree and a max solution tree. The union of the two solution trees is called the critical tree, or the proof tree (see Table 4.2). It is the smallest tree that proves the value of the position.

The critical tree is the optimal policy. An optimal algorithm would search not more than this critical tree. The critical tree proves that the minimax value is not less than the value of the min solution tree, and not larger than the value of the max solution tree. Every algorithm that wishes to find the minimax value, has to traverse at least this critical tree, otherwise it has not seen enough of the tree to determine its value for certain [355].

The size (or rather, the number of leaf nodes) of a max rooted max solution tree is $w^{\lceil d/2 \rceil}$, or the square root of the size of the minimax tree. The number of leaves of a max rooted min solution tree is $w^{\lfloor d/2 \rfloor}$.

Likewise, the number of leaves of the critical tree (the union of the two solution trees) is $w^{\lceil d/2 \rceil} + w^{\lfloor d/2 \rfloor} - 1$, the square root of the size of the minimax tree, $\sqrt{w^d}$. Thus, the size of the critical tree that defines the minimax value is the square root

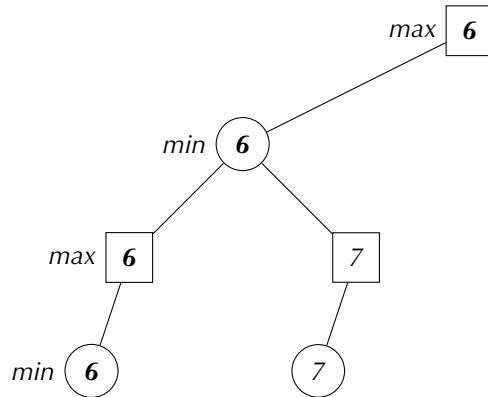


Figure 4.8: Min Solution Tree: Lower bound determined by all children at min nodes, a single child at max nodes

of the size of the minimax tree.

In order to determine the value of a minimax tree with 1000000 leaves, at least 1000 leaves have to be examined. In the same time that minimax would search to depth 5, an optimal algorithm would search to depth 10, double the depth.

In Listing 4.2 we have seen an $O(w^d)$ algorithm for traversing the tree and finding the minimax value.

Conclusion

We have now looked in depth at the part of the state space that defines the minimax value. It is now time to look at enhancements to the basic minimax algorithm.

In Section 4.3.1 we will describe alpha-beta, a best-case $\sqrt{w^d}$ algorithm for traversing the tree and finding the minimax value.

4.3 Search Enhancements

The previous sections provided us with the basic building blocks for heuristic planning in games, with the search-eval architecture and the minimax algorithm. Minimax is a fixed-width fixed-depth algorithm that is too inefficient to achieve high performance. The heuristic function is domain dependent. In the enhancements in this section, we will see a trend towards variable-depth search and generalization of the evaluation function (foreshadowing a bridge to the next chapter on adaptive sampling).

An important role in search enhancements is played by the alpha-beta algorithm, which introduced a powerful exponential pruning mechanism.

Table 4.3 gives an overview of the search enhancements that are presented in this section, to help you find your way in the myriad of terms and approaches.

Name	Principle	Applicability	Effectiveness
alpha-beta	backward pruning	all minimax games	$w^d \rightarrow \sqrt{w^d}$
iterative deepening	iterate search depth	with TT: move ordering	helps alpha-beta
transposition table	cache of states	with ID: move ordering	helps alpha-beta
null-window	alpha-beta	all minimax games	more cut offs
forward pruning	variable width	domain specific	variable
capture moves	ordering	games with captures	good
killer moves, HH	ordering	domain specific	good
backup rule	probability	imperfect info	imperfect info feasible

Table 4.3: Search Enhancements

The effectiveness of the approaches depends to such an extent on the domain that it is hard to quantify, hence the qualitative and vague, terms. The enhancements are presented in the order shown in the Table. Most search enhancements are related to alpha-beta, and improve its performance.

Heuristic positional evaluations are good at evaluating static positions. When a position contains tactics (a large difference in material balance between parent and child position, i.e., a capture) static heuristics miss this. (Note that heuristics are optimized for speed, they are just like Kahneman's System 1, they are like reflexes. All reasoning/looking ahead is handled by the search.) In order to make sure that static heuristic evaluators are only applied to static positions, a number of techniques for variable depth search have been developed that we will discuss.

Section 4.3.5 covers heuristic ways to improve pruning, and Section 4.4 covers heuristic feature evaluation. The alpha-beta algorithm is the core of decades of high efficiency game playing search. The difference between the best case and the worst case performance of alpha-beta is large, $\sqrt{w^d}$ versus w^d . This best case is achieved on well ordered trees, where the best successor positions are the first that are expanded. On well ordered trees alpha-beta can find cut offs. (On badly ordered trees it performs as bad as minimax.) Many of the search enhancements that we will now discuss have been developed to improve the successor ordering, in order to improve the performance of alpha-beta.

Let us now start with the alpha-beta algorithm.

4.3.1 Alpha-Beta

Listing 4.4 shows the alpha-beta algorithm. Let us see how alpha-beta works, to understand how it finds the true minimax value without searching the entirety of the tree.

Alpha-Beta Window

Alpha-beta maintains a *window* on the value of the position. The window consists of two bounds, named α and β . The α and β bounds together form lower and

```

INF = 99999

def eval(n):
    if n['type'] == 'LEAF':
        return n['value']
    else:
        error("Calling eval not on LEAF")

def alphabeta(n, a, b):
    if n['type'] == 'LEAF':
        return eval(n)
    elif n['type'] == 'MAX':
        g = -INF
        for c in n['children']:
            g = max(g, alphabeta(c, a, b))
            a = max(a, g) # update alpha
            if g >= b:
                break # beta cutoff, a>=b
    elif n['type'] == 'MIN':
        g = INF
        for c in n['children']:
            g = min(g, alphabeta(c, a, b))
            b = min(b, g) # update beta
            if a >= g:
                break # alpha cutoff, a>=b
    else:
        error("Wrong node type")
    return g

print("Minimax value:", alphabeta(root, -INF, INF))

```

Listing 4.4: Alpha-Beta

upper bounds on the possible values of the position, or node, that the algorithm searches.

At the root node, alpha-beta starts out with an $\langle \alpha, \beta \rangle$ -window with the widest possible window, which we write as $(-\infty, +\infty)$.⁸ Alpha-beta is a recursive algorithm to find the minimax value of a node. In order to find this value of a node, it calls itself recursively, to find the values of the children of the node in order to compute its own value. Each call to alpha-beta uses the latest (tightest) version of the window. The window for recursive calls is updated with results of the child nodes. When, at a node n , the window collapses (the bounds become equal) the sub-tree below n does not have to be explored since then the value of node n is known, and thus search of n is stopped. Alpha-beta is then said to *cut off* the search of the remaining children of n . The alpha-beta cut-off is the central mechanism to achieve the $w^d \rightarrow \sqrt{w^d}$ best-case potential.

⁸Note that ∞ is shorthand for a number outside the range of numbers returned by the evaluation function. Also note that evaluation functions are assumed to return integer values.

Alpha-Beta Example

We now look in detail at the code, to see how alpha-beta traverses the same example tree of Figure 4.9.⁹ The root r is a max node, the level below the root are two min nodes, representing the two child positions of the player. Each of these has two child positions (again max nodes) for which the opponent moves. These are leaf positions (it is a very small tree). The leaves have values 3, 6, 2, . Figure 4.9 shows an example tree and a detailed example run. Working through the example in the caption of the Figure will show you how an alpha-beta cut off works.

In this tiny example tree only one node was cut off, but in larger trees, the maximum gain of alpha-beta is to cut off $w^d - \sqrt{w^d}$ leaf nodes. (So, in a $w = 10, d = 6$ tree with 1000000 leaves, 999000 leaves can be cut off, searching only a critical tree of 1000 leaves. If minimax can see important capture moves three moves in advance, alpha-beta, in the best case, can see them six moves in advance, with the same computational effort.) Typical high performance game playing programs come close to the best case, at most of the nodes (more than 90%) the first child is also the best child [405].

Alpha-beta Cut-off

In addition to the walk-through in the Figure we will explain how alpha-beta works intuitively. The concept of *cut-off* can be understood intuitively as follows. Let us assume we are the MAX player searching the value of a MAX node (as in Figure 4.9).

As we perform our look ahead, we encounter child positions. Until all children have been searched, the values of the children seen so far provide a bound on the final value. In our case this intermediate value is a lower bound (since we are in a MAX node our value can only increase with each child value that comes in). In alpha-beta, this lower bound is recorded in the α value of the $\langle \alpha, \beta \rangle$ -window as soon as the first sub-tree of the root has been searched. The α -value thus increases from $-\infty$ to the value of the best child c_α that we have seen so far. In the example tree, this c_α is node a , the first child of the root r . The alpha-beta window at root r , after a has been searched, is $\langle 3, +\infty \rangle$. So far, so good.

Now, we continue the search with child b with window $\langle 3, +\infty \rangle$. The positions that our opponent searches can only lower the values it finds, since our opponent is the minimizer. Thus, in MIN nodes, the intermediate search results are upper bounds. In MIN nodes, the search results update β of the $\langle \alpha, \beta \rangle$ -window. As soon as the opponent finds child positions c_β equal to or below that good move c_α that we found in the previous paragraph, the search at our opponent's node can be stopped, since its remaining children can not influence our value anymore. In the example this child is node e , whose value 2 causes the β bound to fall below

⁹You will note that the code for the MAX and the MIN node is quite similar. A clever reformulation exists, NegaMax, that switches the point of view each time that it is called. NegaMax has no code duplication and is functionally equivalent. The switching of sides complicates reasoning about trees, bounds, and values somewhat, and we do not use it in this book.

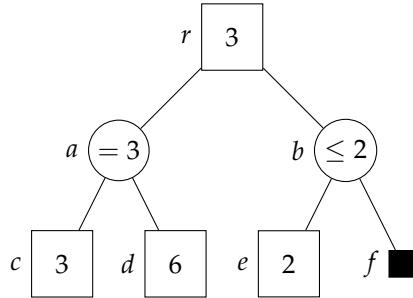


Figure 4.9: Alpha-beta starts at the root, with a window of $(-\infty, +\infty)$. In other words, alpha-beta is called with $n = r, a = -\infty, b = +\infty$. At the root it expands the MAX node's first child a . Node a is of type MIN, and MIN node a 's first child, leaf node c is visited. Node c is a leaf node with value 3. Coming out of the recursion back in node a alpha-beta updates variables $g = \min(+\infty, 3)$ and $b = \min(+\infty, 3)$ to 3. It expands the second child of MIN node a , leaf node d . This node d is expanded with window $(-\infty, 3)$, a tighter beta bound. Its value is 6, coming back in a alpha-beta updates the variables $g = \min(3, 6)$ and $b = \min(3, 6)$ to 3, finds no other children at a , and returns $g = 3$ to r . Back at the root r alpha-beta updates $g = \max(-\infty, 3)$ and $a = \max(-\infty, 3)$ to 3. There is another child b , which it expands with window $(3, +\infty)$, it expands its first child b finding leaf node e . The value of 2 is returned to parent b , who updates the variables $g = \min(+\infty, 2)$ and $b = \min(+\infty, 2)$ to 2. Now the if statement is tested and $3 \geq 2$ is true, so the break statement breaks out of the loop, returning the value of $g = 2$ to parent r , cutting off the expansion of child f . At parent r , the root, the variables are updated to $g = \max(3, 2)$ and $a = \max(3, 2)$ to 3. Alpha-beta is ready and returns $g = 3$ to the caller as the minimax value n without having expanded all nodes of the tree.

α . For us, the remaining child positions have become irrelevant as soon as has been proven that their MIN parent's value will be lower than the good move c_α that we had encountered earlier. *The subsequent children of the MIN node are said to be cut off by the good child $c_\alpha = a$ of the MAX node that we found earlier and the strong reply-move $c_\beta = e$ that the opponent found.*

After I have found a strong move $c_\alpha = a$, then a strong opponent's move $c_\beta = e$ can cut off the search for further opponent's moves. The reason is that my opponent's move $c_\beta = e$ is so strong (bad for the first player), that searching further for other children of b that are even worse for me is useless, since I will no longer play b , but choose instead to play my own strong move $c_\alpha = a$ (or possible other children of the root, if they exist).

In considering my own move, as soon as I have identified a strong enough reply by my opponent to one of my options, I move on, cutting off the search for other replies to that option. If I have not yet found a strong counter reply by my opponent, I continue searching for one in this option. If there is no strong reply,

then I am happy, since I have found a position where my opponent has no strong replies to my move.

Cut-offs are possible due to the intermeshing of maximization and minimization. The impact of alpha-beta on the development of early Chess and Checkers programs was large. Through this simple algorithmic trick programs could search twice as deep as before, achieving a much better quality of play.

It should be noted, however, that there is a catch. Alpha-beta is a left-to-right algorithm, and it can perform cut-offs only if the trees are well ordered. The example tree in Figure 4.9 is well ordered: the root has two sub trees, one with value 3 and one with value ≤ 2 . It is well ordered since the highest (best) child occurs first. If the sub trees had been ordered the other way around, then node b would have been expanded first, and leaf f would have been expanded, along with all other leaves, arriving at the same minimax value of 3. The ordering of successor positions does not influence the value of the position, but it does influence the efficiency with which alpha-beta can compute the value. Alpha-beta's worst case is still w^d .

There is always an ordering of the leaves on which alpha-beta achieves the best-case bound, this tree is called a perfectly ordered tree. In a perfectly ordered tree all children of MAX nodes are ordered high-to-low, and all children of MIN nodes are ordered low-to-high (or to be more precise, all that is needed is that the first child is the best child). Consequently, much research in move ordering heuristics has been performed. Two of the best known heuristics are iterative deepening and transposition tables, which we will discuss next.

4.3.2 Iterative Deepening

Iterative deepening is a search enhancement that, together with transposition tables, can achieve excellent move ordering, allowing alpha-beta to achieve high efficiency. A principle of the search-eval architecture is that looking deeper improves the quality of the answer. A shallow search provides a rough estimate of value and best move, a deeper search provides a better estimate. Likewise, a shallow search provides an approximation of the ordering for a deeper search.

In all practical applications of reinforcement learning, there is a time limit to the search. In typical Chess tournaments a move has to be produced every three minutes on average. The more time we have, the deeper we can search, and the better the playing strength will be. Thus it would be useful if we could stop our search at *any time* and produce a good answer [264]. Iterative deepening provides a solution. Instead of performing one large search to a certain fixed search depth d , iterative deepening performs a sequence of quick searches, starting with depth 0, increasing to depth d . It may seem wasteful to perform these $d - 1$ shallow searches, but due to the exponential nature of the search, the shallow searches take negligible time.

Iterative deepening is a technique to transform a standard fixed-depth depth-first search algorithm into an any time algorithm, that can be stopped at any time and still provide a useful answer.

```
def iterativedeepening(n):
    d = 1
    while not keypressed() and not time_is_up():
        f = minimax(n, d)
        d = d + 1
    return f
```

Listing 4.5: Iterative Deepening

In addition, when the search results (especially the best moves) of all the shallow searches are retained in a large cache (the transposition table) the move ordering of each subsequent $d+1$ search will benefit from the shallower d searches.

Thus, iterative deepening has two advantages: (1) it transforms alpha-beta into an anytime algorithm, and, together with transposition tables, (2) it improves the move ordering for alpha-beta, increasing its efficiency.

The code of iterative deepening is simple, see Listing 4.5.

Though useful by itself, iterative deepening really shines when combined with transposition tables, since then it can improve move ordering for alpha-beta significantly.

4.3.3 Transposition Tables

In many games there are positions that can be reached through multiple parent positions. For example, the position in Figure 4.10 can be reached through the move sequence d4,d5,c4 but also through the move sequence c4,d5,d4. The position has two parent positions (d4,d5) and (c4,d5). Arriving at the same position through a different move sequence is called a *transposition*. So (c4,d5,d4) is a transposition for the state that we get by playing (d4,d5,c4). It is an identical state with a different move history.¹⁰

In Chess, the state space is not a tree but a graph. Searching a transposition is wasteful, and exponentially so. Tree traversal algorithms such as minimax and alpha-beta can be transformed to graph traversal algorithms through the addition of a cache or table. Such a cache or table is called a transposition table. The table stores positions and their search results, in order to prevent transpositions to be searched. Before a new node is searched, the graph algorithm checks if the search result is already known in the table. Search results can be stored in lookup tables (hash tables or dictionaries). For board games often the fast Zobrist hash function is used [753].

Listing 4.6 shows how transposition table results are used to transform a tree search algorithms into a graph search algorithm. Two italic lines of code are added to the code. (The code shown is the minimax function, because it is shorter than alpha-beta, although in practice the latter will be used.) The two lines show basic versions of the use of lookup and store functions, only returning the search value g . In a real program more elements would be added. For one, real programs

¹⁰To transpose means to exchange.

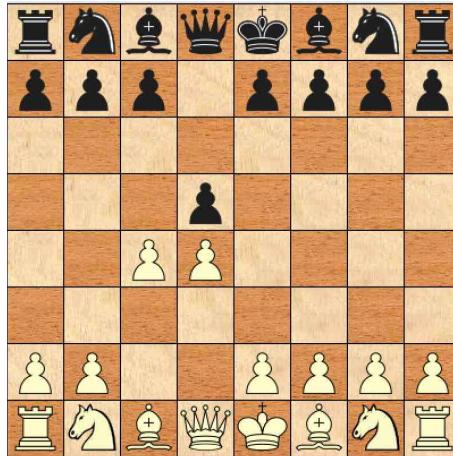


Figure 4.10: Transposition: c4,d5,d4 or d4,d5,c4?

would use a null window variant of alpha-beta (to be discussed soon). The search results that are stored would be upper bounds, lower bounds, or minimax values. Furthermore, in real programs not only the values of the best successor state would be stored, but also the actions leading to them. In this way, information on the *best action* in a shallower search depth is acquired, which will greatly help with move ordering to produce many alpha-beta cut-offs.

Apart from the efficiency improvement due to the prevention of duplicate search of transposed states, it is these best actions of shallower searches that make transposition tables so important. In an iterative deepening search, at all inner nodes best action information will be present in the transposition table. In most situations search values and best moves are moderately stable between iterations: the previous search depth is a good approximation of the current search depth. The consequence is that the best action information from the transposition table from a previous search depth will frequently be the best action for a deeper search depth and therefore cause an alpha-beta cut off at inner nodes.¹¹

Together, iterative deepening and transposition tables allow for the search of a well ordered tree and an efficient alpha-beta search, approaching the best case of alpha-beta in practice [573]. Note that this is an empirical finding, as pathological counter examples can be constructed, but do not appear to occur frequently in practice.

¹¹The opposite situation, where shallow searches are not good predictors for deeper searches, is considered a pathological situation. See the works of Nau, Pearl and others [470, 503, 47, 732, 556].

```

def minimax(n):
    (hit, g) = lookup(n) # find in transposition table
    if hit:
        return g
    elif n[‘type’] == ‘LEAF’:
        g = eval(n)
    elif n[‘type’] == ‘MAX’:
        g = -INF
        for c in n[‘children’]:
            g = max(g, minimax(c))
    elif n[‘type’] == ‘MIN’:
        g = INF
        for c in n[‘children’]:
            g = min(g, minimax(c))
    else:
        error(“Wrong node type”)
    store(n, g) # store in transposition table
    return g

```

Listing 4.6: Use of Transposition Table (best action and depth not shown)

4.3.4 *Null Window Search

Before we go to more heuristics in the next section, we will look further into enhancements based on the alpha-beta window. The window is a powerful concept, with which more efficiency can be gained. The move ordering can be improved in order to approach a perfectly ordered tree, improving the number of cut offs that alpha-beta can find. There is another way to achieve more cut offs, and that is through null windows.

This is a starred section (*) that covers advanced material. If you only want to follow the big picture then this can be skipped (or skimmed). For an understanding of how heuristic planning can achieve good performance, to get an appreciation of the amount of complexity and effort that went into current state of the art game playing programs, please read on.

Narrow Windows, Bounds, and Solution Trees

Alpha-beta searches with a wide window, which causes cut offs at internal nodes, returning the minimax value at the root node. The tighter the window, the more cut offs. So far we have only considered alpha-beta as a routine that searches with a wide window. This is the case at the root. At most inner nodes, alpha-beta is called with tighter search window. Let us have a closer look at what happens: how much of the tree is searched, and how we should interpret the return value that falls outside the alpha-beta window with which it was called. Figure 4.11 shows the familiar tree, whose minimax value is 3. In the Figure we will traverse the tree now two times, first with a narrow alpha-beta window $\langle -\infty, 2 \rangle$ just below that minimax value, and second with a narrow window $\langle 4, +\infty \rangle$ just above the minimax value, to show what happens when the return value falls outside the alpha-beta window. Please follow the example in the Figure, and refer to the

alpha-beta code in Listing 4.4.

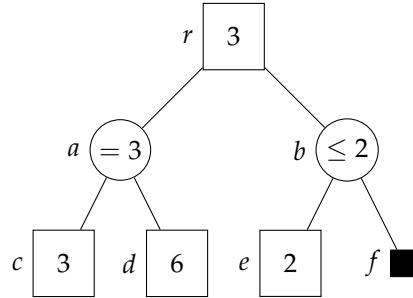


Figure 4.11: Alpha-beta starts at the root with narrow window $(-\infty, 2)$. In other words, alpha-beta is called with $n = r, a = -\infty, b = 2$. At the root it expands first child a . Node a 's first child, leaf node c is visited. Coming out of the recursion back in node a alpha-beta updates variables $g = \min(+\infty, 3)$ to 3 and $b = \min(2, 3)$ to 2. No cutoff happens since $-\infty \not\geq 3$. We then expand the second child of MIN node a , leaf node d . Its value is 6, coming back in a alpha-beta updates the variables $g = \min(3, 6)$ and $b = \min(3, 6)$ to 3, finds no other children at a , and returns $g = 3$ to r . Back at the root r alpha-beta updates $g = \max(-\infty, 3)$ and $a = \max(-\infty, 3)$ to 3. Now a cutoff happens, since $3 \geq 2$. The root returns the value 3, a lower bound since the MAX node has unexpanded children. Let us also try a second narrow window, this time with a high window: $(4, +\infty)$. At the root node a and leaf c are expanded, which returns 3 at node a . There a cutoff happens, since $4 \geq 3$. MIN node a returns upper bound 3 to the root. There, no cutoff happens since $3 \not\geq +\infty$ and node b is expanded, with window $(4, +\infty)$. Node e is expanded and returns 2, which causes a cutoff in b since $4 \geq 2$, and upper bound 2 is returned to the root, which returns $\max(3, 2) = 2$. This value is an upper bound, since the two children returned upper bounds.

The lower narrow window $(-\infty, 2)$ yields as return value a lower bound $3 \leq g$ on the minimax value (future child expansion can still increase the value at MAX nodes, hence the value so far is a lower bound on the final return value). As we can see, the subtree expanded so far is a min solution tree (all children of MIN nodes, not all of MAX nodes). The higher narrow window $(4, +\infty)$ yields as return value an upper bound $g \leq 3$ since the MIN children of the root returned upper bounds, as only some of their children were expanded. The tree traversed in this second pass for this upper bound is a max solution tree (all children of MAX nodes, some of MIN nodes). The first (low) window yielded lower bound $3 \leq g$ and the second (high) window yielded upper bound $g \leq 3$. Since, in this case, the lower and the upper bound are equal, we have found the minimax value $3 \leq g \leq 3$ and the proof tree is the union of the min solution tree and the max solution tree.

Post-condition and Bounds

A narrow window causes more cut offs, but returns a bound if the return value is outside the window. Can we construct useful search algorithms using only bounds?

We will now formalize these notions in the alpha-beta post-condition, in order to see if we can create a useful and efficient algorithm based on the narrow window idea. We know that alpha-beta returns the minimax value if it is called with the window $\langle -\infty, +\infty \rangle$. When the return value falls outside the window, then the return value bounds the minimax value.

The post-condition of $g = \text{alpha-beta}(n, \alpha, \beta)$ is as follows [355, 518]:

1. $\alpha < g < \beta$: the window was right. g is the minimax value of the tree rooted in n . Alpha-beta has traversed at least the nodes in the critical tree, the union of a max and a min solution tree.
2. $g \leq \alpha$: the window was too high. g is an upper bound on the minimax value of the tree rooted in n . Alpha-beta has traversed at least the nodes in the max solution tree whose value is upper bound g on the minimax value.
3. $\beta \leq g$: the window was too low. g is a lower bound on the minimax value of the tree rooted in n . Alpha-beta has traversed at least nodes in the min solution tree whose value is lower bound g on the minimax value.

This post-condition allows the construction of the ultimate in efficient decision procedures: the Null Window Search, or NWS. A null window, of $\alpha = \beta - 1$ guarantees¹² that alpha-beta performs the most efficient search possible, at the cost of returning only a Boolean answer: the minimax value is either above or below the input window. Finding the minimax value with the NWS decision procedure requires a sequence of NWS calls. The literature contains two main algorithms, Scout, which uses NWS recursively, and MTD(f), which uses NWS iteratively from the root.¹³ Scout is also known as Principal Variation Search [213].

Scout

Scout [502, 536] works on the assumption that the first child of a node is the best child. This first child is called the *principal variation*. Assuming such a well ordered tree, Scout searches the first child with a wide window, and then uses NWS to check efficiently if the other children are indeed inferior. If not, they are searched with a wide window, to find the value. Listing 4.7 shows the code.

¹²We assume that all values are integer values, so with $\alpha = \beta - 1$ there is no room for an integer value to fit in the window. A null window of $\alpha = \beta$ would not work, since alpha-beta would return immediately and not traverse any nodes. The term *one window* would perhaps have been better, but history has decided otherwise.

¹³The efficiency gain for null windows versus wide windows depends to a large degree on the quality of the move ordering. Experiments with tournament quality programs yielded differences of around 20% in leaf count, increasing with search depth [517, 518]. For badly ordered trees the difference is much larger.

```

INF = 99999

def eval(n):
    if n['type'] == 'LEAF':
        return n['value']
    else:
        error("Calling eval not on LEAF")

def scout(n, a, b):
    if n['type'] == 'LEAF':
        return eval(n)
    elif n['type'] == 'MAX':
        g = -INF
        for c in n['children']:
            if c == n['children'][0]:
                g = max(g, scout(c, a, b)) # first child wide window
            else:
                g = max(g, scout(c, a, a+1)) # other children nws
                if g >= a+1:
                    g = max(g, scout(c, g, b)) # if better, wide window
            a = max(a, g) # update alpha
            if g >= b:
                break # beta cutoff, a>=b
    elif n['type'] == 'MIN':
        g = INF
        for c in n['children']:
            if c == n['children'][0]:
                g = min(g, scout(c, a, b)) # first child wide window
            else:
                g = min(g, scout(c, b-1, b)) # other children nws
                if g <= b-1:
                    g = min(g, scout(c, a, g)) # if better, wide window
            b = min(b, g) # update beta
            if a >= g:
                break # alpha cutoff, a>=b
    else:
        error("Wrong node type")
    return g

print("Minimax value:", scout(root, -INF, INF))

```

Listing 4.7: Scout [502, 536]

```

def mtdf(n, f):
    ub = +INF
    lb = -INF
    while lb < ub:
        if f == lb:
            b = f+1
        else:
            b = f
        f = alphabeta_with_tt(n, b-1, b)
        if f < b:
            ub = f
        else:
            lb = f
    return f

```

Listing 4.8: MTD(f) [518]

The re-researches may seem inefficient, and they would be, if alpha-beta would recompute search results for each search anew. Actual game playing versions of alpha-beta use transposition tables, large lookup tables in which all nodes and their search results are efficiently stored. Re-searching is therefore not more expensive, since no new nodes are expanded. (The code in the Listing omits the lookup and store calls, and the depth parameter.)

MTD(f) and MTD-bi

MTD(f) [518] takes as input a first guess to the minimax value, and then iteratively calls a sequence of NWS decision procedures to home in on the minimax value. If the first guess is too high, then NWS will return a lower value, an upper bound, which is fed into the next NWS call, a sequence of lower return values (upper bounds) follows, until the minimax value is found and NWS returns a higher return value (a lower bound equal to the previous upper bound). If the first guess is too low, the converse occurs.

In contrast to Scout, which uses some wide window searches, MTD(f) uses only NWS. Experimental comparisons showed indeed that MTD(f) expands even fewer nodes than Scout on average, although trees can be constructed in which either algorithm performs best [518, 516]. Even more so than Scout, MTD(f) relies on a well functioning lookup table to store intermediate search results in, which requires care with inconsistencies in the tree.

Listing 4.8 shows the code for MTD(f).

MTD(f) starts the search for the minimax value at a heuristic first guess, and then iterating in the direction of the minimax value based on the outcome of the null window search. When the granularity of the evaluation function is fine, the number of iterations can become large, and the overhead of traversing through the tree also increases.

An alternative approach is to use a binary search [356] to interpolate between the upper and the lower bound. That value can then be used as the pivot for

the null window search, which will produce a new bound, and to repeat the binary search. This may result in fewer iterations of null window search to find the minimax value. This algorithm is called MTD-bi [518]. The idea of bisecting the search goes back to C* [148, 725] where it is used in end game search. MTD-bi is used in Sunfish, a tiny fully functional Chess program in 111 lines of Python code (see Section 4.6.1).

After this elaborate discussion of alpha-beta and related move ordering enhancements, it is time to discuss other enhancements that have been developed to improve the performance of heuristic planning.

4.3.5 *Heuristic Forward Pruning

If we want to search deeper, the first idea that often springs to mind is to only search “important” nodes. This method is called forward pruning in contrast to alpha-beta’s backward pruning. In forward pruning, the idea is to use domain dependent heuristics to determine these important nodes. If for certain moves we can immediately see that they are not good, then they should not be searched. In early game playing programs this was used since compute power was so limited relative to the large number of sub-trees to be searched. This is the approach taken in Go playing programs such as GNU Go, where the move generator generates only the “sensible” moves such as connect, corner, jump, and territory making moves. The disadvantage of heuristic forward pruning is that the static heuristics may be wrong, and valuable moves are missed, that only a search would reveal to be valuable. In practice it turned out that static board heuristics can all too easily miss deeper tactical game dynamics. As soon as compute power allowed, programmers of Chess and Checkers programs started to use full width move lists.

Plain heuristic forward pruning misses too many dynamics in most games. In Section 4.4.3 further developments of forward pruning are introduced, named null move pruning [178] and Prob-cut [114]. In contrast, heuristic evaluation does work well in Chess and Checkers, and when more compute power became available, the backward pruning power of alpha-beta turned out to achieve good results with full width (no selectivity) search.

4.3.6 *Capture moves

Another ordering mechanism that is highly domain specific, but in contrast is often highly effective, is capture move ordering. For alpha-beta it is important to search an ordered sequence of moves. An obvious move ordering enhancement is to prioritize capture moves, since in many games the dominating factor of the heuristic evaluation function is material balance, e.g., how many pieces of a color are still on the board. It stands to reason that in positions where among the available moves there are capture moves, the capture moves lead to a better position, and should be tried first.

Many Chess programs therefore employ a move generator that first generates capture moves, and if no cut off has occurred, then the non-capture moves.

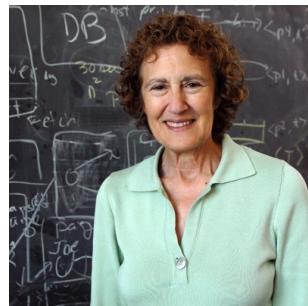


Figure 4.12: Barbara Liskov

Note that this enhancement is heuristic. It uses domain dependent knowledge, and only works in domains where captures are important and frequent. In Go, for example, captures are less frequent, and the capture enhancement is of no importance in this domain.

4.3.7 *Killer Moves and History Heuristic

Another well-known and effective enhancement is called the *killer move*. In many domains actions that are good actions in one position, are also good in another position. For example, if I am in a position where I find a strong action, say by moving my pawn from e4 to f5 I can capture my opponent's queen, then the move pawn e4-f5 is likely to be a good move in many other positions, provided the board has not changed too much. This move is called a killer move. The enhancement works by recording moves that are so strong as to cause an alpha-beta cut off, and then first trying these moves in subsequent positions [309].

A generalization of the killer move is the history heuristic, which adds counters to the moves that cause cut offs, making the heuristic more refined. The history heuristic maintains a table indexed by *from* and *to* squares, and when there is a cutoff, the entry in the table is incremented, possibly weighted by search depth [568]. The history heuristic came out of basic experiments on the interplay of search and knowledge in Chess, one of the earlier works with a principled empirical approach to this question [567].

Killer moves are even easier to implement than the history heuristic, are more general than capture moves, and work in many domains. They are present in most Chess and Checkers playing programs. Killer moves were first described by Barbara Liskov, who would (much later) win the 2008 Turing award for work on object oriented and type safe programming (Figure 4.12). The history heuristic was developed by Jonathan Schaeffer, who would later lead the Chinook Checkers effort.

Most search enhancements focus on selection of moves as a way to improve efficiency.

To conclude our discussion of enhancements of the search function, we will look at a different element of the search, the backup rule.

4.3.8 *Backup Rules other than Minimax

Most enhancements in this chapter are applicable to two-person zero sum perfect information games, games that can be modeled well in the minimax paradigm. For other games, different backup rules are needed. Interestingly, even for some zero sum games a non-minimax backup rule works best, as we shall see in the next chapter on MCTS. In this section we will discuss alternative backup rules.

The minimax backup rule has dominated game playing research for a long time (1950-2006). For two-person zero-sum games with a single heuristic reward value it is intuitively appealing, since it follows the idea of how humans play a game in which they try to pick the best move.

For imperfect information games/games of chance, such as card games or dice games, minimax is not an obvious choice. Many minimax researchers have tried to find work-arounds to force these games into the minimax paradigm, with limited success.

Statistical sampling can be used to cope with the randomness of the game, and be able to use the familiar minimax. A disadvantage is that in order to get reliable outcomes, the number of samples must be high. Alternatives to minimax have been developed. Already in 1966 Michie published the Expectimax algorithm [436] for computing probabilities with the product rule. In 1983 Ballard published the *-minimax rule, which augmented the alpha-beta algorithm with chance nodes, allowing distributions to be propagated through the product rule (by multiplication) while still allowing the efficiency of cutoffs [32, 535]. Subsequent experiments are reported in [135, 469, 271]. Expectimax and *-minimax did not become popular after these experiments, although in 2013 Lanctot et al. [386] reported good results with a combination of Expectimax and statistical sampling.

In 1988 McAllester [431] introduced conspiracy number search as an alternative to alpha-beta, which is based on backing up the size of the sub-tree in addition to the value (how many nodes *conspire* to change a node's value). It thus explicitly searches for small sub-trees. It did not become popular, but it did lead to the development of proof number search [6, 5] which is based on the search for Boolean values such as win/loss. Proof number search is often used for solving a game, where the state space is fully searched to terminal states, without a heuristic evaluation function [541, 570].

The next chapter, on Monte Carlo Tree Search, will introduce the *averaging of statistical samples* as backup rule. MCTS and its backup rule did become quite popular and successful.

To summarize the different backup rules, minimax is an efficient algorithm for backing up scalar values. Expectimax works with probability distributions. Averaging works well for statistical sampling. For Boolean values such as win/loss, proof number search is an efficient algorithm.

Name	Principle	Applicability	Effectiveness
search extensions	variable depth	domain specific	good
quiescence search	variable depth	domain specific	good
null-move search	variable depth	no Zugzwang	good
odd-even effect	move advantage	minimax games	unstable values
piece-square tables	evaluation coding	heuristic evaluation	efficiency
end-game database	database	end-game	perfect knowledge
coefficient learning	machine learning	evaluation tuning	automated learning

Table 4.4: Eval Enhancements

4.3.9 Trends in Search

We are coming to the end of the section on search enhancements and will close with some conclusions.

Please refer to Table 4.3. Minimax is a rigid fixed-width, fixed-depth algorithm, distributing the search effort equally over all parts of the tree, good or bad. The enhancements reported in this section all point to a trend of variable-width, variable-depth, using forward or backward pruning, and move ordering, to devote most of the search effort to the most promising part of the tree, to find the best move, disregarding bad moves as early as possible.

Furthermore, there is a trend towards generality, away from domain specific heuristics, towards mechanisms such as the history heuristic, iterative deepening, and transposition tables, methods that work in many domains.

4.4 Evaluation Enhancements

After elaborating on the many search enhancements that have been invented, it is time to see how heuristic evaluation can be enhanced. Table 4.4 gives an overview of the evaluation function enhancements. The enhancements try to ameliorate the limitations of static heuristics with search extensions, and improve the evaluation function by smart implementations and databases.

4.4.1 *Search Extensions

The heuristic evaluation function computes a score of the static board position. These scores are then used in the search. Potentially important tactical dynamics in the position cannot be seen by a standard static evaluator, since the tactical problem is over the *search horizon*. It would help if this situation could be detected, and the search be extended somewhat to see the effect of the tactical dynamics. Singular extensions were devised by the Deep Thought team (the predecessors of Deep Blue) for Chess [9, 302]. There are many variations of search extensions, all are based on domain specific heuristics. Two of the best known

search extensions for Chess are check extensions and capture extensions. Positions that are in check or where some kind of capture happened are searched one level deeper. A problem with capture extensions can be that in positions where many capture opportunities exist, the search expands too much. For this reason partial depth extensions have been devised. Tuning the amount and type of search extensions is an important part of the tuning effort of game playing programs, especially in Chess, which is a highly tactical game.

Singular extensions are related in some sense to forward pruning. Forward pruning uses heuristics to restrict search to certain “good” moves. Search extensions do a full-width search up to a certain depth, and then use heuristics to extend certain positions for deeper search, combining full-width and selective search.

Related to search extensions are search reductions [83], where in the multi-cut algorithm, a shallow search is used to determine if a cut-off is taken early in a node. Related to multi-cut is Prob-cut [114] and a pruning technique in Chinook [571] which uses a shallow search to identify nodes to terminate the search. Finally, search extensions are reported to reduce the problem of search pathology [337, 8, 590].

4.4.2 *Quiescence Search

Quiescence search [44] and search extensions both try to reduce the impact of the search horizon. Quiescence search tries to distinguish between so-called *quiet* from *noisy* positions. A noisy position is a position whose value cannot be reliably estimated with static heuristics, for example because of the presence of threats or other dynamics. Noisy positions should be searched deeper until a quiet position arises, that can more reliably be evaluated. Quiescence search is most often implemented by looking at differences between positions in the heuristic evaluation function. As with search extensions, getting quiescence search to work requires much experimentation, and a deep understanding of the effects of the static evaluation heuristics that are used in one’s program. When the evaluation function is changed, then the quiescence and extension tuning may need to start over again.

4.4.3 *Null-Move Heuristic

We will now introduce the null-move heuristic. A null-move is a pass, not making a move, a position where the only difference with the successor position is the player to move [43]. In some games passing is legal, such as in Go, in some games it is not, such as in Chess. In most positions not making a move is detrimental to the score of the position. Being allowed to move typically provides opportunities for improvement. If it is not, and all possible moves make a position worse, it is called *Zugzwang*, or forced moving. In this case the introduction of a pass-move causes an unfair advantage to occur.

The null-move heuristic tries to get alpha-beta cut offs cheaply by seeing if a pass creates a situation that is strong enough to cause a cut-off before the regular moves are tried. If a position is so strong that not moving already provides a cut

off, then it is a good idea to take that cut off. In that case the cut off has been found without an expensive search.

Null-moves typically work well, at least in Chess. (In other games, such as Othello, Zugzwang plays a bigger role, and null-moves are not used, or in a different way [114].) The null-move idea has been developed further by extending zero-depth cut off searches to small-depth cut off searches, with success [243, 178, 279, 114, 97]. ProbCut [114] in particular has been important in Othello.

4.4.4 *Odd Even Effect

As was noted before, in most positions having the opportunity to make a move is advantageous. If we are white, then positions with white-to-move are typically better than black-to-move positions. This fact causes pronounced swings in the score between search depths. The odd-even effect is caused by a simple statistical fact, that the expected value of the maximum of a distribution is greater than the expected value of the minimum of a distribution: $E(\max(\text{random}[-999, +999])) > E(\min(\text{random}[-999, +999]))$.

The odd even effect is a statistical effect, it is possible to construct counter example trees where the value at the node is stable as the search depth increases. This so-called odd even effect is also related to the search horizon. Many game trees have a pronounced odd even effect, especially when the trees are fixed depth. As iterative deepening searches deeper (Listing 4.5) the scores at the root node go up and down as the leaf values switch between being minimized and maximized.

The odd-even effect occurs in “normal” trees, for non-Zugzwang positions, and without search pathology. The odd-even effect is not caused by search instability, but is a consequence of the min-max structure of the minimax tree [187].

The value-swings do complicate empirical analysis and finding errors in programs, and comparing search results from different search depths becomes less reliable.

The odd even effect can be reduced by search extensions, that transform the fixed-depth search into a variable-depth search. For this reason it can be advantageous for search extensions and shallow searches to extend and reduce depth in multiples of 2.

After discussing enhancements that are concerned with the search horizon, we will now look at the core of the evaluation function, the efficient computation of the heuristic value.

4.4.5 *Piece-Square tables

Piece-square tables are a popular method for creating efficient evaluation functions. They are used in many strong Chess programs [244].

A piece-square table is a table for each piece on each square for each color. Values for the piece \times square \times color combination indicate the value of the piece being on this square. Piece-square tables are efficient since moving a piece from



Figure 4.13: Fabien Letouzey

a square to another amounts to subtracting the source value and adding the destination value to the heuristic evaluation score.

Piece-square tables are quite popular because of their efficiency. A well known implementation of piece-square tables is by Fabien Letouzey (see Figure 4.13) in his open source program Fruit. The source code is available here: [Fruit](#).¹⁴

Furthermore, piece-square tables are well-suited for learning the evaluation function [45, 183], sometimes leading to surprising differences between manually tuned and machine-learned values.

The creation of an evaluation function that encodes heuristics in a correct way and combines their values efficiently is a challenging problem. Let us look at how this can be automated.

4.4.6 Coefficient Learning

As has been noted before, tuning the coefficients of evaluation function terms (see Section 4.1.3) by hand is a difficult and error-prone process. Recall the general form of the evaluation function:

$$h(s) = c_1 \times f_1(s) + c_2 \times f_2(s) + c_3 \times f_3(s) + \dots + c_n \times f_n(s)$$

An alternative is to train the coefficients by supervised learning. This works by creating a large database of test positions from grand-master games, of which the value or the best move is known. The difference between the evaluation of the state and the correct answer constitutes an error value. Then an iterative optimization process is used to adjust the evaluation function coefficients such that the sum of the error values is minimized.

¹⁴<http://arctrix.com/nas/chess/fruit/>



Figure 4.14: Ken Thompson, winner of the 1983 Turing award for his work on the UNIX operating system

Many authors report on machine learning methods that train the coefficients against databases of grand-master games or in self-play [560, 41, 118, 528, 664, 218, 481, 226, 219, 159, 653]. Frequently, it is reported that the coefficients deviate considerably from hand-tuned coefficients (the machine learning co-efficients “work, but don’t make sense”). Many Chess programmers have tried to find a middle ground by adjusting the coefficients to values that make intuitive sense, and also have low error scores, after they have been optimized by machine learning.

Most work in automated Chess evaluation optimization is in the automated tuning of the coefficient of the features. There is also some work that goes a step further, and tries end-to-end learning of evaluation functions [157, 607, 380] directly from board positions, without the intermediate step of hand crafted heuristic features. This work is quite recent and often related to the work reported in Chapter 7.

We are coming to the end of this overview of evaluation function enhancements. There is one more to discuss: the end-game.

4.4.7 End-game Database

In Chess and Checkers the end-game is an important phase of the game. Inaccurate play can often result in loss of the game. It is also a phase of the game where few pieces are on the board, allowing the exact minimax value to be computed (without heuristic approximation). Sometimes highly “artificial” lines of play turn out to be the correct line of play, with non-trivial and counter-intuitive moves. Work on end-game databases was proposed early, e.g., by Bellman [57], and Thompson [662], see Figure 4.14.

Much of the strength of world champion Checkers program Chinook came

from the 8 piece end-game database [572]. Chess program Deep Blue used a database that contained 5 piece endings and some 6 piece endings.

End-game databases are often computed by retrograde analysis, working backwards from the end [201, 384, 662]. Retrograde analysis is used frequently in solving games (determining the outcome in the case of perfect play by both sides, by full enumeration of the state space). Games that have been solved are Nine Men's Morris [229], Awari [541] and Checkers [570].

Culberson and Schaeffer generalized end-game databases to pattern databases, to be used in all stages of a search [153, 367]. Pattern databases play an important role in the playout phase of MCTS (see Section 5.2.1).

4.4.8 Trends in Eval

We are now at the end of our discussion of heuristic evaluation enhancements. Let us see if we can find a common trend in the enhancements.

Heuristic evaluation function are domain specific. One would think that enhancements of evaluation functions trend towards more specific knowledge. This is not the case. The techniques that are used to enhance evaluation functions trend towards finding automated methods for feature creation: retrograde analysis for creating end-game databases, machine learning of coefficients or piece-square tables, and many variations trying to overcome the search horizon problem. In many cases, the heuristic knowledge is compartmentalized in features that are created with automated methods. Furthermore, many game programmers use automated testing whenever a change is made to the code base. Even in search extensions and quiescence search, whose implementation typically involves domain specifics, there have been efforts towards general methods [44].

The overall trend in evaluation function enhancements is that enhancements start as domain specific, and then work is being undertaken towards automated methods and automated learning.

4.5 System 1 and System 2

We have come to the end of this chapter on heuristic planning. The search-eval architecture was originally inspired by the way in which humans are thought to play a game. Can we see a relation between the heuristic planning methods and human thinking as it is described by Kahneman?

Since the earliest days of game playing research, the search-eval architecture has been the cornerstone of game playing programs. The architecture is simple, see Figure 4.15. The search function traverses possible game positions, and the evaluation function scores the position with a heuristic value function. In the early days of Turing's and Shannon's papers, when search-eval was conceived [679, 600], the search and eval functions were quite straightforward. The heuristic function was the most basic function that worked, and the search function was a minimax(-like) function to traverse the nodes in standard depth-first

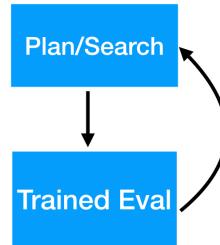


Figure 4.15: Search–Eval Architecture

Chapter	Algorithm	Select	Backup	Eval
Ch 4	alpha-beta	left-to-right	minimax	heuristic
Ch 5	MCTS	adaptive	average	sample
Ch 6	DQN	-	-	generalize
Ch 7	Self-Play	adaptive	average	generalize

Table 4.5: Heuristic-Sample-Generalize

order. This simple model proved quite versatile and amenable to successive refinements. Over the years many enhancements were introduced and performance has reached world champion level in games such as Chess and Checkers. These enhancements are very important—they elevate the level of performance from being barely able to follow the rules to state of the art world champion beating programs.

Table 4.5 gives an overview of the different paradigms in this book. The entry for Chapter 4 shows the heuristic search approach in reinforcement learning. The focus of this chapter has been on full-width search and manual heuristic evaluation functions, and value backup via the minimax rule. Later on we will discuss variable width and variable depth search and learning evaluation functions.

As the table indicates, in the following chapters different, more adaptive, techniques will be introduced. In Chapter 7 this will culminate in functions that are able to self-learn to play a game as complicated as Go with only the rules of the game known to the program.

At the time when early AI researchers such as Turing, Shannon and Samuel discussed Chess playing computers, it was believed that by developing better Chess programs we would learn how human thought worked, and we would learn about human intelligence. When 40 to 50 years later Deep Blue finally beat Kasparov, this belief had changed. It was said that brute force had won over human intellect. We had not learned about how humans think, but we had learned how to make

a fast computer search many positions.

The work of psychologist Kahneman can be interpreted to mean that this view may not tell the whole story. Perhaps Chess computers do operate in a way that is related to the way that humans think. Humans have a fast, heuristic, System 1, and a slow, deliberative, System 2. This corresponds closely to Chess computers that have a fast heuristic evaluation function, and a slow planning function, that painstakingly reasons through some of the possible positions. In this sense there is a correspondence in how humans and computers think or play Chess. The search-eval architecture is not only a good architecture to organize the functions of a chess program in a clean way, it also fits how Kahneman describes that humans play and think. Perhaps, at a high level, Turing and Shannon's dream of creating a mechanical model of thought has succeeded after all.

Of course the "brute force" argument is not without merit. Computers calculate fast and deep. It is true that computers always blindly follow their program, and do not deviate. Human System 2 thinking is not so perfect, and is easily distracted by System 1 associations. Indeed, computers can search many more positions than humans can.

Computer Chess has also taught us that writing error-free heuristic programs is hard. It has taken four to five decades of research by a dedicated community of researchers before artificial intelligence has risen to the challenge. Too much of this time has been spent in removing errors from and improving heuristic evaluation functions that were hard to understand and hard to debug.

Conceptually the way in which computers and humans play chess is highly related. In the details, there are important differences.

Let us now review this chapter, and get our hands dirty with exercises about a concrete Chess program.

4.6 Practice

Below are some questions to check your understanding of this chapter. Each question is a closed question where a simple, one sentence answer is possible.

Questions

1. What is a heuristic?
2. Why is an exponential state space problematic?
3. How can you search an exponential search space efficiently?
4. What is the role of a search function?
5. What is the role of an evaluation function?
6. Why is minimax a form of self-play?
7. Why is minimax recursive?

8. What is the number of leaves of a regular tree of width w and depth d ?
9. What is the critical tree?
10. What is the number of leaves of a regular tree that prove the value at the root?
11. What is a solution tree?
12. What is material balance?
13. What is Elo?
14. What is forward pruning?
15. What is backward pruning?
16. Draw a tree with an alpha-beta cutoff.
17. What causes the performance of alpha-beta to go from worst case to best case?
18. What is a transposition?
19. How do iterative deepening and transposition tables help the performance of alpha-beta?
20. What is a null window?
21. What is a null move?
22. What is a killer move?
23. What is a search extension?
24. How can you use supervised learning in a Chess program?

4.6.1 Sunfish—Chess in 111 lines of Python

Chess programs are often large pieces of code. For Chess, there are many open source programs available that allow study of the code. However, the size of the codes typically requires a large investment of study time. There are also a few short Chess programs. For example, the program Micro-Max, by Harm Geert Muller, is a minimalist Chess program in 133 lines of C code, in less than 2000 characters (not counting comments). Another example is Sunfish, a Chess program in 111 lines of Python. Sunfish was written by Thomas Ahle. Although the name is similar to Stockfish, Sunfish was inspired by Micro-Max. You can download it from GitHub Sunfish.¹⁵ The entire program code fits in a single file, `sunfish.py`.

¹⁵ <https://github.com/thomasahle/sunfish>

The program consists of a heuristic evaluation function based on piece square tables (see Section 4.4.5). It uses null window search algorithm MTD-bi (see Section 4.3.4) enhanced with iterative deepening, transposition tables, null moves, and killer moves. These enhancements are all discussed in the next Chapter. Sunfish is small enough to play around with and study, to learn how a real working Chess program functions.

The small size and organization of the code also allow easy modification. Things to try are, for example, changes to the piece square table, to see the effect on playing style of a different evaluation function (see the exercises in section 4.6.2).

4.6.2 Playing Strength: Elo

The playing strength of a program can be determined by playing tournaments, and computing a rating based on the outcome and the strength of the opponent. A well-known internet tournament is the Top Chess Engine Championship (TCEC), where strong computer Chess programs are compared and a rating is calculated. TCEC can be found at here.¹⁶ In computer Chess the Elo rating is frequently used to compare playing strength.

The Elo rating is calculated using tournament outcomes (introduced by Arpad Elo, see Section 2.1.3). For each game, the winner's rating increases, and the loser' rating decreases. The change depends on the difference between the original ratings. A win by a stronger program increases its rating by less than a surprise win by a weaker program. If player 1 has a rating R_1 and player 2 has a rating R_2 then the expected score of player 1 is $E_1 = \frac{1}{1+10^{(R_2-R_1)/400}}$. If the actual score of player 1 was S_1 then the new rating of player 1 is $R'_1 = R_1 + K(S_1 - E_1)$ where K is a constant that determines the maximum adjustment per game. It is typically set from 16 for master players to 32 for weak players.

Code for computing the Elo rating is provided at BayesElo.¹⁷

Exercises

1. Download and install Sunfish and BayesElo for Python. Play your first game against Sunfish. Write a script that performs a tournament of two identical instances with the same time controls of Sunfish. Suggestion: 10 seconds per move. Look at the games. Do they make sense? [Only if you know how to play Chess] Does Sunfish at this time setting look like a strong program to you? Try different settings.
2. Setup BayesElo to allow you to run a tournament between two programs to determine their playing strength.

¹⁶<https://tcec.chessdom.com/live.html>

¹⁷<https://github.com/ddugovic/BayesianElo>

3. Running tournaments costs compute time. How long should a tournament be? Add confidence intervals or error bars to the graph showing the standard deviation of the Elo rating.
4. Make a change: halve the playing time of the second player. How does the Elo rating change?
5. Read Part 1 of Kahneman [336], on the Two Systems (less than a hundred pages of enjoyable and illuminating reading. Part 2 is about heuristics, but not the kind we have used here in our evaluation functions). How is the thinking of a computer chess program related to human thinking? Is minimax like System 1 or System 2? Is heuristic evaluation like System 1 or System 2?

Sunfish implements null-window search, null-moves, iterative deepening, transposition tables, and killer moves. Throughout this chapter it was often claimed that certain enhancements work well. Using the Sunfish and Elo code from the previous chapter, the following exercises are to create a graph showing the effect of individual search enhancements. In the exercises we study the influence of search enhancements and evaluation heuristics. The baseline is standard Sunfish with all enhancements and all heuristics. Keep in mind the question how many runs of a program you need to get statistically significant results [36].

1. Try the following: Replace MTD-bi by MTD(f) (see Listing 4.8). How is performance influenced? If the tournaments take too long to run, what can you do to get quicker results?
2. Replace the null window search by wide window alpha-beta. How is performance influenced?
3. Replace the move ordering heuristics: put killer moves last in the move list that is generated by the move generator.
4. Show the Elo rating of the program without enhancements. Remove killer moves, then remove null-moves, then try removing the transposition table and iterative deepening. Are the enhancements related?
5. Study the code of the piece square tables. Make a change to one of the squares, describe in words the intended meaning, and compare the difference on playing strength.
6. Create test positions that contain Zugzwang (not moving is advantageous). Check if turning off null-moves improves performance.

4.7 Summary

In this chapter we introduced the first algorithms to address the reinforcement learning problem. The main problem is the size of the state space. We have calculated the size of the entire state space, and found that for most games a standard

minimax enumeration of the full state space is infeasible. Heuristic methods are introduced that can artificially reduce the state space. Static board evaluation functions do not capture game dynamics, and search remains necessary. Thus we arrived at a solution architecture consisting of two functions: the search-eval architecture.

The search functions covered in this chapter have been conceptually simple, in the sense that they are directly based on the maximization and minimization of the reward values. We have discussed the minimax procedure, and have looked at the concept critical tree, the smallest set of nodes that has to be searched to be certain of the value of the state space. This smallest tree concept has then been used to arrive at efficient search algorithms.

The efficiency of both the search and the evaluation function can be greatly enhanced with general and domain specific enhancements. We covered a wide range of heuristic planning enhancements, both search enhancements and eval enhancements.

The first search enhancement is alpha-beta, an algorithm that can search twice as deep as minimax, if the tree is optimally ordered.

Achieving world champion level of performance with alpha-beta required researchers and game programmers to invent many search enhancements. For alpha-beta cut-offs to work effectively, the tree must be ordered. Perhaps the most important enhancements for alpha-beta are iterative deepening and transposition tables. They work together by storing best-move information of previous searches. In this way, alpha-beta searches an ordered tree.

Another important area is relaxing full-width search. A basic approach is heuristic forward pruning. For Chess, it was found that a better approach is to use a mix of full-width and selective search. The first few levels of the tree are searched full-width, so that no move is left untried, and after a certain depth extensions (or reductions) are used.

On the eval side, we discussed quiescence, search extensions, piece-square tables, null moves, end game databases, and evaluation function learning.

A popular heuristic that works in positions is null-move pruning, where before the legal moves an extra (perhaps shallow) pass-move is searched. If a position is so strong that even passing yields a cut off, then it is safe to take the cut-off.

Researchers have a large imagination, and many ideas have been tried. Not all have been successful in the minimax framework and in the games in which they have been tried. Among them are forward pruning (in Chess and in Go), different backup rules (such as star-minimax), Monte Carlo evaluation (such as on Go), and neural network evaluation (in Chess and in Go). Interestingly, as we will see in the upcoming chapters, all these ideas have been used to achieve a high level of play, in different frameworks or games.

We noted a trend towards variable depth search and automation in evaluation. Enhancements typically start with an idea to exploit a domain specific feature (such as the killer move, or hand tuned coefficients). After some more research, often a more general version arises, that performs even better (such as the history heuristic, or machine optimized evaluation coefficients).

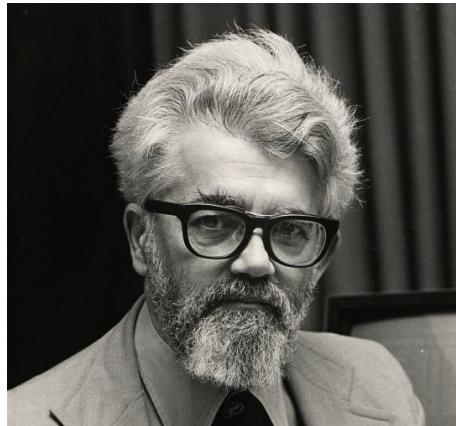


Figure 4.16: John McCarthy

In the next chapter we will continue the trend towards more generally applicable methods and more selective search. We will see how Monte Carlo evaluation, a different backup rule, and selective search together form an algorithm that works well in domains in which minimax and heuristics do not work.

Historical and Bibliographical Notes

One of the earlier works in Heuristic Planning is Pearl’s Heuristics [504] (he would later co-author *The Book of Why*, an accessible text on his scientific work on causation [505]). Minimax is described in most standard artificial intelligence undergraduate texts, such as Russell and Norvig [554], although without solution trees. There is an extensive literature on solution trees, see for example [377, 515, 626, 504, 518].

One of the first open source programs is Crafty [313]. Chess programs have always used the most compute power that was available, and many have stimulated search into parallel programs, where multiple processors search the state space collectively. Deep Blue used special hardware and software for parallel search. Parallel Chess has also stimulated the development of research into parallel programming environments, such as Cilk [400]. The Parallel Cilkchess program achieved some success in its days. Fruit is an influential open source program by Fabien Letouzey, the well-known open source program GNU Chess is now based on Fruit [401].

The Chess programming wiki contains a wealth of information on null-moves, search extension, and transposition tables.¹⁸ Many conferences, workshops, and journals have published papers on computer games. A specialized series of con-

¹⁸https://www.chessprogramming.org/Main_Page

ferences is devoted to computer games, the Advances in Computer Chess conference and the Computer Games conference. The journal of the International Computer Chess (later: Games) Association has published news, tournament reports, and scientific articles.

Alpha-beta is reported to have been reinvented independently by several authors, among them McCarthy (see Figure 4.16), Samuel, Brudno, Richard, Hart and Levine [476]. John McCarthy won the 1971 Turing award for work on AI.

A seminal analysis of the alpha-beta algorithm is Knuth & Moore [355]. This analysis was without the benefit of the concept of solution trees, which were introduced in later publications on Stockman's SSS* algorithm [626, 504, 515]. The relation between SSS*, solution trees, critical trees and alpha-beta is described in [518].

Killer moves were described by Barbara Liskov (Huberman) in her 1968 thesis [309]. The history heuristic is described in Jonathan Schaeffer's thesis and subsequent paper [568].

An overview paper of search enhancements is [575]. A paper on search versus heuristics is [330]. Null moves are described by Donninger [178]. Buro introduced Prob-Cut [114]. Search pathologies, where searching deeper gives worse quality answers, were introduced by Nau. They have been further studied by Pearl, Beal, and others [470, 503, 47, 732, 556].

Chapter 5

Adaptive Sampling

The previous chapter discussed the basic search-eval architecture and the many enhancements to the search function and to the evaluation function. The search enhancements were necessary because of the limitations of fixed-depth fixed-width minimax search. Many of the evaluation enhancements focused on efficiency and the search horizon.

The two main challenges that are addressed in this chapter are (1) adaptive node selection and (2) evaluation of games when it is hard to find an efficient heuristic.

In minimax, all parts of the tree are searched, no matter how un-promising. The enhancements attempted to fix this, to allow a search effort of promising parts of the state space. This approach worked very well for games in which tactical play is important, such as in Chess, Checkers and Othello, where a single move can often change the static score significantly. For other games, notably Go, this approach did not work. Go is a more strategic game where moves can have hidden long term effects, that are not easily visible with a short tactical search as in Chess. For Go, researchers did not find efficient heuristics for variable depth and width search, nor for value estimation of states.

Addressing these two problems required a new algorithm, based on a new paradigm. In this chapter the new algorithm, Monte Carlo Tree Search (MCTS) [150, 108], is introduced. MCTS combines three techniques: it (1) performs adaptive exploration (it is not fixed-width fixed-depth) it (2) uses sampling to determine an evaluation (it does not need heuristics) and it (3) uses averaging as backup function of the statistics to aide node selection (not does not follow minimax at all). MCTS works well for Go and many other applications, although for games such as Chess, Checkers and Othello, alpha-beta with a heuristic evaluation function performs better [108].

Core Problems

- What if there is no efficient heuristic function?

- How to go beyond fixed-depth fixed-width node selection?

Core Concepts

- Monte Carlo sampling
- UCT adaptive node selection
- Averaging statistics backup rule

First we will discuss the history and the challenges that led to the creation of Monte Carlo Tree Search and how, after decades of minimax, a different paradigm emerged. Then we discuss the algorithm, including the UCT selection rule, that governs the exploration/exploitation trade off. Finally, we discuss MCTS enhancements such as RAVE and play out pattern databases.

5.1 Monte Carlo Evaluation

When Kasparov was defeated in 1997 by Deep Blue, computer game researchers went looking for a new *Drosophila Melanogaster*, a new test bed to help them make progress in understanding intelligent reasoning.

Another game of strategy, the game of Go, became the focus of much dedicated research effort. Creating a player for Go seemed a daunting task. The branching factor of Chess is around 35 in the middle game, for Go it is around 200. The average length of a Chess game is 80 moves by a player, in Go it is closer to 250. Indeed, the state space of Chess is estimated to be 10^{47} , for Go it is 10^{170} .

Go programs of around that time played at amateur level. They were like Chess programs but with forward pruning: a highly selective alpha-beta search, and heuristic evaluations based on the concept of territory, as humans are assumed to do [463]. The program GNU Go is a well-known example of this approach, typical designs of that time are described by Müller, Chen and Chen, Boon and Kierulf et al. [463, 133, 86, 349]

Since a full-width look-ahead search is infeasible, most early Go programs used aggressive forward pruning based heuristics: (1) generate a limited number of heuristically likely candidate moves,¹ and (2) search for the optimal policy in this reduced state space [463]. A problem with this approach is that too many good candidates are missed. For evaluation, territory was calculated for each leaf. Such a territory computation used slow flooding-style algorithms and life/death calculations, much slower than the piece-square tables in Chess, necessitating the highly selective search. A different approach was needed.

In 1993, inspired by work on randomized solutions for the traveling salesperson problem, physicist Brügmann wrote a 9×9 Go program based on simulated

¹such as: play in corner, connect via 2 point jump, break ladder

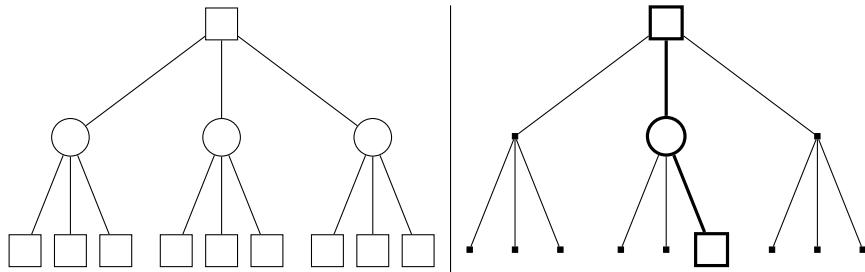


Figure 5.1: Searching a Tree vs a Path



Figure 5.2: Rémi Coulom and Crazy Stone (left)

annealing [109], and in 1990 Abramson explored random evaluations in Othello [4]. These programs did not have a heuristic evaluation function, but played a series of so-called *Monte Carlo playouts*: random moves until the end of the game was reached, where the position was scored as win or loss. This was repeated many times and results were averaged.

This approach differed fundamentally from the minimax approach, where all moves are searched, resulting in an exponential search of a tree of the order of w^d . In Brügmann's program, instead the program searched a linear *path* of size d , see Figure 5.1. Instead of using the minimax function, the program took the average of scores. In contrast to the heuristic knowledge based programs of the time, this program had no heuristic knowledge. Although the program did not play great Go, it played better than random. The field of computer Go generally considered this attempt at connecting the sciences of statistical mechanics and artificial intelligence to be a curiosity. The hand-crafted knowledge based programs performed better. For ten years not much happened with Monte Carlo Go.

Then, in 2003 Bouzy and Helmstetter experimented again with Monte Carlo



Figure 5.3: Sylvain Gelly

Chapter	Algorithm	Select	Backup	Eval
Ch 4	alpha-beta	left-to-right	minimax	heuristic
Ch 5	MCTS	adaptive	average	sample
Ch 6	DQN	-	-	generalize
Ch 7	Self-Play	adaptive	average	generalize

Table 5.1: Heuristic-Sample-Generalize

playouts, again finding that a program can play reasonable Go moves without a heuristic evaluation function [91].

Three years later, with the introduction of Monte Carlo Tree Search (MCTS) a breakthrough occurred: a new kind of tree search was added to the Monte Carlo playouts. Rémi Coulom (Figure 5.2) introduced MCTS as a tree based algorithm [151, 152, 150].² Rules for node selection, expansion, playout and backup were specified.

The early works by Brügmann [109] and Bouzy and Helmstetter [91] showed that averaged random playouts do provide at the least an indication of the quality a position. A “flat” algorithm, with playouts at the root only, did not provide great results, but a recursive, tree based version, combined with a smart exploration-/exploitation selection rule, did.

Two of the first Monte Carlo programs were Sylvain Gelly’s MoGo [234] (Figure 5.3) and Rémi Coulom’s CrazyStone [150]. CrazyStone and MoGo were instantly successful, and many other new Go playing programs were written based on MCTS [151, 128, 127, 464, 194, 39, 230, 539, 396]. Since the introduction of MCTS the playing strength of programs improved rapidly to the level of weak master (2-3 dan) and stronger on the small 9 × 9 board.

There has been a large research interest in MCTS. Browne et al. [108] provide an extensive survey, referencing 240 publications.

²following work by Chang et al. [125], Auer et al. [22] and Cazenave and Helmstetter [123].

Chapter	Name	MDP-Tuple	Reinforcement Learning
Ch 4	alpha-beta	$(S, A, 1, R, 1)$	policy, backup
Ch 5	MCTS	$(S, A, P, R, 1)$	pol, b/u, exploration/exploitation
Ch 6	DQN	(S, A, P, R, γ)	pol, b/u, expl/expl, discount, off-policy
Ch 7	Self-Play	(S, A, P, R, γ)	pol, b/u, expl/expl, discount, off-pol, self-play

Table 5.2: MDP Tuple and Reinforcement Learning in the Chapters

Reinforcement Learning

Let us see how MCTS fits into the general picture of reinforcement learning methods for games (see Table 5.1). MCTS is a reinforcement learning algorithm, as are heuristic planning algorithms. In the Introduction we mentioned three basic methods: heuristics, sampling, and generalization. MCTS represents the sampling method. All three methods are reinforcement learning methods, and with MCTS we see the second main method of reinforcement learning.

Likewise, in Chapter 3 the Markov Decision Process 5-tuple was introduced. Table 5.2 repeats the table from that chapter. We see how the exploration/exploitation trade-off is an integral part of MCTS, in contrast to minimax, where domain specific enhancements were needed to achieve variable-depth search. With MCTS, we are using more elements of MDPs.

5.2 Monte Carlo Tree Search

We will now look at the MCTS algorithm in depth. MCTS consists of four operations: select, expand, playout, back-propagate. See Figure 5.4. Note that the third operation (playout) sometimes goes by the names *rollout* and *simulation*. Back-propagation is sometimes called *backup*. Select is the trial part, backup the error part of the algorithm. We will discuss the operations in order.

Data Structure

As in heuristic planning, in MCTS the state space is tree shaped, starting from the initial state s_0 , using the rules of the game to generate successor states. Unlike in heuristic planning, where the tree is built by recursively adding sub-trees, in MCTS the state space is traversed iteratively, the tree data structure is built in a step by step fashion, node by node. A typical size of an MCTS search is to do 1000–10,000 iterations. In MCTS each iteration starts at the root s_0 , traversing the tree down to the leaves using a selection rule, expanding an extra node, and performing a playout. The result of the playout is then propagated back to the root. During the backpropagation statistics at all internal nodes are updated. These statistics are then used in future iterations by the selection rule to go to the currently “most interesting” part of the tree.

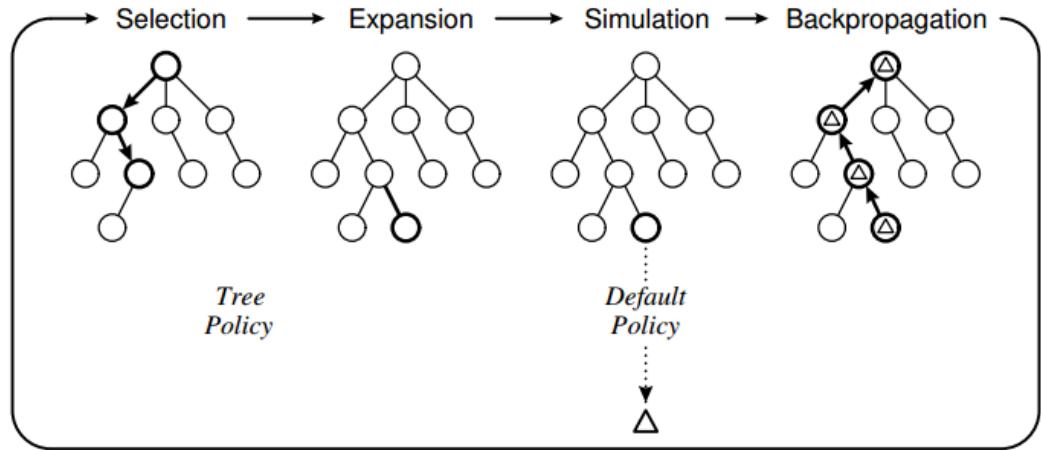


Figure 5.4: Monte Carlo Tree Search [130]

The statistics consist of two counters: the win count w and the visit count v . During backpropagation, the visit count v at all nodes that are on the path back from the leaf to the root are incremented. When the result of the playout was a win, then the win count w of those nodes is also incremented. If the result was a loss, then the win count is left unchanged.

The selection rule uses the win rate w/v and the visit count v to decide whether to exploit high win rate parts of the tree or to explore low visit counts parts. An often used selection rule is UCT, whose formula can be found in Section 5.2.4. It is this selection rule that governs the exploration/exploitation trade-off.

5.2.1 The Four MCTS Operations

Let us now look in more detail at the four operations.

As we saw in the Listing, the main steps are repeated as long as there is time left. Per step, the activities are as follows.

Select

In the selection step the tree is traversed from the root node down until a leaf is reached where a new child is selected that is not part of the tree yet. At each internal state the selection rule is followed which action to take and thus which state to explore next. The UCT rule is the best known selection rule [359].

The selections at these states constitute the policy π of actions in the look ahead simulation leading to the states that currently are the best.

Expand

Then, in the expansion step a child is added to the tree. In most cases only one child is added. In some MCTS versions all successors of a leaf are added to the tree [108].

Playout

Subsequently, during the playout or simulation step moves are played in self-play until the end of the game is reached.³ The reward r of this simulated game is +1 in case of a win for the first player, 0 in case of a draw, and -1 in case of a win for the opponent. Originally, playouts were random (the Monte Carlo part in the name of MCTS) following Brügmann's [109] and Bouzy and Helmstetter's [91] original approach. In practice, most Go playing programs improve on the random playouts by using databases of small 3×3 patterns with best replies [234, 151, 127, 603, 153].

Small amounts of domain knowledge are used after all, albeit not in the form of a heuristic evaluation function.

Backpropagation

In the backpropagation step, reward r is propagated back upwards in the tree, through the nodes that were traversed down previously. Note that two counts are updated: the visit count, for all nodes, and the win count, depending on the reward value.

Finally, when the search budget or the time has run out, the action returned by the MCTS function is the action of the root with the highest visit count, since this node has been favored most often by the selection rule.

Code

The website mcts⁴ contains many useful resources on MCTS, including freely available example code. The following code is from an example Othello program from this site. Listing 5.1 shows the main loop of MCTS [108]. The four main operations of MCTS are clearly indicated.

5.2.2 Policies π

At the end of MCTS, after the required number of iterations has been performed, or when time is up, MCTS return the value and the action with the highest visit count. The action of this initial state s_0 constitutes the policy $\pi(s_0)$ of the reinforcement learning (see Section 3.2.1). It should be pointed out that there are

³Note that this self-play is like the self-play in Chapter 4 where in minimax and alpha-beta the opponent moves for finding the best reply are found by the same algorithm. In Chapter 7 we will see a different kind of self-play, where AlphaZero uses self-play loops to train a neural network evaluation function from scratch.

⁴<https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/>

```

def MCTS( rootstate , itermax ):
    # Conduct an MCTS search for itermax iterations starting from rootstate .
    # Return the best move from the rootstate .
    # Example code from MCTS.AI website
    rootnode = Node(state = rootstate)
    for i in range(itermax):
        node = rootnode
        state = rootstate.Clone()

        # Select
        while node.untriedMoves == [] and node.childNodes != []:
            # node is fully expanded and non-terminal
            node = node.UCTSelectChild()
            state.DoMove(node.move)

        # Expand
        if node.untriedMoves != []:
            m = random.choice(node.untriedMoves)
            state.DoMove(m)
            node = node.AddChild(m,state) # add child and descend tree

        # Playout – this can often be made orders of magnitude quicker
        # using a state.GetRandomMove() function
        while state.GetMoves() != []:
            state.DoMove(random.choice(state.GetMoves()))

        # Backpropagate
        while node != None:
            # backpropagate from the expanded node and work back to the root node
            node.Update(state.GetResult(node.playerJustMoved))
            # state is terminal. Update node with result from POV of node.playerJustMoved
            node = node.parentNode

        # Output some information about the tree – can be omitted
        if (verbose): print rootnode.TreeToString(0)
        else: print rootnode.ChildrenToString()

    return sorted(rootnode.childNodes, key = lambda c: c.visits)[-1].move
    # return the move that was most visited

```

Listing 5.1: MCTS main loop [108]

multiple policies. During the MCTS look ahead simulation a second policy was used. The actions selected by the selection rule in the tree formed a selection policy. Additionally, the actions in the playout phase are a policy. This third type of policy is sometimes called the *default policy*.

5.2.3 Example

Let us illustrate the working of MCTS with an example. Figure 5.5 gives an example of a few MCTS iterations. In the example we see how MCTS traverses the state space in iterations, how the UCT formula adapts between exploiting high win values and exploring unseen parts of the tree, and how the statistics are averaged in the backpropagation phase.

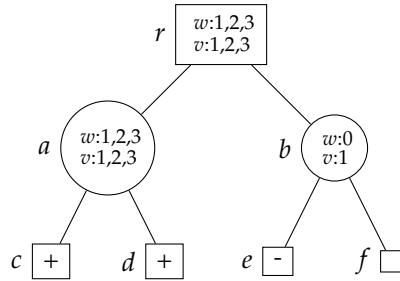


Figure 5.5: MCTS starts at the root, does not select a move because there are no children in the tree, but expands a random child. Let's say a is expanded. Then a random playout is performed at node a . Let's say node d is the first node in the playout. Since this is a tiny state space, the random path that is played out consists only of node d and is a terminal node. The value at node d is a win, so backpropagation increments all win values of the path to the root by 1 (nodes a, r) and also the visit counts. The second iteration starts at root r , where there are unexpanded children so it expands a node, which is node b . At b a playout is performed, which ends up at, say, e , whose value is a loss. This gets propagated (no win values change, and the visit counts of b, r are incremented). Next, the second iteration starts. At the root all children are expanded, so we select a child. We select a since its UCT value is $1/1 + 1 \times \sqrt{\ln 1/1} = 2$, and b 's value is $0/1 + 1 \times \sqrt{\ln 1/1} = 0$. A playout is performed at a , let's say it traverses node c , which is also a win. The values are updated (a, r 's win and visit count are incremented to 2 and 2). Next, the third iteration selects node a since the UCT value of a is $2/2 + 1 \times \sqrt{\ln 2/2} = 1.588$ and b is $0/1 + 1 \times \sqrt{\ln 2/1} = 0.832$. It finds that a has no children, so it expands a child, say node c . It then performs a playout and finds a win, and updates the win and visit values of c, a, r accordingly. It continues with more iterations until time is up.



Figure 5.6: Levente Kocsis and Csaba Szepesvári

5.2.4 UCT Selection Rule

The adaptive exploration/exploitation behavior of MCTS is governed by the selection rule. One of the most popular selection rules is the UCT formula.

UCT was introduced in 2006 by Kocsis and Szepesvári [359] (see Figure 5.6). They provide a theoretical guarantee of eventual convergence to the minimax value. The selection rule was named UCT, for Upper Confidence bounds for multi armed bandits applied to Trees.

The exploration/exploitation trade-off is central in many reinforcement learning algorithms. The selection rule determines the way in which the current values of the children influence which parts of the tree will be explored more. Many rules have been proposed for MCTS. It is important that the rule strikes a good balance. UCT has a tunable parameter to shift between more or less exploration. The UCT formula is as follows:

$$\text{UCT}(j) = \frac{w_j}{n_j} + C_p \sqrt{\frac{\ln n}{n_j}}$$

where w_j is the number of wins in child j , n_j is the number of times child j has been visited, n is the number of times the parent node has been visited, and $C_p \geq 0$ is a constant (the tunable exploration/exploitation parameter). The first term in the UCT equation, the winrate, is the exploitation term. A child with a high winrate receives through this term an exploitation bonus. The second term is for exploration. A child that has been visited infrequently has an exploration bonus. The level of exploration can be adjusted by the C_p constant. A low C_p does little exploration, a high C_p has more exploration. Section 5.3.1 goes deeper into the choice for values for C_p . The selection rule then is to select the child with the highest UCT sum (the standard argmax function).

The UCT formula balances *winrate* ($\frac{w_j}{n_j}$) and “*newness*” ($\sqrt{\frac{\ln n}{n_j}}$) in the selection of nodes to expand. Many alternative selection rules have been proposed. Alternatives for UCT are variants such as Auer’s UCB1 [21, 22, 23] and P-UCT [544, 430]. Most rules are based on bandit theory, which we will explain next.



Figure 5.7: A Multi-Armed Bandit

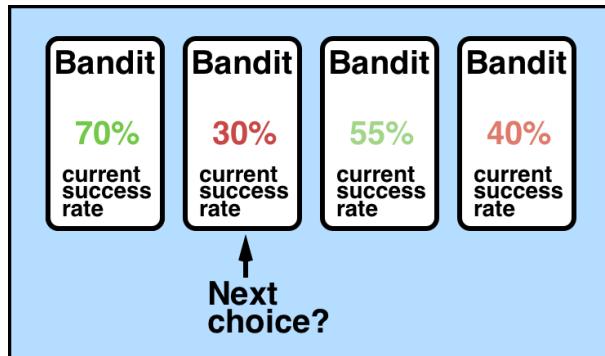


Figure 5.8: Explore or Exploit? [735]

Multi Armed Bandit Problem

The work on UCT and Upper Confidence Bounds (UCB) is based on bandit theory, a topic well studied in the field of stochastic scheduling and decision theory [22, 343, 125]. The theory provides a systematic means to optimize the search strategies to explore more promising parts of the tree, while still ensuring exploration. Seminal work has been done by Lai [381, 382], proving important optimality results.

A multi-armed bandit is a mathematical model whose name refers to a casino slot machine, but with more than one arm (Figure 5.7). Each arm can have a different payout. The probability distribution of the payout of the arms is also not known. As arms are played, the player gets more information as to their payout probability. However, each play also costs money. The multi-armed bandit problem then is to select the best arm in as few tries as possible (Figure 5.8). Multi-

armed bandit theory thus analyses situations of short term cost versus long term gain.

Consider the following situation: One arm gives 10 dollars with probability 0.1, while another gives 1000 dollars with probability 0.002. After 50 tries, the first arm probably yielded some win, whereas the second probably did not. You may then be forgiven for playing mostly the first arm, but this is not optimal in the long run, since the expected reward of the second one is actually twice as high. But to learn this, you will need to sample the second arm for quite a while.

Multi-armed bandits have a classic reinforcement learning exploitation/exploitation trade-off: should I play the arm that I know to give high payout so far, or should I play arms whose payout I do not know but may have higher payout? In a seminal paper Lai and Robbins [381] construct an optimal selection policy (optimal in the sense that it has maximum convergence rate to the arm with the highest mean) for a case where the reward distributions of the arms is known. An active field of research subsequently studied relaxations of this condition.

In MCTS the multi-armed bandit problem is applied in a recursive setting, in a tree of decisions. Each state is a bandit, and each action (child) is an arm.

5.2.5 The Impact of MCTS

Before 2006, mainstream game programming research was based on heuristic planning. The minimax backup rule was the main paradigm, spawning a large effort to formalize and code heuristic human knowledge. The conventional wisdom was that human knowledge should be captured in heuristics, and that no move should be missed (due to the tactical nature of the popular perfect information games that were used at the time). There was no overarching principle behind the many heuristic enhancements, except that they should be efficient to code, and a means to make the search more selective should be tried, in order to overcome the horizon problem.

When MCTS was introduced, it provided an elegant framework for selective search, amenable to theoretical analysis and successful in practice. MCTS follows a different, non-exponential paradigm. Minimax is based on searching all successor states and is an inherently exponential tree searcher, with a $\sqrt{w^d}$ time complexity. MCTS is a sampling algorithm. It searches paths from the root to the terminals, not trees. This non-exponential principle circumvents the need for the plethora of heuristic search enhancements trying to curtail the fundamental exponential time complexity of minimax. Equally important, the sampling also removed the need for heuristic evaluation.

The success of the new approach of selective playout sampling created a paradigm shift in the world of computer game playing. In Go, and many other games, it turned out that some moves *can* be missed, or rather, that *most* moves can be missed, as long as on average a reliable picture of the state space emerges. The paradigm changed from an exponential $O(w^d)$ to a polynomial $O(d)$ complexity. Exponential worked for Chess, polynomial worked for Go.⁵ The new

⁵In practice, both minimax and MCTS are made into anytime algorithms, that produce a best move

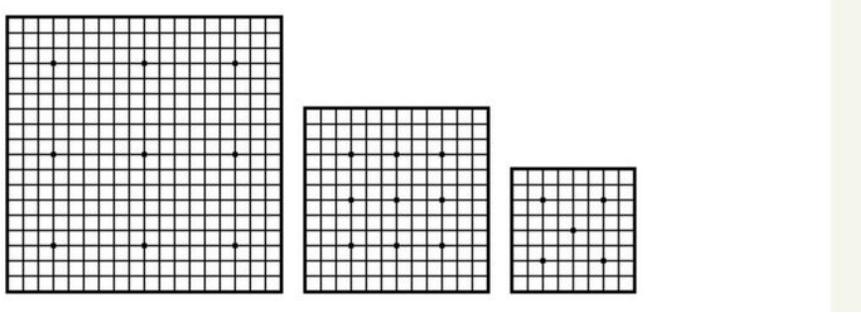


Figure 5.9: Go Boards 19×19 , 13×13 , 9×9

paradigm worked so well, that no human knowledge in the form of heuristics was needed (or at least, initially).

The dominant language of the field changed with the introduction of MCTS. The graph-based language of nodes, leaves, and strategies, changed to state, action, and policy.

Applications of MCTS

The introduction of MCTS improved performance of Go programs considerably, from medium amateur to strong amateur. Where heuristics-based GNU Go played around 10 kyu, Monte Carlo programs progressed to 2-3 dan⁶ in a few years time.

On the small board, 9×9 , Go programs achieved very strong play (see Figure 5.9 for the standard Go board sizes). Figure 5.10 shows a graph of the improvements in the performance of Go programs on the large 19×19 board from [230]. Performance did not improve much beyond this level, despite much effort by researchers. It was thought that perhaps the large action space of the 19×19 board may be too hard for MCTS. Many enhancements were considered, for the playout phase, and for the selection. One of the enhancements that was tried was the introduction of neural nets. As we will see in Chapter 7, in 2015 this became successful.

MCTS revolutionized the world of heuristic search. Previously, the paradigm had been that in order to achieve best-first search, one had to find a domain specific heuristic to guide the search in a smart way. With MCTS this was no longer necessary. Now a general method existed that could find the promising parts of the search without a domain specific heuristic, just by using statistics of the search itself.

MCTS quickly proved successful in other games, both two-agent and single-agent: for video games [128], for single player applications [565], and for many

when stopped. For minimax, iterative deepening transforms it into an anytime algorithm, for MCTS, the algorithm is already an anytime algorithm that performs an iteration of playouts.

⁶Absolute beginners in Go start at 30 kyu, advancing to 1 kyu. Stronger players then achieve 1 dan, progressing to 7 dan, the highest amateur rating for Go. Professional Go players have a rating from 1 dan to 9 dan, written as 1p–9p.

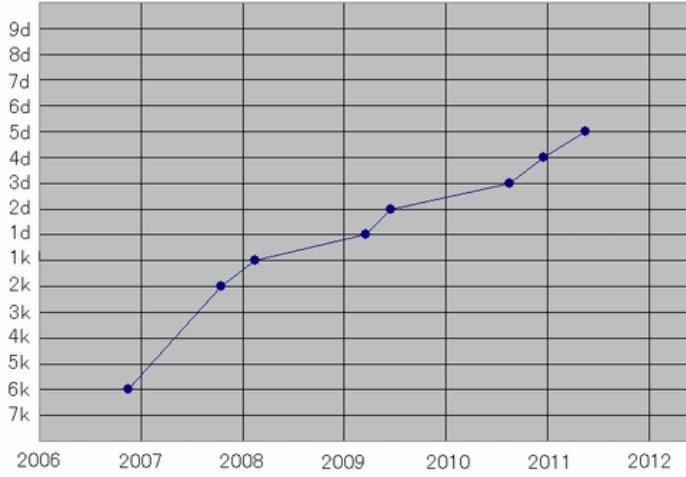


Figure 5.10: Performance of MCTS programs [230]

Name	Principle	Applicability	Effectiveness
expl/expl	expl/expl/N	small/large searches	all MCTS
RAVE	statistics sharing	early phase	all MCTS
play-out db	small patterns	domain specific	domain dependent
adaptive expl/expl	variable C_p	general	all MCTS
parallelism	parallel search	general	all MCTS

Table 5.3: MCTS Enhancements

other games such as Einstein würfelt nicht [418], Settlers of Catan [645], Hannnah [417], Amazons [353], and mathematical games such as the Voronoi game [92].

5.3 *MCTS Enhancements

Basic MCTS already showed much promise, and researchers have immediately found ways to improve performance of the basic algorithm further. They focused on action selection and playout. We will discuss a few of them. Table 5.3 shows MCTS enhancements that we will discuss.

First we will discuss tuning of C_p , the exploration/exploitation parameter, and then we will discuss RAVE, a method to speedup the distribution of statistics at the early stage of an MCTS search. After that, we will discuss play-out databases, adaptive C_p , and parallelism.

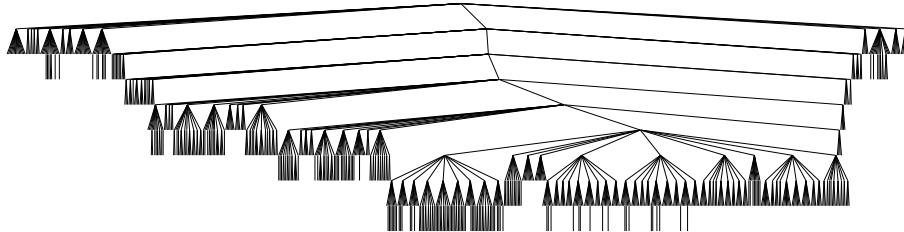


Figure 5.11: Adaptive MCTS tree [374]

5.3.1 Exploration/Exploitation

The search process of MCTS is guided by the values in the tree. MCTS discovers during the search where the promising parts of the tree are. The tree expansion of MCTS is an inherently variable-depth and variable-width search process. To illustrate this we show a picture of a snapshot of the search tree of an MCTS optimization from [374, 697]. In Figure 5.11 we see that some parts of the tree are searched more deeply than others. The tree in the illustration is part of an expression optimization problem in a symbolic algebra system [696].

A key element of MCTS is the exploration/exploitation trade-off that can be tuned with the C_p parameter. The effectiveness of MCTS depends greatly on the choice of this parameter. In follow-up research Kuipers et al. [374, 697] have performed experiments to plot the relation between C_p and the number of MCTS node expansions N . When C_p is small, MCTS favors parts of the tree that have been visited before and are known to be good; when C_p is large, unvisited parts of the tree are favored. Figure 5.12 shows four plots of an optimization for a large symbolic algebra polynomial. The plots show results from 4000 of optimization runs, each run is one dot, all starting with a different random seed. On the y -axis the number of operations of the optimized polynomial is shown, the lower this number is, the better. The lowest number found was close to 4000. This minimum is achieved in the case of $N = 3000$ iterations for a value of C_p with $0.7 \leq C_p \leq 1.2$. Dots above this minimum represent a sub-optimal search result.

For 300 iterations per data point (left upper panel), some structure is visible, with a minimum emerging at $C_p \approx 0.4$. With more tree expansions (see the other three panels) the picture becomes clearer, and the value for C_p for which the best answers are found becomes higher, the picture appears to shift to the right. For low numbers of tree expansions (see again upper left panel) there is almost no discernible advantage of setting the exploration/exploitation parameter at a certain value. For 1000 iterations (see upper right panel) MCTS works best when exploitative (the left part of the plot achieves the lowest number of operations). As the number of iterations N is larger (the two lower panels) MCTS achieves better results when its selection policy is more explorative, to try to get out of local minima. For the graphs with iterations of 3000 and 10000 the range of good results for C_p becomes wider, indicating that the choice between exploration/exploitation

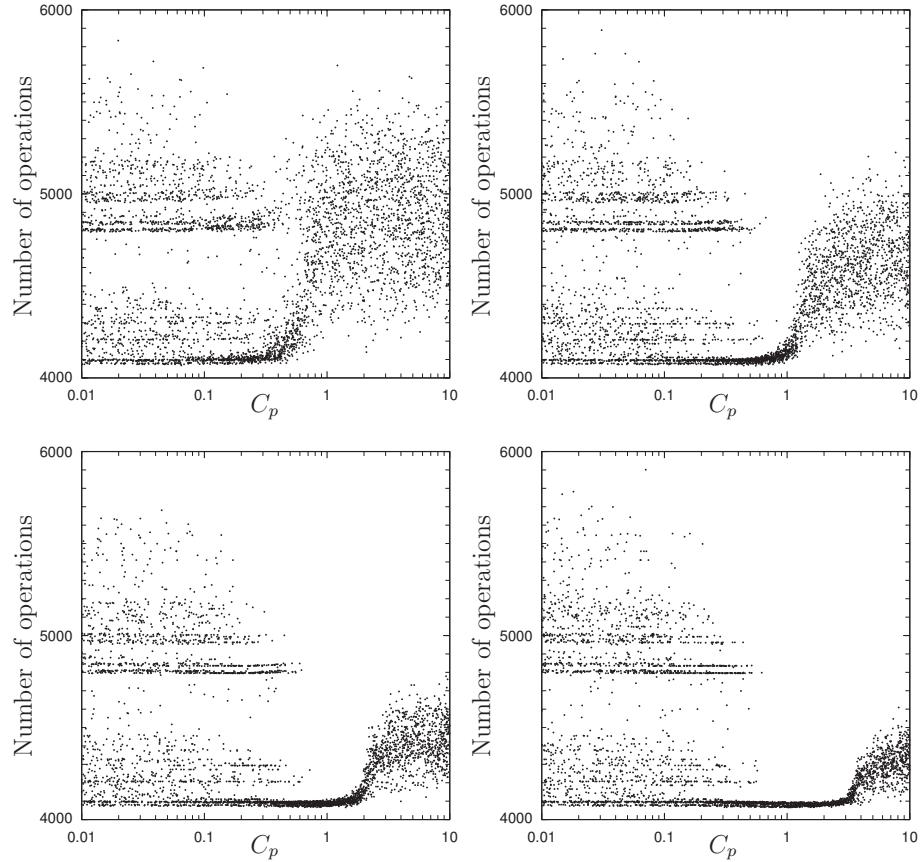


Figure 5.12: Scatter plots of C_p for $N = 300, 1000, 3000, 10000$ node expansions per MCTS run. Each dot is one MCTS run to optimize a large polynomial. Lower is better. Exploitation (small C_p) works best in small trees ($N = 300$), exploration in large trees ($N = 10000$) [374]

becomes less critical.

For small values of C_p , such that MCTS behaves exploitatively, the method gets trapped in one of the local minima as can be seen from scattered dots that form “lines” in the left-hand sides of the four panels. For large values of C_p , such that MCTS is more exploratively, many of the searches do not lead to the global minimum as can be seen from the cloud of points on the right-hand side of the four panels. For intermediate values of $C_p \approx 1$ MCTS balances well between exploitation and exploration and finds almost always an ordering that is very close to the best one known [374, 697].

Selection: RAVE

One of the best known enhancements to MCTS is RAVE, or Rapid Action Value Estimation. RAVE addresses the problem that initially, at the start of the search, all visit and winrate counts of the nodes are zero and MCTS is expanding without little guidance of where to go. The early part of MCTS thus progresses in the dark. The sooner search information is available throughout the tree, the better the children that will be selected and the more efficient the search will be. It is important to quickly learn values for many actions. Two closely related techniques have been developed: All Moves As First (AMAF) and Rapid Action Value Estimation (RAVE). They allow for the node statistics (win count and visit count) to be updated quicker.

Already in [109] Brügmann gives the first description of AMAF in his Monte Carlo Go. Gelly and Silver [231] report on the use of AMAF in MCTS, combining it with tree search and UCT, in a Go program. Helmbold and Parker-Wood [281] also find AMAF to be effective in Go. An AMAF update of the win and visit counts updates not only the counts of nodes on the playout path to the root of the MCTS tree, but also siblings of those nodes that occurred at any position in the playout. Statistics are updated for all actions of a state that occurred in the playout. Since playouts tend to touch about half the board for a side, this means that many extra nodes in the MCTS tree are updated by AMAF. AMAF works in the backup operation, in order to influence node selection in the UCT formula. Through AMAF many more nodes receive the statistics, allowing UCT to make a more informed exploration/exploitation decision.

The cost of AMAF is a loss of precision. Branching out the win and visit statistics to nodes that are not exactly on the path, but are loosely related, reduces precision of the algorithm. A better situation would be to only use AMAF at the start of the search, as warm up, and then to not use AMAF anymore when MCTS has sufficient statistics to work well. This is the approach that RAVE follows. At the start it uses AMAF-like spreading of updates, whose use is governed by a parameter that decreases as nodes are expanded. See [281, 127, 108] for a more in depth discussion of AMAF and its variants, and [231, 233] for their variant RAVE. Gelly and Silver further extend the UCT formula to take advantage of prior knowledge, making use of pattern databases that are trained from grand master games.

Another enhancement concerns the play-out phase.

Play-Out: Pattern Database

The playout phase chooses moves randomly, this achieves quick play-outs and no heuristic has to be devised. However, should such quick heuristics exist, they will almost surely achieve better play than random moves.

Most Go programs replace random playouts by heuristics or databases of small (often 3×3) patterns. The patterns are learned off-line by supervised machine learning from grand master games. Gelly et al. [234, 232] describe how pattern databases are used for better playouts in MoGo, achieving grand master performance on the small 9×9 Go board.

Adaptive Exploitation/Exploration

The C_p parameter of MCTS governs the exploration/exploitation trade off. Ruijl et al. [551] describe a scheme in which C_p decreases towards zero over the course of the search, starting out favoring exploration, and becoming more exploitative towards the end of the search. This focuses search effort towards building deeper trees towards the end. It is reported that this is beneficial for an application where large formulas are optimized with MCTS [551, 552, 550].

Such adaptive parameters are also used in simulated annealing, ϵ -greedy exploration, and other reinforcement learning methods.

Ensembles and Parallelism

One way to improve performance without actually having to change the algorithm, is to parallelize it. A parallel algorithm can run faster when parallel hardware is available, such as a multi-core or cluster computer.

MCTS consists of many search iterations starting at the root. These searches are semi-independent. Through the win and visit counts the iterations communicate search results. The searches can in principle be run independently, although the individual iterations miss out on some of the accumulated win and visit statistics in the other iterations. Nevertheless, MCTS appears quite well-suited for speedup through parallelization. Indeed, most large Go programs run some of the iterations in parallel.

Traditionally, the literature identifies three approaches to parallelization: root parallelism, tree parallelism, and leaf parallelism [129, 445]. In root parallelism, processors perform MCTS searches independently, not sharing a tree (thus reducing synchronization and communication, at the cost of less efficient searches through no sharing of information between searches). In tree parallelism the tree is shared, at the cost of more synchronization and communication, but also leading to a more efficient search. Leaf parallelism again shares no information, and performs independent playouts in parallel.

Mirsoleimani et al. [443, 447] provide a unified view on tree and root parallelism, solving some of its problems, and allowing hybrid versions of root and tree parallelism to be created with Pipeline Parallel MCTS.

Although root parallelism may seem inefficient due to the lack of sharing of information, there is an interesting relation with ensemble search. Fern and Lewis [204] and Mirsoleimani et al. [442] study ensembles of independent MCTS searches and the relation with exploration, in which the tree is searched independently. Ensembles of MCTS searches explore more of the tree than a single MCTS search does, because the searches are independent. Root parallel searches are also independent, since they share no information, just as ensemble search. Thus, if one would like to achieve a certain exploration/exploitation balance in a parallel setting, all one has to do is to use root parallelism with a reduced C_p parameter. This dials back the exploration of each individual search, which one gets for free because of the higher exploration inherent in the independent (ensemble or parallel) searches.

Roll Out Algorithms

We are at the end of this section on MCTS enhancements. There is a final observation to make, which is not an enhancement, but more a philosophical note, about algorithms that are considered very different but are less so when viewed from a different perspective.

Alpha-beta and MCTS are usually regarded as algorithms from different classes, of an almost incomparable nature. Alpha-beta is a depth-first tree traversal algorithm. MCTS is a path-traversal algorithm, iteratively invoked until the search budget is exhausted, starting from the root and returning to it after each pass. Alpha-beta is rigid left-to-right, MCTS is inherently adaptive and selective, using UCT.

MCTS is part of a family of *roll out* algorithms [73]. In roll out algorithms the starting point is a policy, the base policy, whose performance is evaluated in some way, possibly by simulation. Based on that evaluation, an improved policy is obtained by one-step look-ahead.

In 1995, Rivest [540] published an iterative approach for approximation of the minimax value, with good results. Twenty years later, inspired by MCTS, Huang [304] presents a roll out formulation (path-based) of alpha-beta. In this formulation the algorithmic structure of the roll out version of alpha-beta and MCTS are strikingly similar. The two differences are the selection rule (trial) and the type of values that are backed-up (error). Alpha-beta has a left-first selection rule, MCTS selects the child with the highest UCT value. Alpha-beta backs-up upper bounds and lower bounds, MCTS backs-up node counts and win counts.

Listing 5.3 shows alpha-beta roll out code in pseudo-Python (after [304]).⁷ Listing 5.2 shows MCTS in compact pseudo-Python (after [304]).

Huang's formulation is important because it elegantly captures the difference between these two important algorithms in a single algorithmic framework. His formulation also highlights that the two algorithms have more in common than previously assumed, even though they come from different paradigms.

Conclusion

Basic alpha-beta is a fixed-depth fixed-width algorithm. It spends all search effort equally in the tree. Researchers had to come up with enhancements to break out of this rigid mold, to allow targeting search effort to interesting parts of the tree. Enhancements such as forward pruning, null-move pruning, search extensions and quiescence search are needed to achieve variable-depth and variable-width search. In contrast, MCTS is inherently variable-depth, variable-width. It uses an intricate selection rule to find parts of the tree that are promising, trading-off win rate of explored states versus exploring new states, achieving an adaptive kind of greedy search (improving on the basic fixed ϵ -greedy model of Section 3.3.2).

⁷Note that the code is pseudo-Python. It looks like Python, but, for the sake of compactness, some essential functions are missing, such as initializations of counters and bounds, tree manipulation code, and max and min functions for all children. The listing contain some code duplication to highlight similarity of algorithmic structure.

```
# python-like pseudo-code of MCTS. After [Huang 2015]
def mcts(root, budget):
    while root.n < budget:
        g = rollout(root)
    return g

def rollout(s):
    if s.isleaf():
        g = playout(s)
    elif s.ismax():
        c_select = s.argmax_c(c.avg + cp * sqrt(log(s.n)/(2*c.n)))
        g = rollout(c_select)
        s.avg = s.n/(s.n+1)*s.avg + 1/(s.n+1)*g
        s.n   = s.n + 1
    elif s.ismin():
        c_select = s.argmin_c(c.avg - cp * sqrt(log(s.n)/(2*c.n)))
        g = rollout(c_select)
        s.avg = s.n/(s.n+1)*s.avg + 1/(s.n+1)*g
        s.n   = s.n + 1
    return g
```

Listing 5.2: Roll out MCTS in pseudo-Python [304]

```
# python-like pseudo-code of alpha-beta. After [Huang 2015]
def alphabeta(root):
    # all bounds are initialized to (-inf, +inf) in initialization code (not shown)
    while root.lb < root.ub:
        rollout(root, root.lb, root.ub)
    return root.lower

def rollout(s, alpha, beta):
    if s.isleaf():
        s.lb = s.playout()
        s.ub = s.playout()
    elif s.ismax():
        c_select = first([c for c in s.child() if max(alpha, c.lb) < min(beta, c.ub)])
        rollout(c_select, max(alpha, c_select.lb), min(beta, c_select.ub))
        s.lb = max([c.lb for c in s.child()])
        s.ub = max([c.ub for c in s.child()])
    elif s.ismin():
        c_select = first([c for c in s.child() if max(alpha, c.lb) < min(beta, c.ub)])
        rollout(c_select, max(alpha, c_select.lb), min(beta, c_select.ub))
        s.lb = min([c.lb for c in s.child()])
        s.ub = min([c.ub for c in s.child()])
    return
```

Listing 5.3: Roll out alpha-beta in pseudo-Python [304]

MCTS was first developed for Go. Soon other games and optimization applications were also found to benefit from the flexible adaptivity of MCTS and especially from the fact that no domain specific heuristic evaluation function was needed. See for some examples [108, 19, 128, 611, 374, 127, 457, 458].

5.4 Practice

Below are some questions to check your understanding of this chapter. Each question is a closed question where a simple, one sentence answer is possible based in the text in this chapter.

Questions

1. How are Go players ranked?
2. What is the difference between tactics and strategy in games?
3. What is the difference between exponential and polynomial time complexity? Can you name an algorithm for each?
4. What are the four steps of MCTS?
5. Describe two advantages of MCTS over rigid heuristic planning.
6. How is MCTS recursive?
7. Describe the function of each of the four operations of MCTS.
8. Give the UCT formula.
9. How does UCT achieve trading off exploration and exploitation, which inputs does it use?
10. When C_p is small, does MCTS explore more or exploit more?
11. For small numbers of node expansions, would you prefer more exploration or more exploitation?
12. What does RAVE do?
13. What is the role of pattern databases in the play-out phase?
14. Give three ways in which the performance of MCTS can be enhanced.
15. What is a roll out algorithm?

Exercises

For the exercises in this chapter we will use two code bases: Browne et al. [108] and AlphaDoge. The MCTS implementation of Browne is simple, clear and well suited to learn MCTS with. The example code implements Othello and Nim, but not Go. The AlphaDoge implementation does implement Go, and is more advanced, but will also take more time to study.

1. Go to here⁸ and download the code. Familiarize yourself with the code, and play a game of Othello against the MCTS program as preparation for the other exercises.
2. Study the impact of search effort. Make a graph of Elo rating on the y -axis and N the number of node expansions on the x -axis (play different N numbers of node expansions against each other).
3. Study the impact of exploration and exploitation. For a fixed search budget, plot Elo rating for four different C_p values: $\{0.1, 0.7, 1.0, 3.0\}$.
4. Implement a Hex game player with MCTS. See the Hex page⁹ for information. See also [106, 274].
5. Go to AlphaDoge¹⁰ and download the code. We will only use the MCTS player, not the AlphaZero trainer. Play a game against the computer on 9×9 and on 19×19 . Who won? Is your computer fast enough for the big board?
6. Do the preceding Othello exercises (1-3) also for 9×9 Go.

5.5 Summary

In this chapter the dual challenges of (1) rigid search and (2) lack of an efficient heuristic evaluation function were addressed. We introduced a new paradigm, the adaptive sampling paradigm, with the MCTS algorithm. Where minimax needed enhancements and tweaks to focus the search effort on the important part of the state space, MCTS is inherently selective. The most popular selection rule is UCT, which provides a good exploration/exploitation trade-off. UCT is based on a large body of bandit theory.

The development of MCTS was driven by Go, which emerged as the new Drosophila of AI, after Deep Blue beat Kasparov in Chess.

The fact that MCTS does not need a heuristic evaluation function allows it to work for Go, and for other domains for which no efficient heuristic function was found. MCTS caused a breakthrough in Go performance, achieving master level play on the small 9×9 board.

⁸<https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-a/>

⁹<https://www.maths.ed.ac.uk/~csangwin/hex/index.html>

¹⁰<https://github.com/VainF/AlphaDoge>

Traditionally, MCTS has four operations: select, expand, playout, and backup (although we also showed a more compact roll out formulation, providing a link to alpha-beta [304]).

As always, enhancements are important to achieve the highest possible performance. Two well-known enhancements are RAVE, to speedup the propagation of node statistics in the first part of the search, and databases of small patterns in the playout operation.

The previous chapters focussed on model-based reinforcement learning. Heuristic planning and adaptive sampling need a simulator with the rules of the game as the transition function.

Historical and Bibliographical Notes

A highly cited survey on MCTS is Browne et al. [108]. It has more than 200 references to papers on MCTS. Coulom introduced the MCTS algorithm [150], and he implemented it in his program Crazy Stone. Gelly et al. wrote another successful Monte Carlo based program, MoGo [234, 230], which was successful in tournaments against other programs, and against human Go players. Gelly and Silver published an influential paper on pattern databases and AMAF/RAVE [231, 233].

Of prime importance in MCTS is the selection rule, which gives its adaptive selectivity. UCB1 and UCT are selection rules based on bandit theory, derived in [359, 21]. In their seminal 2006 paper, Kocsis and Szepesvári introduce UCT [359]. Most Go programs use some form of parallel search. There has been quite some research into parallelization of MCTS, see, e.g., Chaslot et al. and Mirsoleimani et al. [129, 447]. Ensemble MCTS is related to parallel MCTS and exploration, see [204].

On-line there are many resources to be found for programming a computer Go program. At Senseis there is a wealth of information on how to write your own Go program.¹¹

¹¹<https://senseis.xmp.net/?ComputerGoProgramming>

Chapter 6

Function Approximation

In this chapter we focus on model-free reinforcement learning, when no transition function and no reward function are known. We focus on end-to-end feature learning, without hand-crafted heuristics and without sampling. The main challenge in this chapter is to find functions that can provide good approximations of the value of large, high-dimensional, state spaces.

This chapter is about learning an evaluation function through automatic feature discovery. The techniques that we describe in this chapter are responsible for much of the large interest in AI. Deep learning has been able to find approximation functions for image classification that classify images better than humans. In games, deep learning has found policy functions that play vintage Atari arcade games. Deep learning has been very successful. It is a driving force behind the recent AI revolution.

Deep learning is a method for automated discovery of features to approximate an objective function. This objective function can be a regression function, classifier, a value function or a policy function. Regression and classifiers are typical for supervised image recognition tasks, and value and policy functions are used in reinforcement learning.

The previous chapters discussed hand crafted heuristic functions. If there was learning, then typically the coefficients of a polynomial of hand crafted heuristic features were learned, or the importance of patterns for the play out phase of MCTS.

The aim of this chapter is more ambitious: here we focus on *end-to-end* learning, the learning from output labels or actions based directly on the raw inputs, un-pre-processed, without intermediate hand crafted heuristics. End-to-end learning is computationally intensive. It has been made possible by advances around the turn of the millennium in compute power in CPUs and especially in GPUs. Furthermore, around that time large data sets of hand-labeled training data were developed, providing the necessary training data. Finally, algorithmic advances solved key problems in deep learning. Together with the increase in compute power and big data these developments enabled the deep learning revolution.

To quote one of the central works of this chapter, Mnih et al. [453]: “Recent breakthroughs in computer vision and speech recognition have relied on efficiently training deep neural networks on very large training sets. The most successful approaches are trained directly from the raw inputs, using lightweight updates based on stochastic gradient descent. By feeding sufficient data into deep neural networks, it is often possible to learn better representations than hand crafted features.”

Altough our goal in this book is reinforcement learning, many of the advances in deep reinforcement learning started in supervised learning. Therefore the first part of this chapter is devoted to an introduction of the advances in deep supervised learning. A good understanding of deep learning is important in modern reinforcement learning, we will therefore spend quite some effort on this topic.

Core Problems

- Can we find end-to-end value/policy functions for large, high-dimensional, state spaces?

Core Concepts

- Automated feature discovery through deep learning
- Convolutional neural network (CNN) and Deep Q-learning network (DQN)
- Over-fitting and stable learning

In this chapter we focus on two pieces of research at the center of the deep learning breakthrough. For supervised learning it is the 2012 work by Krizhevsky et al. [372] on end-to-end image recognition. For reinforcement learning it is the 2013/2015 work by Mnih et al. [453, 454] on end-to-end playing of Atari video games. For both we first discuss the main approach, and then we look at enhancements and more advanced algorithms. We will now start with supervised learning.

6.1 Deep Supervised Learning

Towards the end of Chapter 3, in section 3.3.1 we discussed ways to approximate functions when the state space becomes so large that it is unlikely that states will have been seen before in training. Statistical learning theory provides a framework for machine learning methods such as supervised learning and reinforcement learning [554, 225].

Supervised learning is the task of learning a regression or classification function that maps an input to an output based on example input-output pairs [554, 245, 80]. For example, the training set may be a set of pictures, labeled with a

Chapter	Algorithm	Select	Backup	Eval
Ch 4	alpha-beta	left-to-right	minimax	heuristic
Ch 5	MCTS	adaptive	average	sample
Ch 6	DQN	-	-	generalize
Ch 7	Self-Play	adaptive	average	generalize

Table 6.1: Heuristic-Sample-Generalize

description, such as *CAT* or *DOG*. Because the learning process learns a function from an explicit set of labeled training data it is called *supervised*, since the labels supervise the learning process.

6.1.1 Generalization and Features

Supervised learning is a basic and widely studied problem in artificial intelligence. A wide range of algorithms exist for supervised learning: Support Vector Machines, linear regression, Bayesian classification, decision trees, random forests, nearest neighbor, and artificial neural networks, to name just a few [554, 80].

Since around 2010 artificial neural networks have achieved impressive results in image recognition, speech recognition, and game playing. We will focus on deep neural networks in this chapter.

We will first discuss learning as a generalization task, and then how generalization can be viewed as a feature discovery task.

Learning as Generalization from Training Set to Test Set

Learning is the adaptation of behavior from experience. Supervised learning is concerned with training for an input/output mapping, to categorize input data into a number of output categories or regression function. Supervised learning is closely related to generalization, which is the formulation of common properties in instances of data. Table 6.1 shows (again) the main paradigms of this book, and how generalization is related to the other paradigms.

Supervised learning learns a continuous regression or a discrete classification function. Regression finds the relation between examples and a continuous variable, classification finds the relation between examples and a set of discrete classes. When the classification domain is small enough that all test examples might be seen at training time, the function can be learned exactly. When the application domain is so large that test examples will not have been seen at training time, then the function must be approximated.¹

A standard method in machine learning to assess the quality of a categorization algorithm is to divide the data set in a training part and a test part (*k*-fold cross validation) [80]. A typical split is 90% of the data for the training set, and the

¹Also in continuous regression problems the test examples will not be seen at training time, and approximation is necessary.

remaining 10% for the test set. The algorithm is trained on examples from the training part. The quality of the resulting approximation function is then tested on the remaining (unseen) 10%, to see how well it generalizes. If the percentage of correct predictions on the test set is about the same as on the training set, then the function generalizes well to the test set. If not, the training process may have under-fitted or over-fitted (captured too little of the signal or too much of the noise, see Section 6.1.7).

Deep learning provides methods to generalize over data to approximate regression and classification functions. It does so by discovering common features. Let us look how.

Generalization as Feature Discovery

Generalization can be achieved by the discovery of common features. Traditionally, feature discovery is a labor intensive task. Common features in a data set are painstakingly identified by hand, by one by one looking at the examples in the data set, and by writing small algorithms that recognize the features. For image data, such hand-identified features are typically simple features such as lines, squares, circles, and angles. These features can then be used in decision trees to construct recognizers to classify an image.

As we shall see in the next subsections, deep learning has allowed this process to be automated. We have just described machine learning as a generalization task. In generalization we extract the essence of a concept by looking for similarities with other objects. By finding such commonalities we can simplify the instances into a smaller number of higher-level concepts. These concepts are more abstract, more general, than the many lower level instances.²

In the context of AI and generalization, the commonalities are called features, and the process of learning higher level abstractions is called representation learning [393]. Deep nets can perform feature recognition through a layered hierarchy of increasingly abstract representations [287, 393]. Today deep learning is one of the core technologies that are used in automated function approximation.

From a scientific point of view, the hand crafting and hand tuning is undesirable as it is hard to reproduce. Hand-crafting depends on expert knowledge, in contrast to the training of approximators by an optimization algorithm such as stochastic gradient descent.

Furthermore, and most importantly, the hand crafted heuristics do not generalize well to other domains, whereas deep feature recognizers have been shown to work well in many domains (although they typically have to be re-trained for each domain).

Finally, for some large problem domains exist where our domain-intuition fails to come up with efficient features. We may not understand our own human interpretations well enough to come up with efficient algorithms. This was the case in Go, as we saw in the previous chapter.

²Feature discovery is related to dimensionality reduction, a technique that can be used for visualization of complex data [547, 650, 423].

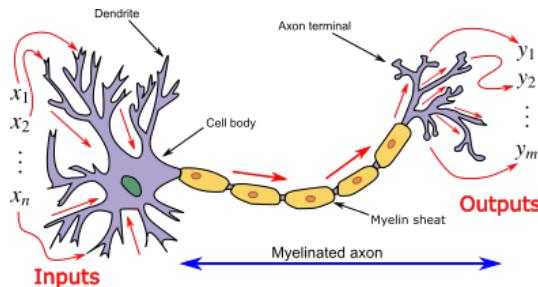


Figure 6.1: Biological Neuron [709]

There is a flip side to the advantages of automated feature discovery with deep learning, since for many domains much time has to be spent finding a neural network architecture for the domain that works, before the training process results in features that perform well. The time spent hand crafting heuristic features is now spent hand crafting network architectures to automatically learn the features.³

End-to-end learning, or the discovery of features directly from high dimensional raw pixel data is computationally quite demanding, as opposed to learning based on intermediate (lower dimensional) hand crafted features. You may recall that in Chapter 4 heuristic value functions were used to search the large state spaces of Chess and Checkers (section 4.1.3). Automated tuning of features was discussed to counter the manual tuning effort, a hybrid method between hand crafted heuristic functions and end-to-end feature learning has been developed, where the features themselves are still hand crafted, but the coefficients (weights) c_i of the features in the polynomial evaluation function $h(s) = c_1 \times f_1(s) + c_2 \times f_2(s) + c_3 \times f_3(s) + \dots + c_n \times f_n(s)$ are learned by supervised learning (see Section 4.1.3). The difference is that the features in the Chess evaluation function are manually designed heuristics, combined in a linear function whose coefficients are manually tuned, whereas the features in a deep net are trained (generalized) on a large set of examples in an automatic training process.

We will now see in more detail how neural networks can perform automated feature discovery. First we will describe the general architecture for shallow artificial neural networks.

6.1.2 Shallow Network

The architecture of artificial neural networks is inspired by the architecture of biological neural networks, such as the human brain. Neural networks consist of neural core cells that are connected by nerve cells (see, for example, [48]). Figure 6.1 shows a drawing of a biological neuron.

³Of course, research has now focused on ways to automate the hand crafting of network architectures. Section 7.5.3 describes ways to automatically generate appropriate network architectures for different domains.

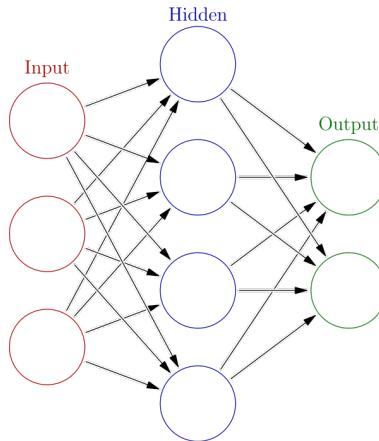


Figure 6.2: A Shallow Neural Network

Figure 6.2 shows a simple artificial neural network, with an input layer, an output layer, and a single hidden layer. Artificial neural networks have achieved fame through the success of deep learning. A neural network with a single hidden layer (or a few, 2–3) is called shallow. When the network has more hidden layers, it is called deep.

Neural networks have a long history, and much work has been done before they were successful. Already in 1943 McCulloch and Pitts [434] provided a computational model for neural networks. The history of development of neural networks is quite interesting and rich, with high ups and deep downs. Training neural networks is not a trivial problem, and many years of research were necessary to achieve the current day impressive results. The 2018 Turing award was awarded to Bengio, Hinton, and LeCun for their work in this area (see Figure 2.3). Section 6.6 provides pointers to some of the literature.

The most popular training method is based on gradient descent, of which the stochastic version performs very well in deep neural networks. Much research has been done on algorithms for efficient gradient descent algorithms [728], culminating in the backpropagation algorithm. Stochastic gradient descent (SGD) and backpropagation have allowed the training of multi-layer networks, paving the way for deep learning’s success.

Stochastic Gradient Descent and Backpropagation

We will now discuss the networks and their training algorithms in more detail.

Neural networks consist of neurons and connections, typically organized in layers (as in Figure 6.2). The neurons act as a function that process their input signal as a weighted combination of inputs, producing an output signal. This function is called an activation function. Popular activation functions are the rec-

tified linear unit (ReLU: partly linear, partly zero), the hyperbolic tangent, and the sigmoid or logistic function $\frac{1}{1+e^{-x}}$ (these will be discussed in more detail shortly). The neurons are connected by weights. At each neuron j the incoming weights ij are summed \sum and then processed by the activation function σ . The output o of neuron j is therefore: $o_j = \sigma(\sum_i x_i w_{ij})$ for weight ij of predecessor neuron x_i . The outputs of this layer of neurons are fed to the inputs for the weights of the next layer.

In supervised learning, a network is trained on a set of example pairs (x, z) : inputs x and labels, or targets, z . Typical examples are images of digits, in which case the labels are the correct digit.

The neural network is a black box function $f_\theta(x) \rightarrow y$ that converts input to output. The behavior depends on the weights θ . Training the network is adjusting the weights such that the required input-output relation is achieved. This is done by minimizing the error (or loss) function, that calculates the difference between the output y and the network target z .

The training process consists of training *epochs*, individual passes in which the network weights are optimized towards goal. When training starts, the weights of the network are initialized to small random numbers. Each epoch consists of a forward (recognition) pass, and a backward (training) pass. The forward pass is just the regular recognition operation for which the network is designed. The input layer is exposed to the input (e.g., the image), which is then propagated through the network to the output layers, using the weights and activation functions. The output layer provides the answer, by having a high value at the neuron corresponding to the right label (such as CAT or DOG).

Training stops when the error function has been reduced below a certain threshold for a single example, or when the loss on an entire validation set has dropped sufficiently. More elaborate stopping criteria will be discussed later in relation to over-fitting.

The backward pass calculates the difference between the forward recognition outcome and the true label. At the output layer the propagated value y is compared to the other part of the example pair, the label z . The difference with the label is calculated, yielding the *error*. Two common error functions are the mean squared error $\frac{1}{n} \sum_i^n (z_i - y_i)^2$ (for regression) and the cross-entropy error $-\sum_i^M z_i \log y_i$ (for classification of M classes). See, for example, Goodfellow et al. [245] for much more on error functions.

Then, in the backward phase, this error is propagated back to the input layer, adjusting the weights in the direction so that the error becomes smaller. This method uses the gradient over the weights, and is called gradient descent.

Most neural nets are trained using a stochastic version of gradient descent, or SGD [623], that approximates the gradients by sampling. SGD speeds up the process and introduces some noise, reducing over-fitting. Again, Goodfellow et al. [245] provide more details.

The theory and practice of gradient descent algorithms is quite elaborate and subtle. (For example, in Section 6.1.6 we will discuss exploding and vanishing gradients.) Many articles and books have been written on the subject of gradient

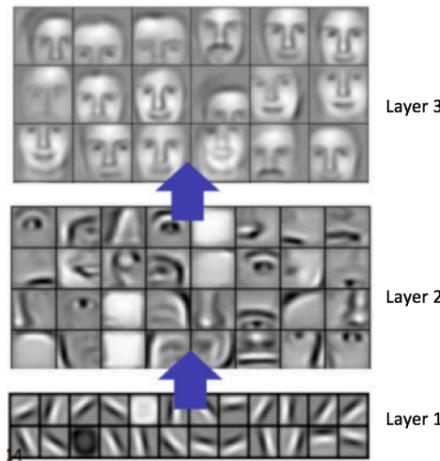


Figure 6.3: Layers of Features of Increasing Complexity [397]

descent, see, for example [245, 273, 288, 81, 393].

Now that we have discussed essential concepts of shallow neural networks, we will turn to deep learning, and see which technology was able to achieve super-human performance in certain recognition tasks.

6.1.3 Deep Learning

There has been quite some theoretical work on the question of which functions can be represented with neural networks.⁴ Some functions can only be represented with an architecture of at least a certain depth [63, 62, 269]. For complex tasks such as image recognition single layer networks do not perform well, or need pre-processing with hand-coded feature recognizers. A breakthrough in end-to-end (direct image) recognition was achieved only when multi-layer (convolutional) networks were being used, and when ways were found to train them efficiently. This breakthrough created a surge of interest into neural nets and deep learning [584, 393].

In 2006 Hinton et al. [287] published work on efficiently learning a high-level representation using successive layers of variables in a restricted Boltzmann machine. Later, researchers were able to create multi level networks that learned to recognize high level concepts such as cats from watching unlabeled images from Youtube videos [392].

After impressive results in 2012 [372], LeCun, Bengio and Hinton published

⁴The universal approximation theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate a wide variety of continuous functions[155]. Although it states that a simple neural networks can represent a variety of interesting functions when given appropriate parameters, the theorem does not say how those parameters can be learned. Learning may take very long, and may take an impractical number of neurons in the hidden layer.

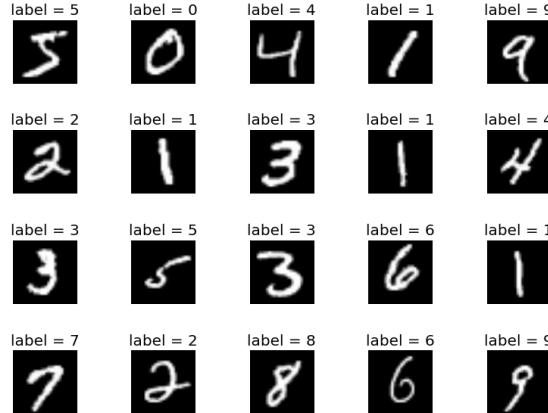


Figure 6.4: Some MNIST images

an influential paper on deep learning [393], explaining how multiple hidden layers in networks learn increasingly abstract representations. In deep learning, each layer transforms its input data into a (slightly) more abstract representation. The hierarchy of network layers together can recognize a hierarchy of low to high level concepts [393, 397]. For example, in face recognition (see Figure 6.3) the first hidden layer may encode edges; the second layer then composes and encodes simple structures of edges; the third layer may encode higher level concepts such as nose or eyes; and the fourth layer works at the abstraction level of a face. Note that deep feature learning finds what to abstract at which level on its own [61], and can come up with classes of intermediate concepts, that work, but look surprising and highly counterintuitive upon first inspection by humans.

Especially in image and speech recognition impressive results have been reported by deep networks [59]. Popular variants of SGD are AdaGrad [185] and Adam [351], methods that adaptively change the learning rate and momentum of the backpropagation. Section 6.2.1 goes deeper into the features of deep layers.

For deep learning architectures to perform recognition tasks at a level comparable to human performance, a number of problems had to be solved, that we will now discuss.

6.1.4 Scaling Up

The universal approximation theorem states that a feed forward network with a single hidden layer can approximate a wide variety of interesting continuous functions with an arbitrary accuracy. However, the theorem does not give an indication on how many nodes are necessary to learn this representation [155], and complex functions can only be represented with an architecture of at least a certain depth. Therefore multi level networks are used in practice.

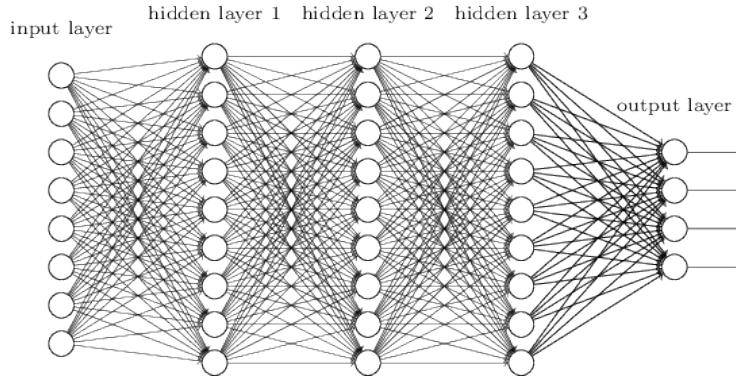


Figure 6.5: Three Fully Connected Hidden Layers [557]

The development of deep learning benefitted greatly from handwriting recognition efforts. The standard test set for handwriting recognition was MNIST (for Modified National Institute of Standards and Technology). Standard MNIST images are low-resolution $x \times y = 32 \times 32$ pixel images of single handwritten digits (Figure 6.4). Of course, researchers wanted to process more complex scenes than single digits, and higher resolution images. To achieve higher accuracy, and to process more complex scenes, researchers needed to grow the networks in size and complexity.

We will describe three ways in which this can be done. First, assuming we wish that the layers have the same width as the input size, the width $x \times y = n$ of the network can be increased, by having more neurons in each layer. Second, the depth d of the network can be increased, by having more layers. And third, the complexity of the network can be increased, by using more complex type of processing at the neurons, and a different interconnection structure.

As you might expect, all three were necessary, in unexpected and non-trivial ways.

1. Increasing Width: Slow learning and Over-fitting

Simply increasing the width n of a fully connected network of depth d increases network size $s = n^2 \times d$, but is not a scalable solution (see Figure 6.5). More width creates three problems. First, in a fully connected network it causes a quadratic increase in weights and the same holds for the training time, which becomes unacceptably long. Second, over-fitting becomes a problem, since after increasing the capacity of the network, the (quadratically higher) number of parameters $n^2 \times d$ may become greater than the number of observations, in which case the network will over-fit: it will perfectly memorize training examples, and be unable to correctly generalize to different test examples. Figure 6.6 illustrates over-fitting, which will be described more fully in Section 6.1.7. Third, only increasing the width does not address the problem that some complex functions can only be

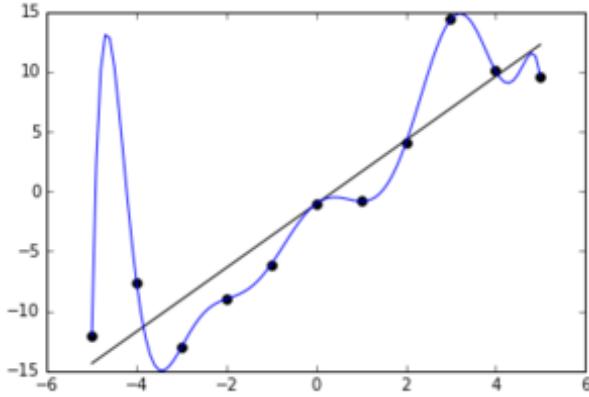


Figure 6.6: Over-fitting: Does the Curvy Blue Line or the Straight Black Line Generalize the Information in the Data Points Best?

approximated with a network of a certain depth [63, 62, 269].

2. Increasing Depth to prevent Weights Increase

Increasing the width of a fully connected network runs into problems since the number of weights increases quadratically. Let us look what happens when we increase the depth d of the network.

As the depth increases, the number of weights $s = n^2 \times d$ increases linearly (see Figure 6.5) and so does the training time. The over-fitting problem also increases linearly. And, complicated functions can be approximated better by the deeper network.

Therefore, increasing the depth of the network is the way forward. However, it does not solve all our problems. If we want to approximate a function, especially one with a high dimensional input such as for higher definition images, then at least our input layer has to have a correspondingly high number of inputs. So, we still have not completely solved the problem of the increasing number of weights of fully connected networks. The number of weights just keeps increasing with high resolution input, causing long training times, over-fitting, and exploding and vanishing gradients.

For our solution we need the third element: a different connection structure. Therefore, instead of fully connected networks, we will now look at convolutional networks, and related improvements.

6.1.5 Convolutional Neural Nets

As we wish to process higher resolution images, increasing n or d increases the number of weights too rapidly, leading to unacceptably long training times and over-fitting when the network is fully connected.

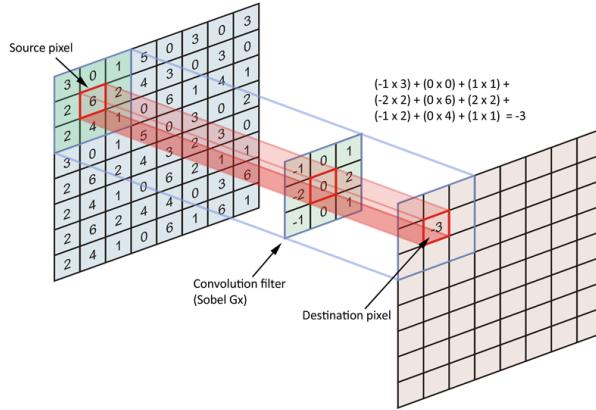


Figure 6.7: Convolutional Filters [245]

The solution lies in the third element: using a different (sparse) interconnection structure.

Convolutional neural nets take their inspiration from biology. Early work on monkey and cat visual cortices indicated that neurons respond to small regions in the visual field [307, 308]. The visual cortex in animals and humans is not fully connected, but locally connected, in a receptive field.

The connectivity pattern of CNNs resembles the animal visual cortex [307, 429]. A CNN consists of convolutional operators or filters. A typical convolution operator has a small *receptive field* (it only connects to a limited number of neurons, say 5×5) whereas a fully connected neuron connects to all neurons in the layer below. Convolutional filters detect the presence of local patterns, or features. The next layer therefore acts as a feature map. The filters detect small local features. A CNN layer can be seen as a set of learnable filters (or kernels), invariant for local transformations [245].

Real-world images consist of repetitions of many smaller elements. Due to this so-called *translation-invariance*, the same patterns re-appear throughout an image. CNNs can take advantage of this. The weights of the links are shared, resulting in a large reduction in the number of weights that have to be trained. Mathematically CNNs are constraints on what the weight values can be (some zero, some have to be equal). This is a significant advantage of CNNs, since the computational requirements of training the weights of many fully connected layers would be prohibitive. In addition, statistical strength is gained, since the effective data per weight increases.

Deep CNNs work well in image recognition tasks, for visual filtering operations in spatial dependencies, and for feature recognition (edges, shapes) [394]. For recognition tasks with temporal dependencies, such as in speech or text recognition, other networks have been developed. CNNs perform well on image recognition and other tasks. Convolutional neural nets (CNNs) are an integral part of

the deep learning revolution [394].⁵

Convolutional Filter

In Figure 6.7 we see an example filter. Filters can be used to identify certain features. Features are basic elements such as edges, straight lines, round lines, curves, and colors. To work as a curve detector, the filter should have a pixel structure with high values indicating a shape of a curve. By then multiplying and adding these filter values with the pixel values of the image we can detect if the shape is present. The sum of the multiplications in the input image will be large if there is a shape that resembles the curve in the filter.

Now this filter can only detect a certain shape of curve. Other filters can detect other shapes. A filter layer can be visualized in an activation map, showing where a specific filter (layer) fires. Larger activation maps can recognize more elements in the input image. Adding more filters increases the size of the network, which effectively is a large activation map. The filters in the first network layer process (“convolve”) the input image and fire (have high values) when a specific feature it is built to detect is in the input image. Training a convolutional net is training a filter that consists of layers of sub-filters.

By going through the convolutional layers of the network, increasingly complex features can be represented in the activation maps.

Before the CNN starts, the weights or filter values are initialized to random values. The filters in the lower layers are “empty,” they are not trained to look for edges and curves. The filters in the higher layers are not yet trained for eyes and mouths. During training, the filters in these layers take on their task as recognizers for respective representations.

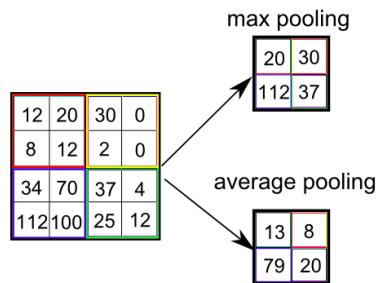
Once they are trained, they can be used for as many recognition tasks as needed. A recognition task consists of a single quick forward pass through the network.

Let us spend some more time on understanding these filters.

Shared Weights

In CNNs the filter parameters are shared in a layer. Each layer thus defines a filter operations. A filter is defined by few parameters but is applied to many pixels of the image; each filter is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and form a feature map. This means that all the neurons in a given convolutional layer respond to the same feature within their specific response field. Replicating units in this way allows for features to be detected regardless of their position in the visual field, thus constituting the property of translation invariance.

⁵Interestingly, this paper was already published in 1989. The deep learning revolution happened twenty years later, when publicly available data-sets, more efficient algorithms, and more compute power in the form of GPUs was available.

Figure 6.8: Max and Average 2×2 Pooling [245]

This weight sharing is also very important to prevent an increase in the number of weights in deep and wide nets, and to prevent over-fitting, as we shall see later on.

CNN Architecture

Convolutions compute features—the deeper the network, the more complex the features. A typical CNN architecture consists of a number of stacked convolutional layers. In the final layers, fully connected layers are used to then classify the inputs.

In the convolutional layers, by connecting only locally, the number of weights is dramatically reduced in comparison to a fully connected net. The ability of a single neuron in a CNN to recognize different features, however, is less than that of a fully connected neuron.

By stacking many such locally connected layers on top of each other we can achieve the desired non-linear filters whose joint effect becomes increasingly global, as more layers are added.⁶ The neurons become responsive to a larger region of pixel space, so that the network first creates representations of small parts of the input, and from these representations create larger areas. By stacking convolutional layers on top of each other, they can recognize and represent increasingly complex concepts without an explosion of weights.

Max Pooling

A further method for reducing the number of weights is pooling. It is an operation related to convolving. Pooling is a kind of non linear down sampling (expressing the information in lower resolution with fewer bits). Typically, a 2×2 block is down sampled to a scalar value (see Figure 6.8). Pooling reduces the dimension of the network. The most frequently used form is max pooling. It is an impor-

⁶Non-linearity is essential. If all neurons performed linearly, then there would be no need for layers. Complex recognition functions able to discriminate between cats and dogs would not be possible.

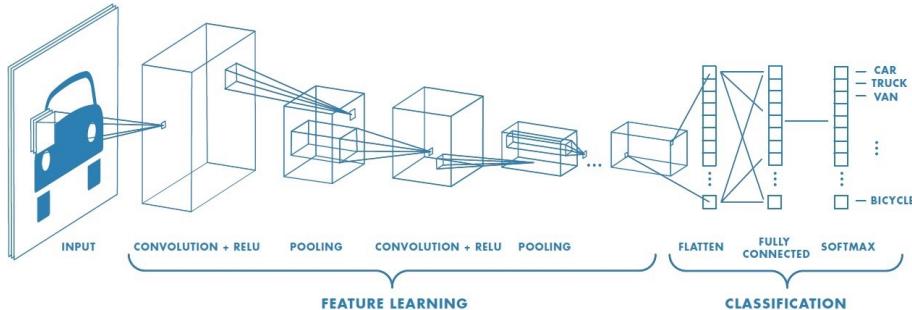


Figure 6.9: Convolutional Network Example Architecture [557]

tant component for object detection [140] and is an integral part of most CNN architectures.

Max pooling also allows small translations, such as shifting the cat by a few pixels, or scaling, such as putting the cat closer to the camera.

A typical CNN architecture consists of an architectures of multiple layers of convolution, max pooling, and ReLU layers, topped off by a fully connected layer (see Figure 6.9).⁷ Section 6.2.1 will discuss more concrete examples of well known CNN architectures.

6.1.6 Exploding and Vanishing Gradients

The deep convolutional architectures solved the problem of processing higher resolution and more complex scenes by controlling the number of weights organized in layers of filters. The deep architectures brought a new problem, though. The popular training algorithm, backpropagation, in combination with the conventional sigmoid activation function and deep networks, suffers from the vanishing gradient problem.

Gradients

The vanishing gradient problem was one of the problems that has held back progress in multi-level neural networks for some time.

The backpropagation algorithm computes the gradient of the error function. Backpropagation computes gradients (derivatives) by the chain rule. Originally, most neural networks used the sigmoid function as their activation function. The problem is that the derivative of the sigmoid function $\frac{1}{1+e^{-x}}$ on the domain $[0, 1]$ is a value between 0 and 0.25. Backpropagation multiplies these values with the weights for each layer, starting from the output layer working backwards to the input layer. Each time that a value between 0 and 0.25 is multiplied by the weights, the derivative gets smaller, approaching zero or infinity for the deepest

⁷Often with a soft-max function. The soft-max function normalizes an input vector of real numbers, to a probability distribution $[0, 1]$.

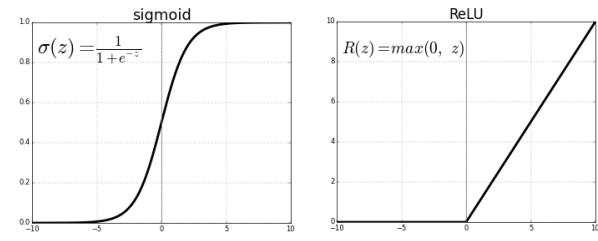


Figure 6.10: Sigmoid and ReLU

layers. Long products of many layers yield a low gradient value at the deep layers. The gradient tends to get smaller as we move backward through the hidden layers. This means that neurons in the earlier layers learn much more slowly than neurons in later layers. This effect was first described in Hochreiter’s Master’s thesis [291, 290].

The result is that gradient descent with a traditional activation function (sigmoid or tanh) cannot learn over many layers. For the deepest layers the gradients are vanishing to zero, negating the use that an extra layer would give.

The gradient in deep neural networks is unstable, tending to either explode or vanish in earlier layers. This instability was a fundamental problem for gradient-based learning in deep neural networks.

Multiple solutions to the vanishing gradient problem have been found. One solution is to model the network as a multi-level hierarchy of networks pre-trained one level at a time [581]. Hinton’s work on training deep networks also addresses this problem [287].

ReLU

Since the problem of vanishing gradients is caused by the derivative of the activation function, alternative functions were proposed. Traditionally the activation function of neurons is a well differentiable function such as tanh or the sigmoid function. A simpler activation functions is ReLU function (short for rectified linear unit). The definition of ReLU is $f(x) = \max(0, x)$. Over the positive part of its input ReLU is linear, otherwise it is zero. See Figure 6.10. ReLU is a simple function that is quick to calculate, is non-linear, and removes negative values from the activations [372]. The major advantage of ReLU is that in deep networks it suffers much less from the vanishing gradient problem. Deep ReLU networks are much easier to train than sigmoid networks.⁸

⁸The derivative of the ReLU function has a discontinuity. This discontinuity has lead to the introduction of soft-ReLU, which replaces the sharp transition from 0 to linear with a small smooth part.

Batch Normalization

Another solution to the vanishing gradient problem is batch normalization. Batch normalization periodically re-scales the data. It performs a batch standardization and assures that each batch is in a safe state, taking the gradients away from the zero, the one, or infinity. This technique was introduced in [315]. At the cost of some extra computation, the effect of periodical normalization is manyfold. Training is sped up, over-fitting is reduced, and, finally, the vanishing gradient problem is reduced substantially, since gradient values are normalized.

Batch normalization is at the moment the most powerful method to prevent over-fitting.

6.1.7 Over-fitting

Over-fitting is a major problem in large neural networks. The intuition behind under-fitting and over-fitting is as follows. Let us assume that the training data represents a noisy signal, as most data generated by measurements of natural processes does. The learning process performs well, when the signal from the training data is learned, and the noise is disregarded. Only then will it generalize well to the test data. Under-fitting occurs when the capacity of the network is too small to learn, model, or represent the signal. Over-fitting occurs when the capacity is so large that it learns even the noise in the training data (see Figure 6.6). Both under-fitting and over-fitting result in limited accuracy at test time.

Since the number of weights in deep networks is often in the millions, it is easily greater than the number of observations. In such a situation over-fitting is a problem that reduces the generalization performance of the network. Over-fitting is said to have happened when a trained model corresponds too closely to a particular data set, and fails to reliably predict a future observation. The network has been fitted to the particular training set, the signal and the noise, but not to the underlying structure of the data. Over-fitting occurs frequently when a model has more parameters than the training data set has examples.

Figure 6.6 illustrates this problem. The blue curvy line perfectly fits all data points, but is unlikely to perform well on a separate test set. The black straight is more likely to have caught the essence of the training domain, even though it misses some of the data points.

Over-fitting can be prevented in a number of ways, some of which are aimed at restoring the balance between the number of network parameters and the number of training examples. We will discuss *data augmentation* and *capacity reduction*.

Another approach is to look at the the training process. Examples are *regularization*, *early stopping*, *dropouts* and *batch normalization*.

Data Augmentation

Over-fitting occurs when there are more parameters in the network than examples to train on. One possible solution is to increase the amount of data. This method

is called data augmentation. The training data set is increased through manipulations such as rotations, reflections, noise, rescaling, etc. A disadvantage of this method is that the computational cost of training increases.

Capacity Reduction

Another easy solution to over-fitting lies in the realization that over-fitting is a result of the network having too large a capacity, the network has too many parameters. A cheap way of preventing this situation is to reduce the capacity of the network, by reducing the width and depth of the network.

L1 and L2 Regularization

A standard method to try when we suspect that the network is over-fitting, is regularization. Regularization involves adding an extra term to the loss function. The regularization term forces the network to not be too complex. The term penalizes the model for using too high weight values. This limits flexibility, but also encourages building solutions based on multiple features. Two popular versions of this method are L1 and L2 regularization [479, 245]. For different situations different preferred methods exist.

Early Stopping

Perhaps the easiest solution to over-fitting is the early stopping solution. Early stopping is based on the observation that over-fitting can be regarded as a consequence of so-called over-training (training that progresses beyond the signal, into the noise). By terminating the training process earlier, for example by using a higher stopping threshold for the error function, we can prevent over-fitting to occur [120].

Finding the right moment to stop may take some experimenting [524, 525]. A convenient and popular way is to add a third set to the training set/test set duo which then becomes a training set, a test set, and a holdout validation set. The role of the training set and the test set remains the same, to train the network. However, after each training epoch, the network is evaluated against the holdout validation set, to see if under- or over-fitting occurs, or if we should stop training. Finally, the test set is used in the familiar former role to see how well the trained network generalizes to unseen instances. In this way, over-fitting can be prevented dynamically during training [525, 80, 245].

Dropout

A popular method to reduce over-fitting is to introduce dropout layers into the networks. Dropout reduces the effective capacity of the network by stochastically dropping a certain percentage of neurons from the backpropagation process [289, 624]. Dropout is an effective and computationally efficient method to reduce over-fitting [245].



Figure 6.11: Fei-Fei Li

Section 6.4.2 further discusses the trade-off between generalization and overfitting, including references to recent insights.

Batch Normalization

Batch normalization periodically normalizes the data [315], as we just mentioned. This has many benefits, including a reduction of over-fitting.

6.1.8 Datasets and Networks

Now that we have discussed in depth the deep learning algorithms and their challenges and solutions, it is time to look at datasets and methods that have been instrumental in the progress of the field.

Datasets have been of great importance for the progress of supervised learning. The algorithmic advances that were discussed in Section 6.1.2 were facilitated by the availability of large training sets of labeled examples. Deep learning has benefited from a number of Drosophila. A well known repository of data sets is the University of California at Irvine Machine Learning repository UCI.⁹

The MNIST database may well be the best known of these data sets. MNIST is a database of 60,000 handwritten digits provided by the National Institute of Standards and Technology. The original MNIST database is available at MNIST.¹⁰ CIFAR is one of the most widely used set of images in machine learning and vision [371]. CIFAR-10 has also 60,000 images in 10 classes: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Another major data set of central importance to deep learning is ImageNet [203, 170]. ImageNet is available at ImageNet.¹¹ It is a collection of more than 14 million URLs of images that have

⁹<https://archive.ics.uci.edu/ml/index.php>

¹⁰<http://yann.lecun.com/exdb/mnist/>

¹¹<http://www.image-net.org>



Figure 6.12: Rina Dechter

been hand annotated with the objects that are in the picture. It contains more than 20,000 categories. A typical category contains several hundred training images.

The importance of ImageNet for the progress in AI is large. The availability of labeled images allowed algorithms to learn, and new algorithms to be created. ImageNet was conceived by Fei-Fei Li in 2006, and in later years she developed it further. Since 2010 ImageNet runs an annual software contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [170]. Since 2012 ILSVRC has been won by deep networks, starting the deep learning boom in science and industry.

Fei-Fei Li directed the Stanford AI Lab from 2013-2018 (see Figure 6.11).

We will now mention some influential networks architectures. We will start with Yann LeCun's LeNet-5.

LeNet-5

Towards the end of the 1990s the work on neural networks moved into deep learning, a term coined by Rina Dechter [169] (Figure 6.12). Some twenty years after the introduction of deep convolutional neural nets by LeCun et al. [395] CNNs became highly popular. This paper introduced the architecture LeNet-5. LeNet-5 is a 7 layer convolutional neural net trained to classify handwritten MNIST digits from 32×32 pixel images. It is a successful network that was used commercially to recognize digits in banking checks. The paper has a thorough comparison of LeNet-5 to other methods such as principal component analysis (PCA).

At GitHub a modern Keras implementation of LeNet is available at Models Keras (and also of AlexNet and VGG).¹² The code straightforwardly lists the layer definitions, shown in listing 6.1.

¹²<https://github.com/eweill/keras-deepcv/tree/master/models/classification>

```

def lenet_model(img_shape=(28, 28, 1), n_classes=10, l2_reg=0.,
                 weights=None):

    # Initialize model
    lenet = Sequential()

    # 2 sets of CRP (Convolution, RELU, Pooling)
    lenet.add(Conv2D(20, (5, 5), padding="same",
                    input_shape=img_shape, kernel_regularizer=l2(l2_reg)))
    lenet.add(Activation("relu"))
    lenet.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    lenet.add(Conv2D(50, (5, 5), padding="same",
                    kernel_regularizer=l2(l2_reg)))
    lenet.add(Activation("relu"))
    lenet.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    # Fully connected layers (w/ RELU)
    lenet.add(Flatten())
    lenet.add(Dense(500, kernel_regularizer=l2(l2_reg)))
    lenet.add(Activation("relu"))

    # Softmax (for classification)
    lenet.add(Dense(n_classes, kernel_regularizer=l2(l2_reg)))
    lenet.add(Activation("softmax"))

    if weights is not None:
        lenet.load_weights(weights)

    # Return the constructed network
    return lenet

```

Listing 6.1: LeNet-5 code [395, 724]

AlexNet

In 2012 Krizhevsky et al. published the AlexNet architecture [372] with breakthrough performance on the ImageNet dataset. They used a deep network of 8 layers. The first 5 layers were convolutional layers, some with max pooling, and 3 layers were fully connected layers. Krizhevsky et al. overcame problems of vanishing gradients and over-fitting with ReLU, data augmentation and L2 regularization, and effectively used the power of GPUs to train a large network. Their work created many follow up works showing even better performance, and attracted much attention, setting off the general interest in deep learning.

Deep networks have since grown further in size, with more layers, and different learning algorithms and connection structures. The number of weights in such networks is often in the millions, greater than the number of observations on which the network is trained. This makes over-fitting an important problem, and may of the methods that we discussed previously have been developed because of these larger networks.

TensorFlow

The deep learning breakthrough was caused by the co-occurrence of three major developments: (1) algorithmic advances that solved key problems in deep learning, (2) the availability of large data sets of labeled training data. A third crucial ingredient in the deep learning breakthrough is the availability of computational power.

The most expensive operations in image processing and neural network training are essentially operations on matrices. Matrix operations are some of the most well-studied problems. Their algorithmic structure is well understood, and of basic linear algebra operations high performance parallel implementations for CPUs exist, such as the BLAS [177, 136].

Graphical Processing Units, or GPUs, were originally developed for fast processing of image and video data, largely driven by the video gaming industry. Modern GPUs consist of thousands of processing units optimized to process linear algebra matrix operations in parallel [561, 415, 622], offering matrix performance that is orders of magnitude faster than CPUs [484, 141, 641].

Many modern neural network training packages support GPU parallelism. Well known dedicated deep learning packages are Berkeley's Caffe [321], Facebook's PyTorch [500], Theano [65] which has been subsumed by Google's TensorFlow [1, 2], and its user friendly add-on Keras [137]. Most packages have Python interfaces. Some machine learning and mathematical packages also offer deep learning tools, such as MATLAB and R.

TensorFlow provides high quality implementations of many machine learning and neural network algorithms and operations. (A tensor is a multi dimensional array, often used for transformations.) The programming concept of TensorFlow takes some getting used to. Programs are constructed as a data-flow graph in which the sequence of tensors defines the operations. A higher level, easier, interface is provided by Keras. Keras comes with TensorFlow. It is recommended

to start with Keras.

Implementing a full working deep learning algorithm that performs well on a new problem is a challenging task. Many problems of backpropagation, gradient descent, over-fitting, vanishing gradients, and numerical stability have to be solved. It is because of the free availability of high quality deep learning methods that so much progress has been made in the recent years. Advances in image recognition, speech recognition, game playing, automated translation, and autonomous vehicles is made possible in a large part by these software suites. The free availability of high quality implementations may be the fourth reason for the deep learning breakthrough. Appendix A contains overviews of relevant machine learning packages.

Conclusion

We have discussed in depth methods for deep supervised learning. This has been a long section; many problems had to be overcome to achieve end-to-end learning, en only by the convergence of better algorithms, more computational power, and large labeled data sets, was the deep learning breakthrough in image recognition possible.

6.2 *Advanced Network Architectures

The breakthroughs in deep learning have prompted much further research in fascinating areas. Let us now look deeper in more advanced methods for generalization and stable learning.

This is a starred section, with more advanced material, which is may be skipped when in a hurry.

Much of the work in supervised learning is driven by image recognition tasks. Just as in game playing, the availability of clear benchmarks and competitions has facilitated progress. The ImageNet database [170] has been at the center of this field. Guo et al. provide an excellent review [259] of the state of the art of visual recognition.

The field of pattern recognition is a rich field. Many methods for image classification have been devised. Some work by feature extraction and selection, some work directly on the raw data. Traditionally, the field of image recognition used hand-crafted heuristic features such as edges, corners, and others (just as Chess and Checkers evaluation functions used hand crafted features).

6.2.1 Examples of Concrete Convolutional Networks

As we saw, after the turn of the millennium two important developments happened: (1) large data bases of (hand) labeled images became available, and (2) powerful GPUs became available and were used for training neural networks. The large labeled datasets provided crucial training material, and the GPU power

allowed algorithms to go through the large training sets much quicker than before. Together, these two developments allowed major improvements in training speed and accuracy. Among the many works describing impressive progress, the AlexNet paper [372] is often cited as an example of the breakthrough of deep learning. The highlights of its architecture were described previously. We will now look deeper into its architecture.

The 2012 ImageNet database as used by AlexNet has 15 million labeled images. The network featured a highly optimized 2D two GPU implementation of 5 convolutional layers and 3 fully connected layers. The filters in the convolutional layers are 11×11 in size.

The neurons follow a ReLU function. In AlexNet images were scaled to 256×256 RGB pixels. The size of the network was large, with 60 million parameters. This causes considerable over-fitting. AlexNet used data augmentation and dropouts to prevent over-fitting.

Krizhevsky et al. won the 2012 ImageNet competition with an error rate of 15%, significantly better than number two, who achieved 26%. Although there were earlier reports of CNNs that were successful in applications such as bioinformatics and Chinese handwriting recognition, it was this convincing win of the 2012 ImageNet competition for which AlexNet is well known.

AlexNet has become an important network architecture for research. Many resources are available for AlexNet. The original AlexNet code is available on GitHub at AlexNet.¹³ Berkeley's Caffe project maintains a "model zoo" where multiple trained models are available, complete with Caffe code that can be studied. The AlexNet Caffe model is available on GitHub at BVLC AlexNet.¹⁴ Also, as mentioned before, a Keras implementation of AlexNet, LeNet and VGG is available at Models Keras.¹⁵

ZFnet

A year later a paper by Zeiler and Fergus [745] significantly improved upon AlexNet. Their network has become known as ZFnet. The work also provided an enlightening explanation about how convolutional nets work, including a visualization of the inner layers (see Figure 6.13). In a sense, this allows a look into the brain of the neural net, to see what features it recognizes at which layer. Afterwards many papers have provided these kinds of insights into the hidden layers of their network.

VGG

Simonyan and Zisserman [612] followed a different philosophy than AlexNet and ZFnet, with small filters (3×3) and a deep hierarchy of 16-19 layers. Their VGG net was also successful. VGG faithfully implements the idea behind deep nets as

¹³<https://github.com/akrizhevsky/cuda-convnet2>

¹⁴https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet

¹⁵<https://github.com/eweill/keras-deepcv/tree/master/models/classification>

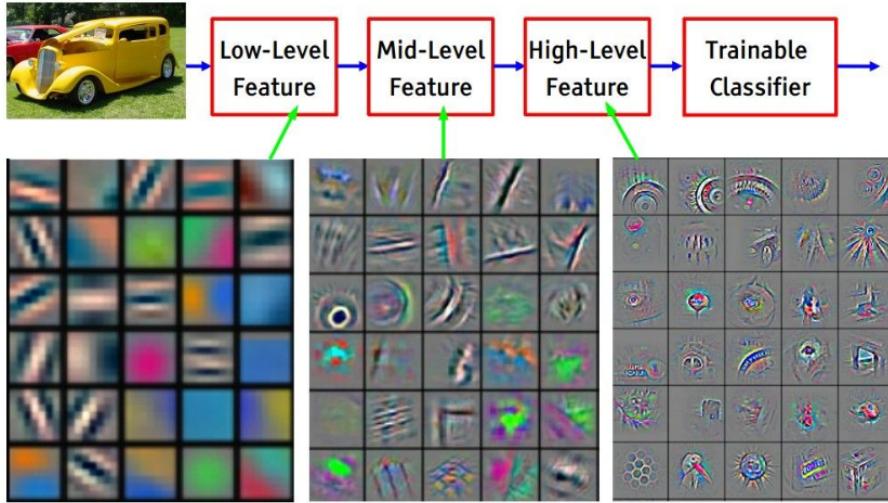


Figure 6.13: ZFnet layers [745]

needing many layers (16) in order to build a hierarchical representation of visual data [393].

GoogLeNet

A further notable network is GoogLeNet [642], which is even deeper, with 22 layers. The new architecture and algorithms perform better than AlexNet, yet have 12 times fewer parameters, showing how strong the new architecture and algorithms are. The GoogLeNet architecture is about sparse networks, allowing a deeper network without increasing the total number of parameters. The network is built around *inception* modules. In previous architectures, convolutional layers were stacked sequentially on top of each other. Inception modules are small networks inside a network, where different sized convolutions are placed in parallel to each other. Through the inception modules sparse network structures were achieved, reducing training time. The paper provides more details. In addition to the inception modules, batch normalization was used to prevent vanishing gradients [643] and dropouts for regularization.

Residual Networks

After GoogLeNet, another important innovation is the residual network architecture, or *resnets*. This was introduced in the 2015 ImageNet challenge, which He et al. won with a very low error rate of 3.57%. This error rate is actually lower than what most humans achieve: 5-10% [275]. ResNet has no fewer than 152 layers, and, as before, this is achieved through a *simpler* architecture with fewer connections (less parameters than VGG). The problem of vanishing gradients, which

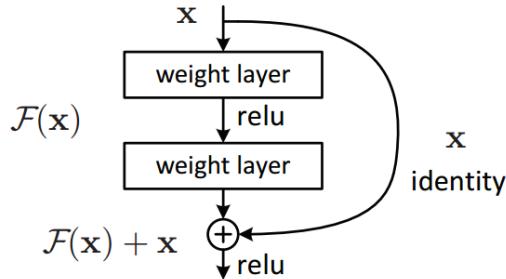


Figure 6.14: Residual Cell [275]

deep networks suffer from, is addressed with intermediate normalization steps.

The main contribution of resnets is based on the following observation. The authors observed that when adding more layers to their net, they were obtaining *lower* training accuracy. This is counterintuitive, since more parameters should allow the net to achieve at least the accuracy of the shallower net. The deep network suffered from degrading training accuracy due to reasons related to overfitting [275].

Residual nets introduce *skip links* to cure this degradation (see Figure 6.14). Skip links are connections skipping one or more layers, allowing the training to skip layers. Skip links create a mixture of a shallow and a deep network, preventing the accuracy degradation and vanishing gradients of deep networks. Furthermore He et al. note that normal layers having a harder time to learn an identity function, than learning a mapping to 0, and the skip layers allow the net to more easily learn an identity function.

Following up on ResNets, DenseNets were devised, building on the main insight behind residual nets, and achieving even more impressive results [305].

These architectural innovations have again caused performance improvements. As a result, image recognition tasks now frequently exceed human performance [259, 175]. This is an active area of research, and better architectures are presented at the major machine learning conferences each year.

6.2.2 Other Architectures

In addition to mainstream supervised network architectures, there are other architectural innovations of different kinds of networks.

R-CNN

The previous networks are designed for object recognition and classification. A related problem is object detection, where a bounding box is drawn around an object. Region based CNNs addresses this problem (R-CNN). They are described in [240].



Figure 6.15: Deep Fake [739]

Generative Adversarial Networks

Another active research area is deep generative modeling. Generative models are models from which a new example can be sampled. (A contrast to generative models are the regular discriminative models, that output a label, for classification.) There are several classes of generative models. An important and popular class that has made quite an impact is the Generative Adversarial Network, or GAN [246].

It was found that if an image is slightly perturbed, and imperceptibly to the human eye, deep networks can easily be fooled to characterize an image as the wrong category [644]. This brittleness is known as the one-pixel problem [631, 644]. The one-pixel problem has spawned an active area of research to understand this problem, and to make image recognizers more robust.

GANs are generative models that generate adversarial examples. The purpose of adversarial examples is to fool the discriminator (recognizer). They are designed to cause a wrong classification. In GANs one network, the generator, generates an image, and a second network, the discriminator, tries to recognize the image. The goal for the generator is to mislead the discriminator. In addition to making recognizers more robust against the one-pixel problem, one of the other uses of GANs is to generate artificial photo-realistic images such as *deep fake* images [739], see Figure 6.15 and *deep dreaming*, Figure 6.16¹⁶ [342]. GANs have

¹⁶Deep Dream Generator <https://deeppdreamgenerator.com>



Figure 6.16: Deep Dream

significantly increased our theoretical understanding of supervised training.

6.2.3 Sequential processing

Image recognition has had a large impact on network architectures, leading to innovations in network architectures such as convolutional nets.

Other applications, such as time series analysis and speech recognition, have also caused new architectures to be created. Image recognition is a single-time modeling task. In contrast, speech recognition and time series analysis concern sequential data that must be modelled. Such sequences can be modeled by recurrent neural nets. Some of the better known recurrent neural nets are Hopfield networks [298], recurrent neural nets (RNN) [72, 205], and Long Short-Term Memory (LSTM) [292].

Figure 6.17 shows a basic recurrent neural network. An RNN neuron is the same as a normal neuron, with input, output, and activation function. However, RNN neurons have an extra pair of looping input/output connections. Figure 6.18 shows the internal structure of an RNN neuron (a tanh activation function) with their loop connections unrolled as previous and next time steps. Through this structure, the values of the parameters in an RNN can evolve. In effect, RNNs have a kind of variable-like state.

To understand how RNNs work, it helps to unroll the network, as has been done in Figures 6.18 and 6.19. The Figures illustrate time-unrollment. The recurrent neuron loops have been drawn as a straight line to show the network in a deep layered style, with connections between the layers. In reality the layers are

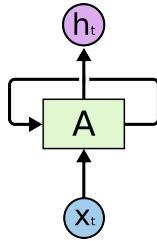


Figure 6.17: RNN, x_t is an input vector, h_t is the output/prediction, and A denotes that RNN which feeds back into itself at the next time step [485]

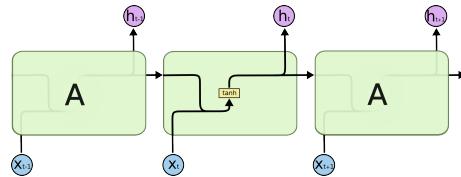


Figure 6.18: Time-unrolled RNN neuron, with a \tanh activation function; shown are the previous and next time steps as well [485]

time steps in the processing of the recurrent connections. In a sense, an RNN is a deep neural net folded into a single layer of recurrent neurons.

Where deep convolutional networks are successful in image classification, RNNs are used for more demanding tasks, such as captioning challenges. In a captioning task the network is shown a picture, and then has to come up with a textual description that makes sense [706]. Figure 6.20 gives examples of captioning tasks, from Chen et al. [134].

Captioning tasks are hard, and a computer succeeding in “seeing” an image and “interpreting” it with a caption that makes sense, describing what can be seen on the picture, can be a startling experience. Seeing one’s own RNN learning to generate the first appropriate descriptions of images can be a satisfying experience for machine learning researchers (see the exercises at the end of this chapter). Indeed, simple models can deliver surprisingly good results, and RNNs can achieve very good results.

A step up from image captioning is video captioning, where a description has to be generated for a sequence of images [695].

The main innovation of recurrent nets is that they allow us to work with sequences of vectors. Figure 6.21 show different combinations of sequences that we will discuss now. There can be sequences in the input, in the output, or in both. Karpathy has written an accessible and well-illustrated blog on the different RNN configurations [341]. The figure shows different rectangles. Each rectangle is a vector. Arrows represent computations, such as matrix multiply. Input vectors

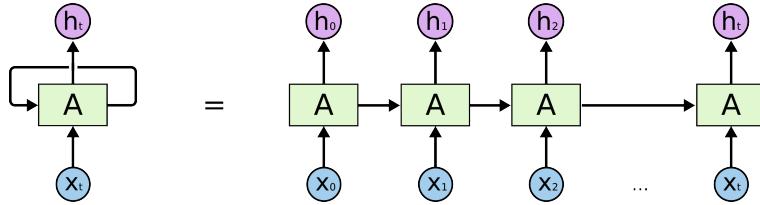


Figure 6.19: RNN unrolled in time [485]

are in red, output vectors are in blue and green vectors hold the state. Following Karpathy [341], from left to right we see: (1) *one to one* the standard network without RNN. This network maps a fixed-sized input to fixed-sized output, such as image classification. (2) *One to many* adds a sequence in the output. This can be an image captioning task that takes an image and outputs a sentence of words. (3) *Many to one* is the opposite, with a sequence in the input. Think for example of sentiment analysis (a sentence is classified for words with negative or positive emotional meaning). (4) *Many to many* has both a sequence for input and a sequence for output. This can be the case in machine translation, where a sentence in English is read and then a sentence in French is produced. (5) *Many to many* is a related situation, with synchronized input and output sequences. This can be the case in video classification where each frame of the video should be labeled.

State

Time series prediction is a difficult problem, for which sequential processing is well suited. Sequential processing is more powerful than fixed networks, that can only perform a predetermined number of steps. Moreover, where deep normal networks represent a single stateless functional mapping from input to output, RNNs combine the input vector together with the state from the previous time step to produce a new state vector, to be fed into the next time step. In this particular sense RNNs have state. The fact that RNNs have such state in addition to a learned function brings them closer to a conventional imperative computer program (such as a Python program with variables). An RNN can be interpreted as a fixed program with certain inputs and some internal variables. In this way, RNNs are simple computer programs that do not have to be programmed, but learn their functionality by training. RNNs are appealing for building intelligent systems [236]. This is an active area of research.

Many different successful situations for recurrent neural networks have been found. Graves et al. describe speech to text processing [252]. Machine translation is described by Sutskever et al. [636]. Earlier work described text generation [634]. RNNs for video classification are described by Donahue et al. [176]. Mnih et al. describe RNNs for visual attention [452]. Important work on RNNs has been done by Sutskever [633], Mikolov [440] and Graves [249].



The man at bat readies to swing at the pitch while the umpire looks on.



A large bus sitting next to a very tall building.



A horse carrying a large load of hay and two people sitting on it.



Bunk bed with a narrow shelf sitting underneath it.

Figure 6.20: Captioning Tasks [134]

LSTM

Unlike conventional regression or classification, time series prediction also adds the complexity of a sequence dependence among the input variables.

LSTM (Long Short-Term Memory) is a more powerful type of neural network designed to handle sequences. Figure 6.22 shows the LSTM module, allowing comparison to the simple RNN of Figure 6.18. LSTMs are designed for sequential problems, such as time series, and planning. LSTMs were introduced by Hochreiter and Schmidhuber [292].

RNN training suffers from the vanishing gradient problem. For short term sequences this problem may be controllable by the same methods as for CNN (see Section 6.1.6). For long term remembering LSTM are better suited.

LSTM building blocks avoid the vanishing gradient problem by introducing skip-connections (just like residual nets), connecting the gradients over layers unchanged. Since LSTMs have skip-links that make them less susceptible to the vanishing gradient problem, this makes LSTMs better suited for large architectures.

LSTMs are frequently used to solve diverse problems [585, 251, 254, 237]. A more detailed explanation of LSTMs can be found in [292, 341]. Researchers have experimented with LSTMs for creating hybrid architectures that combine planning and learning, with some initial success [610]. See also Section 7.5.5 for

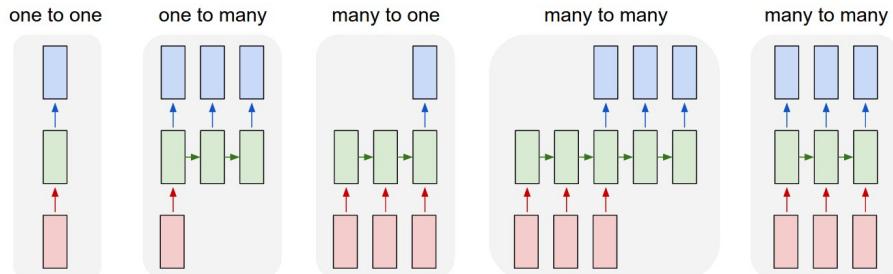


Figure 6.21: RNN configurations [341]

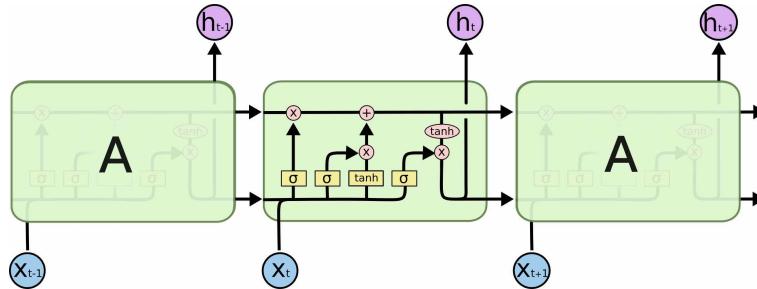


Figure 6.22: LSTM [341]

a few more details on combined planning and learning.

Conclusion

There has been a wealth of research into deep neural networks for image recognition (and beyond). Many complex deep architectures were created, and achieved impressive recognition results. Especially research on generative adversarial networks and sequence processing is a highly active area of supervised learning research.

Let us now go to deep reinforcement learning, to see how this field fared.

6.3 Deep Reinforcement Learning

Deep reinforcement learning is one of the major success stories of artificial intelligence. Before it became a success, there were quite some obstacles that had to be overcome. Successes from supervised learning were not as easy to transfer to reinforcement learning as one might have hoped. In this section we will discuss these problems and the methods that have been found to address them.

The need for deep reinforcement learning came out of the wish for improving performance beyond heuristics and sampling, beyond what can be achieved with (automated tuning of) hand crafted features. End-to-end learning requires training of high dimensional state spaces, far exceeding the capabilities of exact or heuristic planning methods, or of shallow networks or other conventional machine learning methods.

In Section 6.3.2 function approximation for reinforcement learning will be introduced, as a method that is suited for high dimensional state spaces. As we will see, reinforcement learning function approximation suffers from training instability. Section 6.3.2 discusses this so-called deadly triad of unstable training. Next, we will describe a solution introduced in 2013. This solution, experience replay, effectively performs de-correlation of learning examples, improving the stability of the training. Experience replay was introduced in the context of a program achieving human-level play of Atari arcade video games such as Space Invaders, achieving quite some publicity by itself at the time [453, 454].

Let us now start at the beginning, with the need for end-to-end learning in a reinforcement learning context.

6.3.1 End-to-end Reinforcement Learning

The state space of hand crafted features is typically of manageable size. Much of the complexity of the problem domain is abstracted away by the hand crafted heuristics. The heuristics translate raw board features into a few focused heuristic parameters, going from a state space with many dimensions to a low dimensional state space. End-to-end learning does away with the hand crafted heuristics, and takes the problem of learning from the raw inputs head on. The computational complexity of end-to-end learning problems is correspondingly much higher.

The wish for learning beyond that which could be attained with (machine learning on top of) hand crafted heuristics has been driving progress in deep reinforcement learning.

In the previous section we have seen the problems that had to be overcome in supervised learning to achieve end-to-end learning. Only after convolutional neural networks had been invented, and after solutions for over-fitting and vanishing gradients had been found, did deep end-to-end learning work.

In reinforcement learning a similar situation existed. Here the problem was the stability of the training process. Let us now see why stable deep reinforcement learning is such a difficult problem.

6.3.2 Reinforcement Learning and Function Approximation

Reinforcement learning is a form of machine learning, where just like in supervised learning, a function is learned from examples through generalization. The way in which these examples are presented to the learning algorithm, however, is different.

Supervised Learning	Reinforcement Learning
(example-label) pair	(state-action) pair
database of ground truths	environment gives reward
static learning process	interactive learning process
fixed set of examples	changing set of examples
dense: ground truth for all examples	sparse: reward for some examples

Table 6.2: Supervised versus Reinforcement learning

There are two important differences between supervised learning and reinforcement learning: (1) the way that information is presented (static database or interaction) and (2) whether the information is present for all or for some states.¹⁷

First, supervised learning infers functions from a database of example-label pairs, when the ground truth is given. Reinforcement learning infers a function for state-action pairs. It has no fixed database of examples with ground truths. Positive or negative rewards are only provided after interaction with an environment. This interaction influences the learning process. The set of examples is not fixed, and may change from learning process to learning process. See Table 6.2.

The second difference is that in supervised learning ground truths are present for *all* example-label pairs. The state space is fixed and dense, and learning methods are stable. In reinforcement learning problems the state space is often sparse, the outcome of an action may only be known after many further steps, and the rewards are only available for some of the state-action pairs. For other state-action pairs the rewards have to be inferred, propagated over a long distance of intermediate states before the reward of a state-action pair is known. This gives rise to the credit assignment problem, and, as we shall see next, to instability.

Much research into the stability of reinforcement learning has been performed. We will discuss the deadly triad, long range credit assignment, correlation between states, and fluctuations in the data distribution.

Deadly Triad

Deep reinforcement learning can be unstable. Tsitsiklis and Van Roy [677] showed in 1997 that combining off-policy reinforcement learning with non-linear function approximators (such as neural networks) could cause the training to diverge. Sutton and Barto [638] go further, and summarize three main reasons for unstable training: function approximation, bootstrapping, and off-policy learning. Together, these form the so-called *deadly triad* of instability. When these three techniques are combined, training can diverge, and value estimates can become

¹⁷Also note that the functions to be learned are different. In supervised learning a classification or a regression function is typically learned. In reinforcement learning the function is a policy or a value function.

unbounded.¹⁸

Function approximation may have inaccuracies in the attribution of values to states. Whereas exact methods are designed to recognize individual states, neural networks are designed to recognize individual features of states. These features may be shared by different states, and values attributed to those features are shared as well by states. Function approximation may cause confusion or mis-identification of states. In a reinforcement learning process where new states are generated on the fly, this sharing of values among states may cause loops or other forms of instability. If the accuracy of the approximation of the true function values is good enough, then states may be identified well enough to prevent most loops or divergent training processes.

Bootstrapping of values increases the efficiency of the training because values do not have to be calculated from the start, but previously calculated values are reused. Bootstrapping is at the basis of the recursive Bellman equation. However, errors or biases in initial values may persist, and even spill to other states, as values are propagated Bellman-style, and error values become correlated between states. With function approximation inaccuracies in values are almost guaranteed. Bootstrapping can thus lead to unstable learning, and to training loops getting stuck in one area of the state space.

Off-policy learning uses a behavior policy that is different from the target policy that we are optimizing (see Section 3.3.5). When the target policy is improved, the off-policy values are not, and the algorithm does not converge by this action. Off-policy learning converges independent of the behavior policy, and converges generally less well than on-policy learning, especially when combined with function approximation [638].

The consequence of the deadly triad is that stable learning may not occur, especially when training takes place in an environment where the distance of credit assignment is long, or when function approximation is imprecise.

In a divergent training process error values do not decrease, but increase. The network does not learn. It does not find a mapping between inputs and outputs, between states and actions.

For some time, most further research in reinforcement learning focused on linear function approximators, which have better convergence guarantees. Later, methods that achieve convergence in the face of the deadly triad were found, causing many further techniques to address these problems to be developed (see Section 6.4).

As we discussed in the introduction of this section, a further problem of reinforcement learning is that it is a process that generates examples on the fly, by interaction. This gives rise to correlations between training examples. Let us have a look at the consequences of these correlations.

¹⁸Actually, as Tsitsiklis and Van Roy showed, already two of the three may be enough for instability. All three together increases the chance of unstable training more, unless special measures are taken.

Correlated States

In supervised learning data samples (states) are independent and are assumed to be distributed uniformly over the state space. In the database of static images, there is no relation between subsequent images, and examples are independently sampled. Each image is separate (in contrast to a movie, where many inter-temporal relations between frames are present). In model-based reinforcement learning a sequence of states is generated by a simulator or game playing program. The features of the states are often correlated, differing only by a single action, one move or one stone. Such a correlated training set may be myopic, covering only a part of the state space. Such a training set will not work well with standard supervised training algorithms. The correlated states create the possibility of feedback loops, and the training algorithm may well get stuck in a part of the feature space.

To illustrate myopic learning, let us consider an example of correlated training in Chess. Imagine a program that knows everything about center control but nothing about king safety, and therefore loses if it encounters a differently trained agent that will outplay it in the king safety domain. We can try to counter this effect, as we will see soon.

In addition to the problems of loops and local optima, learning from correlated consecutive samples is inefficient. Little new information is contained in two states that are almost the same. The low learning efficiency translates into a smaller effective data set, much training is done, yet little is learned. Randomizing the samples breaks these correlations and therefore reduces the probability to get stuck.

The deadly triad suggests us to use on-policy instead of off-policy learning, to improve the training convergence. However, when learning on-policy the current network determines the next data sample to be trained on. Mnih et al. [453] give an example of how on-policy learning also leads to training divergence. Assume that the maximizing action is to move left. With on-policy learning the training samples will be dominated by samples from the left-hand side; if, for some reason, the maximizing action then switches to the right, then the training distribution will also switch. In this way unwanted feedback loops may arise and the network will get stuck in self reinforcing features, or diverge.

Fluctuating Data Distribution

A consequence of the deadly triad problem is that in reinforcement learning the data distribution may be fluctuating. In supervised learning for each training example an immediate output label is available, allowing an error function to be calculated, and weights to be adapted. In reinforcement learning output signals from the environment are infrequent or derived from other states, and may thus be “polluted” by old values, and are calculated using values that may be approximations.

Since in reinforcement learning training examples are generated in interaction with the environment, divergent training processes have consequences for the set

of training examples. In reinforcement learning the size of the training set may vary in size, or may be infinite, as the training process itself influences which examples are generated.

This has two consequences, one desired, one undesired. In the short term, learning new behaviors may cause myopia or getting stuck in the state space by feeding on its own behavior as described in on-policy learning. This short term training behavior can be smoothed with a replay buffer by providing data from a larger landscape [453].

In the longer term, learning new behaviors may actually be desirable. Forgetting bad behaviors can also be a way to escape ineffective parts of the state space. Thus, the replay buffer should not be too large. In Chapter 7 we will learn more about this in the context of self-play.

TD-Gammon and the Quest for Stable Training

Although all this theory suggests many reasons why function approximation may preclude stable reinforcement learning, there were early indications to the contrary that stable training was possible in practice. TD-Gammon [653] used self-play reinforcement learning, achieving stable learning in a shallow network. Perhaps some form of stable reinforcement learning was possible, at least in a shallow network? TD-Gammon’s training used a temporal difference algorithm similar to Q-learning, approximating the value function with a network with one hidden layer, based on heuristic features.

TD-Gammon’s success prompted attempts with TD-learning in Checkers [132] and Go [635, 142]. Unfortunately the success could not be replicated in these games, and it was believed for some time that Backgammon was a special case well-suited for reinforcement learning and self-play [522, 588]. One suggestion was that perhaps the randomness of the dice rolls helped exploration and smoothing of the value function of the state space.

A few other reports of early successful applications of deep neural networks in a reinforcement learning setting suggest also that stable deep reinforcement learning is possible [277, 559], prompting further work. The results in Atari (2013) and Go (2016) have now provided clear evidence that stable and general reinforcement learning is indeed possible, and why.¹⁹ (See also the historical and bibliographical notes of this chapter.)

Conclusion

So far, we have discussed reasons why deep reinforcement learning processes may suffer from instability. Deep reinforcement learning is a process where the training examples to be learned from are influenced by the training process itself, introducing the possibility of unstable or self-reinforcing learning. We have also seen that, despite these reasons, some papers reported stable learning (in addition

¹⁹However, achieving stable learning still requires elaborate tuning and experimentation; the fundamental reasons for instability have not disappeared.



Figure 6.23: Atari 2600 Console

to TD-Gammon’s success with shallow reinforcement learning). Dealing with divergence and instability is of prime importance in deep reinforcement learning.

As noted before, attempts to follow up on TD-gammon’s success with neural nets and self-play in Checkers and Go were not successful at first. We will now discuss the work of Mnih et al. [453] who used techniques to achieve stable deep reinforcement learning, achieving end-to-end reinforcement learning on a computationally challenging problem in a convincing way, by playing Atari 2600 games.

6.3.3 Atari 2600 Games

Let us now look in more detail into the Atari 2600 experiments.

Learning actions directly from high dimensional sensory inputs such as sound and vision is one of the long standing challenges of artificial intelligence. In 2013, one year after the AlexNet success in supervised learning, a paper was published that took reinforcement learning a major step further. Mnih et al. [453] published a ground breaking work on *end-to-end* reinforcement policy learning in Atari games. Their work is end-to-end in the sense that they learn joystick actions directly from the visual state, the raw pixels, without an intermediate step of hand crafted heuristic features. The network architecture and training algorithm were named DQN, for Deep-Q-network.

Mnih et al. applied deep learning to play 1980s arcade Atari 2600 games, such as Space Invaders, Pong, and Breakout, directly from the raw television screen pixels [55]. Figure 6.23 shows a picture of the Atari 2600 console.

Being able to close the loop from pixels to policy is a major achievement. In their original 2013 workshop paper Mnih et al. [453] were able to achieve human level play for six games. In a follow up article in 2015 they improved on their work and were able to achieve a level of play equal to humans for 49 of the games that were in the test set [454].

It should be noted that there are a few Atari games that proved difficult to

achieve strong play at. Notably games that required more long range planning because long stretches of the game do not give rewards, such as Montezuma's Revenge, were a problem. Delayed credit assignment over long periods is still hard.

Nevertheless, a single architecture was able to successfully learn control policies for many different games. Minimal prior knowledge was used, and the neural network only processed the pixels and the game score. The same network architecture and training procedure was used on each game, although a network trained for one game could not play another game well.

Arcade Learning Environment

The results by Mnih et al. stimulated much further work in the area of deep reinforcement learning. Successful end-to-end reinforcement learning created much interest among researchers, and many related algorithms were developed. We will look into some depth into their methods and experiments.

The games that were used for DQN are from a standard benchmark set, the Arcade Learning Environment (ALE) [55]. ALE is a test-bed designed to provide challenging reinforcement learning tasks. Among other things it contains an emulator of the Atari 2600 console. ALE presents agents with a high dimensional²⁰ visual input (210×160 RGB video at 60Hz) of tasks that were designed to be interesting and challenging for human players (see Figure 6.24). The game cartridge ROM typically holds the game code (2-4 kB), while the console memory is small, just 128 bytes. This is not a typo, the console memory really is only 128 bytes. The actions can be input via a joystick, and a fire button (18 actions).

The original experiments were performed with seven Atari games. Training was performed on 50 million frames in total.

In their 2013 work [453] the neural network performed better than an expert human player on Breakout, Enduro, and Pong. On Seaquest, Q*Bert, and Space Invaders performance was far below that of a human. In these games a strategy must be found that extends over longer time periods. In their 2015 work [454] the net performed better than human level play in 29 of 49 Atari games. Again, games with longer credit delay were more difficult.

Let us look in more detail at the network architecture of the 2013 experiments.

Network Architecture

The Atari task is a control task: the network trains a behavior policy directly from pixel frame input. The task of processing raw frames involves a high computational load. Therefore, the 2013 training architecture contains a number of reduction steps. The network that is used consists of three hidden layers, which is simpler than what is used in most supervised learning tasks. Figure 6.25 shows the architecture of DQN.

²⁰That is, high dimensional for machine learning. 210×160 pixels is not exactly High Definition video quality.



Figure 6.24: Screenshots of 4 Atari Games (Breakout, Pong, Montezuma’s Revenge, Private Eye)

The images are high dimensional data. Since working with the full resolution of 210×160 with 128 colors at 60 frames per second would be computationally too intensive, the images are preprocessed. The 210×160 with 128 color palette is reduced to gray scale and 110×84 pixels cropped to 84×84 . The first hidden layer convolves 16 8×8 filters with stride 4 and ReLU neurons. The second hidden layer convolves 32 4×4 filters with stride 2 and ReLU neurons. The third hidden layer is fully connected and consists of 256 ReLU neurons. The output layer is also fully connected with one output per action (18 actions for the joystick). The outputs correspond to the Q-values of the individual action. The network receives the change in game score as a number from the emulator, derivative updates are reprocessed to $\{-1, 0, +1\}$ to indicate decrease, no change, or improvement of the score (Huber loss [37]).

To reduce computational demands further frame skipping was used. Only one in every 3-4 frames was used, depending on the game. For history, the net takes as input the last four resulting frames, allowing movement to be seen by the net. As optimizer RMSprop was used. A variant of ϵ -greedy is used, that starts with an ϵ of 1.0 (exploring) going down to 0.1 (90% exploiting).

The network architecture consists of convolutional layers and a fully connected layer. Recall that convolutional nets are shift-invariant feature recognizers. Fully connected nets allow $n \times m$ mapping. The feature recognizers together with the $n \times m$ mapping allow mapping of any feature (shape) to any action.

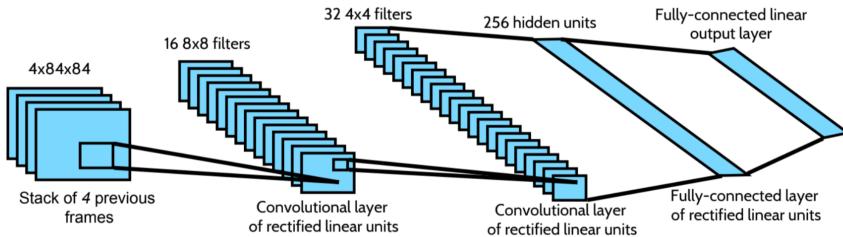


Figure 6.25: DQN Architecture

6.3.4 De-correlating States

Let us now look in more detail at how stable learning was achieved. The focus of DQN is on breaking correlations between states. The DQN algorithm has two methods to achieve this: experience replay and infrequent weight updates. We will first look at experience replay.

Experience Replay

Recall that in reinforcement learning training samples are created in a sequence of interactions with an environment, and that subsequent training states are correlated. The networks are trained on too many samples in a certain area, and unknown parts of the state space remain unexplored. Furthermore, through function approximation, some behavior may be forgotten. When an agent reaches a new level in a game that is different from previous levels, the agent may forget how to play the other level.

DQN uses experience replay, with a replay buffer, a cache of previously explored states.²¹ The goal is to increase the independence of training examples, by sampling training examples from this buffer. The next state to be trained on is no longer a direct successor of the current state, but one in a long history of previous states. In this way the replay buffer spreads out the learning over all seen states by sampling a batch of states at random, breaking temporal correlations between samples.

Experience replay improves on standard Q-learning in two respects: (1) it randomizes the order in which samples are used, breaking the correlations between consecutive samples; (2) it prevents unwanted feedback loops, which would arise when the game makes moves in a certain area that it cannot escape from, getting stuck in local areas of the state space. By averaging the behavior distribution over the previous states, experience replay smooths out training and avoids oscillations or divergence in the parameters.

Note that training by experience replay is a form of off-policy learning, since

²¹Originally experience replay is a biologically inspired mechanism [432, 486, 410].

the current parameters are different from those used to generate the sample.²² Off-policy learning is one of the three elements of the deadly triad. It is curious to see that this solution to unstable learning is to use more of one of the causes of the deadly triad.

In practice, experience replay stores the last N examples in the replay memory, and samples uniformly when performing updates. A form of importance sampling might differentiate important transitions. Experience replay works well in practice in Atari [454]. However, further analysis of replay buffers has pointed to possible problems. Van Hasselt et al. [688] and Zhang et al. [750] study the deadly triad with experience replay, and find that larger networks resulted in more instabilities, but also that longer multi-step returns yielded fewer unrealistically high reward values. Stability in reinforcement learning is still an active area of research, as we shall see in Section 6.4.

Infrequent Weight Updates

The second improvement in DQN is *infrequent weight updates*. Infrequent updates of the target weight values also reduce correlations and oscillations caused by loops and self reinforcing features. This method works by using a separate network for generating the targets of the update of the Q value. Every x updates the network Q is cloned to obtain a target network \hat{Q} , which is used for generating the targets for the following x updates to Q .

This second network improves stability of Q learning, where normally an update that increases $Q(s_t, a_t)$ often increases $Q(s_{t+1}, a)$ for all a . This also increases the target, quite possibly leading to oscillations and divergence of the policy. Generating the targets using an older set of parameters adds a delay between the time an update to Q is made and the time the update changes the targets, breaking the feedback loop or at least making oscillations less likely.

6.3.5 Conclusion

It should be noted that the fundamental reasons for instability in reinforcement learning with function approximation still exist; the deadly triad has not evaporated into thin air. Although DQN and related algorithms have achieved stable learning for some games, the algorithms require much tuning and experimentation. The field is still young, and getting good performance requires much hard work.

The topic of this chapter is how deep learning enabled automated feature discovery to achieve true end-to-end machine learning, from pixels to labels and actions. We have seen how a combination of new algorithms, large training sets, and GPU compute power has enabled researchers to achieve breakthroughs in image recognition and game playing. These breakthroughs have resulted in applications that we use in our daily lives, and that have inspired much further research (see also Section 6.4).

²²But since DQN uses ϵ -greedy exploration, in a percentage of the actions the behavior policy is used. DQN is a mix between on-policy and off-policy learning.

Let us take a step back and reflect on what has been achieved in supervised and reinforcement learning.

End-to-End Training

Both the works on Atari and ImageNet succeed in end-to-end training. It is interesting to compare end-to-end training in supervised and in reinforcement learning. The breakthroughs in computer vision with AlexNet [372] and other ILSVRC-works rely on training on large labeled training sets. In these and later research, networks are trained directly from the raw inputs, using updates by stochastic gradient descent. The layered representations that are learned in this way are often better representations than hand-crafted or heuristic features.

Mnih et al. achieved an end-to-end behavior policy from raw visual input, also without any heuristics or intermediate features. The Atari 2600 games are games of skill. The games are difficult for humans, because they require dexterity and quick hand-eye coordination. Playing the games builds reflexes. The games are not so much designed as games of strategy (such as Chess, Checkers, Go, Backgammon, Othello, or even Tic Tac Toe) where credit assignment is long and small differences between states can have large consequences, and where planning algorithms shine. The Atari 2600 games are reflex games (thinking fast, not slow) with short temporal credit assignment, that could be learned with a surprisingly simple neural network. In this sense, the problem of playing Atari well is not unlike an image database categorization problem: both problems are to find the right answer (out of a small set) that matches an input consisting of a set of pixels. Apparently, mapping pixels to categories is not that different from mapping pixels to joystick actions, and the long range credit assignment problem has been overcome. To see such a deep connection between supervised and reinforcement learning may be a surprising side effect from the research into deep supervised and reinforcement learning.

The algorithmic feat of breaking through the divergence and oscillation of reinforcement learning with long delays in credit assignment is impressive, and the Atari results are highly imaginative, and have stimulated much subsequent research. Many blogs have been written on replicating the result, which is still not an easy task at all [37].

We will now turn to advanced deep reinforcement learning enhancements.

6.4 *Advanced Deep Reinforcement Learning

As we have seen in Section 6.3.2, stability of the training process is the main challenge in deep reinforcement learning. The breakthrough of DQN was to show how experience replay improves the training stability, allowing end-to-end learning in Atari. The Atari results have spawned much activity among reinforcement learning researchers to improve training stability further, and many refinements of experience replay have been devised. In this section we will discuss some of these enhancements.

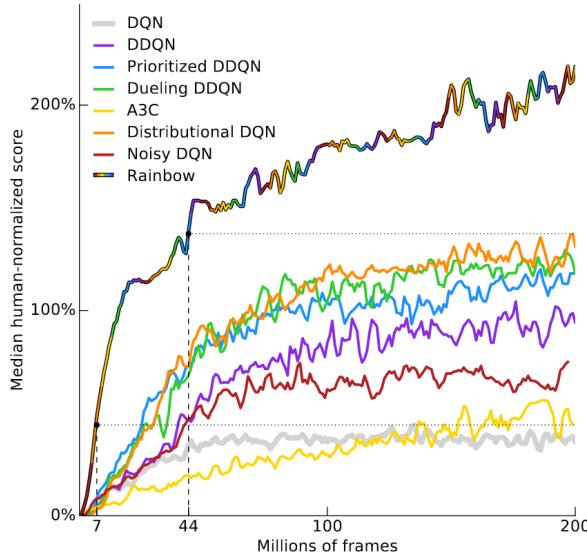


Figure 6.26: Rainbow Graph: Performance over 57 Atari Games [285]

6.4.1 Rainbow

Just as in planning the basic concepts and algorithms in deep reinforcement learning are simple. And just as in planning, the basic ideas offer a large room for further enhancement, to achieve high levels of performance [575].

After methods were found to achieve stable deep reinforcement learning [453] many further enhancements were introduced. And just as in planning, many of these enhancements are independent. The concepts do not interfere and can be added together.

In 2017 Hessel et al. published the Rainbow paper [285], in which they combined seven important enhancements. The paper is so called because the major graph showing the cumulative performance over 57 Atari games of the seven enhancements is multi colored (see Figure 6.26).

Reducing Correlation

The baseline algorithm of the methods described in the Rainbow paper is DQN, which was described in detail in this chapter. The original DQN architecture is based on a simple neural network with two convolutional layers and one fully connected layer, a fully connected output layer²³ and ϵ -greedy exploration. The main challenge that DQN overcomes is an unstable learning process. We recall that DQN does so by the combination of four techniques [453, 454]:

²³The 2015 architecture uses three convolutional layers[454]

1. an experience replay buffer for randomized sampling to break consecutive state correlations
2. a separate target weight network to break target value correlations
3. clipping rewards to $\{-1, 0, +1\}$
4. skipping frames and reducing pixel resolution to reduce the computational load

The following algorithms improve on various aspects of DQN.

DDQN

Van Hasselt et al. introduce double deep Q learning (DDQN) [689]. DDQN is based on the observation that Q-learning may over estimate action values. They find that, in practice, such over-estimations are common. On Atari 2600 games they show that, due to the use of a deep neural network, DQN suffers from substantial over-estimations. Earlier Hasselt et al. [268] introduced the double Q learning algorithm in a tabular setting. The later paper shows that this idea also works with a large deep network. This DDQN algorithm not only reduces the over-estimations but also leads to much better performance on several games.

DDQN was tested on 49 Atari games and achieved about twice the average score of DQN with the same hyper-parameters, and four times the average DQN score with tuned hyper-parameters [689].

Prioritized DDQN

Prioritized experience replay, or PEX, was introduced by Schaul et al. [577]. In the Rainbow paper its results are shown in combination with DDQN.

Recall that in DQN experience replay lets agents reuse examples from the past. In DQN experience transitions are uniformly sampled from a replay memory. Therefore, actions are simply replayed at the same frequency that they were originally experienced, regardless of their significance. Schaul et al. develop a framework for prioritizing experience. Important actions are replayed more frequently, and therefore learning efficiency is improved.

Schaul et al. use standard proportional prioritized replay, where the absolute TD error is used to prioritize actions. This can be computed in the distributional setting, using the mean action values. In the Rainbow paper all distributional variants prioritize actions by the Kullback-Leibler loss [285].

Dueling DDQN

Vanilla DQN uses a single neural network as function approximator. Dueling DDQN [720] improves on this architecture by using two separate estimators: a value function and an advantage function. This allows learning across actions. Results show that the advantage function leads to better policy evaluation when there are many similar valued actions.

A3C/Multi Step

The Rainbow experiments include Asynchronous Advantage Actor Critic (A3C) [451] using a multi step bootstrap target [638, 637].

In Section 3.3.3 the actor-critic methods was discussed, a method combining value functions and policy functions.

In reinforcement learning, the amount of computational effort required for training is often substantial, and can be a prohibiting factor in an experiment. Although the speedup provided by GPUs enabled much progress, the training times are still large, and researchers looked for better and faster algorithms. One way to speed up algorithms is through parallelization, running parts of the search effort on multiple computers in parallel.

In Mnih et al. [451] a method for asynchronous gradient descent is presented. The method allows implementation of different DQN variants, and related algorithms. One of these, an actor critic algorithm in which a separate policy and value function are learned [640], performs exceptionally well. The asynchronous version is named A3C, for Asynchronous Advantage Actor-Critic.

One of the effects of actor-critic [640] is that it stabilizes training, just like experience replay. A3C has parallel actor learners that stabilize the controllers. The parallelism also decorrelates the samples into a more stationary process, since the parallel agents experience a variety of unrelated states. Asynchronous parallelism allows on-policy methods, actor critic, and off-policy methods. Mnih et al. report on extensive experiments on Atari games and found asynchronous actor critic to perform the best [451]. A related method is DQV learning, which also shows better performance than DQN and DDQN [555].

The Rainbow experiments include A3C and show its performance against advanced DQN algorithms, and in combination.

Distributional DQN

DQN learns a single value, the estimated mean of the state value. Distributional Q-learning [54] learns a categorical distribution of discounted returns, instead of estimating the mean. This is in contrast to most reinforcement learning algorithms that model only the expectation of this value. Bellemare et al. use the distributional perspective to design a new algorithm which applies Bellman's equation to the learning of approximate distributions. Performance results of Distributional DQN on Atari are good, showing the importance of the distributional perspective. Other relevant research into distributional perspectives is by Moerland et al. [455, 456].

Noisy DQN

Noisy DQN [220] uses stochastic network layers for exploration. In NoisyNet parametric noise is added to the weights. This noise induces randomness in the agent's policy, increasing exploration. The parameters of the noise are learned with gradient descent along with the remaining network weights. The authors

replace the standard exploration heuristics for A3C, DQN and dueling agents (entropy reward and ϵ -greedy) with NoisyNet. The better exploration yields substantially higher scores for Atari.

Conclusion

The enhancements of the Rainbow paper were all developed separately. Interestingly, they do not interfere, but complement each other. The Rainbow paper [285] shows that they can be added on top of each other, enhancing basic DQN performance greatly (see Figure 6.26).

In the Rainbow paper the authors have restricted themselves to value based Q-learning methods. They note that purely policy based algorithms such as trust region policy optimization [591], proximal policy optimization [592] or different actor critic methods [451, 483] provide opportunities for further enhancements.

Progress has been made in deep reinforcement learning by many researchers in a short period of time. The Rainbow paper illustrates how much our understanding of DQN has improved. As in planning and in supervised learning, the presence of clear benchmarks was instrumental for progress. Researchers were able to see clearly if ideas worked, and to check if their intuition and understanding were correct. OpenAI’s Gym [98] and the ALE [55] are responsible to a great extent for enabling this progress.

6.4.2 On Generalization and Over-fitting

After we have delved into much detail on advanced supervised learning and reinforcement learning methods, it is time for some notes on recent work on the fundamental relation between generalization and over-fitting.

The central goal of machine learning is to learn functions from data, to generalize the essence from the input, to learn the signal without the noise. To learn complex patterns, the state of the art is to use deep, high capacity, neural networks. A problem is that if too little varied input is available in relation to the learning capacity of the net, that the network starts to over-fit, to memorize the input, to learn the input *including* the noise. This principle, known as the *bias-variance trade-off*, is a fundamental element in statistical learning theory [270]. Models with more parameters have lower bias but higher variance and once models have more parameters than observations, they will over-fit on the test set (as has been argued many times in this chapter).

The study of over-fitting is attracting attention in reinforcement learning. In Section 6.1.7 we discussed practical solutions such as regularization, dropouts, and early stopping. Especially the early stopping approach suggests that over-fitting can be viewed as a balancing problem between underdoing and overdoing generalization [265].

Relevant recent studies are [146, 748, 747]. These studies note that data for most supervised learning applications comes from the natural world. Patterns may be highly or even infinitely complex, and the true level of noise may not be

known. To put it differently, in this view the distribution of the test set differs from the training set.

Zhang et al. [748] find that deep neural networks, with their substantial over-parameterization and massive capacity, can indeed memorize data, and thus generalize poorly. They also note that the problem of over-fitting by high-capacity models also applies to long stretched training time and recurrent models. They emphasize that there is a fundamental trade-off between generalization and over-fitting in machine learning, and properly controlling (regularizing) the training is key to achieving generalization [748].

The question why deep learning works, why such large models do not continuously over-fit but actually work well, is an active area of research [52, 50]. Experimental studies have shown that large networks do not necessarily over-fit, and theoretical studies showing a regime of high parameter/low over-fitting are appearing. Zhang et al. [747] note that reinforcement learning algorithms can over-fit both in simulated environments and on natural images. They find that as soon as there is enough diversity in the simulated environment, deep reinforcement learning generalizes well, and speculate that this may be since synthetic domains have relatively little noise. On the other hand, they note that deep reinforcement learning algorithms show more prominent over-fitting when observing natural data, and the development of new benchmarks to study over-fitting in deep reinforcement learning is advocated.

Belkin et al. have performed theoretical studies on interpolation in SGD. Their studies suggests an implicit regularization regime of many parameters beyond over-fitting where SGD generalizes well to test data. This effect may in part explain the good results in practice of deep learning. Nakkiran et al. report similar experimental results, termed the *double descent* phenomenon [467, 468]. Research into the nature of over-fitting is quite active, see, e.g., Belkin et al. [52, 422, 50, 51].

6.5 Practice

Below are some questions to check your understanding of this chapter. Each question is a closed question where a simple, one sentence answer is possible.

Questions

1. What is the difference between supervised learning and reinforcement learning?
2. What is the difference between a shallow network and a deep network?
3. Why is function approximation important in game playing?
4. What phases does a learning epoch have? What happens in each phase?
5. What is the difference between heuristic features and learned features?
6. What is MNIST? What is ImageNet? What is TensorFlow?

7. Name three factors that were essential for the deep learning breakthrough, and why.
8. What is end-to-end learning? Do you know an alternative? What are the advantages of each?
9. What is under-fitting and what causes it? What is over-fitting and what causes it? How can you see if you have over-fitting?
10. Name three ways to prevent over-fitting.
11. What is the difference between a fully connected network and a convolutional neural network?
12. What is max pooling?
13. Why are shared weights advantageous?
14. Describe the vanishing gradient problem. What are ReLU units?
15. Why is deep reinforcement learning more susceptible to unstable learning than deep supervised learning?
16. What is the deadly triad?
17. Why may reinforcement learning with function approximation be unstable?
18. What is the role of the replay buffer?
19. How can correlation between states lead to bad generalization?
20. How did AlexNet advance the state of the art in deep learning?
21. Name three architectural innovations/characteristics of AlexNet.
22. What is an inception modules? What is a residual network? What is the relation between the two?
23. What is a GAN?
24. What is an RNN, how is it different from a deeply layered network?
25. What are the advantages of LSTM?
26. Why is the Rainbow paper so named, and what is the main message?

Exercises

Let us now start with some exercises. Install TensorFlow and Keras. Go to the TensorFlow page at TensorFlow.²⁴ The assignments below are meant to be done in Keras.

1. Download and install TensorFlow and Keras. Check if everything is working by executing the MNIST training example. Choose the default training dataset. Run it on the training set, and then test the trained network on a testset. How well does it predict?
2. Which optimizer does the MNIST example use? Try a different optimizer, an adaptive optimizer such as Adam and compare it to simple Backprop. Does training speed improve?
3. OpenML is a research project to compare different machine learning algorithms with different datasets. Go to OpenML and download different datasets to see which problems may be harder to learn.
4. The learning rate is a crucial hyper-parameter. Always tune this one first. You may find that learning goes slowly. Try to increase the training speed by increasing the different learning rate? Does performance increase? Does the error function drop faster or slower? Can you explain your observations?
5. Explore options for Cloud processing. The Cloud provides ample compute power at cheap prices, often with free student or sign up bonuses. Go to AWS, the Google Cloud, or Microsoft Azure, and explore ways to speed up your learning.
6. One of the biggest problems of supervised learning is over-fitting. Can you check if over- or under-fitting occurred in your experiments? Create an experiment in which you force over-fitting to occur. What will you do: change the size of the network, change the algorithm (change CNN, ReLU, dropout) or change a parameter (learning rate, dropout percentage, training time)? How can you see if over-fitting has occurred? When has it reduced?

Atari Breakout Code

Importantly, the Atari papers come with source code. For reference, the original DQN code from [454] is at Atari DQN.²⁵ This code is the original code, in programming language Lua, without new developments. Before you start working with this code, it should be noted that DQN can be difficult to get working. There are many little details to get right. In addition, the experiments often have to run for a long time which means that debugging and improving is slow if you have just one GPU.

²⁴<https://www.tensorflow.org>

²⁵<https://github.com/kuz/DeepMind-Atari-Deep-Q-Learner>

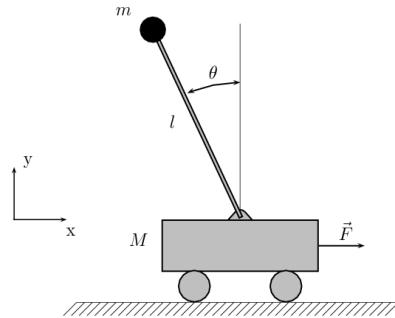


Figure 6.27: Cartpole Experiment

A better start is the TensorFlow implementation in Python of DQN for the Atari game Breakout, at Atari Breakout Code.²⁶ This may get you up to speed more quickly. The code is readable, and follows the concepts discussed in this chapter. However, the code is too large to cover in detail here.

To get really started with deep reinforcement learning, we suggest to use DQN and many of the other algorithms that are available at the OpenAI Keras RL library at Keras RL²⁷ [520, 236].

Cartpole

OpenAI provides baseline implementations of important reinforcement learning algorithms [171]. A good implementation of DQN for the Atari Learning Environment is available. Installation instructions for Ubuntu and MacOS are provided.

Cartpole is a basic experiment in which an agent must learn to balance a stick on a cart by reinforcement learning. A nice video of a Cartpole experiment is at Cartpole Video²⁸ (this is a real-life cartpole, as opposed to the simulated cartpole that is most often used in reinforcement learning experiments, see Figure 6.27). At OpenAI Cartpole baseline²⁹ working example code for the Cartpole experiment is available.

The high level code is simple, as Listing 6.2 shows. To really understand what is going on you have to look into the implementations of `deepq.learn` which has implements DQN in a few hundred lines of code.

Progress in deep reinforcement methods was greatly helped by the existence of suitable benchmark suites and competitions [55]. Just as Chess for heuristic planning and MNIST for supervised learning, the Arcade Learning Environment facilitated reinforcement learning progress for agents playing video games.

²⁶<https://github.com/floodsung/DQN-Atari-Tensorflow>

²⁷<https://github.com/keras-rl/keras-rl>

²⁸<https://www.youtube.com/watch?v=5Q14Ejn0JZc>

²⁹<https://github.com/openai/baselines/tree/master/baselines/deepq>

```

import gym
from baselines import deepq
def callback(lcl, _glb):
    # stop training if reward exceeds 199
    is_solved = lcl['t'] > 100 and sum(lcl['episode_rewards'][−101:−1]) / 100 >= 199
    return is_solved
def main():
    env = gym.make("CartPole-v0")
    act = deepq.learn(
        env,
        network='mlp',
        lr=1e-3,
        total_timesteps=100000,
        buffer_size=50000,
        exploration_fraction=0.1,
        exploration_final_eps=0.02,
        print_freq=10,
        callback=callback
    )
    print("Saving model to cartpole_model.pkl")
    act.save("cartpole_model.pkl")
if __name__ == '__main__':
    main()

```

Listing 6.2: Cartpole code [171]

Other well known benchmark suites are the OpenAI Gym [98] for video games and reinforcement learning, and MuJoCo [670] for simulated robot tasks. The benchmarks can be found here:

1. Arcade Learning Environment³⁰ [55]
2. OpenAI Gym³¹ [98]
3. MuJoCo³² [670]

The use of benchmarks is also of great importance for reproducible reinforcement learning experiments [282].

More Exercises

For the following exercises, if you have not done so, download and install Gym.

1. Go to the Mountain Car example, and run it. Experiment with different learning settings to understand their effect on the learning.
2. Try the same for Cartpole.
3. Install ALE. Train a player for Space Invaders. Did the training process work well? How do you find out if the trained player is a good player? Can you compute an Elo rating?
4. Identify the Replay buffer in the code. Reduce the size, or otherwise turn it off. Does DQN still converge, what does it do to the quality of the player that is trained?
5. *Try a few other Atari games. Does DQN train well? Try Montezuma's Revenge. Does DQN work well? Have a look at the alternative deep reinforcement methods. How could you improve DQN to play well in Montezuma's Revenge? (This is a hard research question.)

The open availability of TensorFlow, Keras, ImageNet, and OpenAI Gym/ALE allow for easy replication, and, importantly, improvement of the works covered in this chapter. Keras implementations of VGG, ResNet are available in Keras.applications (see Keras Applications).³³ AlexNet is here: AlexNet,³⁴ GoogLeNet is here: GoogLeNet.³⁵

³⁰<https://github.com/mgbellemare/Arcade-Learning-Environment>

³¹<https://gym.openai.com/>

³²<http://www.mujoco.org/>

³³<https://keras.io/applications/>

³⁴<https://gist.github.com/JBed>

³⁵<https://gist.github.com/joelouismarino>

Stable Baselines

The Gym GitHub repository OpenAI Baselines³⁶ contains implementations of many algorithms covered here. You can easily download them and experiment to gain an insight into their behavior. Stable Baselines is a fork of the OpenAI algorithms, it has more documentation and other features. It can be found at Stable Baselines³⁷ and the documentation is at Docs.³⁸ An implementation of the Rainbow paper in Torch is available here.³⁹ The RL Baselines Zoo even provides a collection of pre-trained agents, at Zoo.⁴⁰

Advanced Exercises

Due to the advanced nature of some parts of this chapter, the following exercises are large, project sized. These topics and these exercises are fascinating, but answering these questions may require a substantial amount of research and work (potentially thesis-size).

1. Reproduce the basic AlexNet, VGG, and ResNet results on ImageNet in Keras. Do deeper ResNets give better results? How do deeper ResNet impact training time?
2. Read GAN papers on how to generate artificial images [246, 342, 674]. Try to generate fake cats and dogs images. Can you also use a discriminator to separate real images from fakes?
3. Use a RNN to perform a Captioning Challenge in ImageNet.
4. Read the Rainbow paper [285]. Go to the OpenAI baselines and download and run the code and perform an ablation study, leaving out the constituting methods one by one.

6.6 Summary

In this chapter we faced the challenge of finding function approximators for large, high-dimensional, state spaces. Deep learning allowed automated feature discovery for end-to-end learning. The field of function approximation with neural networks is large, rich, and deep. This chapter can only give a glimpse of the work that has been done.

The goal of function approximation is to find a reliable value function for large state spaces, that are so large, that test examples can not be all seen during training. In this way, the function approximators of this chapter are an alternative to

³⁶<https://github.com/openai/baselines>

³⁷<https://github.com/hill-a/stable-baselines>

³⁸<https://stable-baselines.readthedocs.io/en/master/>

³⁹<https://github.com/Kaixhin/Rainbow>

⁴⁰<https://github.com/araffin/rl-baselines-zoo>

the heuristic feature function of Chapter 4 and the averaged random values of Chapter 5.

We have discussed how supervised learning provides a suitable paradigm for learning by example. The representation power of networks can be improved by deep hierarchies of layers of neurons.

Training deep networks is not easy. Among the problems to be overcome are efficient training methods, over-fitting, and vanishing gradients. Many solutions have been discussed, from ReLU, batch normalization (to prevent vanishing gradients) to dropouts (to prevent over-fitting) to convolutional layers (to both reduce the number of weights and allow efficient translations and rotations), and we have discussed training methods such as SGD.

Function approximation has blossomed when large labeled data sets became available *and* when the necessary computational power in the form of GPUs arrived. In addition to the data sets, easy to use free software packages such as TensorFlow and Keras have lowered the barrier of entry for research significantly. These packages have played a large role in the success of the field.

We also discussed deep reinforcement learning, where correlations between states cause divergent and unstable learning.

Deep supervised learning constructs complex features from example databases. The examples in these databases are independent, and training converges towards an optimum. In reinforcement learning states are correlated and the training process easily diverges. Two techniques, replay buffer and infrequent weight updates, have been introduced to decouple states enough to allow the training to converge.

Mnih et al. state the following in their paper [454]: “a single architecture can successfully learn control policies in a range of different environments with only very minimal prior knowledge, receiving only the pixels and the game score as inputs, and using the same algorithm, network architecture And hyper-parameters on each game, privy only to the inputs a human player would have.” This architecture, DQN, was demonstrated on Atari games to achieve end-to-end learning from the raw pixels on a wide variety of games that are challenging for humans.

For better feature discovery new methods such as VGG and ResNet have been introduced, achieving very high levels of performance on image recognition tasks. Recurrent neural nets, or LSTMs, are well suited for sequence processing, such as caption generation.

For reinforcement learning, many methods have been devised to improve generalization and reduce divergence. The methods typically use two networks to break possible oscillations between the search and the target values, and they use a form of replay buffers. Many variants for better generalization and more stability have been developed, as the aptly named Rainbow paper documents.

The existence of so many methods suggest the need for a better theoretical understanding of the generalization and convergence processes. Perhaps future work will lead to a unifying theory and methods for deep supervised and reinforcement learning.

Fast and Slow

At this point it is useful to step back and compare the artificial learning methods that we discussed with human learning.

We discussed how deep learning can create function approximators that encode trained knowledge, trained features, just as the heuristics did in heuristic planning. If we look at Kahneman's System 1 (Thinking Fast) and System 2 (Thinking Slow) then using function approximators is like Thinking Fast. Function approximators act fast, like reflexes, providing quick answers delivered by association or lookup. This is in sharp contrast to planning, which provides exact answers through tree traversal. System 2's Thinking Slow performs planning or reasoning.

Historical and Bibliographical Notes

Learning in Games

In game playing, the advantages of neural networks for function approximation have been recognized early. Although the best known success of neural networks in games is undoubtedly AlphaGo, there has also been interesting work performed with supervised learning in games. Before we move the deep reinforcement learning and AlphaGo, let us briefly look at the earlier approaches that have been tried.

Twenty years before AlexNet, in 1990, Tesauro published a paper on Neurogammon [652], his program for playing Backgammon. Neurogammon used supervised learning to train the coefficients of human designed heuristic input features, by using 400 example games played by Tesauro himself. A later version of Neurogammon, TD-Gammon, used reinforcement learning to train the same heuristic input features. TD-Gammon achieved world class human tournament level play [653, 654].

In 1996 Enzenberger created the NeuroGo [192] Go playing program. NeuroGo uses temporal difference learning and backpropagation to train the coefficients of manually crafted heuristics features. Its performance did not surpass that of heuristic programs of its time. In 1990 there was insufficient computer power for training of large networks, nor were there large databases of example games to achieve strong play.

In 1999 Chellapilla and Fogel [132, 217] used a fully connected network to evolve board features in Checkers, achieving a good results. Maddison et al. [425] show how convolutional neural nets can be used in the context of Go to generate moves. Sutskever and Nair [635] used CNNs to learn from Go expert games.

Convolutional neural nets were also used in Go by Clark and Storkey [142, 143] who had used a CNN for supervised learning from a database of human professional games, showing that it outperformed GNU Go and scored wins against Fuego. David et al. [157] report on end to end deep learning in Chess, achieving results on par with Falcon and Crafty, two alpha-beta based programs.



Figure 6.28: Jürgen Schmidhuber

Supervised and Reinforcement Learning

Good review papers on deep learning are [584, 393]. In 1958 Rosenblatt [543] introduced the perceptron, a simple neural network for pattern recognition.

Much work has been done to formulate this backpropagation algorithm [728]. It allowed the training of multi layer networks, paving the way for deep learning's success. In the 1980s McClelland and Rumelhart published an influential work coining the terms parallel distributed processing and connectionism [553]. However, Support Vector Machines and other simpler classifiers became popular [87, 618]. They were easier to reason about and performed better [546]. Later, max-pooling was introduced, and GPU methods were created to speed up backpropagation. The performance of neural nets was further plagued by the vanishing gradient problem, especially in deeper (multi level) networks.

Among many other contributions to both theory and practice of deep learning, Schmidhuber (see Figure 6.28) devised a hierarchy of networks that were pre-trained one network at a time, to overcome this problem [581]. Other methods were later designed, such as ReLU threshold functions and batch renormalization [315]. A good review paper on deep learning is Schmidhuber [584].

There are many books on Keras and TensorFlow, see, e.g., [236].

NeuroGo [192] and Neurogammon [652] used small nets to learn the coefficients of heuristics evaluation functions of the state. They were written in a time when little compute power was available, nor had the necessary advances in deep learning training algorithms occurred, to learn features directly from the state.

Silver et al. [609] discuss the use of temporal difference search in Go, comparing MCTS with TD reinforcement learning. Matthew Lai [380] experiments with DQN in Chess.

Experience replay is an important technique for the success of deep reinforcement learning [453]. Related ideas can be traced to earlier works. Gradient TD methods have been proven to converge for evaluating a fixed policy with a non



Figure 6.29: Andrew Ng

linear value approximator [75]. Relaxing the fixed control policy in neural fitted Q learning algorithm (NFQ) has been reported to work for non linear control with deep networks [538]. NFQ builds on work on stable function approximation [247, 199] and experience replay [409], and more recently on least-squares policy iteration [379].

Andrew Ng (see Figure 6.29) is one of the most active researchers in AI, especially in deep learning and robotics. He is co-founder of Google AI and deeplearning.ai.⁴¹ He received the 2009 IJCAI Computers and Thought award.

Note that pattern recognition has created a wealth of pattern recognition machine learning algorithms, of which deep learning is just one. Other well-known and successful approaches are principal component analysis, decision trees, random forests, support vector machines, k-means clustering, Bayesian networks, Kalman filters, and hidden Markov models. Each of these methods works well in different situations. Good reference works on these methods are [80, 344], or more practically oriented works such as [236, 462]. For large and high dimensional spaces, such as we find in end-to-end learning in games, (deep) neural networks have achieved impressive results, which is the reason we focus on them in this part.

Pointers to further work on recurrent networks (RNN and LSTM) can be found in [254, 251]. An excellent paper describing the various improvements to deep Q-learning is the Rainbow paper [285].

For more theory on deep supervised learning the book by Goodfellow et al. is indispensable [245]. Goodfellow et al. [246] introduced Generative Adversarial Networks, which spawned much follow up research (see Figure 6.30) including dreamlike (but also highly realistic) fake images.⁴²

⁴¹<http://deeplearning.ai>

⁴²Perhaps GANs are the answer to the question posed by Philip K. Dick in one of his science fiction



Figure 6.30: Ian Goodfellow

Sutton and Barto is a main reference on reinforcement learning [638].

stories: “Do Androids Dream of Electric Sheep?” [172].

Chapter 7

Self-Play

The previous chapter showed us how deep learning can find good function approximations, with impressive feature discovery in image recognition and Atari game play. In Chapter 5 we saw how the UCT formula provided effective adaptive node selection to trade off exploration and exploitation. Chapter 4 taught us the basics of how to search large state spaces, such as the search-eval architecture, and the importance of enhancements. We will now see how to combine adaptive search and evaluation by function approximation in a reinforcement learning system that is able to beat the strongest human Go players.

In this chapter planning and learning come together, as do exact and approximate reinforcement learning. The node selection part of MCTS and deep value and policy function approximation join forces. Together they form a Go player that is strong enough to beat the best humans, and beyond. We will discuss in detail how such a high level of play is achieved.

The underlying problems that we discuss in this chapter are how to achieve (1) high performance and (2) stable generalization in a very challenging state space. A state space that is large and high dimensional, where credit assignment distances are long—a state space much more difficult than Atari. First, we will see an approach combining and extending supervised learning and reinforcement learning. We will see how, with meticulous care, correlations between states are avoided in database learning in AlphaGo. Second, we will see how, when learning from databases runs out of steam, another approach is tried in AlphaGo Zero, based on pure reinforcement learning. AlphaGo Zero learns to play Go from scratch, tabula rasa, without any pre-encoded heuristic knowledge.

Third, we will see how yet another program, AlphaZero, is even able to learn to play from scratch in Chess and Shogi, beating the best conventional heuristic planning players. It is quite remarkable that this new self-learning approach beats conventional computer players, built on decades of conventional game playing research. It is even more remarkable that the new players learn to play at world champion level in a matter of days, where humans need years.

The learning goal of this chapter is to understand how the modern self-play

program	planning	learning	input/output	achievement
AlphaGo	MCTS	superv. & self-play	Go	beat human champions
AlphaGo Zero	MCTS	self-play	Go	learn tabula rasa
AlphaZero	MCTS	self-play	Go, Chess, Shogi	generalize three games

Table 7.1: AlphaGo, AlphaGo Zero, AlphaZero

architectures of AlphaGo and AlphaZero work. In the first sections we discuss the AlphaGo achievements, and how the programs work. Then, in Section 7.4 we will look in more depth at current self-play, on earlier works leading up to the successes, and on some open questions. To start answering the open questions we look in Section 7.5 into curriculum learning and transfer learning. Finally, in Section 7.5.2 we will look at areas for further development.

We start with a short impression of the AlphaGo matches, to get an idea of how the achievements were received by the world.

Core Problem

- Can we improve adaptive sampling with function approximation?
- Can we find target examples beyond supervised examples?

Core Concepts

- Self-Learning and Self-Play
- Curriculum learning and transfer learning

This chapter discusses three related programs, all with names that confusingly start with Alpha, and all developed by a team of researchers of the company DeepMind. Each program achieved a major scientific result, published in science' most prestigious journals. The programs are AlphaGo [604], AlphaGo Zero [607], and AlphaZero [606]. See Table 7.1.¹ We will discuss them in subsequent sections of this chapter. Afterwards we present a general discussion of how the methods combined to achieve such high performance (Section 7.4), and conclude with advanced topics in reinforcement learning (Section 7.5).

The information in this chapter is based on their publications [604, 607, 606]. Appendix E contains more technical details of the three systems. Discussing these details would distract from the main ideas in the main text, but they are important for a thorough appreciation of the magnitude of the work.

¹Work on a fourth program, AlphaStar, for StarCraft, is also under way [701]. See Section 8.2.5. Also, a related program, AlphaFold, is successful in protein folding [482].

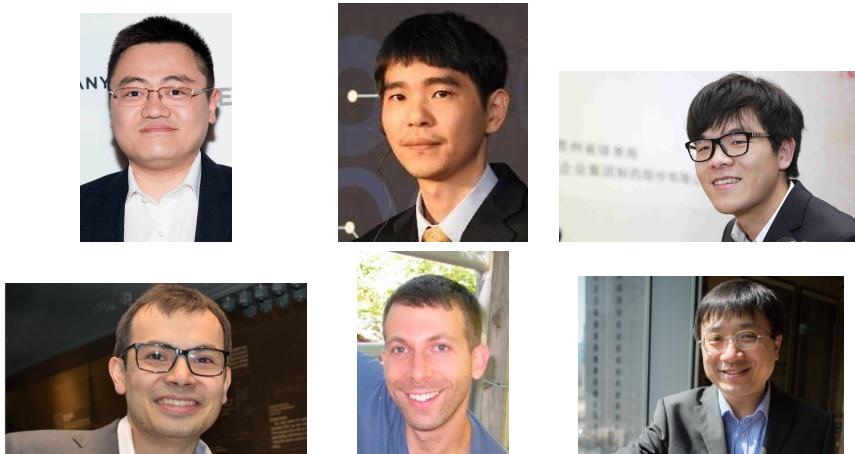


Figure 7.1: Fan Hui, Lee Sedol, Ke Jie; Demis Hassabis, David Silver, Aja Huang
(Left-to-right, top-to-bottom)

7.1 Strong Play: AlphaGo

An early version of AlphaGo beat European champion Fan Hui in London in 2015 in a training match, while AlphaGo was still being developed. A year later, after the program had been developed further, it beat top-ranked Lee Sedol in Seoul in March 2016. In 2017, AlphaGo beat top-ranked Ke Jie in Wuzhen in May 2017. The pictures of the three human Go champions are shown in Figure 7.1, together with three prominent members of the AlphaGo team: Demis Hassabis, David Silver, and Aja Huang.

7.1.1 The AlphaGo Matches

To get an appreciation for the achievement of AlphaGo, and to introduce the program, we will now have a closer look at the matches that it played. Please go to Appendix D for a record of all games played in the Fan Hui tournament, the Lee Sedol tournament, and the Ke Jie tournament. First we discuss the Fan Hui games, then the Lee Sedol games, and finally the Ke Jie games.

Fan Hui

The games against Fan Hui were played in October 2015 in London as part of the development effort of AlphaGo. Fan Hui is the 2013, 2014, and 2015 European Go Champion, then rated at 2p dan. He described the program as very strong and stable. “It seems like a wall. I know AlphaGo is a computer, but if no one told me, maybe I would think the player was a little strange, but a very strong player, a real person.”

Lee Sedol

The games against Lee Sedol were played in May 2016 in Seoul as part of a highly televised media circus. Although there is no official world wide ranking in international Go, in 2016 Lee Sedol was widely considered one of the four best players in the world. Lee Sedol has been a 9p dan professional Go player since 2003. When he played AlphaGo, he had 18 international titles, and was confident that he would win against the computer.

To his surprise, the match ended in a 4-1 win for AlphaGo, with the program playing very strong games. The games against Lee Sedol attracted a large amount of media attention, comparable to the Kasparov-Deep Blue matches two decades earlier. A quite entertaining movie of was made, of one and a half hour length. It can be found on YouTube movie.²

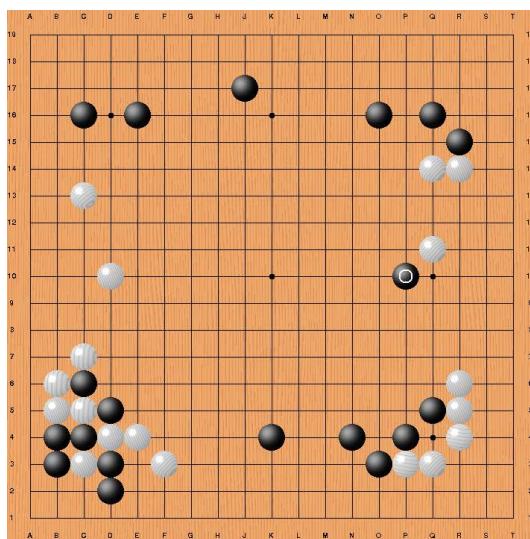


Figure 7.2: “Speechless” Move 37 by AlphaGo in Game 2

The match started with Game 1, which was a win for AlphaGo. Then, in Game 2, Alphago played a move on the right-hand side of the board, that shocked spectators (see Figure 7.2). Commentators said that it was a very strange move, many thought that it was a mistake. Lee Sedol took nearly fifteen minutes to formulate a response. Fan Hui explained that, at first, he also could not believe his eyes. But then he saw the beauty in this rather unusual move. Indeed, the move changed the course of the game and AlphaGo went on to win. At the post-game press conference Lee Sedol was in shock. He said: “Yesterday, I was surprised,” referring to his loss in Game 1. “But today I am speechless. If you look at the way the game was played, I admit, it was a very clear loss on my part. From the very beginning of the game, there was not a moment in time when I felt that

²<https://www.alphagomovie.com>

I was leading."

Fan Hui gives more commentary, showing why this move looks so profound from a human perspective. He writes: "Black 37 is one of the two moves from this match sure to go down in Go history. This move proved so stunning that, when it appeared on the screen, many players thought the stone had been put down in the wrong place."

"On seeing Black 37, I wrote down the following: "Here?! This goes beyond my understanding. Globally, there's nothing wrong with it, it's going in the right direction [...] and AlphaGo always pays special attention to coordinating the stones. It seems anything is possible in Go! Everyone will be talking about this move! A human would never dare play it, it's too difficult to estimate. But AlphaGo can. Perhaps this move is a sign of its confidence."

"This move made a deep impression on me during the game. I experienced first confusion, then shock, and finally delight. It reminded me of an old Chinese saying: "A beginner plays the corners, an average player the sides; but a master controls the center." These days, due to the convergence of strengths and the pressure of competition, something close to the opposite is true, with most players focusing on the corners and sides. In contrast, AlphaGo's talent for central control is second to none. Perhaps, through AlphaGo, we too can become the 'masters' of which the proverb speaks."

"Returning to the game, we may say that Black 37 casts an invisible net across the board. Together with the lower side, Black's shoulder hit creates potential all across the center. Although it helps White make territory on the right, the presence of White means that a Black invasion there would not have been valuable anyway. Of course, Black should be reluctant to give away fourth-line territory too easily, but one must give to get."

"After the match, when I examined the data back at DeepMind, I saw that AlphaGo had not even been thinking about 37 only a few moves before. It had been expecting a different move, and its data indicated that a human player would hardly consider the shoulder hit a possibility. It was only when White played 36 that AlphaGo discovered 37, and boldly decided that this move would work even better."

"The pace of the game was much slower than the previous day, so Lee had already gone out to smoke before 37. The minute he caught sight of AlphaGo's reply, he stared blankly at the board. Then he smiled, sat down, and started thinking. The longer he thought, the more serious his expression became, while the clock ran down little by little" [310]. And indeed, this game was also won by AlphaGo.

Lee continued to play three more games in the match, winning Game 4 by a beautiful move, but losing the match 1–4, the first time in history that a Go champion of his caliber had lost a match to a computer program.

Ke Jie

A year later another match was played, this time in China, against the Chinese champion. Ke Jie was ranked number one in the Korean, Japanese, and Chinese

ranking systems at the time of the match. He was also first among all human players worldwide under Rémi Coulom’s ranking system [149].

On 23–27 May 2017 a three game match was played in Wuzhen, China. The match was won 3–0 by AlphaGo. AlphaGo was subsequently awarded a professional 9-dan title by the Chinese Weiqi Association.

As preparation for this match DeepMind had organized an online 60 game tournament against top pros, from 29 December 2016 to 4 January 2017. This version of AlphaGo ran on a strong machine that used 4 specially developed Tensor Processing Units (TPU). TPUs, specially developed by Google for fast and efficient tensor processing, are described in Section E.1. This version of AlphaGo achieved a very high Elo rating of 4858 against the online pros [607], it showed that the program was 3 stone stronger than the version that defeated Lee Sedol. DeepMind named this match version *AlphaGo Master*. It was a refined and further developed version of AlphaGo, although the basic architecture was unchanged, with a combination of supervised and reinforcement learning (even though DeepMind was, at the time, already working on a re-design, AlphaGo Zero).

After this impression of the matches and achievements of AlphaGo, it is time to have a look at the methods that were used to achieve these phenomenal results.

7.1.2 What

What technology did DeepMind use in AlphaGo to achieve such a strong level of play, 10 years before most experts had expected this milestone to be reached?

It turns out that in AlphaGo there was no magic bullet, no single groundbreaking new technology that had been invented. What the DeepMind team did, was to take all existing approaches, and improve and refine them greatly, and find a way for the different technologies to work together in a new and highly impressive way.³

The AlphaGo design follows the search-eval architecture. For search, it employs a version of MCTS. For eval, it uses an elaborate and highly advanced combination of first supervised learning and then reinforcement learning, and it uses shape databases for the MCTS-playouts. For the training positions for both supervised and reinforcement learning great care is taken to de-correlate the training positions, to achieve stable training. Three neural nets are used: a fast shape policy net for the playouts, a slow policy net for exploration/exploitation selection, and a value net for the leaf values of the MCTS tree. In addition, the slow policy net is trained by supervised and by reinforcement learning, giving a total of no less than four training methods for three networks. Figure 7.3 reproduces the familiar AlphaGo networks picture from [604].

After this high-level description, let us look in more detail into the AlphaGo methods.

³The term *magic bullet* can conceivably apply to the second version, AlphaGo Zero, a re-design fully based on self-learning, that is more revolutionary in design.

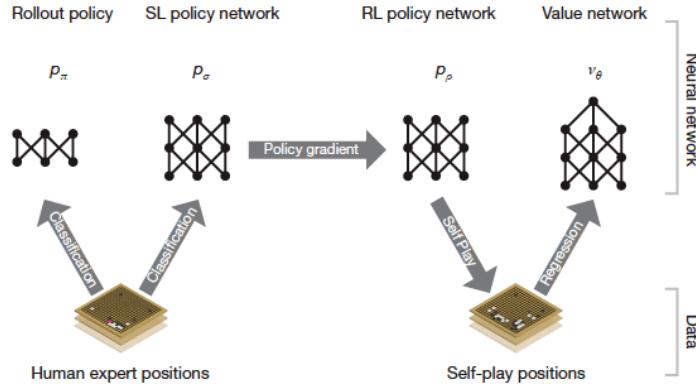


Figure 7.3: AlphaGo Networks: [604]

7.1.3 How

AlphaGo is a complicated program, that magnificently combines supervised and reinforcement learning in a winning way.

First, we will discuss the supervised learning in AlphaGo. AlphaGo uses supervised learning to learn from grand-master games. A database of 160,000 grand-master games is used to train a slower selection policy network by supervised learning (28 million positions). The positions from grand-master games consist of a board position and a best action (the move played by the grand-masters). This policy network is used in the MCTS selection operation.⁴ The supervised learning is used to pre-train the network to a certain (high) level of play, and then positions from self-play are used to improve it further.

Second, in addition to supervised learning, AlphaGo uses reinforcement learning for three networks: for two separate policy networks and for a value network. The reinforcement learning agent is an MCTS/neural net program that trains the slow policy network against an “environment” that really is a database with games that were played against a copy of itself. Note that quite some preparation is used to tune and de-correlate the training positions in the database. This part of the training, against a database of self-play games, is often simply called self-play. It is not, however, straight, unattended, autonomous, self-play, but a kind of learning from a database of self-play games, and extensive tuning, selection and de-correlation activities.

Third, a value network is trained based on a database of 30 million self-play positions. For the MCTS leaf values a mix is used of the value network and of the outcome of MCTS roll outs using the fast roll out policy network. The mixing of these values is governed by the new λ parameter. Note that in AlphaGo the random playouts of MCTS are replaced by playouts using the fast policy net.

⁴Recall the four MCTS operations: select, expand, roll-out and backup.

Fourth, a game database is used to train a fast policy network by supervised learning. The fast policy network is trained with positions from separate games to prevent over-fitting and loops due to correlation between subsequent game states. A fast roll-out policy network based on 3×3 patterns is created for MCTS playouts.

To summarize, AlphaGo uses the four training methods to train three networks:

1. a “slow” selection policy net, first trained by supervised learning from grandmaster games, then further refined through self-play positions
2. a value net for position evaluation, trained by self-play positions
3. a “fast” roll out policy net trained by supervised learning from human games, used for MCTS playouts

Performance of the Three Networks

We will now discuss these networks in more detail.

Fast Roll Out Policy Network The fast roll out policy network is based on 3×3 patterns. Its function is to replace (or enhance) the random roll outs of MCTS. It achieves only 24% accuracy on a test set, but it replies with an answer in $2\mu s$, which is 1500 times faster than the slow policy network, on the hardware used by the AlphaGo team.

Selection Policy Network The network trained by supervised learning is trained on 29 million positions for classification of expert moves on a data set of internet games (split into a test set of one million positions, and a training set of 28 million positions). This policy network achieves 57% accuracy of correct actions after training. Its time to compute an answer is 3 ms. It is trained by asynchronous stochastic gradient descent. The supervised training took three weeks; 340 million training steps were performed.

The policy network is further refined by reinforcement learning with self-play positions. It was trained for one full day of self-play of 10,000 mini-batches of 128 games, carefully managed to prevent correlations using positions from a pool of opponents. The training algorithm is policy gradient descent, with the REINFORCE algorithm [638]. This network performs quite well. Without any search at all, it won 85% of games against Pachi, a strong open source MCTS program.⁵ The reinforcement learning is important, since a no-search network based on supervised learning won only 11% against Pachi. Curiously, it is the weaker supervised net that was used in MCTS selection in AlphaGo. The paper speculates that this is because of greater variation in the supervised network, it chooses from a wider repertoire of “best” moves, whereas the reinforcement network chooses always the single best move [604]. Still, it is surprising that a weaker network is used in favor of a stronger network and more work is needed to fully understand the reasons.

Value Network The value network is trained as a regression of the policy network. The value represents the probability of winning. For the value network 30

⁵Pachi code is at <http://pachi.or.cz>

million training positions were used from self-play games. To avoid over-fitting the training positions were created so as not to be correlated, using a small self-play search based on the positions that were used for the supervised selection policy network. The training algorithm is stochastic gradient descent as in the supervised training phase, based on the mean squared error between the predicted values and the observed rewards. This training phase took 50 GPUs one week.

Many more details of the various training methods can be found in Appendix E or in the original publication [604].

7.1.4 Results

Quite some development and experimentation were needed for AlphaGo to reach the level of play that it did.

Figure 7.4 shows the Elo rating of AlphaGo compared to other programs and compared to Fan Hui. AlphaGo is clearly superior. The graph indicates a playing strength of 5p dan. Since publication of the article in 2016, AlphaGo has been further refined, and the performance has continued to improve. The version playing against Lee Sedol and Ke Jie was significantly stronger, at 9p dan.

AlphaGo has three networks. Therefore, it is instructive to see how each network influences performance. Figure 7.5 shows the impact of turning some of the three networks off. We see that each network has a significant contribution, and that the contribution of the networks is independent in the sense that they all complement and improve each other.

AlphaGo uses a significant amount of compute power, for training, but also during tournament play-time. Therefore, it is interesting to see how more compute power influences playing strength. Figure 7.6 shows the impact of the extra hardware for the distributed version of AlphaGo. Clearly the use of more than one GPU is advantageous, as is the use of multi-threading. The use of a cluster of machines (dark blue, distributed) gives an increase in performance of around 200 Elo points.

To give an idea of what AlphaGo's hardware looks like, Figure 7.7 shows a CPU, a GPU, and a TPU.

Playing Strength in Go

At this point it is interesting to compare the playing strength of Go programs that have been written. See Figure 7.8. The programs fall into three categories, each with their own programming paradigm. First are the programs based on heuristic planning, the Chess-style programs. GNU Go is a well-known example of this group of programs. The heuristics in these programs are hand-coded. Their level of play was at medium amateur level. Then come the adaptive sampling MCTS-based programs, that reached strong amateur level. Examples of these programs are MoGo and CrazyStone. Finally come the AlphaGo programs, of MCTS combined with neural nets. They reach super-human performance. The Figure also shows the program Fine Art, by Tencent.

The Figure clearly shows how the performance improved by the different paradigms.

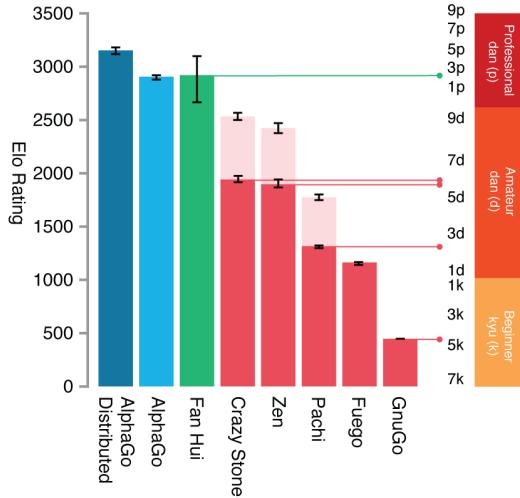


Figure 7.4: AlphaGo Elo Rating (pale red is 4 handicap stones) [604]

Open Questions

The AlphaGo team made important scientific progress, they strongly pushed the field of reinforcement learning forward, and the attention generated by the 2016 and 2017 matches did much to strengthen the worldwide interest in AI. The successes and the design of AlphaGo attracted much attention by AI researchers and others. However, AlphaGo is a complicated program, and a natural question is if its design could be simplified. Among the questions raised by the AlphaGo success were the following:

- The overall architecture is complicated, with four training methods for three nets. Is a simpler architecture possible? Can self-play replace supervised learning entirely? Can the intermediate, highly processed, database of self-play positions be skipped, to achieve true self-play?
- AlphaGo trained its networks for weeks. Can the training be made faster? This question was already addressed in part by Google, with the development of TPUs. It was clear that AI and machine learning needed all the power they could get, and TPUs provide more speed, specifically optimized for neural network training.
- Looking more specifically at AlphaGo, a natural question to ask is: “Why does the supervised learning policy network work better in MCTS selection, whereas the reinforcement learning policy network has higher accuracy?”
- Furthermore, at the leaf nodes of the MCTS tree both fast roll outs are computed and a value function from the value network is used. The two are mixed by a new parameter, λ . Would a more elegant solution exist?

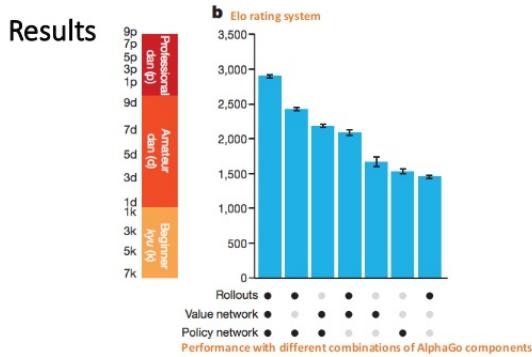


Figure 7.5: AlphaGo Networks [604]

- The network architecture is elaborate, with 13 or 14 layers (see Appendix E). The training has many hyper-parameters. The design and tuning cost time and effort. Can this network architecture and the tuning of hyper-parameters be made simpler?
- Then, just as with Deep Blue-Kasparov, the natural question was raised: “What next?” With the human Go champions defeated, what was to be the next Drosophila? Was it to be learning from scratch, would it be taking on general intelligence, or would there be another game?

As to this last question, we will see the answer in the next sections.

7.2 Self-Play: AlphaGo Zero

Although AlphaGo achieved an impressive level of play, its architecture was complicated, with different kinds of training, extensive de-correlation, many hyper-parameters, and very large computational demands. The engineering and tuning effort was quite large, and hindered reproducibility and generalization of the approach to other games and other domains. Therefore, the next step was to see if the architecture could be simplified, to see if a fully-reinforcement learning approach would work, and if it could learn cheaper.

The reinforcement learning-only version of AlphaGo was called AlphaGo Zero. Instead of learning from grand-master games, it learned to play the game of Go from scratch [607]. Only the rules of the game were known, and the program learned to play beyond world championship level all by itself. The program played *tabula rasa*, from a blank slate, with no heuristics, and no grand-master games or supervised learning. Of course, the rules of the game were coded into the program in some way, and the input and output layer of the net are specifically designed for Go, and quite some refinement went into the network architecture and choice

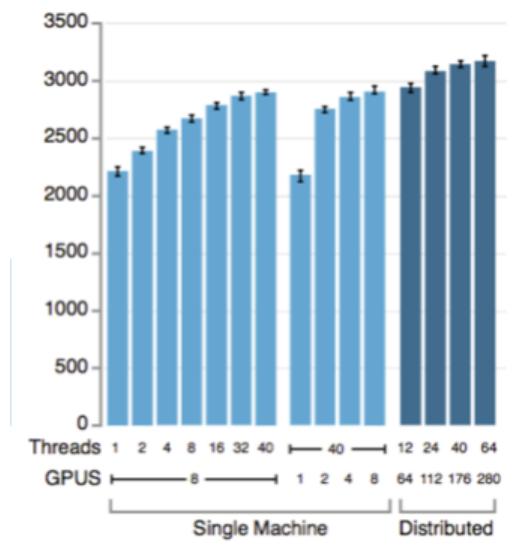


Figure 7.6: Elo Rating of AlphaGo Distributed Versions [604]

of hyper-parameters for the self-play to work. All these are described by Silver et al. [607] and have been discussed in talks at scientific conferences.⁶

7.2.1 What

Less is More

AlphaGo Zero is based purely on learning by self-play. A conceptual image with the training loop is shown in Figure 7.9 on page 225, to be explained shortly. It is reinforcement-learning only, with a single neural network, and all knowledge is built up by playing against itself, again, and again, and again. As such, the design of the program is one of elegant simplicity, certainly when compared to the four training methods and three nets of AlphaGo.

The great British computer scientist Tony Hoare is credited with saying: “Inside every large problem there is a small problem struggling to get out.” This certainly applies to AlphaGo. The team fully redesigned the program from four training methods and three function approximators to one each: only training by self-play, and a single unified policy/value net (a two-headed net, a net with a policy head and a value head).

Furthermore, MCTS in AlphaGo Zero no longer does any play outs at all. At the leaves of the MCTS tree the value function provided by the value head of the

⁶The field of General Game Playing also has the goal to learn to play a game with zero heuristic knowledge, and more: to learn how to play any game from scratch. Although GGP methods are more general, Go is a significantly more complex game than typical GGP games are. See Section 8.2.6 for GGP.

Difference Between

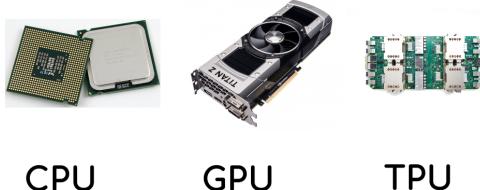


Figure 7.7: CPU, GPU, and TPU

single network is used. The mixing parameter λ is no longer needed. In fact, since MCTS no longer does random playouts, the Monte Carlo in its name is no longer accurate. The algorithm should really be called something else. DQTS for Deep Q-network Tree Search would be better.

AlphaGo Zero no longer pre-trains its nets with grand-master games before it starts the reinforcement learning. It learns all knowledge from scratch. From a reinforcement learning point of view the fact that the entire spectrum of play was learned, in one long stable training sequence from absolute beginner to world class, is a very impressive achievement. Clearly, problems with generality of training, the stability of the training and long range credit assignment were overcome in AlphaGo Zero.

7.2.2 How

To understand what is inside AlphaGo Zero, we will delve into self-learning and self-play. First, a word on terminology. We know that reinforcement learning is learning by interaction with an environment. *Self-learning* is when this environment that we are learning from is the same model as the one that is being learned. A dragon biting in its own tail. When our problem domain at hand is a game we call this kind of self-learning reinforcement learning by *self-play*.

Self-learning uses both planning and training. In self-play, the search-eval architecture gets a new element: an extra learning loop is added around the search-eval functions, creating a self-learning, recursive, architecture that is able to do tabula rasa learning. Table 7.2 shows the functions of the search-eval architecture and the main algorithms as they are discussed in the chapters of this book.

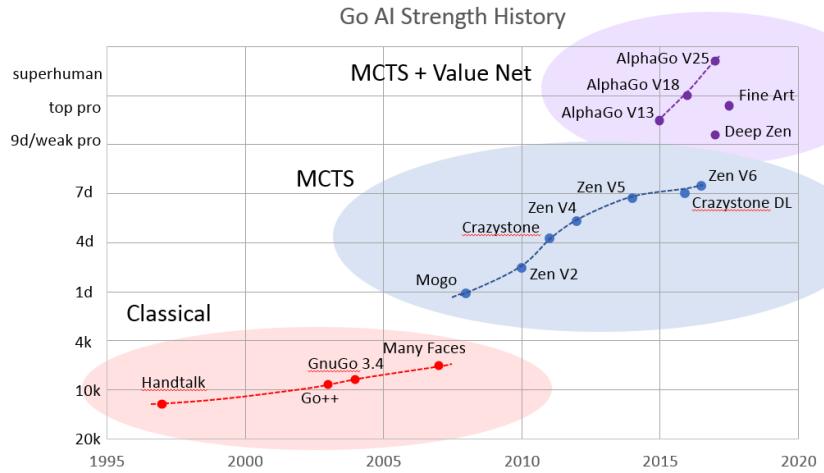


Figure 7.8: Go Playing Strength over the Years [12]

	Ch 4	Ch 5	Ch 6	Ch 7
exact: search	alpha-beta	MCTS	-	MCTS
select	left-to-right	UCT	-	P-UCT/policy net
backup	minimax	average	-	average
approx: eval	heuristic	roll-out	DQN	value net
self-play	-	-	-	self-play

Table 7.2: Search-Eval Architecture

7.2.3 Self-Learning Loop

Conceptually self-learning and self-play are as simple as they are elegant: a training loop around a standard search-eval player. See Listing 7.1 for pseudo code and Figure 7.9 for a conceptual picture. In the first line of the code there is an outer loop that determines how many tournaments the self-play learning is performed. This single line of code (line 1) is the self-play loop.

Figure 7.9 illustrates the concept of self-play. For each position a best move is found by a searcher that does look ahead and uses a value network as evaluator. With this mechanism a large number of games is played, generating many positions, best moves, and game outcomes. These are then used as examples that are input for the subsequent training phase of the policy/value net, that will be used in the next self-play iteration.

The structure of self-play is a straightforward onion-like system of functions wrapped around each other (see Figure 7.10). The arrows in the Figure point to the lower functions that are used by the higher function to generate the training

```

1 for it in range (1, max_iterations): # do a curric. of self-play tourn.
2   for game in range(1, max_games): # play a tourn. of games; then train
3     trim(triples) # if buffer full: replace old entries
4     while not game_over(): # generate the moves of one game
5       game_pairs += mcts(eval(net)) # move is a (state,action) pair
6       triples += add(games_pairs, game_outcome()) # add win/lose to buf
7       net = train(net, triples) # retrain with (state,action,outc) triples

```

Listing 7.1: Self-Play Loop

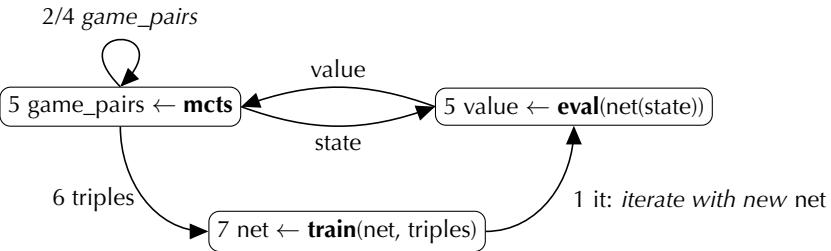


Figure 7.9: A Diagram of Self-Learning

examples. The dashed lines indicate the policy/value network, and its training.

Let us do an outside-in walk-through of this onion using the code in Listing 7.1. Line 1 is the main self-play loop. It iterates the execution of the curriculum of self-play tournaments. Line 2 executes the training iterations (tournaments). The network is trained with $(state, action, outcome)$ triples. The game outcomes are collected in the triples buffer. When the buffer is full, the oldest entries are deleted (it is a replay buffer with randomized sampling to break correlations). Line 3 performs the replay buffer replacement of old entries. Then an inner loop performs games of self-play with the current net as eval (a mini tournament, fourth line). The games of this tournament result in $(state, action)$ pairs. Together with the game outcome z (win, loss, or draw) they are stored in the triples buffer. Line 4 plays this game to create the pairs for each move, and the outcome of the game. Line 5 calls MCTS to generate an action in each board position. MCTS performs tree simulations where it uses the policy net in P-UCT selection, and the value net at the MCTS leaves. Line 6 adds the outcome of each game to the $(state,action)$ pairs, to make the triples for the network to train on. Note that since the network is a two headed values/policy net, both an action and a outcome are needed for network training. On the last line this triples buffer is then used to train the network. The newly trained network is then used in the next self-play iteration as the evaluation function by the searcher. Then another tournament is played, using the searcher's look-ahead to generate a next batch of higher quality examples.

We can also peel the onion by looking outside-in at the data that is generated. A self-play curriculum consists of tournaments. At the end of a curriculum a trained player is ready to be used. Each tournament consists of games (25,000 in AlphaGo Zero), at the end of the tournament the buffer is filled with example

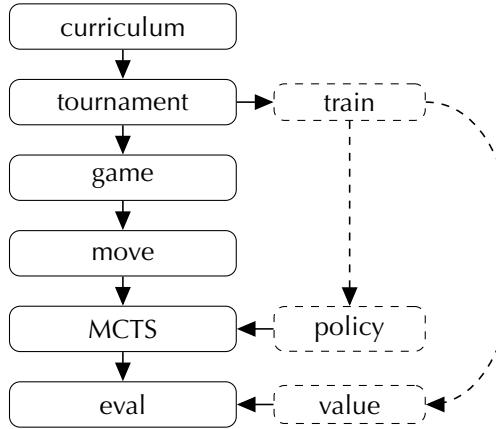


Figure 7.10: Levels in Self-Play

positions to train the network further. Each game consists of positions and moves, at the end of a game the *outcome* can be added to the moves (*state,action*), ready to be added to the buffer. Each move is computed by an MCTS search, at the end of the search a best action is returned, to be added to the list of pairs of the game buffer. Each MCTS search consists is simulations (1600 in AlphaGo Zero), at the end of each simulation another node is added to the MCTS tree.

Although conceptually simple, there are subtle interactions occurring during self-play that affect generalization and stability. To achieve good generalization and stable training much research and understanding of the games is required. We will see some later in this chapter.

Among those challenges are the creation of good quality training targets of sufficient generality (diversity) so that quality of play keeps improving, without hitting a plateau, or degenerating into a limited kind of play. The task of MCTS look-ahead is to generate good quality training samples. Another challenge is to achieve convergence in training. The randomized replay buffer is used to break correlations between consecutive states.

The principle of self-play is simple. Getting the details to work to yield a very high level of play required many years of research. Section 7.4.4 provides an overview of earlier works, many of which tried self-play.

7.2.4 Search-Eval Architecture

Let us look at self-play from the perspective of the search-eval architecture. Self-play programs have a search function and an evaluation function just like all game playing programs. In AlphaGo Zero the search function is MCTS, and the evaluation function is a deep network (or more than one, as in AlphaGo).

A self-play program has an extra element, the self learning loop (see Figure 7.9).

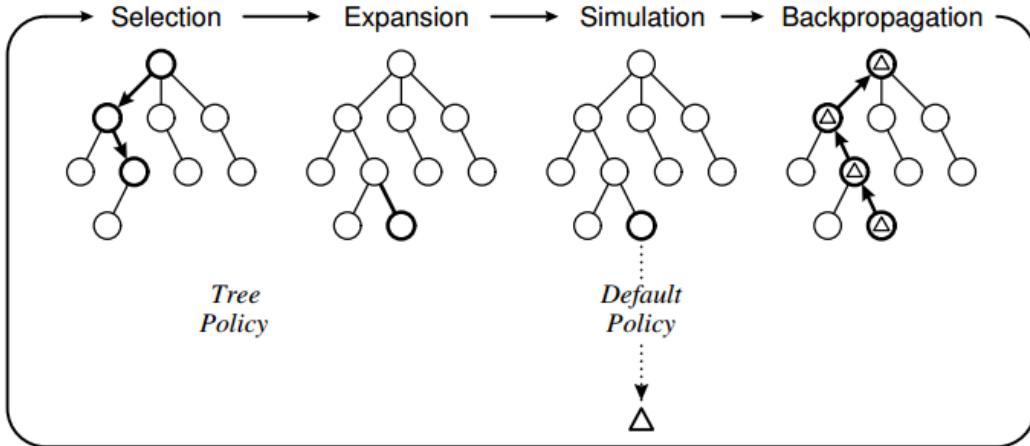


Figure 7.11: Monte Carlo Tree Search [130]

Evaluation

Eval is a function approximator, for modern end-to-end systems often a deep neural net. The function approximator is trained, and is then queried during play. Thus, eval has two roles:

1. eval is the *learning element* of the self-play scheme, that is trained in the training phase of self-play;
2. eval is a *policy/value function* to be queried for the policy and the value of a state during the play-phase.

In AlphaGo Zero the function approximator is used in two places: the policy head of the net is used in the P-UCT move selection, and the value head of the net is used in the backpropagation leaves of the MCTS tree. See Figure 7.11 for the four operations of MCTS.

Search

In self-play the search function also has two roles: to look ahead to generate examples in the training phase and to look ahead during actual play. For both roles the look ahead algorithm does exactly the same thing. It is only its *role* in the self-play scheme that is different during training and use:

1. search generates the examples for the training phase. By looking ahead, it generates better quality examples for the training.
2. next, when the self-play training has finished, and the program is used for normal game play, the search function performs look-ahead using the net to find the best action in actual play.

The two roles might be compared to the training and test phases in supervised learning.

Neural Network Training

AlphaGo used separate convolutional neural networks (policy nets and a value net). AlphaGo Zero switched to a single residual net with two heads (ResNets are discussed in Section 6.2.1). To understand why the two nets could so easily be folded into a single two-headed network we should return to Chapter 3 where it was noted that policy and value are really two sides of the same coin: the best action is the one with the highest reward and vice versa.

It may seem counter-intuitive that a simpler architecture, with just a single two headed evaluation network would work better for such a complex game (Tony Hoare would say: “I told you so”). The simpler architecture is easier to understand and improve. Simplicity may have helped the development team to reach the impressive reinforcement learning results.

The two-headed optimization target is related to multi-task learning problems [119, 491]. In multi-task learning related learning tasks are learned at the same time. Of course, in AlphaGo Zero the two headed targets must learn a single problem. The two heads have a shared representation in the single network, benefitting from the improved regularization, as in multi-task learning.

The main change in the training scheme of AlphaGo Zero is that MCTS is now an integral part of a reinforcement training loop, where in the previous program examples were taken from a database of self-played positions. Each MCTS search gets 1600 simulations, which is enough to generate high quality training examples, giving better generalization.

It may again seem counter-intuitive, but to play stronger, AlphaGo Zero does *not* use the boost that initialization with grand-master games provides. In order to reach higher, it starts lower. Since all learning is now by reinforcement, the training process must now be even more stable. The slightest problems in overfitting or correlation between states that throws off the training will show up as levelling off of the Elo rating. AlphaGo Zero uses MCTS to do more exploration early in the search, and extra noise was added to the root node to ensure all moves are explored.

The AlphaGo Zero paper provides a comparison with the old AlphaGo showing how in relatively little training time self-play reinforcement learning is able to surpass supervised database learning (see Figure 7.12).

Of course, it remains important to ensure diversity, since it is all too easy for the self-play opponents to co-teach each other into a certain obscure part of Go knowledge, without discovering the breadth of the game.

7.2.5 Results

We will now turn to the playing strength of AlphaGo Zero.

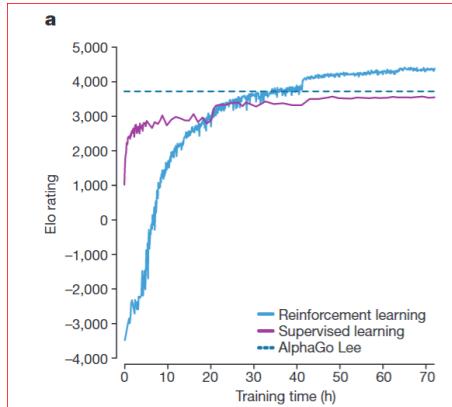


Figure 7.12: Comparison of Supervised Learning and Self-Play Reinforcement Learning [607]

Playing Strength

In their paper Silver et al. [607] describe that learning progressed smoothly throughout the training, and did not suffer from the oscillations or catastrophic forgetting that have been noted in previous literature [215, 278].

Training also progressed quickly. AlphaGo Zero outperformed the original AlphaGo after just 36 hours. The training time for the version of AlphaGo that played Lee Sedol was several months. Furthermore AlphaGo Zero used a single machine with 4 tensor processing units, whereas AlphaGo Lee was distributed over many machines and used 48 TPUs. Figure 7.13 shows the performance of AlphaGo Zero. Also shown is the performance of the raw network, without MCTS search. The importance of MCTS is large, around 2000 Elo points.

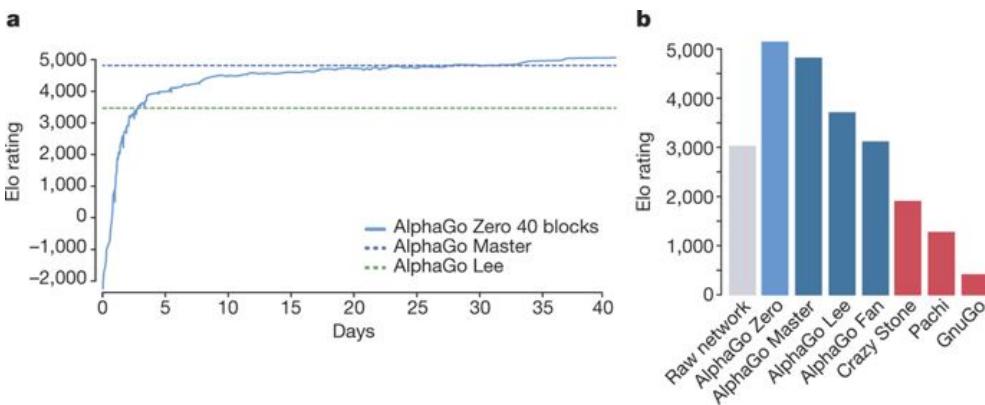


Figure 7.13: Performance of AlphaGo Zero [607]

Curriculum Learning

The result of the more accurate self-play is not only *better* but also *faster* training, from months for AlphaGo to days for AlphaGo Zero. In part this is because of what is known as *curriculum learning* [63]: it is quicker to learn a goal in many small steps from easy to hard, than in one large step. In self-play, AlphaGo Zero's neural network always trained against an opponent of a comparable level, or slightly better. Learning to play against a slightly better opponent costs less computational effort than (supervised) training against grand-master games, where the network has to learn to overcome a large error without pre-training. In self-play, the net is always training against an opponent of its own level, allowing a smoother learning path, and, importantly, requiring less computational effort [607]. In Section 7.5.1 we will discuss curriculum learning in more depth.

Tabula Rasa

AlphaGo Zero's reinforcement learning is truly learning from scratch. The paper notes that AlphaGo Zero discovered a remarkable level of Go knowledge during its self-play training process. This knowledge included not only fundamental elements of human Go knowledge, but also non standard strategies beyond the scope of traditional Go knowledge.

Joseki are standard corner openings that all Go players are familiar with as they are learning to play the game. There are beginner's and advanced joseki. Over the course of its learning, AlphaGo Zero learned beginner to advanced joseki. It is interesting to study how it did so, as it reveals the progression of AlphaGo Zero's Go intelligence. Figure 7.14 shows how the program learned over the course of a few hours of self-play. Not to anthropomorphize too much, but you can see the little program getting smarter.

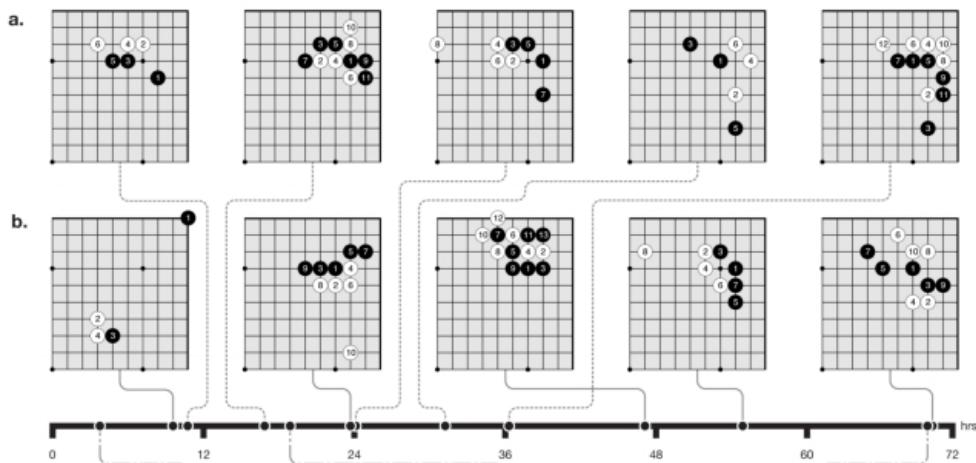


Figure 7.14: AlphaGo Zero is Learning Joseki [607]

The top row shows five joseki that AlphaGo Zero discovered. The first joseki is one of the standard beginner's openings out of Go theory. As we move to the right, more difficult joseki are learned, with stones being played in looser configurations.

The bottom row shows five joseki favored at different stages of the self-play training. It starts with a preference for a weak corner move. After 10 more hours of training, a better 3-3 corner sequence is favored. More training reveals more, and better, variations.

For a human Go player, it is remarkable to see this kind of progression in computer play, reminding them of the time when they themselves discovered these joseki. With such evidence of the computer's learning, it is hard not to anthropomorphize AlphaGo Zero (think of the computer as a human being).

Open Questions Revisited

Most of AlphaGo's open questions on page 220 are addressed by AlphaGo Zero. The new program has a simpler architecture, and it addressed a major AI question: learning from scratch, with surprising success.

The architecture of the program has changed fundamentally: from a combination of supervised and reinforcement learning based on databases of positions, to a self-play loop in which the searcher was incorporated in the training loop (Figure 7.9 and Listing 7.1). Surely earlier works on self-play were an inspiration for AlphaGo Zero (see Section 7.4.4). In any event, a set of important open questions was answered.

A remaining question was the Drosophila question: can we construct a player that plays different games (the general game player question [509, 235])? We will now turn to one answer.

7.3 General Play: AlphaZero

After defeating the best human Go players, and after having created the first Tabula Rasa self learning Go computer, the team went on to study this other long standing challenge in AI: constructing a general game player. If AlphaGo Zero's architecture was able to learn Go from scratch, then perhaps it would be able to learn any game? A year after the AlphaGo Zero publication they published a positive result, showing how AlphaGo Zero's design could also play Chess and Shogi.

7.3.1 What

At the end of 2017 a pre-print [605] reported that the AlphaGo Zero approach also worked on Chess and Shogi (a kind of Japanese Chess). A year later the formal publication followed [606]. The team had been able to teach the same search-eval architecture (MCTS with 20 ResNet blocks) to learn Go, Chess and Shogi, thus achieving a form of *general* tabula rasa learning, albeit limited to three

games. The program, named AlphaZero, beat the strongest conventional players in Go, Chess and Shogi (respectively AlphaGo Zero, the program Stockfish, and the program Elmo).

Go		Chess		Shogi	
Feature	Planes	Feature	Planes	Feature	Planes
P1 stone	1	P1 piece	6	P1 piece	14
		P2 piece	6	P2 piece	14
		Repetitions	2	Repetitions	3
Colour	1	P1 prisoner count	7	P1 prisoner count	7
		P2 prisoner count	7	P2 prisoner count	7
		Colour	1	Colour	1
		Total move count	1	Total move count	1
		P1 castling	2		
Total	17	P2 castling	2		
		No-progress count	1		
Total		119		362	

Table 7.3: Different Input Planes (first set repeated 8 times to capture history) [606]

Note that the hand-crafted input and output layers of the networks are different for each game. Table 7.3 shows the differences, the more complicated move patterns of Chess and Shogi cause the input planes to be significantly more elaborate than for Go (see the paper for details). Furthermore, the weights in the trained networks are also different. Strict transfer learning was not achieved: a trained Shogi net is not able to play Go well.

AlphaGo Zero’s achievement of a general game learning architecture is remarkable for more than one reason. First, it is remarkable that self-play is so powerful that it can learn from scratch *and* become so strong that it beats the current world champions. Second, it is remarkable that different games can be learned by the same network architecture. Perhaps, some kind of element of general learning must be present in the structure of the ResNets of AlphaZero, although the networks that are learned differ between games: a network trained for Chess cannot play Go well. Third, for Chess and Shogi all previous strong programs had been based on the heuristic planning approach of alpha-beta with domain specific heuristics [533], now they are beaten by a general learning approach using self-play function approximation.

Go, Chess and Shogi

That AlphaZero can play three different games with one architecture is surprising. The three games, though all board games, are quite different. Go is a static game of strategy. Stones, once placed, do not move, and are rarely captured. Each stone played is therefore of strategic importance. In Chess the pieces move, it is a dynamic game where tactics are important, and the possibility of sudden death

is an important element of play: in Chess it is possible to win in the middle of the game by capturing the king. No such concept exists in Go. Indeed, one element of human Chess play is to move the king to a safe position to reduce the level of stress in the game. Shogi, sometimes called Japanese Chess, is even more dynamic. Pieces, once captured, can be returned in the game, changing loyalty, creating even more complex game dynamics.⁷

As all top Chess programs, Stockfish follows the classic search-eval design of alpha-beta and a heuristic evaluation function, that we have seen in Sunfish (see Section 4.6.1) and Deep Blue (see Section 2.3.3). Among the enhancements of Stockfish are forward pruning, iterative deepening, transposition tables, quiescence, piece-square tables, null window, null move, history heuristics, search extensions, search reductions, static exchange evaluation, an opening book and an end game database. Chapter 4 provides background explanations on most of these enhancements. In Stockfish, all these enhancements have been carefully tested and tuned, for many years [542].

The state space complexity of Shogi is substantially larger than Chess: 10^{71} versus 10^{47} [5, 314]. The strongest Shogi programs have recently started defeating human champions [682]. Elmo, one of these programs, is designed like Chess programs, around alpha-beta search and a heuristic evaluation function, with many carefully tuned and tested game dependent enhancements.

It would seem that since the move patterns, the tactics, and the strategies to be learned are very different, that it would require a different kind of program to play well. Indeed, this is the conventional approach: a developer crafts a program specifically for a certain game, incorporating all kinds of game specific knowledge and heuristics into the code. For Go, the game is so subtle and difficult that no scientist was able to elicitate successful heuristics, and knowledge had to be learned by machine learning from carefully crafted game databases (initially, in AlphaGo) or through self-play (eventually, in AlphaGo Zero).

The conventional game-specific approach has worked well for Chess, Shogi and other games so far. The general programs created by GGP generally trade-off generality for playing strength (see Section 8.2.6). AlphaZero is the first to show that it is possible to be general *and* strong. Indeed, AlphaZero has shown that the differences in move patterns are not such that a different kind of program is needed to *learn* to play well.

7.3.2 How

One General Architecture

Although it would seem that the different kinds of games require a different program design, AlphaZero's results have proven this conventional wisdom wrong, at least for the *learning* of three board games. All that needed to be changed were the input and output layers of the function approximator. The self-play training algorithms, network architecture, search algorithm, most training hyper-parameters⁸

⁷Pieces are of the same color, but are wedge-shaped, and change orientation when switching sides.

⁸The learning rate decay scheme was slightly different for Go than for Chess and Shogi.

and compute hardware are all the same. The self-play program was then able to learn to play the game beyond world class level. Either the game dynamics of Go, Chess and Shogi are less different than they seemed, or the structure of self-play with ResNets is general enough to allow it to learn very different non-linear value/policy functions.

It is this last aspect that makes the results so inspiring from an AI point of view, since it reminds us of how humans learn to play.

7.3.3 Results

Playing Strength

The Elo rating of AlphaZero in Chess, Shogi and Go is shown in Figure 7.15. Also shown is the Elo rating of the programs Stockfish, Elmo, AlphaGo Zero [607] and AlphaGo Lee [604].

We can see how AlphaZero is stronger than the other programs, all through learning by self-play. In Chess, a field that has benefited from work by a large community of researchers for as long as computers existed, the difference is the smallest. For Shogi the difference in playing strength is larger.

Interestingly, AlphaZero slightly outperforms AlphaGo Zero. There are small differences between AlphaZero and AlphaGo Zero. AlphaZero updates the neural network continually whereas in AlphaGo Zero the new net is only accepted if it wins 55% of the times against the old network. AlphaZero does not exploit Go symmetries. The paper does not provide other reasons for the differences.

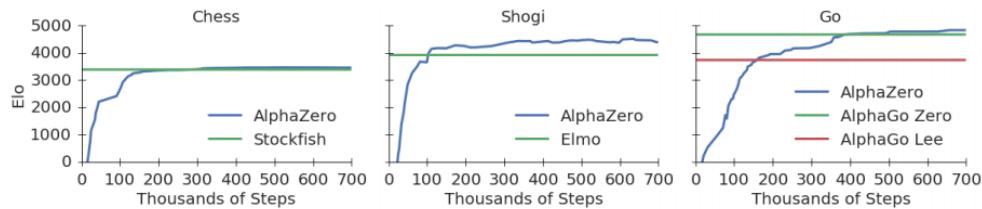


Figure 7.15: Elo Rating of AlphaZero in Chess, Shogi, and Go [606]

7.4 Self-Play Techniques

Now that we have had an in-depth look into self-play and the achievements of AlphaGo, we will take a step back and discuss the implications. We will start with a short summary of the impact on science and society. Then, we will look in more depth at the different technologies and how they worked together to achieve the breakthrough. Finally, we will look at the relationship with other forms of reinforcement learning, and, in order to provide a look into the future, we will have a short look at the history of self-play.

7.4.1 Impact

Let us first take a brief look at how science and society reacted to the AlphaGo breakthrough.

Popular Media

The Lee Sedol match of AlphaGo has had a large impact in the popular media. The idea of computers being “smarter” than highly intelligent human champions continues to attract attention, as much as it did in the days of Deep Blue–Kasparov. Artificial intelligence shot to prominence and many AI researchers were interviewed on questions ranging from when we would see self driving cars, to the future of the human race.

Newspaper articles appeared on AlphaGo, a movie⁹ was made, and many blogs were written by enthusiasts on how to build your own AlphaGo and AlphaZero. This, by the way, turns out to be quite doable on the algorithmic side (see Appendix A.3 for a list of open source self-play systems inspired by AlphaGo) but acquiring the necessary computational power to train the function approximators is a very significant challenge.

Impact on Go

It is interesting to see how human Go players respond to the matches, and to see the effect on the style of human play. Of course, there was sadness and melancholy when the human champions lost against the machines at this beautiful game, a hold-out of the supremacy of human intelligence. However, most reactions were positive, and not just for the magnificent achievement in artificial intelligence. Go players were enthusiastic about the innovations and new style of play that AlphaGo exhibited. Players were eager to learn more about how they could improve their beloved game even more. Books have been written about the new Go theory that was created (see, for example, Zhou [751]). Furthermore, the publicity around the matches created much new interest in Go.

Impact on Science

The AlphaGo matches have also had quite an impact on society and science. The interest from society in artificial intelligence has increased significantly, with companies and governments investing in research labs. Also within science there has been much interest in multidisciplinary collaborations, to see how machine learning can transfer to fields ranging from astronomy, physics, chemistry, pharmacology, medicine, biology, psychology, sociology, economics, archaeology, linguistics, literature, history, public administration, law, philosophy, and to the arts.

⁹<https://www.alphagomovie.com>

Impact on AI

On the field of artificial intelligence itself the results in reinforcement learning, including the AlphaGo matches and the self-play results of AlphaGo Zero, have had a large impact as well. Machine learning has attracted many new talents, and especially reinforcement learning has benefited. Attendance at scientific conferences has increased manifold, to the extent that some conferences started limiting attendance.

The use of the game of Go as a benchmark for intelligence has led to an increased interest in benchmarking and reproducibility [282, 348, 316]. Popular suites are ALE [55], Gym [98], ELF [668], StarCraft [681, 360, 704], and Mario [340]. See also the list of frameworks in Appendix A. There is much interest in further challenges of self-play (see the open questions in Section 7.4.5 on page 244).

Weak, Strong and Ultra Strong AI

At this point it is appropriate to put the AlphaGo achievements in a more historical perspective. In 1988 Donald Michie defined three types of AI: weak, strong, and ultra strong AI [437]. Michie’s criteria were meant to discern various qualities of machine learning, beyond predictive performance. Michie’s criteria stressed understanding, or comprehensibility, of learned knowledge. In weak AI the agent produces improved predictive performance as the amount of data increases. In strong AI the agent must also provide hypotheses, an explanation. In ultra-strong AI the agent should be able to teach the hypothesis to a human.

Clearly, most of modern machine learning has only achieved weak AI according to Michie’s criteria, AlphaGo included. For AI to be truly useful in society, it must achieve more. The current interest in explainable AI aims to achieve strong AI. See also Section 7.5.5.

7.4.2 Methods

Let us now look in more detail at the methods behind this “weak AI” breakthrough. The three AlphaGo programs use advanced reinforcement learning methods that together achieve a large performance leap. Even though the technology of the programs is elaborate, and the three papers are quite technical and detailed, there are conclusions to be drawn, to allow us to see the forest through the trees.

To understand how the large step in performance arose, we will highlight some main elements. We will discuss how the methods of this book fit together, adaptivity, correlation, recursion, and curriculum learning.

All Together

First, in AlphaGo techniques from all previous chapters come together. From Chapter 3 we see the concepts of policy and value functions. From Chapter 4 we see the search-eval architecture and we see shape enhancements and pattern

databases. From Chapter 5 we see MCTS. From Chapter 6 we see deep supervised database learning, value function approximation, advanced generalization and de-correlation techniques.

Taking all these techniques together, AlphaGo is a program that consists of a wide variety of techniques. Then again, if we regard the history of the field of reinforcement learning in games, is AlphaGo but a logical next step after TD-Gammon, that was waiting to happen?¹⁰

For the second program, AlphaGo Zero, another element is added: self-play. Self-play improved strength of play even more, and, remarkably, reduced training time significantly.

Adaptive Search

Second, we will look at the search-eval functions. Here the element to note is adaptive search. Searching the large state space is expensive, it is important to focus the search effort on promising parts of the state space (best-first search [267]). In Chess this was achieved by enhancing the fundamentally left-to-right fixed-depth fixed-width minimax algorithm with a heuristic value function for successor ordering and leaf-evaluation. AlphaGo uses the fundamentally adaptive MCTS algorithm, with trained function approximators for UCT policy selection and value functions for leaf-evaluation. Through the UCT mechanism MCTS can focus its search effort greedily to the part with the highest winrate, while at the same time balancing exploration of parts of the tree that are under-explored.

MCTS is used in the three Alpha-* programs in different ways, from standard to innovative. In the first program it is used in the conventional way, of an adaptive planner that still does playouts. The training is done without MCTS, off-line, based on grand-master games, and databases of self-play games. The databases are carefully de-correlated before they are used for training.

In the second and third program true self-play is used, without storing games in intermediate databases that are tuned, searched, and de-correlated separately. MCTS is used inside the training loop, as an integral part of the self-generation of training examples, to enhance the quality of the examples for every self-play iteration.

Also, in the *-Zero programs MCTS relies fully on the value function approximator, no playout is performed anymore. Only the selective statistics-based exploration/exploitation trade-off of MCTS is left. The MC part in the name of MCTS, which stands for the Monte Carlo playouts, really has become a misnomer for this adaptive tree searcher.

¹⁰Well, maybe with the benefit of hindsight. However, the fact that most experts had not expected this breakthrough to happen within ten years is testament to the deep understanding of all elements of the field by the DeepMind team, and to their deep insights that only become apparent with the benefit of hindsight. Also, the importance of the ability to execute on a clear vision is impressive, as the many prizes that the team has since collected underline.

De-Correlation

Third, de-correlation of training examples is of great importance in reinforcement learning. Whereas in supervised learning the individual training examples in databases are typically uncorrelated—the images in a sequence are often unrelated—in reinforcement learning, when positions come out of game sequences, de-correlation must be done explicitly. If not, training on correlated examples causes tunnel vision, training loops and low training efficiency. For example, a Chess player who is an expert in pawn play but does not know how to use the knight well is not an all-round player. The Chess player may win against a similarly myopic player trained in the same specialization, but will lose against a fresh test opponent trained in other areas. The AlphaGo Zero publication provides hints on how to achieve this.

In the three programs we see a move from explicit to implicit de-correlation. At first, in AlphaGo, databases of reinforcement learning positions were used that were carefully de-correlated with extra processing steps, and even small extra searches. In self-play a replay buffer is used, out of which example positions are sampled randomly to de-correlate, and to achieves stable training. In the second and third program, in self-play, the task of covering a wide area of the state space also rests with MCTS. The exploration part of MCTS should make sure that enough new and unexplored parts of the state space are covered. Extra noise terms are added to increase exploration. The C_p parameter in the UCT formula, that controls the level of exploration should be sufficiently high (see also Sections 5.3.1 and Appendix E.1).

Triple Recursion

Fourth, a striking element of the second program is the simplicity of the network architecture.

The new self-play architecture goes from 3 networks and 4 training methods to 1 network (with two heads) and 1 training method: self-play reinforcement learning.

The single netwwork with two heads goes back to Chapter 3, where policy functions and value functions were introduced. Policy and value are intimately related (in a deterministic setting good actions have high values and vice versa). This close connection clearly holds for Go, as the same network is used for both outcomes in AlphaGo Zero and AlphaZero.

Two crucial elements of self-play are general coverage and training stability. AlphaGo Zero learns from scratch, it does not use grand-master games to initialize the training, but goes right to the reinforcement learning phase. In AlphaGo the program structure and hyper-parameters had been tuned such that the stability of the self-play was good enough to reach world champion level play. After the program was redesigned after TD-Gammon-style self-play in AlphaGo Zero, the stability turned out to be good enough to achieve a long learning trajectory, from zero knowledge to grand-master level, all by itself.

AlphaGo Zero uses one neural net, and one training method. The simple

and elegant architecture allows a triple recursive, self-play training method. The training is triply recursive in the following sense. (1) The program self-improves the players in a curriculum of tournaments. (2) In each tournament the players perform self-play games, generating examples that are used to train the evaluation function. (3) In each game MCTS uses the same opponent model in its look-ahead, for itself and for the opponent, just as minimax. Listing 7.1 and Figure 7.9 provide the code and the conceptual picture of this triply recursive training structure in line 1, line 2, and line 5 (the third level of recursion is implicit inside the MCTS look ahead).

From Easy to Hard

This brings us to the fifth element: curriculum learning.

In AlphaGo the networks are trained from random initialization to advanced, in two large steps, first by grand-master games, then by self-play games. As in most training (Atari, ImageNet) inside each step the examples are not sorted from easy to hard. The program has to achieve the optimization step from no-knowledge to human-level play in one big, unsorted, leap. Overcoming such a large training step (from beginner to advanced) costs much training time. In AlphaGo Zero, which uses self-play, the network is trained in many small steps. The program starts with easy examples to train on, which goes quickly. Subsequently, harder problems are generated and trained for, based on the network that has already been pre-trained with the easier examples.

Self-play naturally generates a curriculum from easy to hard examples. The net result is that training times go down and the ultimate playing strength benefits. Curriculum learning has been studied before (see, e.g., [63]) but has not been popular since in most training situations it is difficult to find well-sorted training examples. In self-play this occurs naturally. Section 7.5.1 will go deeper into curriculum learning.

Generalized Game Play

Finally, we note that the simpler architecture of AlphaGo Zero generalizes well, to Go, Shogi, Chess.

It is hard to imagine how the complex architecture of AlphaGo (3 nets, 4 training methods, many hyper-parameters) could easily be adapted to different games. With the simpler architecture this is more straightforward, requiring only different input and output layers.

7.4.3 Reinforcement Learning versus Self-play

The previous section described the methods of AlphaGo Zero. We will now compare the AlphaGo methods to other methods. We will start with (1) traditional reinforcement learning, then we will (2) discuss cognitive science, and finally (3) we will discuss the history of self-play.

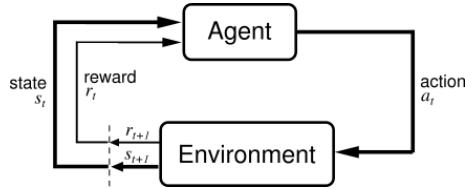


Figure 7.16: Reinforcement Learning

We will start with reinforcement learning. Taking a step back, it is instructive to compare the self-play of AlphaGo Zero and the DQN of Atari to the original reinforcement learning frame of agent and environment (see Figure 7.16).

Environment In Atari the ALE simulator is the environment. For each DQN action Gym provides a new state and reward.

In AlphaGo Zero the MCTS player is the simulator that provides the new states and rewards. Samples from different games are used in the value function to prevent correlations and over-fitting.

Agent In Atari the DQN net is the agent. The agent is trained to have good actions and values by playing against the environment, provided by the Gym environment.

In AlphaGo Zero the MCTS-network player is the agent. The net is trained against MCTS generated plays of earlier versions of itself (the MCTS environment).

The principle of self-learning in the self-play scheme is not new. Training by self-play, by self-generated examples, uses the same principles as TD-Gammon did in 1995, and has been tried in different ways by others. What is new is the success of this particular implementation, with MCTS, deep networks, and GPUs, TPUs, on the full game of Go, far surpassing human ability.

Cognitive Science: 1+2=3

We will now turn to cognitive science.

The correspondence between reinforcement learning in artificial intelligence and in cognitive science has been noted in a number of publications, for example [14]. When we look at self-play and the search-eval architecture we see three elements: the search procedure, the evaluation procedure, and the self-learning loop. Evaluation corresponds to system 1: thinking fast, a reflex-like network. Planning corresponds to system 2: thinking slow, a reasoning-like rational brain. Self-learning takes evaluation (system 1) and planning (system 2) and adds a third system, a learning loop, to create a self-learning system. The third system, the self-training loop, allows the reasoning system to feed training examples to the network to improve the reflexes. Note that Kahneman [336] does not mention a “system 3.”

7.4.4 History of Learning and Self-Play

To close this section on methods, we will put the AlphaGo self-play work in the context of the long history of different learning methods in the game playing literature. We will discuss examples of database and supervised learning that were used before self-learning and self-play.

Self-play is exciting. It is a holy grail of artificial intelligence: the emergence of intelligent behavior out of basic interactions [399]. Self-play has intrigued researchers since the start of game playing research.

Self-play is a recursive method, it feeds off itself. In minimax we determine our best move by calculating our opponent's best reply, which is based on our best reply, etc. In TD-Gammon and AlphaGo Zero style self-play we use a search-eval player to train our own evaluation function, by playing a small tournament against ourselves, using the game record as training input, yielding an improved evaluation function and playing for improved gain. Then we continue our path of self-improvement by doing another iteration of self-play, with the improved value function. The planning that is performed by the search function ensures that the game record will be of a higher quality than the previous level of play on which the current evaluation function is trained, so that the network is always trained against higher quality examples.

It can be difficult to get self-learning to work, because of instability. When a supervised learning system does not learn, then we can try to find out why it is not learning certain examples by checking them one by one. A self-learning reinforcement system produces its own training examples, and it can be hard to find out why a system is not learning, when there is no database of known knowledge that the system should have learned.

Therefore, we will see many examples of database learning, before we see self-play appear in game playing programs.

Earlier Works

Turing's original article on a paper Chess program introduced the search-eval architecture, with the basics of minimax state space search, and feature-based eval. Already in 1959 Samuel [560] introduced the idea of learning by self-play in his Checkers program. The coefficients of the heuristic function were updated towards the value of the search after black and white had each played one move. Today, more and more programs are based on end-to-end learning.

We will now have a look at previous game playing programs that used supervised and reinforcement learning. Table 7.4 lists notable works on games. The first column contains the name of the program, if given in the paper. The second column lists which game(s) the program plays. The third column is the planning algorithm, if applicable. The fourth column is the training algorithm, if applicable. CNN stands for convolutional neural net, FCN for fully connected net. The fifth column describes the data with which the learning took place. For supervised learning this is typically a database of grand-master games, and for reinforcement learning a self-play loop is used, or databases with self-played games. The sixth

Program	Game	Plan	Learn	Data	Net	Eval	Reference
Samuel	Checkers	AlphaB.	coeff.	Self-Play	-	coeff. feat.	Samuel 1959 [560]
Chinook	Checkers	AlphaB.	-	endgames	-	heur. feat.	Schaeffer '92 [571, 572]
Logistello	Othello	AlphaB.	regress.	patterns	-	coeff. pat.	Buro 1995 [112, 113]
Deep Blue	Chess	AlphaB.	coeff.	manual	-	coeff. feat.	Hsu 1990 [303, 118]
NeuroCh.	Chess	AlphaB.	TD(0)	db/Slf-Ply	1 layer	coeff. feat.	Thrun 1995 [664]
Neurogam.	B.gam.	-	FCN	Games db	1 layer	coeff. feat.	Tesauro 1989 [651]
TD-gam.	B.gam.	AlphaB.	TD(λ)	Self-Play	1 layer	end to end	Tesauro 1995 [653]
-	Go	-	TD(0)	Self-Play	2 layer	end to end	Schraudolph '94 [588]
MoGo	Go	MCTS	TD(0)	Self-Play	-	coeff/shape	Gelly 2008 [234, 232]
NeuroGo	Go	-	FCN	Games db	1 layer	coeff. feat.	Enzenberger '96 [192]
Blondie24	Checkers	AlphaB.	FCN	Evo Play	3 layer	end to end	Fogel 2001 [217]
Giraffe	Chess	AlphaB.	CNN	Self-Play	3 layer	end to end	Lai 2015 [380]
DeepChess	Chess	AlphaB.	FCN	Games db	4 layer	end to end	David 2016 [157]
-	Atari	-	CNN	DQN	3 layer	end to end	Mnih 2013 [453]
Exlt	Hex	MCTS	CNN	Self-Play	15 layer	end to end	Anthony 2017 [14]
-	Go	MCTS	CNN	Games db	5 layer	end to end	Clark 2015 [143]
DarkForest	Go	MCTS	CNN	Games db	12 layer	end to end	Tian 2015 [669]
AlphaGo	Go	MCTS	CNN	Games db Self-Play	$\pi:12C/1F$ $v:13C/1F$	end to end	Silver 2016 [604]
AlphaGo Z.	Go	MCTS	ResNet	Self-Play	20 block heads: π, v	end to end	Silver 2017 [607]
AlphaZero	Go Chess Shogi	MCTS	ResNet	Self-Play	20 block heads: π, v	end to end	Silver 2018 [606]

Table 7.4: Evolution of Planning, Learning and Self-play in Game Playing

column is the network architecture. Only the hidden layers are counted. For the older papers this is often a small number of hidden layers. After the advent of the use of GPUs in 2012 larger nets became feasible. The seventh column lists the evaluation method. This can be a linear combination of heuristic features, a system where the network learns the coefficients of heuristic features, or a full end-to-end learning system from raw board input. Last is a reference to a major work describing the approach of the program. Only the name of the first author is mentioned. Some programs were developed over many years by different authors. Please refer to the original papers for the often fascinating histories.

We will now discuss how the methods of these earlier programs developed. These methods are mostly aimed at position evaluation, although some are also used for selective search.

Learning Shapes and Patterns

Game playing programs have been using many types of machine learning to improve the quality of play. Shapes and patterns are of great importance in many games. For this reason, many early works focused on learning shapes and patterns. This learning was typically off-line, using months and months of training time. The Checkers program Chinook employed an endgame database that con-

tained all board positions with 8 or fewer pieces. As soon as the database was reached by the search, perfect knowledge was available. The endgame database greatly contributed to the strength of Chinook [572]. In Chess, Deep Blue had a database that was accessed as soon as 5 or fewer pieces were on the board [118].

In Go, the emphasis of databases is on small local patterns. Stoutamire used hashed sets of patterns to find good and bad shapes [628]. Van der Werf used neural networks and automatic feature extraction to predict good local moves from game records [687, 686].

Supervised Learning

Many researchers have tried supervised learning with databases of grand-master games. Tesauro's 1989 Neurogammon achieved good results in Backgammon with supervised learning of the coefficients of an evaluation function of hand-crafted features. He used a single hidden layer [652].

In Go, Enderton used neural networks in the search in his Go program Golem, for move ordering and for forward pruning of selective search [191]. Enzenberger's NeuroGo used a neural network for position evaluation [192].

Much later, Maddison et al. published work on a 12-layer convolutional neural net for move prediction in Go [425], laying a foundation for the supervised learning of AlphaGo. Facebook's AI Research lab created Darkforest for end-to-end deep learning of Go positions and MCTS [669].

Reinforcement Learning

Supervised learning, as in Neurogammon, uses a database to learn from good moves. Self-play is different. In minimax and MCTS, the look ahead searcher is alternately playing the role of the player and of its opponent, to determine the best move based on the outcome of actual games. This requires games to be actually played out, which is computationally expensive.

After the success with supervised learning in Neurogammon, Tesauro tried temporal difference learning from the raw board in a small neural network in 1992. This was further augmented with expert designed features, resulting in play at world class level. TD-Gammon's success was the first widely published success of self-play [654, 655].

In Go Schraudolph et al. [588] used temporal difference to learn position evaluation from board networks. They note that to be successful, networks should reflect spatial organization of input signals. Huang et al. [306] used self-play of 16 different players to learn opening strategies in Go.

Silver, in his PhD thesis [603], applied reinforcement learning to search in Go, and Silver et al. [608] and Gelly & Silver [232, 231, 230] studied reinforcement learning of local shape patterns. Silver et al. [609] further combined temporal difference learning in MCTS.

In Chess, Thrun's NeuroChess [664] tried temporal difference feature learning. Inspired by the progress in deep learning, in 2015 Matthew Lai tried end-to-end

learning in Chess: he went beyond evaluation function weight tuning with Giraffe [380], using DQN, and self-play, including feature extraction and pattern recognition for end-to-end reinforcement learning. Giraffe reached master level play. Lai later joined the AlphaZero team.

In some of the early works the small size of the networks limited success. In others the temporal difference learning or self-play is based on hand crafted or heuristic features of the state space, possibly introducing bias or errors. End-to-end learning, from the raw board or pixels requires more computational power. When first done successfully end-to-end (AlexNet [372], Atari [453, 454]) the papers generated much research interest.

Concurrent to the work by the DeepMind team, Anthony et al. [14] worked on a similar idea for tabula rasa learning based on self-play, combining MCTS and DQN. The name of their work is ExIt, for Expert Iteration. They did not use Go, but Hex, a simpler game, to show their methods. They do make the link with Kahneman's work explicit, even in the title of their paper.

The relation between planning and training is also studied in model-based and model-free reinforcement learning. See Section 3.3.4. The interplay between planning and training, and especially of model-based and model-free approaches, is an active area of research [14, 723, 498, 710, 403, 722].

7.4.5 Future of Self-Play

We have now looked more broadly at self-play, and at earlier programs that have tried self-play. AlphaGo Zero showed the large potential of this method. We will now look at directions for future research on self-play.

Self-play has been an active area of research for many years. The AlphaZero results have reinforced the power of the paradigm, showing that TD-Gammon's success was not an exception. The successes have created much research interest into fundamental questions and applications of self-play.

Many questions remain. Among them are:

- Why is performance of self-play levelling off? How can learning be improved further? Can learning be sped up, can learning efficiency be improved?
- Self-play works for two-agent board games. For which other domains can self-play work? Does self-play also for other Markov Decision problems, or for non-Markovian problems?
- Did AlphaGo/DQN self-play kill the deadly triangle? Is training stability solved problem?
- Why do ResNets perform so well in self-play? Are there other architectures that work better?
- Can the network architecture be optimized automatically?

- So far, general methods have been enhanced with domain dependent methods (Chapters 4 and 6). Which enhancements work well in self-play?
- Does self-play work well in imperfect information games such as Bridge, Poker, and StarCraft?
- Are self-play results easy to transfer? How can concepts that are learned in one game be transferred to other games? (Note that in AlphaZero each game needed to start training from scratch.)
- Hierarchical reinforcement learning studies how multiple layers of abstraction can work in large state spaces, and solve more complicated problems. Will it work for self-play?
- There is much research in flat reinforcement learning (training without planning). In AlphaZero self-play crucially depends on planning to generate high quality examples. Will exact methods make a come back or will DQN find ways to train for the long credit assignment distances that planning is so good at?
- Evolutionary Strategies have been successful in some studies. Will they play a role of importance in (general) self-play?
- Self-play is computationally very demanding, which is a major impediment to progress. Can parallelization speed up self-play, allowing faster experimentation and larger games?

Future work will show if and how these questions can be answered.

Let us now look more deeply into some of these questions, specifically, curriculum learning, transfer learning, and other future areas of research.

7.5 *Enhancing Deep Reinforcement Learning

As we have seen, self-play has a long history in artificial intelligence. The recursive nature of minimax is a form of self-play, Samuel’s Checkers program used self-play to adjust its evaluation coefficients, and TD-Gammon learned patterns generated by self-play. Despite this long history, the AlphaGo results showed a novel use of self-play, in end-to-end training of a deep function approximator.

To answer the open questions of the previous section, we will look at the kind of learning that occurs in self-play. A better understanding of curriculum learning, transfer learning and meta learning will pave the way towards answering the open questions.

7.5.1 Curriculum Learning

When training examples contain hard to learn concepts, there is a large gap between the training target and the output state of the approximator, at first. As we saw in the previous chapter with AlphaGo, learning can be slow in this situation.

AlphaGo Zero showed that it is possible to create a different convergence process. We start small, with easy to learn targets. In a curriculum it is easier to explore and to achieve a reward, and thereby a learning signal. The training targets are generated in lock step with the trainee, both trainer and trainee grow in playing quality together. Such a convergence process is related to curriculum learning.

Training deep neural networks is hard, as we saw in the previous chapter, many years of research were needed before efficient training methods were found.

From a conceptual point of view, deep learning attempts to learn feature hierarchies. Features at higher levels are formed by composition of lower level features. Deep Learning automatically learns multiple levels of abstraction, creating a complex function mapping from input to output.

Many different approaches have been tried. As we saw in Section 6.1.3, Hinton et al. [287] introduced an efficient pre-training approach for training one layer greedily at a time. Before Hinton there were no efficient algorithms for training fully-connected deep networks. By training layers one by one, the network starts by learning the simpler concepts in the first layer, then more abstract concepts in the second layer, etc. Further research introduced other efficient algorithms for training deep networks, showing advantages of deep networks over shallow ones [245, 394].

Pre-training can help the optimization process in a network that is difficult to train. Some networks have good solutions in local minima that are hard to find by random initialization [198].

One Layer at a Time

The idea of training a deep network one layer at a time, starting with the lower layers with simpler concepts, is related to curriculum learning. Curriculum learning is a concept from didactics and psychology. A curriculum is a sequence of learning tasks, assumed to be ordered from easy to hard, since this is how learning normally progresses.

In artificial intelligence, curriculum learning arose out of the studies for better deep learning algorithms, and the success of layer-by-layer unsupervised pre-training. Pre-training with a curriculum strategy has been found to act as unsupervised pre-training. It finds better minima and works as a regularizer. It helps to reach faster convergence to a minimum of the error function.

Related to curriculum learning is *self-paced learning* [376]. We will now look more deeply into curriculum learning, following Bengio et al. [63]. Bengio et al. propose a curriculum as a weighted sequence of training steps [63]. Each training step in the sequence can be considered to have a different set of weights on the examples. Initially, the weights favor easier examples illustrating the simplest concepts. The next training step involves a small change in the weighting that increases the probability of sampling more difficult examples. At the end of the sequence, the re-weighting of the examples is uniform and training is performed on the target training set.

Training Speed

A curriculum strategy is faster because the learner spends less time with noisy or harder to predict examples. Curriculum learning guides training towards better regions in parameter space, into basins of attraction (local minima) of the optimization procedure associated with better generalization.

Weinshall et al. prove that curriculum learning is expected to significantly speed up learning, especially at the beginning of training [726]. Furthermore, final generalization is sometimes improved with curriculum learning, especially when the conditions for learning are hard: when the task is difficult, when the network is small, or when strong regularization is enforced. These theoretical result are supported by empirical evidence [63, 428, 197, 60].

Human Learning is Curriculum Learning

Curriculum learning has been studied both in human learning and in deep learning. Most structured, supervised, human learning is guided by a curriculum [726]. When a human teacher presents examples, the order in which examples are presented is rarely random. Tasks are typically divided by the teacher into smaller sub-tasks and ordered from easy to hard. This sub-tasking is called shaping [373] and studied in reinforcement learning, e.g., by Graves et al. [250].

Krueger and Dayan [373] note that in human and animal learning, curriculum learning is able to learn higher concepts that are beyond what can be found by trial and error. They use the term *shaping* for the creation of a curriculum of easy and hard targets.

In human learning children are taught easy concepts first. There is a reason that children start in first grade, and not in sixth. Using a neural network for language acquisition, Elman [188] argues for starting small, with a small network, and gradually increasing the difficulty of training examples and the network size. Multi-stage curriculum strategies improved generalization and show faster convergence.

Active Learning

The concept of learning in curriculums has been studied more widely, and not just in self-paced learning [376, 322]. Active learning is a kind of machine learning that is in between supervised and reinforcement learning. Sometimes labels are in principle available, but at a cost. Active learning performs a kind of iterative supervised learning, in which during the learning process the agent can choose to query which labels to reveal. Active learning is related to reinforcement learning and to curriculum learning, and is of interest for studies into recommender systems [598, 548, 156].

GANs

Recent work in generative adversarial networks (GANs) has suggested that the performance of GANs may be due to curriculum learning. The generator and discriminator work in tandem, and in a well-tuned system generate a sequence of easy to hard training examples [602, 174].

Teaching Curricula

Weinshall et al. [726] note that curriculum learning is not often used in the real world, since it is hard to find situations where a teacher can rank the examples from easy to hard. In most supervised and reinforcement learning settings, there is no weighing from easy to hard available for the examples. This may have precluded the widespread adoption of curriculum learning. However, in a self-play situation the order of examples is naturally ordered from easy to hard, and the training process follows a natural curriculum. Furthermore, teaching curricula made by humans may be wrong or unsuited for machine learning. Self-play does not suffer from this problem, and automatically generates the right level of difficulty for the learner.

Convolutional neural networks are easier to train than fully connected networks. Although curriculum learning has remained for the most part in the fringes of machine learning research, curriculum learning has been identified as a key challenge for machine learning throughout [449, 450, 718]. In the context of self-play there is certainly room for further work on this topic.

Self-Play

As has been noted in Section 7.4.2, in tabula rasa self-play, the examples generated by the system automatically follow an easy-to-hard curriculum. At the start of the learning process, when the tabulas are still rasa, none of the networks have learned any knowledge yet, and the examples to train on are necessarily easy. As knowledge is acquired in the networks, the examples are harder, and the learning tasks are harder. Figure 7.14 on page 230 shows how AlphaGo Zero learned a sequence of corner plays of increasing difficulty.

As such, self-play may be one of the first methods to naturally generate easy to train curricula.

7.5.2 Training as Generalization

We will now look at other generalization methods.

In some sense, all training is generalization. The goal is to build an approximator based on the examples. Transfer of training results to initialize a net when this network is from another domain, is transfer learning.

Transfer Learning

In addition to the relation with human learning, curriculum learning is also related to transfer learning and lifelong learning [665]. Curriculum learning strategies can be regarded as a special form of transfer learning. The earlier tasks are used to initialize the network and guide the learner so that it will perform better on the final task. Note that the traditional motivation for transfer learning is to improve generalization by sharing across tasks. In curriculum learning the main goal is to guide the optimization process, to converge faster, and to reach better local minima.

Curriculum learning is also important for life long learning, or never ending learning [448], and has been reported to be necessary for other domains as well [744, 250].

In ordinary learning we are interested in generalization over examples inside the training domain. In transfer learning we look at generalization to other domains outside the training domain. Domain generalization comes in two flavors: system generalization and transfer learning. In system generalization the task is to build a system that can learn to solve multiple problems. This is the kind of generalization that Generalized Game Playing (GGP) seeks to achieve (see Section 8.2.6). AlphaZero succeeded in this aspect for three board games. Algorithms, network architecture and hyper-parameters generalized to three games, although it needed hand crafted input and output layers, and, significantly, the network had to be retrained from scratch for each of the three games. The trained nets do not generalize across games.

In multi-task learning related learning tasks are learned at the same time. Multi-task learning improves regularization by requiring the algorithm to perform well on a related learning task instead of penalizing all over-fitting uniformly [200, 18]. Multi-task learning has also been applied to Atari games [346, 345].

The goal of transfer learning is to transfer knowledge that is learned in one domain to solve problems in another domain, with little retraining [523, 666]. Transfer learning is related to multi-task learning, where one network is optimized for multiple tasks that are often related. However, the two-headed AlphaGo Zero network optimizes for value and for policy at the same time in the same network [491, 119].

Transfer learning is related to many other forms of learning. For further study, see these surveys on transfer learning [119, 491, 648].

Meta-learning

We will now turn to meta-learning, a topic that is related to transfer learning and to curriculum learning.

Efficient learning requires prior knowledge, or priors. In principle, such priors can be engineered into the structure of the network (such as in convolutional neural networks) or, more interestingly, they can be acquired through learning. This approach is called meta-learning in the machine learning literature [580, 579, 666, 293, 11].



Figure 7.17: Chelsea Finn

Duan et al. [184] introduce RL^2 , an approach in which the reinforcement learning algorithm itself is a target of a training process. They achieve encouraging results.

Wang et al. [717] explore the link between meta-learning and reinforcement learning, by having one reinforcement learning algorithm implement a second reinforcement learning algorithm.

Meta-learning is related to few-shot learning. A drawback of high-capacity deep learning networks is the large amount of data that is needed to prevent overfitting. Humans are able to learn with very few examples. Few-shot learning studies methods to achieve this. Finn et al. [210] introduced Model-agnostic meta-learning, or MAML (see Figure 7.17). MAML is a promising method to train parameters so that they can achieve few-shot learning (or even one-shot learning). This work has spawned more interest in meta-learning [15, 49, 211, 749, 703, 532]. Meta World is a benchmark suite introduced for research on multi-task and meta reinforcement learning [743].

Meta-learning and transfer learning are related and active research areas [224, 593, 700, 261, 260, 531, 620, 673].

Limits of Generalization

Now that we have looked at curriculum learning and transfer learning, we will turn to ways to further enhance self-play. This section provides an overview of some of the areas in which active self-play research is being performed.

In all learning graphs the learning performance levels off after some point. This also occurs in self-play. At some point, extra training does not improve performance any more (see, for example, Figure 7.18). A better understanding of why generalization does not improve further may provide insights for further improvement, just like a better understanding of learning principles such as curriculum learning allowed AlphaGo Zero to achieve a higher level of play than AlphaGo.

Perhaps the curve is levelling off because there is nothing left to learn in 19×19

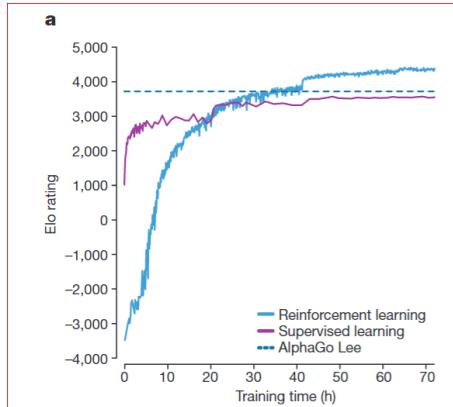


Figure 7.18: Comparison of Supervised Learning and Self-play Reinforcement Learning [607]

Go? If this were true, then AlphaZero would have found the global optimum, the ultimate Go wisdom (assuming such a point existed).

This is an argument that is sometimes heard, but it is mistaken. It may apply in an exact planning setting, where the entire state space could have been traversed. Clearly, the state space of Go is too large for that.

A reason for the levelling off may be that the capacity of the network is too small, and it is under-fitting. This can be tested by increasing the network size. The AlphaGo Zero paper describes a comparison between 20 block and 40 block residual nets, finding the improvement of the larger network to be small [607]. If the smaller network had been under-fitting, then the larger network should have performed better. Another solution is to change the network architecture in a more fundamental way, by adding or removing layers of different types.

Perhaps there is a more fundamental limit in the self-play scheme? Tesauro described in his TD-gammon papers that the single layer network did not learn more complicated patterns, and adding a small amount of planning (a two level search) and certain hand crafted heuristic knowledge improved performance of the program significantly [655].

Perhaps the self-play training scheme can be improved. Already Epstein [195, 196] noted the importance of a good trainer in self-play, and has suggestions on how to avoid self-play training pathologies. Another possibility is that the planner needs to improve as the level of play increases, for example by increasing the number of MCTS simulations, or changing the amount of exploration that MCTS does.

Perhaps the network is over-fitting, and more data is needed. The data in self-play is generated by self-play, and the curve shows a decreasing utility of more training. This argument therefore becomes one of training stability and exploration, to increase generalization by removing correlations and deadly triad problems.

Hyper-parameter	Value
self play games per iteration	25000
MCTS simulations	1600
exploration temperature	1 for first 30 moves, 0 afterwards
Dirichlet noise	$\eta = 0.03, \epsilon = 0.25$
resignation threshold	5%
regularization parameter	$c = 10^{0.4}$
mini batch size	2048, 32 per worker
self play positions	500,000
learning rate	annealed with momentum 0.9
CE loss/MSE weighting	equal in the loss function
optimization checkpoint	1000
self play evaluator	400
win margin for acceptance	55%
network architecture	19 residual blocks

Table 7.5: AlphaGo Hyper-parameters [607]

Perhaps the local minima in the state space have become hard to find. The question how to improve learning further may well be the most important question to ask in machine learning. There are still many parameters and elements of the program left to explore and understand. This remains very much an open question that is actively researched by the field.

A structured way to optimize different parameters that govern the training process is hyper-parameter optimization, which we will now discuss in more detail.

7.5.3 Hyper-parameter Optimization

Deep learning and self-play systems have many hyper-parameters. The AlphaGo Zero paper lists the following, see Table 7.5 [607].

The effects of some of these hyper-parameters may interfere with each other. AI has taught us that some decision making tasks are best left to computers. Finding the optimal setting for all parameters is a daunting task, best left to machine optimization.¹¹ Many optimization algorithms have a large set of hyper-parameters, and hyper-parameter optimization is an active field of research. Well known methods are exhaustive search [297], random search [66, 719], Bayesian optimization [67, 312, 663, 619], and evolutionary strategies [26, 111, 496, 692]. Some software packages for hyper-parameter optimization are SMAC¹² [312],

¹¹just as finding the best coefficients for an evaluation function of hand crafted heuristics is best left to machine learning, and, for that matter, crafting the evaluation features themselves is also best left to machine learning, and crafting a database of training positions is best left to self play. Some would even say that crafting machine learning algorithms is also best left to machine learning, but that is a topic for another book [266].

¹²<https://github.com/automl/SMAC3>

auto-sklearn¹³ [206], Auto-WEKA¹⁴ [368], scikit-learn¹⁵ [506], irace¹⁶ [416] and nevergrad¹⁷ [534].

A parameter sweep of AlphaZeroGeneral (Section A.3) is Wang et al. [714]. AlphaGo Zero optimized its hyper-parameters with Bayesian optimization, AlphaZero used the same set of hyper-parameters.

Network Architecture

The architecture of the network can also be regarded as a hyper-parameter to be optimized. Among the decisions to make are the number and types of the layers, and the number and types of the neurons, whether to have ResNet cells, and how they are connected. Getting the architecture right for your problem can be a time consuming problem, perhaps also best left to machine learning.

Neural Architecture Search is an active field of research, and can be based on the same methods as hyper-parameter optimization. Random search, Bayesian optimization [742], evolutionary algorithms [352, 691, 320, 439, 712], are popular methods, although reinforcement learning has also been used [754, 514, 29]. Two surveys are [741] and [190]. A standard work on evolutionary strategies is Bäck [25].

Network Type

In Chapter 6 we saw an evolution of different network types for image recognition, and the same occurred in reinforcement learning. For AlphaGo Zero Figure E.1 compares convolutional and residual nets. So far, there is little other work reported that compares different network types on the same problem. A complicating factor is that the extra learning loop makes running self-learning experiments computationally expensive (see also Figure 8.3). Running experiments to find out what type of network and which hyper-parameters work best for a problem is currently close to computationally infeasible for large problems.

Network Optimization

An important element of all neural network function approximators is the optimization algorithm. Stochastic gradient descent has been the optimization algorithm of choice, although alternatives exist. Recently works report on evolution strategies [139, 558, 730]. An advantage of evolution strategies is that they are well suited for parallelization, providing possible speedups for the training phases of self-play. Wang et al. [715] study the optimization target of a dual-headed self-play network in AlphaZeroGeneral.

¹³<https://automl.github.io/auto-sklearn/stable/>

¹⁴<http://www.cs.ubc.ca/labs/beta/Projects/autoweka/>

¹⁵<https://scikit-learn.org/stable/>

¹⁶<http://iridia.ulb.ac.be/irace/>

¹⁷<https://code.fb.com/ai-research/nevergrad/>

Related to this topic are distributional DQNs, where not just the expected value is optimized, but also the variance of the error [54, 455, 456] (Section 6.4.1).

Jaderberg et al. [319] report a population based approach to training reinforcement learning agents for a multiplayer real time strategy game.

Parallelism

A major problem of self-play is the large computational requirement, which translates to long training times. Self-play has three obvious areas where parallelism can be used for speed up. First of all, network training can be sped up. Here GPUs and TPUs are already used, TensorFlow and other packages typically include support for parallel training.

Another option is to parallelize MCTS. We have already seen a few parallel MCTS approaches, such as tree-parallelism and root-parallelism, in Section 5.3.1. Their use is not trivial, and careful implementation in current self-play can speed-up training.

Furthermore, the tournament games of the self-play setup are all independent, and can be parallelized. This should give an important speedup of self-play, filling the example buffer in parallel. Advances in this area will enable more experimentation and more insights into self-play processes.

7.5.4 Hierarchical Reinforcement Learning

One of the main challenges in reinforcement learning is scaling up classic methods to problems with large action or state spaces. A problem of model-free reinforcement learning is low sample efficiency, samples generation is slow or samples are not used efficiently (see also Section 3.3.6).

After having seen how different parameters of the training process can be optimized, we will look at hierarchical reinforcement learning, a more principled idea to scale reinforcement learning to larger problem classes, by decomposing large problems into smaller ones. Hierarchical reinforcement learning can be more sample efficient by re-using sub-tasks and abstract actions.

Most reinforcement learning problems are specified at a low level of detail. The steps that we can make are small (move a single piece, perform a single movement of a robot arm). The advantage is that this allows for precision and optimality. The disadvantage is the large size of the state space. The state space of highly detailed problems is large and searching becomes infeasible for larger problems.

Consider the following problem: how can we find good solutions in large state spaces? In the real world, when we plan a trip from A to B, we use abstraction to reduce the state space, to be able to reason at a higher level of abstraction. We do not reason at the level of footsteps we need to take, but we first decide on the mode of transportation. Hierarchical reinforcement learning tries to understand and automate this ability to work flexibly with multiple overlapping time scales: conventional reinforcement learning nets do state abstraction over a single state,

hierarchical reinforcement learning performs *temporal abstraction*, solving sub-problems in sequence. Temporal abstraction is described in a seminal paper by Sutton et al. [639].

Flet-Berliac [214] provides a recent overview of hierarchical reinforcement learning, noting that planning research into hierarchical methods showed that exponential improvements in computation cost can be achieved with methods such as hierarchical task networks [154], macro actions [208] and state abstraction methods [354]. Flet-Berliac summarizes the promise of hierarchical reinforcement learning (1) to achieve long-term credit assignment through faster learning and better generalization, (2) to allow structured exploration by exploring with sub-policies rather than primitive actions and (3) to perform transfer learning because different levels of hierarchy can encompass different knowledge.

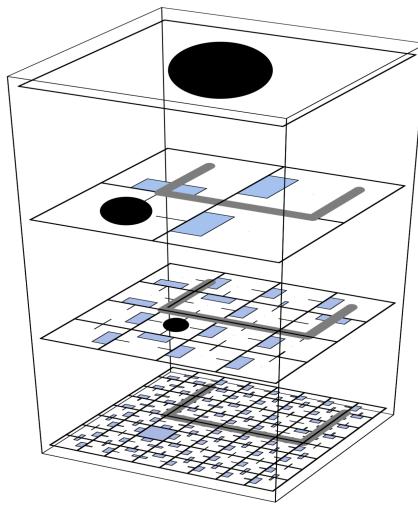


Figure 7.19: Hierarchy in Feudal Reinforcement Learning [161]

Dayan and Hinton introduced Feudal Reinforcement Learning [161], in which learning takes place according to a medieval feudal hierarchical system, see Figure 7.19. Feudal reinforcement learning introduced a specific Q-learning method, and paved the way for other methods [214].

Hierarchical reinforcement learning is an intuitively appealing approach. It is an active field of research and we will discuss some of the better known approaches. Researchers have addressed the need for large-scale planning by introducing various forms of abstraction, e.g., Fikes et al. [209] and Korf [366]. One of the simplest abstractions is a *macro-operator*, which is a sequence of actions that can be invoked as if it were a primitive action and has been explored in robotics and control engineering.

In addition to macros, in reinforcement learning hierarchical abstraction has been mainly studied in the framework of options, MAXQ, and hierarchies of abstract machines (HAM) [35].

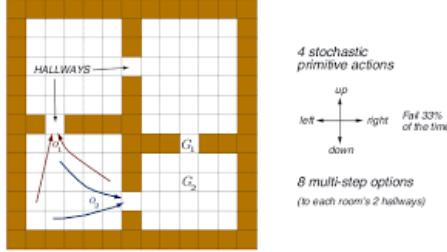


Figure 7.20: A Maze with Rooms [639]

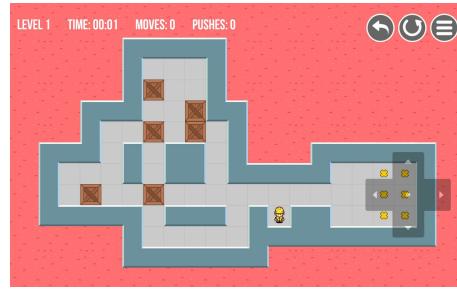


Figure 7.21: Sokoban

Macros

Macros trade-off solution speed for optimality [366]. Familiar application domains are room problems with simulated robots. Figure 7.20 shows a simple room problem, and Figure 7.21 shows an example of a Sokoban puzzle [331]. Both examples show a clear hierarchy, where the agent first has to plan to go to the right room, and then, at a more detailed level of abstraction, how to navigate the room.

A game that is similarly difficult for “flat” reinforcement learning is Montezuma’s Revenge, since it contains a room-like structure with long navigation and few rewards. Rafatti and Noelle [530] present a method for subgoal discovery which they apply to a part of Montezuma’s Revenge. Other relevant works in this domain are [89, 477, 435, 90].

Options

Sutton et al. [639] present *options* as a temporal abstraction for hierarchical reinforcement learning.

Options are policies for taking action over a period of time. Examples of options include picking up an object, going to lunch, and traveling to a distant city [639]. Options are very much like actions, and enable temporally abstract knowledge and actions to be included in the reinforcement learning framework

in a natural way. Options may be used interchangeably with primitive actions in planning and learning methods such as Q-learning.

Hierarchy of Abstract Machines

The idea behind the hierarchy of abstract machines (HAM) approach is that policies of a core MDP are defined as programs which execute based on their own states as well as the current states of the core MDP [497]. The HAM approach limits the actions of the agent to transitions between states of the machine. An example of a simple abstract automaton can be: *constantly move to the right or down*.

Although some achievements have been made [283, 284] automatically discovering hierarchical structure is still an open problem in reinforcement learning. Panov and Skrynnik [494] present a new approach to hierarchy formation, based on HAM.

MAXQ

Dietterich [173] developed another approach to hierarchical reinforcement learning: MAXQ Value Function Decomposition, abbreviated to MAXQ. In MAXQ, a hierarchy of MDPs is created whose solutions can be learned simultaneously. The MAXQ approach assumes a decomposition into a set of hierarchical subtasks, which can then be solved. A task graph summarizes the structure of the hierarchy. Dietterich introduced the now well-known Taxi problem (Figure 3.8 on page 76) to illustrate his approach.

Automatic Discovery of Abstractions

These former works on hierarchical reinforcement learning assume human supervision, for example to provide explicit subgoal states, or expert actions. Other approaches have attempted to automatically discover appropriate temporal abstractions. McGovern et al. [435] proposed a method for finding subgoals in *bottleneck* states that are visited frequently in successful episodes. Bacon et al. [27] proposed the Option-Critic method for the automatic discovery of options. Vezhnevets et al. [699] also discussed the feudal networks reinforcement learning architecture. Rafati and Noelle [530] learn subgoals in Montezuma’s Revenge. Schaul et al. [576]’s universal value function approach suggests the use of function approximation for hierarchical reinforcement learning. Pang et al. [493] describe experiments with macro actions in StarCraft.

Hierarchical reinforcement learning is an active field of research with a long history. See for further reading for example [332, 512, 224, 427, 35, 30, 613, 375, 404, 630, 335, 698]. Applications of hierarchical reinforcement learning to self-play remain to be explored, see also [632].

7.5.5 Further Research

The successes in self-play have been in two-agent zero-sum games. The successes have prompted researchers to look at other domains, to see if there similar progress can be made. We will list some research directions. We start with different kinds of problems, such as multi-agent problems, and then we continue with different representations.

Multi Agent

Two agent games are typically competitive (zero sum) games. Single agent and multi agent games are often (partly) cooperative. They have been the subject of study for some time, and many excellent works have appeared [414, 647, 144, 117, 492, 278, 420, 126]. Applying the recent progress in self-play with multi agent cooperative games will be quite interesting, as we will see in the next chapter.

Sparse Rewards

Many problems are sparse problems: the reward information in state spaces is sparse, and credit assignment can be delayed for a long time. Furthermore, the action space may be large as well. An example of sparse rewards is Montezuma’s Revenge, an Atari game in which many actions can be taken without encountering a reward state or a change in value. This game requires more advanced strategic reasoning, presenting a challenging problem for many reinforcement learning methods (see Section 6.3.3). Some recent references are [490, 527, 3].

Continuous Action Spaces

Many applications have large, continuous action spaces, possibly combined with imperfect information (see Poker and StarCraft in Section 8.2.4). We often encounter this in real life [407, 457, 181, 563]. Examples are highly prevalent in robotics, where an arm movement can be over a continuous angle. A good review of this field is [357].

Knowledge Representation

Another area of interest is the intersection of knowledge representation and reinforcement learning. This is partly driven by the desire for explainable AI, or decision support where the reasoning behind a recommendation is also provided. This field is related to learning Bayesian networks and belief networks [276, 471, 594, 163, 53, 656, 96, 20].

Visualizing Neural Nets

Deep network layers learn a hierarchy of abstract features. The layers learn concepts of increasing generality [61, 393], creating a hierarchy of concepts. How

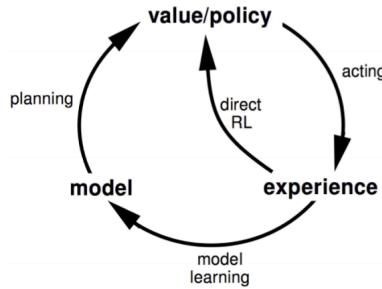


Figure 7.22: Model-free and Model-based methods

general are the concepts learned by self-play? What do they “look” like, can they be represented as a two dimensional picture for us to interpret? A deeper understanding of these concepts might enable us to learn more complex representations, and thus achieve better local minima in the error function [513]. A popular dimensionality reduction and visualization method is t-SNE [423].

Montavon et al. [459] provides a tutorial on methods for interpreting deep nets.

Model-free and Model-based Reinforcement Learning

Section 3.3.4 introduced model-free and model-based reinforcement learning, the inner and the outer loop of Figure 7.22 [638]. Model-free reinforcement learning is quite successful. Much work in reinforcement learning is on *flat* DQN, on learning without self-play or planning (see, e.g., [13]). However, model-free learning has low sample efficiency. Let us look at the question if all reinforcement learning can be folded into a single deep function approximator, or if a two-step model-based approach works best?

Since long range credit assignment and hierarchical structures appear to be difficult to fold into flat function approximators, some forms of abstraction and strategic reasoning are likely to be necessary. The use of exact planning methods will also depend on the dimensionality of the problem. For high dimensional problems approximation is a necessity, for low dimensional problems exact methods will yield better answers. Can these be combined into a single architecture?

RNN and LSTM networks are well suited for modeling sequential problems. There are studies on how look-ahead can be performed by neural networks, obviating the need for the often slow MCTS planning part [256, 255]. Schrittwieser et al. [589] report on MuZero, a system that combines model-based planning and model-free learning. MuZero learns the model that is used during planning online, for board games as well as Atari games. This is an exciting research direction where more can be expected. Kaiser et al. [338] learn a model for Atari games called SimPLe. Hafner et al. [262] use planning in latent space to learn dynamics

in a system called PlaNet.

Network architectures can be created that combine planning and learning. The study of model-based reinforcement learning is an active area of research. Recent experiments are reported in [610, 646, 202, 723, 338, 262]. More research is needed to see if all planning and learning of AlphaGo-like self-play approaches can be integrated into a single network architecture.

Explainable AI

An important emerging research area is explainable AI, or XAI. Explainable AI is closely related to the topics of planning and learning that we discuss in this book. When a human expert gives a recommendation, the expert can be questioned, and can explain the reasoning behind the advice. This is a very desirable property. Most clients receiving advice, be it financial, medical, or otherwise, prefer to receive a well-reasoned explanation.

Most decision support systems that are based on symbolic AI can be made to provide such reasoning. Decision trees [529], graphical models [328, 391], and search trees are eminently suited for following the decision points of an advice.

Connectionist approaches such as deep learning are more opaque. However, the accuracy of deep learning approaches has assured their popularity. The area of explainable AI aims to achieve the ease of reasoning of symbolic AI with the accuracy of connectionist approaches [257, 179, 107]. Hybrid approaches combining aspects of planning and learning are among the techniques that hold promise.

7.6 Practice

We have now come to the end of a long chapter, that covered many advanced topics, and provided pointers to even more research. Let us turn to practice now, to implementations.

First are questions to check your understanding of this chapter. Each question is a closed question where a simple, one sentence answer is possible.

Questions

1. What are the differences between AlphaGo, AlphaGo Zero, and AlphaZero?
2. What is the nationality of Fan Hui, Lee Sedol, and Ke Jie?
3. Which networks does AlphaGo have, and which training methods?
4. Which networks does AlphaGo Zero have, and which training methods?
5. What is a double headed network?
6. What is the error function that is trained in a double headed value/policy net?

7. What is a TPU and how is it different from a GPU?
8. What is the Elo rating of AlphaGo in 2016 (approximately: around 2000, around 2500, or around 3000)?
9. Which three elements make up the self-play loop? (You may draw a picture.)
10. How is self-play recursive?
11. What is the network architecture of AlphaGo?
12. What is the network architecture of AlphaGo Zero?
13. What is tabula rasa learning?
14. How can tabula rasa learning be faster than reinforcement learning on top of supervised learning of grand-master games?
15. What are the main results of the AlphaZero paper?
16. Name a top traditional Chess program.
17. Name a top traditional Shogi program.
18. Was the style of play of AlphaGo described by the Go community as boring or as exciting?
19. What de-correlation efforts does AlphaGo use?
20. What de-correlation efforts does AlphaGo Zero use?
21. What de-correlation efforts does AlphaZero use?
22. Explain similarities between planning, learning, System 1, and System 2.
23. What is curriculum learning?
24. What is transfer learning?
25. What is multi-task learning? Does AlphaGo have multi-task learning? Does AlphaGo Zero have multi-task learning?
26. What is meta-learning?
27. Describe three kinds of recursion in AlphaGo Zero-like self-play?
28. What is hyper-parameter optimization and how does it apply to reinforcement learning?
29. What is neural architecture search and how does it apply to reinforcement learning?
30. What is hierarchical reinforcement learning? Name three approaches.
31. Why are games with sparse rewards difficult for reinforcement learning?
32. Why are domains with continuous action spaces difficult for reinforcement learning?

7.6.1 Exercises

The AlphaZero General (A0G) code base is well suited for experimentation with self-play. We use it in the exercises of this chapter. A0G can be found at A0G.¹⁸ Note that you also have to install TensorFlow and Keras. They are at tensorflow.org.

1. Install and run A0G Othello 6x6. Play against the computer to see if everything works. Look at the source code, especially at `Coach.py`. Can you see a relation with the code in Listing 7.1? What is an episode?
2. Observe the learning processes of the different games Tic Tac Toe, Connect4, and Gobang. See how self learning progresses.
3. Vary learning rate, vary MCTS Nsims, vary iterations. How is performance impacted? How is learning time impacted?
4. A stable self-play process A0G depends in part on the amount of exploration in MCTS. Vary the Cpuct constant, and observe if training is impacted adversely.
5. Try different net architectures. How is performance impacted?
6. Take a traditional alpha-beta Othello program, for example Othello.¹⁹ See if A0G can learn a strong enough player to beat it.
7. Choose a simple game of your own preference, and implement it in A0G. Does it learn quickly?
8. *Implement Chess input and output layers in A0G, learn it tabula rasa, and play against Sunfish. You may want to have a look at LeelaZero Chess or ELF OpenGo for inspiration.
9. Play with tensorboard to look inside the brain/training process and understand training process
10. Implement hyper-parameter tuning of A0G parameters
11. *Write a hyper-parameter optimizer for A0G, trying different net architectures. This is challenging and quite computationally intensive. How can you reduce the computational demands?
12. Can single agent optimization problems benefit from self-play? Implement the 15 Puzzle in A0G, and travelling salesperson. See also [390, 738].
13. See if transfer learning of 4x4 Othello, to 6x6 to 8x8 works, and works faster. Are you seeing a curriculum learning effect?

¹⁸<https://github.com/suragnair/alpha-zero-general>

¹⁹<http://dhconnelly.com/paip-python/docs/paip/othello.html>

7.7 Summary

The challenge of this chapter was to combine planning and learning, to create a self-playing self-learning system. We described the performance of AlphaGo, AlphaGo Zero, and AlphaZero in depth. Three of the strongest human Go players were resoundingly defeated. There was great interest in Go and AI by science and society.

Issues of generalization, over-fitting and training stability (the deadly triad) were overcome by the AlphaGo team in an impressive show of science and engineering prowess, for which they received many accolades.

The main conceptual innovation of this chapter is the impressive performance of self-play, as a combination of a planning and training loop, and as a combination of MCTS and DQN. Self-play, Figure 7.9, takes Evaluation (System 1) and Planning (System 2) and adds a third item, a learning-loop, creating a self-learning search-eval system.

The self-play loop uses MCTS to generate high-quality examples, which are used to train the neural net. This new neural net is then used in a further self-play iteration to refine the network further (and again, and again, and again). AlphaGo Zero thus learns from zero knowledge, tabula rasa. And, just as impressively, the same self-play architecture can learn tabula rasa Chess and Shogi as well, defeating decades of heuristic refinement.

MCTS is changed significantly in the self-play setting. Gone are the random play-outs that gave MCTS the name Monte Carlo, and much of the performance is due to a high-quality policy and value approximation residual network. The name Deep Q-network Tree Search would be more appropriate.

Reproducibility in science is of great importance to progress [282]. The impact of benchmarks and open source algorithms in this respect is important. The reimplementations of the self-play results in AOG, ELF OpenGo, PhoenixGo and Leela will stimulate experimentation and are of importance to progress towards general learning systems.

In this chapter we discussed enhancements of self-play, to understand why it works, and how its performance can be enhanced further. Self-play learns faster than a database approach. Self-play creates a natural curriculum learning situation, where the examples are ordered from easy to hard. We discussed the links between curriculum learning and transfer learning. The self-play success stimulates research interest in curriculum and transfer learning.

We also discussed further enhancements to self-play. Optimization of the network architecture and the training hyper-parameters are active fields of research. Also research into hierarchical reinforcement learning may well further improve performance of self-play, possibly generalizing it to other domains.

Historical and Bibliographical Notes

The work on AlphaGo is a landmark achievement in AI. Every AI researcher is encouraged to study it well. The primary source of information for AlphaGo are

the three AlphaGo papers by Silver et al. [604, 607, 606]. The papers contain a wealth of information, and are well written, although they can be a bit overwhelming at times, especially the detailed methods sections. Many blogs and popular press articles have been written about AlphaGo that might be more accessible.

Do not forget to watch the AlphaGo movie. Although it does not go too deep into the details, it is a fascinating account of the match, and its significance in AI. It is at AlphaGo Movie.²⁰ There are also very nice explanations on YouTube, for example at AlphaZero Explanation.²¹

AlphaGo was not created in a vacuum. There is a large body of research of which we will now discuss some of the more notable successes. Section 7.4.4 gave a brief overview, although there are more relevant works. Please refer to Table 7.4 for an overview of notable games.

Pre-MCTS, there are works on shape and move prediction [625, 68]. In the contexts of MCTS, many researchers worked on combining MCTS with learned patterns, especially to improve the random roll outs of MCTS. Supervised learning on grand-master games was used to improve playouts and also to improve UCT selection. Gelly & Silver et al. published notable works in this area [234, 608, 232], and others [64].

Patterns are also important in Othello. Around 1995 very good results were achieved by Logistello [112]. Logistello used logistic regression on patterns. The author also successfully introduced probabilistic depth reduction methods [114]. Graf et al. [248] describe experiments with adaptive playouts in MCTS with deep learning. Convolutional neural nets were also used in Go by Clark and Storkey [142, 143] who had used a CNN for supervised learning from a database of human professional games, showing that it outperformed GNU Go and scored wins against Fuego. In Chess, the program DeepChess by David et al. [157] performed end-to-end deep learning in Chess, achieving results on par with Falcon and Crafty, two alpha-beta based programs. In Checkers, Blondie24 [132, 217] used a network to evolve board features, also achieving good results.

Tesauro's success inspired many others to try temporal difference learning. Wiering et al. and Van der Ree [729, 685] report on self-play and TD learning in Othello and Backgammon.

The program Knightcap [41, 42] and Beal et al. [46] also use temporal difference learning on evaluation function features. Veness et al.[694] use self-play to learn evaluation function weights. They name their approach *search bootstrapping* and, referencing Samuel's Checkers program, describe it as follows: "The idea of search bootstrapping is to adjust the parameters of a heuristic evaluation function towards the value of a deep search." This sentence captures the idea of self-play, as used in AlphaZero, quite well. Arenz [17] applied MCTS to Chess.

Heinz also reported on self-play experiments in Chess [280]. In other games, Guo et al. [258] report on the use of MCTS to play Atari games, their combination of MCTS and DQN outperforms DQN on Atari.

In Poker impressive results have been published [93, 461] with combinations

²⁰<https://www.alphagomovie.com>

²¹<https://www.youtube.com/watch?v=MgowR4pq3e8>



Figure 7.23: Pieter Abbeel

of planning and learning algorithms, most notably counterfactual regret minimization and self-play.

For more information on hyper-parameter optimization, see [67, 312]. A reference work on Evolutionary Strategies is [26]. To learn more about curriculum learning, see [63, 428]. For more information on transfer learning, see [523, 666].

Ever since his paper on reinforcement learning for aerobatic helicopter flight [3] (there are spectacular videos on the web) Pieter Abbeel's work in robotics and in reinforcement learning has been inspiring and influential. See Figure 7.23.

Readable and insightful introductions into the field of abstraction in deep representation learning are [61, 393]. A good place to start for network visualization is the ZFnet paper [745].

Chapter 8

Conclusion

This chapter is about the future.

In the previous chapters we have discussed in much detail the methods that have been used to achieve intelligence in games. The focus has been on single games, on narrow intelligence. The purpose of this chapter is to see how narrow intelligence can be extended to general intelligence. We will approach general intelligence in two steps. First, in Section 8.1, we will analyze the methods of the previous chapters, to find common threads. Second, in Section 8.2, we will discuss new problem domains that expand our horizon beyond zero-sum perfect-information games. Then, in Section 8.3, we will compare human and artificial intelligence.

Core Problems

- What are the essential methods of intelligent decision making in games?
- What games can be used in our quest towards general intelligence?
- How are artificial and human intelligence related?

Core Concepts

- Automating generalization, levels of abstraction
- Multi-agent, imperfect information games
- Specialized intelligence, general intelligence

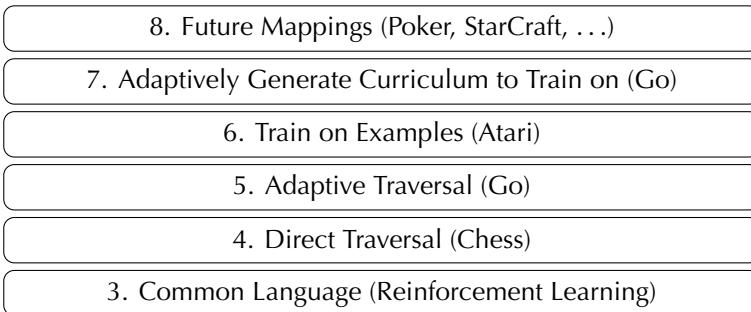


Figure 8.1: Stack of Chapters, with increasingly advanced Mappings from Problems to Methods (from bottom to top)

8.1 Artificial Intelligence in Games

In this section we will review the methods that we have seen in the book so far. We will analyze them to see how they created an artificial kind of intelligence. We will look for common aspects in the hope that we can extend them to create new methods, to create an even more human kind of intelligence, allowing better human-computer interaction and better human-computer understanding.

Figure 8.1 shows the structure of this book as stack of chapters, with the methods building on each other. They start with being closely related to the state space, directly traversing the state space using manually constructed features. Soon enhancements arrive, and adaptive traversal and exploration. Features are learned, and advanced training methods are invented, that balance generalization, overfitting, correlation and curriculum.

All these methods can be understood with a basic search-eval architecture. In this section we will discuss how the search-eval architecture can be extended to allow further generalization of intelligent behavior in games. Planning, learning and self-play were able to succeed in more than one game, and reach world champion level play. What is the essence of these methods that work so well, and how can we extend them further? What are their key elements?

8.1.1 Search-Eval

The search-eval architecture is one of the principles of the game playing programs that we discuss in this book. What exactly makes it work so well, and why?

Search

To determine the value of a state, the basic approach is to follow Bellman's equation, and to recursively follow all actions to all states and find their values. This basic approach should be used when the size of the state space is not too large,

and when it is essential to exactly follow actions and states, such as in games where tactical lines depend crucially on board state. The latter is the case in many games, and most game playing programs therefore make use of exact look-ahead, or search.

The former, a small state space, is often not the case, and a form of approximation must be used, to curtail the exponential time and space complexity of the exact approach.

Storage methods must support exact matching of states and values. A table or a tree, large enough to store all states in the state space can be used.

Eval

For large state spaces, approximation of states must be used. Value of states can be approximated without expanding the subtree of action-states by looking at the features in the state itself. Thus the value is approximated without traversing a full sub tree, but only examining the static features of the current state. This is typically a constant time problem, independent of the depth of look-ahead of the search algorithm, in contrast to an exponential time problem for search.

The features can be hand-coded based on heuristics, or the features can be learned from examples states, for which values (or policies) must be present. The parameters of the approximation algorithm are often stored as the coefficients of a polynomial (heuristics) or as a layered network (neural network).

Various methods are used in thee search and eval function, although most follow one of two principles: the principle of bootstrapping, and the principle of optimization.

8.1.2 Bootstrapping and Optimization

In the methods that we have discussed in this book, there are two principles that occur repeatedly. They are the principle of bootstrapping, and the principle of optimization. We will have a closer look at them.

Bootstrapping

Bootstrapping is often implemented using recursion, and divide and conquer makes use of this principle. Bootstrapping is at the basis of reinforcement learning, in Bellman's equation, defining the value of a state based on the value of future actions. Minimax, the principle of using the same model for the player and the opponent, also finds the value at the root of the game tree by bootstrapping. Finally, bootstrapping (recursion) features three times in AlphaZero self-play, in the move selection of MCTS, in the generation of the network for example states, and in the curriculum of games that are generated in a tournament against an identical copy of the player.

Recursion, or the calling of a function by the same function (with different parameters) is used by the programs to achieve bootstrapping.

	Exact planning (search)	Function approximation (eval)
Optimization	Q-learning	DQN
Bootstrapping	minimax	self-play

Table 8.1: Method Matrix

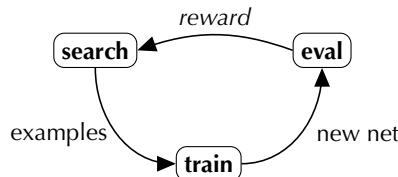


Figure 8.2: Simplified Self-Play Loop

Optimization

Optimization is the principle of choosing the best item out of a set, preferably using some smart means or extra information to do so efficiently. All reinforcement learners and all game playing programs are nothing more than advanced optimizers, using many times the principle of optimization to select best actions, best states, best examples, best network architectures, and best training hyperparameters. They do so within the search-eval architecture. They are bootstrapping optimizers in a search-eval architecture.

Table 8.1 show the principles of optimization and of bootstrapping in the context of the search-eval architecture. The cells of the matrix contain examples of these principles. Other examples exist, many exact algorithms use the principle of optimization, such as SARSA, A*, depth-first search, and many other algorithms exist for the other three cells as well.

To summarize, exact algorithms can be used for *planning*, approximation algorithms can be heuristic, or can be *trained*, and *self-play* uses the plan-train loop that we recall from Chapter 7, and that we show again in Figure 8.2 in a simplified form.

Exact algorithms optimize for the best action for a single state; approximation algorithms optimize for the best action over many example states, and self-play optimizes a player over many games.

Extra optimization dimensions can be added, such as optimization loops for hyper-parameters or network architecture.

Let us now try to extend the search-eval architecture with these ideas.

8.1.3 Intelligence as Layers of Optimization

Both planning and training use optimization methods that find the best element or minimize an error. The methods in this book work on different data structures, from trees for heuristic values, to networks with error values and gradients. Our methods are then typically composed recursively, such as minimax and self-play,

in which they use themselves. Intelligence in games has become recursively structured optimization.

Table 8.1 summarizes the optimization and bootstrapping methods against the search and eval architecture.

The self-play loop of Figure 8.2 showed self-play as a triply nested recursive training structure (see also Section 7.4.2 on page 238). The three levels of recursion are: (1) for computing each move it uses the same MCTS opponent in its look-ahead, (2) for generating its training examples for the function approximator it uses the same function approximators, (3) for training the games to generate the examples it uses self-play, playing against a copy of the player.

A danger of self-referencing systems is the occurrence of feedback loops, amplification of unwanted artifacts or noise drowning out the signal. With a system that uses self-referencing three times, prevention of feedback loops is crucial, as we saw with the emphasis on generality and exploration in the previous chapter.

Correlation

De-correlation of example states is important in this respect. To learn, new information is needed, or little will be learned. There has to be diversity in the states. *Correlation* between states is to be avoided since two correlated states have less new information to learn from than uncorrelated states (and correlated states are myopic, they cover a smaller part of the state space, and they can lead to cycles). If we treat them as if they do contain as much information as independent states, then our outcome is overly influenced, and our picture of reality becomes warped.

Beyond Triply Nested Recursive Optimization

Now that we have seen how one loop can be added to search-eval (an extra layer of optimization for an increase in generalization) it is natural to consider adding more layers of optimization, for even more generalization. See Figure 8.3.

The search-eval architecture allows generalization at more levels. Let us look at the different levels of abstraction that search-eval suggests. We start at the bottom.

We might consider *eval*, the state approximator, as the zeroth level assessor. The goal of *eval* is to directly translate the high dimensional board state into a single value. One level of abstraction higher we get *search and eval*, where the goal is to optimize for the best move in the space of approximated values. The second level is where we use self-play to learn a player; we optimize in the space of levels of play. A third level can optimize over other hyper-parameters. That results in a systems that returns increasingly smart players. The fourth level is another level of optimization. That can be over the network architecture. A fifth level may be optimizing over different domains, addressing the generality part of Artificial General Intelligence. A sixth level might conceivably be added. Perhaps this could be ethical decisions?

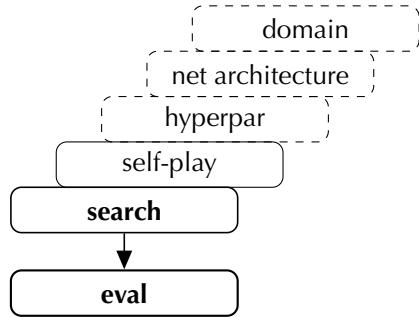


Figure 8.3: Intelligence in Games as Layers of Optimization

The search-eval architecture allows expression and reasoning about increasingly intelligent systems, expressing increasingly general levels of machine intelligence, all based on optimization and approximation.

In the system introduced here the first level uses search (minimax, or MCTS). The second is the self-play loop. The third and higher levels have no agreed upon order in the literature, although hyper-parameter optimization and network architecture evolving is being experimented with in contexts close to (but not yet on top of) self-play. Figure 8.3 illustrates the levels of intelligence that can be built on top of the search-eval architecture.

Implicit in the picture is a nesting of the different levels of abstraction. This is a computationally expensive solution. Already a two level self play system has very large run times. Each extra layer will multiply the run time by the number of instances that will be tried. Many algorithmic enhancements are needed before a higher level of abstraction becomes computationally feasible, or better global optimization methods must be applied (Section 7.5.3). Therefore, instead of nesting, sequencing the levels of abstraction may provide a solution if the levels are independent. If the choice of hyper-parameters does not influence the choice of network architecture this would work. By optimizing a level separately from the other levels, the time complexity of a nest of levels n of size s goes from $O(s^n)$ to $O(ns)$. The trade-off is that if the levels are not independent, then sequencing will miss interactions, losing out on better optima. Section 7.5.3 describes parallelization of self-play.

Jaderberg et al. [318, 319] describe an impressive result of a multi-level learning system. Their system uses a variant of reinforcement learning that learns a population of agents, in a multiplayer real-time strategy capture the flag game, Quake III. The computational demands of their approach are mitigated by not using an MCTS search and using relatively simple and fast RNNs.

Figure 8.1 highlights the importance of the relation between the games and the methods to solve them. Over the chapters of this book there is a progression to more and more advanced methods, driven by the difficulty of the problem domain. We start with a direct mapping between problem and method, and end with yet many layers of optimization.

The methods in this book progress from specific to general: starting with heuristic planning, we continue to adaptive planning, training from examples, and end at generating a curriculum of training examples. Each step adds another layer of abstraction, another layer of optimization.

8.1.4 On the Importance of Enhancements

The search-eval architecture provides a conceptual framework for game playing programs. To achieve a high level of performance, the enhancements, however, are important. High performing game playing programs from 1950-2006 were based on alpha-beta. Without transposition tables and singular extensions there would be no Deep Blue. Without fast playout patterns there would be no AlphaGo, without specific input layers no AlphaZero, without ReLU, dropouts, and GPUs, no deep learning Turing award, and without the replay buffer there would be no success in Atari. The enhancements are the result of deep understanding by the researchers of their domain. Understanding that is more specialized, and less general, than neat principled frameworks such as minimax, reinforcement learning, or function approximation.

In addition to the elegant conceptual frameworks, deep, dirty, domain-specific understanding is necessary for progress in this field [575].

8.1.5 Towards General Intelligence

Let us revisit the problem statement from the Introduction.

What are the machine learning methods that are used in Chess and Go to achieve a level of play stronger than the strongest humans?

Various reinforcement learning methods have beaten human champions in Chess and Go, ranging from heuristic planning, adaptive planning, function approximation to self-play. All methods fit in the classic search-eval architecture, and all use variants of the principles of bootstrapping and optimization. Heuristic planning can achieve levels of play far surpassing human ability in Chess and Checkers. We have also seen how to combine planning and learning into a self-play system that learns to play from scratch, achieving even *higher* levels in Chess, Shogi, and Go.

Did we achieve intelligence? For these three games, we certainly passed the Turing test.¹ Systems were created that behave at a level for which a human would need super human intelligence. The intelligence is, however, single domain intelligence. AlphaGo cannot play Shogi, and it cannot tell an entertaining joke.

AlphaZero showed that the methods generalize to three games, hinting at artificial general intelligence, although it is more precise to speak of training three almost identical systems to become three different systems specialized in their own kind of special intelligence (since the net, once trained for Go, cannot play

¹Actually, since we far surpassed human play, an argument can be made that in doing so we failed the Turing test, since no human can play at this level.

Chess). (GGP systems can play general games, but do not achieve performance close to AlphaZero, see Section 8.2.6.)

8.2 Future Drosophila

The reinforcement learning results in Atari and Go have inspired much further research. The use of concrete games, sometimes called *Drosophila*, has stimulated progress in game playing greatly. In addition to the usual two-person zero sum perfect information games, researchers have looked for games that capture other elements of real life, such as non-zero-sum (collaboration) games, imperfect information, single-agent games, and multi-agent games. In this section we will review some of these aspects.

8.2.1 Single-Agent Games

Reinforcement learning is a natural training paradigm for two-agent games. Reinforcement learning is also used in multi-agent and imperfect information games, such as StarCraft and Poker [278].

However, many optimization problems in science are single-agent optimization problems, such as navigation, the Traveling Salesperson Problem, the 15 puzzle, and the Satisfiability Problem [554]. MCTS and neural networks have been used to solve single-agent combinatorial problems [457, 552, 705, 616]. Single-agent optimization problems, where the shortest solution must be found, can have especially sparse state spaces in which few reward signals are present. Learning in such sparse spaces is hard. A solution can be to introduce intermediate goals.

The advantage of self-play is that it creates a curriculum learning situations, which speeds up learning. Laterre et al. [389, 390] have tried to apply self-play to bin packing, another standard single-agent optimization problem. The self-play method worked, but was outperformed by MCTS. Other work applies approximate reinforcement learning to TSP [365, 58, 227]. Xu and Lieberherr [738] also used self-play in combinatorial optimization. It can be expected that more research into single agent self play approximators will appear, attempting to benefit from the curriculum learning effect that is inherent in self-play approximators [428].

8.2.2 Real-Time Strategy Games

Imperfect information games have been studied extensively in reinforcement learning [621, 333]. The multi-agent imperfect information Drosophila are card games (such as Bridge and Poker) and real time strategy games (such as Defense of the Ancients (DOTA) 2 and StarCraft). StarCraft has been studied for some time [704, 671, 681], as have DOTA 2 [182, 33, 488], and other RTS games [526, 388, 649, 319]. Easy to use Python interfaces have been created for StarCraft and DOTA 2. These games are highly challenging for current methods. They are multi player, imperfect information, have large and continuous action spaces and large and sparse

states spaces, and long time credit assignment challenges. Early results are being achieved. Leibo et al. [399] have written a manifesto for further research into curriculum learning in multi-agent games.

8.2.3 Negotiation and Collaboration

Non-zero-sum games are close to everyday life situations. Many real world situations include negotiation and collaboration and offer win/win situations. AI has studied negotiation and collaboration extensively.

Negotiation and collaboration also occur in card games. Poker and Bridge have large action spaces and a large state space. Poker has been researched for some time and for simpler (two-player) versions of the game successful programs have been written [93, 461, 103, 362]. Recently, success has also been reported in multi-player Poker. Cooperation and negotiation are central elements of Bridge, and research is continuing [617, 10]. Negotiation has also been studied for some time [326, 369, 327], including negotiation games such as Diplomacy [166, 168, 167, 165, 370, 537].

Let us now have a closer look at two games, Poker and StarCraft.

8.2.4 Poker

Poker is an excellent game to show what progress has been made in imperfect information games.

Poker has been studied for some time in artificial intelligence and computer Poker championships have been conducted regularly [78, 79, 238, 549, 34, 93, 79, 238, 94, 562, 461]. No-limit Texas hold'em is the most popular form of poker. The two-player variant (called Heads Up) prevents opponent collusion. Heads-up no-limit Texas hold'em (HUNL) has been the primary benchmark for imperfect-information game play for several years.

Poker features face down cards, whose value is unknown to the other players. The face down cards constitute hidden information. This hidden information creates a large number of possible states, making Poker a game that is far more complex than Chess or Checkers. There are many variants of Poker, the state space of HUNL is reported to be 10^{161} , simply enumerating the state space to find an optimal policy is infeasible [103]. A further complication is that a player reveals information during the course of play. Players may be mislead through bluffing, i.e., deliberately betting a large sum to make the opponent believe that one has stronger cards than one has. Therefore, an AI must balance actions, so that the opponent does not find out too much about the cards that the AI has.

In 2018, one of the top programs, Libratus, defeated top human professional in HUNL in a 20-day, 120,000-hand competition featuring a \$200,000 prize pool [103]. We will now describe this program in more details, based in [103]. Libratus consists of three main modules.

The first module computes a smaller version of the game, and then computes game-theoretic strategies. The algorithm used is a variant of counterfactual regret

minimization (CFR). CFR is an algorithm to find Nash equilibrium strategies [752]. It is based on multi-armed bandit theory.

CFR is a recursive algorithm: it learns a strategy by repeatedly playing against itself. CFR begins with a random strategy. It then simulates playing games against itself. After every game, it improves its decisions. It repeats this process many times, constantly improving the strategy each time. As it plays, it comes closer to an optimal strategy, a strategy that can do (on average) no worse than tie against any opponent. CFR generates strategies that minimize exploitability, which means that it plays conservative. CFR algorithms do not exploit imperfect play from opponents in a way that a human would. From a game-theoretic point of view, a perfect strategy is one that cannot expect to lose to any other player's strategy. A perfect strategy in the presence of sub-optimal players who have weaknesses would be one that exploits those weaknesses.

When a later part of the game is reached during play, the second module of Libratus constructs a finer-grained abstraction for that subgame and solves it in real time.

The third module is the self-improver, which enhances the first policy. It fills in missing branches in the first version of the game and computes a game-theoretic policy for those branches. Computing this in advance is infeasible. Libratus uses the opponents' actual moves to suggest where in the game tree this is worthwhile.

In the experiment against human players Libratus analyzed the bet sizes that were most heavily used by its opponents during each day of the competition. On the basis of the frequency of the opponent bet, the program chose some bet sizes for which it would try to calculate a response overnight. Each of those bet sizes was then added to the strategy, together with the newly computed policy following that bet size. In this way, the program was able to narrow its gaps as the competition proceeded.

Interestingly, Libratus was based on heuristics, abstraction, and game theory algorithms, and not on deep reinforcement learning. However, later work introduced Deep Counterfactual Regret Minimization [102], a form of CFR that approximate CFR equilibria in large games, using a 7 layer neural network. This algorithm is also the basis for Pluribus, the later multi-player program, defeating top players in six-player Poker [104]. The algorithm in Pluribus uses a form of self-play in combination with search.

Another top program, DeepStack, does use randomly generated games to train its deep value function network [461]. In Poker we see two different approaches achieving success.

8.2.5 StarCraft

StarCraft is a multi-player real time strategy game of even larger complexity [487]. It is a good test bed for AI research, since it features decision making under uncertainty, spatial and temporal reasoning, collaboration and competition, opponent modeling, and real-time planning. The companies DeepMind and Blizzard have released a Python interface to stimulate research in StarCraft [704, 702].

The state space of real time strategy games is larger than traditional board games such as Chess, Checkers, or Go. The state space has been estimated to be on the order of 10^{1685} , a very large number [487]. The number of actions in each state is also large. Standard exact planning methods are not feasible, and need to be augmented with approximate methods to be effective. Planning in real time strategy games has multiple levels of abstraction (see hierarchical reinforcement learning in Section 7.5.4). At a high level of abstraction, long-term planning is needed; at a low level of abstraction, tactical battles must be fought and individual units must be moved.

StarCraft offers opportunities to explore many challenging new methods in reinforcement learning. First, it is an imperfect information game. The terrain map is partially visible. A local camera must be moved by the player to acquire information. There is a “fog-of-war” that obscures unvisited regions. The map must be explored proactively to determine the opponents’ state. Second, StarCraft is a multi-level-multi-agent game in which several players compete for influence and resources. StarCraft is also multi-agent at another level. Each player controls many units that need to cooperate to achieve the player’s goal. Third, the action space is large and diverse. There are around 108 actions for a player, which can be selected with a point-and-click interface. There are many different types of units and buildings, each type with its own local actions. Furthermore, the legal actions vary during play as it advances through possible technologies. Finally, games are long. They last for thousands of states and actions. Decisions early in the game, such as which units to build, have consequences that will be seen much later in the game when armies meet in battle. There is a long distance in credit assignment, and exploration is important in StarCraft [487].

In a series of test matches held in December 2018, DeepMind’s AlphaStar beat two top players in two-player single-map matches, using a different user-interface. The neural network was initially trained by supervised learning from anonymised human games, that were then further trained by playing against other AlphaStar agents, using a population-based version of self-play reinforcement learning [701, 702].

8.2.6 General Game Playing

As has already been mentioned a few times, another approach to studying generality in game intelligence is General Game Playing (GGP). GGP takes a very different approach, not using concrete existing games, but using abstract games.

GGP started after Deep Blue won from Kasparov, and the field was looking for their new Drosophila. One of the criticism of the field of game playing was that all programs had to be specifically developed for each game anew. Human intelligence, on the other hand, is general, and can just as easily be applied to different tasks.

General game playing is the design of artificial intelligence programs that must be able to play more than one game successfully. For many Chess-like games, computers are programmed to play these games using a specially designed algorithm, which cannot be transferred to another context. A Chess-playing computer

program cannot play Checkers. A general game playing system should be able to play any game for which it is given the rules. It was hoped that the general methods coming out of GGP research would also be able to help in other areas, such as in assisting search and rescue missions [507, 235].

The Logic group at Stanford University hosts the General Game Playing project [235, 661] and initiates competitions. The group is lead by Michael Genesereth (see Figure 8.4).

The GGP approach is to create a system and a language in which many games can be expressed. The challenge is then to create a single game player that can play any game that occurs in a tournament. GGP is an interesting mix between a model-free and a model-based approach, in which programs must be able to accept any model that can be expressed in de game description language. The GDL is similar to PROLOG. In GGP a parser for the game description must be written, which yields the state/action transition function, and termination rules. The program has to come up with an efficient search procedure, and a good evaluation function. It is hard to program heuristics when the domain is unknown. It was hoped that GGP would stimulate the creation of general problem solving methods. It is therefore curious to note that a method developed in Go, the other Drosophila, has been successful in GGP. Monte Carlo Tree Search is a general search procedure that does not need domain specific heuristics, and turned out to be well suited for GGP.

Thus, in general game playing the goal is more challenging than in Go, since the rules of the games are not a given, but are provided in the form of a formal game description, and the program has to be creative enough to come up with a way to play any game that can be specified in the given language.

Arguably GGP is a less spectacular Drosophila, since there is no human world champion to beat, and there is no active human gaming community. In another sense GGP is more spectacular since it tries to find solutions that work for all games.

GGP has yielded a range of interesting advances towards more general AI. Much research in GGP is being performed [509, 659, 578, 660, 661, 586, 438, 587, 212, 82, 614, 615, 716, 713]. It will be interesting to see if more techniques from the two fields (i.e., general and special game playing) will cross over.

There are other general game playing systems, which use their own languages for defining the game rules. In 1992, Barney Pell introduced Meta-Game Playing, one of the earliest programs to use automated game generation. Pell's Metagamer was able to play Chess-like games, given game rules definition in a language called the Game Description Language [507, 508, 235]. Following the GGP idea a General Video Game Playing competition has been started [511, 510, 402, 378].²

Both AlphaZero and GGP show how general game play can be achieved. The AlphaZero approach to general game playing is less general, requiring hand designed input and output layers, where GGP programs generate these automatically from a description of the rules. The game complexity of Go, Chess, and Shogi is

²<http://www.gvgai.net>



Figure 8.4: Michael Genesereth

much larger than that of typical GGP challenges. GGP is developed to stimulate the research into general methods for game playing.

8.2.7 Domains in Science

As mentioned in Section 7.4.1, the AlphaGo results have attracted attention of many other fields in science. Complex combinatorics is of interest to theoretical physics [552, 495], chemistry and pharmacology, specifically for retrosynthetic molecular design [597] and drug design [693]. Image recognition is of interest to experimental physics [31], biology [478] and medicine [164]. Decision support is of interest to medical decision making [300]. Reasoning and image analysis are of interest to smart industry [239, 323], linguistics and humanities [627], and strategic reasoning is of interest to law, international relations, and policy analysis [24, 411, 690, 121]. Finally, the applications of robotics and reinforcement learning are of great interest to the cognitive and social sciences [325, 100]. Of particular interest are cheminformatics including drug design [597], theoretical physics, cognitive robotics and law and negotiation.

Many collaborations between artificial intelligence and these other fields are emerging.

8.3 A Computer's Look at Human Intelligence

Now that we have discussed the reinforcement learning methods, and discussed future challenges in reinforcement learning in games, it is time to compare the artificial approach to intelligence to the other approach, the human kind.

As we already mentioned in Chapter 2, human intelligence is a complex, multifaceted concept. Many different definitions of intelligence have been proposed. Intelligence involves recognition, reasoning, memory, learning, problem solving,



Figure 8.5: Daniel Kahneman

and creativity. Human intelligence is usually also assumed to include understanding, emotion, self-awareness and purpose (volition). Intelligence is the ability to infer knowledge from information, and to apply that knowledge towards adaptive behavior within a context. See, for example, Legg et al. [398] or Neisser et al. [473].

We will now discuss these elements of intelligence in more detail, and analyze how human and artificial intelligence differ in their approach.

Memory and Recognition

One of the basic elements of intelligence is the ability to recognize objects or features that have been encountered before. This requires sensors, a recognition system, and a functioning memory. All have been studied widely in human psychology. In Kahneman's terminology System 1 is the part of our thinking that recognizes objects. Figure 8.5 shows a picture of Kahneman. Human recognition works fast and associative.

Computers have always had memories, and over the years the amount of memory of computers has grown greatly. Computers have long struggled with the kind of recognition that humans do so easily. Only after CNNs were introduced (fashioned after an animal visual cortex) and after enough training data became avail-

able in MNIST and ImageNet, and after enough computing power in the form of GPUs became available, only then did computer recognition for images start to achieve impressive results.

Reasoning

Reasoning is another core element of intelligence. In Kahneman's terminology, reasoning is System 2, or Thinking Slow. Intelligence is considered a positive trait, and many humans and scientists pride themselves on their capacity for elaborate and deep logic reasoning. It is something that children are trained at in school topics such as calculus, mathematics, and history. It is the reason why Chess classes are present at schools.

Indeed, one of the precepts of classical economics is the assumption of rational behavior of the homo economicus. One of the interesting aspects of Kahneman's book is that it stresses the importance of System 1, the fast, intuitive, type of thinking. The non-rational part of our thinking is guiding much of our daily behavior. (Why do people buy lottery tickets, if we know that the expected value of a lottery is negative?) Indeed, Kahneman's work on prospect theory, together with many other important insights from psychology, has been instrumental in the creation of the field of behavioral economics. Thaler and Sunstein's have written a popular book on nudging [658], applying insights from behavioral economics to influence behavior of people.

In artificial intelligence, systems for logic reasoning were one of the first systems built, right up there with Chess and Checkers programs [736, 383, 475]. Some of the many successful results of this research are the programming language PROLOG [145, 95], research into semantic reasoning [16, 296] and negotiation [412].

Learning

One of the central aspects of intelligence is the ability to adapt one's behavior based on information from the environment. The ability to learn is certainly a crucial aspect of intelligence. When children are young, learning and playing are closely related. Johan Huizinga, an eminent Dutch scholar of history, argues in *Homo Ludens* [311] that play may be the primary educational element in our human culture. Small children learn through playing, through interaction with their environments. Later, they learn in school, in a less-playful regimented curriculum. Some of our learning is by repeated exposure to examples. In supervised learning we teach ourselves associations and reflexes. Children who have been taught the tables of multiplication can immediately answer the question: "What is six times seven?" using their system 1. Others have to use system 2 to answer this question.

In computers, machine learning also uses memory and recognition capacities to build up associations. If we teach (supervise) ourselves, we use our rational reasoning capacities to do so. Learning has been recognized early on as a second central element of artificial intelligence. Indeed, since the impressive results with deep learning in image recognition, the term machine learning has become

almost synonymous with artificial intelligence, sometimes forgetting about reasoning altogether.

Creativity

Creativity is the process in which something new and valuable is created [521]. Creativity is strongly associated with human intelligence.

There have been quite some studies into artificial creativity. A computational view has emerged, in which creativity emerges out of computational processes [582]. Indeed, in addition to automated theorem proving [736, 419, 76], fields such as artificial painting [147] and artificial music [207, 680] have a long history in AI.³ A formal theory of creativity has been put forward by Schmidhuber [583]. If we think back to Fan Hui's description of AlphaGo's moves ("speechless") we see words describing beauty and creativity (Section 7.1.1 on page 214).

Understanding

Understanding is a concept that is related to human intelligence. Clearly, humans can lay claim to understanding something. Insofar as computers are not self-aware or have an identity, computers cannot be said to understand something. Computers can be made to analyze a concept and explain the reasoning that supports a certain conclusion, but it would be harder to claim that an explanation is equivalent to understanding [124].

A Chess end-game analysis can produce the exact subtree proving why a certain move is the best move in a position. That does not mean that the computer has an understanding of what it analyzed, even if the explanation is identical to one given by a human.

Emotion

A similar reasoning applies to emotion. Emotion is an important aspect that determines much of human thought and behavior. Much of our evolutionary survival skills are encoded in our emotional reactions, and much of our daily behavior is too [336, 413, 99].

In artificial intelligence, the field of affective computing studies the formalization, recognition and reasoning with emotions [413, 88]. As with understanding, a computer may be able to recognize emotions, it can be made to behave as if it has emotions, but few people would say that it has emotions.

Affective computing, and the models of emotion, will yield better human computer interactions, and interactions with robots that appear more natural.

³See for example the International Computer Music Association at <http://www.computermusic.org>.

Self-Awareness, Purpose

Most people would argue that computers do not possess self-awareness, humans do. Together with our free will, it guides much of our intelligent behavior. Intelligent computer behavior is guided by one or more goals that it tries to achieve. Humans decide on their own goal, for computers the goal is externally specified.

Note that the strict behavioral approach of the Turing test could allow a situation where a machine is so good at answering questions concerning self awareness and introspection, that we must conclude that it is indistinguishable from a sentient human. The question of self-aware machines raises many interesting neurological and philosophical question. See, for example, [358, 253].

8.4 System 3?

Kahneman describes two kinds of human thinking: Thinking Fast and Slow [336], also known as system 1 and system 2. In this book, we have seen how artificial intelligence in games is created out of planning and learning. The exact and approximate methods in the search-eval architecture all have parallels to Thinking Fast and Slow (see Table 8.2). In addition to these parallels, there are important

System 1	System 2
fast	slow
recognition	reasoning
reflex	rational
approximate	exact
train	plan
eval	search

Table 8.2: Fast and Slow, Eval and Search

differences between human and artificial intelligence. Human intelligence is general, artificial intelligence is mostly specific to one domain. Human intelligence is self-aware, artificial intelligence is judged by behavior, to pass the Turing test.

Let us first look at Thinking Fast. Thinking fast is reflex-like recognition. It is not so much like the training of the function approximator, which is off-line, and takes days, as it is the usage of the approximator, which is on-line, and takes milliseconds.

This brings us to revisit a point made earlier in Section 7.4.3. We want to extend Kahneman's scheme of two kinds of thinking, in which system 1 is Thinking Fast, and system 2 is Thinking Slow. Now self-play has an analogon in Kahneman's world. Self-play consists of exact planning methods being used to generate examples to train the function approximator (see Figure 8.2). System 2 (slow) can be used not only to reason and solve complicated problems such as what is the product of 6 and 7, it can also be used to train system 1 (fast) by repeatedly

exposing system 1 to examples that system 2 generate. In this way we create a third mode of thinking that can be called conscious *learning* or *training*. A system that performs this kind of deliberate learning in which system 2 teaches system 1, might be called system 3. Anyone who has learned words of a different language, or has recited the tables of multiplication at school is familiar with this kind of learning. Practice makes perfect. Learning by self-play exists both in AI and in human intelligence. In Section 8.5.3 we will discuss a consequence of the fact that artificial intelligence is now able to do system 3 training.

Now that we have seen some similarities between human and artificial intelligence, let us have a look at some of the differences.

8.5 General Intelligence

We have superficially explored the limits of artificial intelligence, let us look at the strengths. Some of the above aspects of human intelligence can readily be simulated by a computer, such as recognition, memory, reasoning, and learning.

8.5.1 Artificial Intelligence

For a computer to be artificially intelligent it should behave the same as human intelligence. An intelligent computer must be able to remember, to recognize, to reason, and to learn, in order to pass the Turing test.

As we have seen in this book, there is much research on techniques for recognition, memory, reasoning, and learning. In describing the behavior of Deep Blue and AlphaGo, some Chess and Go players have described it in terms of beauty, divine intervention, and other language usually used for creative works.

In the fields of game playing it can reasonably be argued that genuine artificial intelligence exists. In Backgammon, Checkers, Chess, Atari, and Go, the best games are played by silicon.

8.5.2 Exceptional Intelligence

Exceptional intelligence in a single field is valued highly among humans. Exceptional Soccer intelligence, Tennis intelligence, Violin intelligence, Novelist intelligence, or Chess intelligence is scarce (and can be monetized through prize money and sponsor deals).

Computers have such specialized intelligence. In fact, that is all they have. Computers can do one thing well, humans can do many things. Evolution has endowed humans with intelligence that allows them to do many things, in order to survive in the world in which they live. The intelligence of game playing programs has not evolved. Game playing programs have been programmed to (learn to) excel at one game. They have specific intelligence.

When looking carefully at the history of game intelligence, an interesting pattern emerges. The specific game intelligence started with search algorithms that

exploit the game structure (two player zero sum adversarial game follows the minimax rule). Game specific heuristic search enhancements made the programs even more game specific, and the eval algorithms are based on game-specific heuristics as well. In some cases the evaluation function is tuned with a (general) learning algorithm.

The minimax algorithm itself is general in the sense that it applies to all two player zero sum games.

A subsequent search algorithm was MCTS. It was more general than minimax, since it also works for single agent (optimization) problems. It also is more general because it works without a domain specific evaluation function.

To achieve high performance, domain-specific heuristic methods were added, such as pattern-based playouts, and selection rules that can incorporate priors.

Next are more general learning algorithms, such as supervised and reinforcement learning neural network function approximators. They are even more general, in theory, although tuning a network architecture is a long and tedious process in which different network architectures and hyper-parameters are tried and tested on the specific game for which it should work.

Self-play takes this a step further, forgoing the games databases of grand master games, and learning itself in a curriculum learning fashion to play a game.

The pattern that emerges is that the methods in reinforcement learning in games have progressed from domain specific to general (which is then partly undone by domain specific performance tuning). The trend is towards more generality in methods, and this generality has now reached the point where it is starting to spill over to general application domains. As we saw in Section 8.2, more and more applications emerge where the self-play AI methods work, and a single architecture may even work in a few different games.

8.5.3 General Intelligence

For humans, intelligence is general. The intelligence of humans applies to many different domains. A human being with little general intelligence and high intelligence in one special domain is unusual. However, most individual humans do have certain specialities in which they excel.

Some game playing programs have been designed to exhibit general intelligence. The AlphaZero architecture can learn itself to play three games exceedingly well, provided that the right input and output layer of the neural network are specialized.

General game playing programs are designed to be general enough to interpret all games that can be specified by the Game Description Language.

Practicing, or the 10,000 hour rule

How does one achieve high levels of intelligence? To reach the top in tennis, soccer, violin playing, writing, or Chess requires talent and practice. Indeed, it is sometimes argued that “all” it takes to achieve success is a total of 10,000 hours of focused practice [241]. Scientific studies dispute this point of view and the

number of 10,000, and argue that the focus on practice is an oversimplification. However, it is clear that, in addition to talent, practice is indeed important in games, music and sports (see [424]).

To learn new behavior, deliberate attention of system 2 is necessary. During training we perform the actions repeatedly, under the conscious guidance of system 2. At some point, system 1 will have picked up enough to perform the behavior semi-automatically. Less attention of system 2 is needed. After enough practice, the behavior will be performed without any deliberate attention. System 1 has learned to do it unattended. $6 \times 7 = 42$.

8.5.4 Curriculum Learning

Following a suitable curriculum improves training results. A good curriculum speeds up training, and achieves better end results [188]. Human formal teaching (in schools) follows a curriculum, and the same improvements have been found in supervised and reinforcement learning in artificial intelligence.

Self-play in games creates a curriculum-like setting, where training examples are generated from easy to hard. This allows for quick and effective training. Indeed, as was mentioned in Section 7.2.5, AlphaZero learns to play at a level beyond that of human world champions in days, where human champions take years of dedicated training to reach that level.

AlphaZero is not only better at playing, but also learns much faster than the human champions.

8.5.5 The Future will be Fast and General

We have come to the end of this book. We have discussed a wide array of artificial intelligence technologies, and we have seen programs reach amazing achievements. The technological basis in this book has been the search-eval architecture (or exact planning plus function approximation) plus self-play organized as layers of recursive optimization.

This book has shown how game playing has progressed from game specific heuristic methods to general learning methods that teach themselves to play. Interestingly, a similar basic architecture is able to learn different games, from tactical Chess, strategic Go, to hidden information Poker, to multiplayer StarCraft.

In this chapter we have looked at the link between artificial and human intelligence, in general terms of recognition, reasoning, and self-awareness. We have also looked at the correspondence between planning and learning, and thinking fast and slow.

General Games

A major difference between artificial versus human intelligence is specific versus general intelligence. Artificial intelligence has achieved very strong results in specific domains, and the challenge is now to extend these results to more general

intelligence, for different games, and beyond games, to fields such as negotiation and human-robot interaction.

General Domains

Achieving general learning is studied in fields such as lifelong learning [665] and meta learning [210, 224], fields related to transfer learning [491] and curriculum learning [63].

The game programs studied in this book provide hints at how an architecture for lifelong general curriculum learning might look like, although solutions must be found for excessive computational demands. work.

Fast Training

With self-play, computers fashion their own training curriculums and training progresses faster than before. For the games of Chess, Shogi, and Go, one program was able to teach itself through self-play to world champion level and beyond in a matter of days. Humans take years of dedicated study to reach this level of play. In these games, computers not only play better, but also learn faster.

Future research will show if the same results of fast and powerful learning transfer to other games, to other problem domains, and to real world general intelligence.

8.6 Practice

Below are some questions to check your understanding of this chapter. Each question is a closed question where a simple, one sentence answer is possible.

Questions

1. Explain the search-eval architecture briefly.
2. Describe bootstrapping and optimization.
3. Why is diversity important in learning?
4. Explain triply nested recursive optimization.
5. How does automation lead to generalization?
6. Why are enhancements important?
7. Describe the relation between recursion, optimization, features, generalization, learning, and intelligence.
8. Why is Poker an interesting research topic?
9. What is general game playing?

10. Describe System 1, System 2, and System 3.
11. Describe General and Special intelligence.

At this point, you have become quite proficient at implementing reinforcement learning in games technology. Below are ideas for larger projects, to help you on the path to further research. These are challenging projects, perhaps suitable for a thesis.

Exercises

1. Design a simple imperfect information game and implement it in A0G. For example: Blackjack. Does it learn? Does self-play work in imperfect information?
2. Write a self-play single-agent player for your single-agent optimization problem of choice. [challenging]
3. Interface A0G with the StarCraft interface, and create a multi-player self-play system. [challenging]
4. Design a cooperative multi-player game. For example, merging traffic on a highway on-ramp, with or without signalling/negotiation. [challenging]
5. Write a Negotiation or Diplomacy game. See [24] for negotiation competitions. [highly challenging]

8.7 Summary

In this chapter we have looked back at what we have covered in this book. We have looked at the methods that form the search-eval architecture. The search-eval architecture can also be seen as an architecture with several nested levels of optimization, of which self-play is just one. We see how artificial intelligence methods are used on artificial intelligence, to achieve higher levels of intelligence, or more general intelligence.

We have also briefly discussed other techniques such as hierarchical reinforcement learning and meta learning. Finally, we have looked at new application domains, beyond two-player zero-sum perfect information. Multi-agent imperfect information games approach more aspects of the real world. Our techniques have progressed to the point where creating players for Poker and StarCraft are realistic challenges, that are succeeding.

We discussed an alternative Drosophila, General Game Playing (GGP). Where the challenge of Go lies primarily in the size of the state space, in GGP the challenge lies in being able to play a game for which the rules are not known beforehand.

We have compared artificial intelligence with human intelligence. We have looked at Kahneman's two systems, System 1 for thinking fast, and System 2 for



Figure 8.6: Sarit Kraus

thinking slow. In computers, thinking fast is like a function approximator, and thinking slow is planning. Self-play uses planning to generate examples for the function approximator to learn from. In Kahneman's terms, this could be called System 3.

Self-play learns fast, due to curriculum learning. We have noted that AlphaGo Zero's artificial intelligence learning itself to play Go took days, whereas humans need years of dedicated study to learn this level of Go, if they succeed at all.

We have looked at the elements that comprise human intelligence, and noted again that artificial intelligence is special. At least in the narrow domain of zero-sum board games, we have the situation that computers play better and learn faster than humans. However, human intelligence remains more general.

Future research will try to enlarge the domains in which artificial intelligence works. The techniques covered in this book will undoubtedly contribute to this research.

Historical and Bibliographical Notes

Negotiation and collaboration are important elements of real world decision making that have been studied extensively in the field of AI. Kraus et al. [369] provide an overview. Kraus (Figure 8.6) has contributed extensively to Diplomacy research [370] and to negotiation competitions. She received the 1995 IJCAI Computers and Thought award.

The field of probabilistic reasoning is of central importance in AI. An early model on Poker is Koller et al. [362]. Koller's work on probabilistic reasoning is widely recognized in AI as well as in biomedical sciences. Koller (Figure 8.7) received the 2001 IJCAI Computers and Thought award.



Figure 8.7: Daphne Koller

Part I

Appendices

Appendix A

Deep Reinforcement Learning Environments

Artificial intelligence is an open field of research. Many researchers publish their code, allowing for easy reproduction of experiments, and allowing researchers to build upon each other's progress. This appendix contains pointers to software environments that were meant to help you further in your own research.

We list three types of environments: general programming and learning environments, deep learning environments, and self-learning environments.

We start with general learning environments.

A.1 General Learning Environments

General purpose programming languages such as Python allow us to program the experiments that we wish to perform. One step further are machine learning environments such as Weka [734, 263].

For deep learning, well known environments are Caffe [321], Facebook's PyTorch [500], Theano [65] which has been subsumed by Google's Tensorflow [1, 2], and its user friendly add-on Keras [137]. Keras is perhaps the easiest way to start

Name	Type	URL	reference
Python	general purpose language	https://www.python.org	[545]
Weka	machine learning framework	https://www.cs.waikato.ac.nz/ml/weka/	[734]
Caffe	deep learning framework	http://caffe.berkeleyvision.org	[321]
PyTorch	deep learning framework	https://pytorch.org	[501]
TensorFlow	deep learning framework	https://www.tensorflow.org	[2]
Keras	deep learning library	https://keras.io	[138]

Table A.1: General Learning Environments

Name	Type	URL	reference
ALE	Atari Games	https://github.com/mgbellemare/Arcade-Learning-Environment	[55]
Gym	RL environments	https://gym.openai.com	[98]
Stable Baselines	RL algorithms	https://stable-baselines.readthedocs.io/en/master/	[286]
Dopamine	deep RL framework	https://github.com/google/dopamine	[122]
RLLib	Distributed RL	https://ray.readthedocs.io/en/latest/rllib.html	[406]
Bsuite	RL environments	https://github.com/deepmind/bsuite	[489]
OpenSpiel	RL in Games fwk	https://github.com/deepmind/open_spiel	[385]
Meta World	Metalearning Games	https://meta-world.github.io	[743]

Table A.2: Deep Reinforcement Learning Environments

with deep learning. PyTorch offers great flexibility. Tensorflow may well be the most popular environment. All environments offer seamless integration with CPU and GPU backends, no specialized GPU programming knowledge is needed.

Table A.1 summarizes the general learning environments.

A.2 Deep Reinforcement Learning Environments

Reinforcement learning has seen much interest from researchers. The availability of environments has stimulated this research greatly. Bellemare et al. introduced the Atari Learning Environment [55], that has subsequently been incorporated into OpenAI Gym. The Gym Github page can be found here.¹ OpenAI Gym has, in addition to ALE, classic reinforcement learning examples such as Cartpole and Mountain Car. OpenAI also provides baseline implementations of reinforcement learning algorithms [171]. A great introduction on their use is at Spinning up: OpenAIs Spinning Up, and the code repo, which is here.

Subsequently, refactored versions of baselines are presented as the Stable Baselines. Visit them at Stable Baselines. Note: Stable-Baselines supports Tensorflow versions from 1.8.0 to 1.14.0, and does not work on Tensorflow versions 2.0.0 and above. Support for Tensorflow 2 API is planned.

The baseline Zoo contains trained models for the stable baselines.

Dopamine [122] is a framework by DeepMind for deep reinforcement learning.

RLLib [406] provides abstractions for distributing reinforcement learning on large scale clusters of machines.

Bsuite [489] is a Behavior Suite for reinforcement learning.

OpenSpiel [385] is a framework for reinforcement learning in games, implementing many of the algorithms that are discussed in this book.

¹<https://github.com/openai/gym>

Name	Type	URL	reference
AlphaZero General	AlphaZero in Python	https://github.com/suragnair/alpha-zero-general	[657]
ELF	Game Framework	https://github.com/pytorch/ELF	[667]
Leela	AlphaZero for Chess, Go	https://github.com/LeelaChessZero/lczero	[499]
PhoenixGo	AlphaZero based Go program	https://github.com/Tencent/PhoenixGo	[746]

Table A.3: Self-Learning Environments

Meta World [743] is a benchmark and evaluation suite for multi-task and meta reinforcement learning.

Table A.2 summarizes the deep reinforcement learning environments.

A.3 Open Reimplementations of Alpha Zero

The publication of the AlphaGo papers created great interest in the research community. The DeepMind team has not yet published their code, but the publications [604, 605] (and especially [607]) provide enough detail for researchers to create their own versions, some of which are open sourced on Github. We will describe some of them.

A0G: AlphaZero General

Thakoor et al. [657] created a self-play system as part of a course project. The system is called AlphaZero General (A0G), and is on Github. It is implemented in Python and TensorFlow, Keras and PyTorch, and suitably scaled down for smaller computational resources. It has implementations for 6×6 Othello, Tic Tac Toe, Gobang, and Connect4, all small games of significantly less complexity than Go. Its main network architecture is a four layer CNN followed by two fully connected layers. The code is easy to understand in an afternoon of study, and is well suited for educational purposes. The course project write-up provides some documentation [657].

Facebook ELF

ELF stands for Extensible Lightweight Framework. It is a framework for game research in C++ and Python [667]. Originally developed for real time strategy games by Facebook, it includes the Arcade Learning Environment and the Darkforest² Go program [669]. ELF can be found at Github ELF. ELF also contains the program OpenGo [668], a reimplementation of AlphaGo Zero (in C++).

²<https://github.com/facebookresearch/darkforestGo>

Leela

Another reimplementation of AlphaZero is Leela. Both a Chess and a Go version of Leela exist. The Chess version is based on Chess engine Sjeng. The Go³ version is based on Go engine Leela. Leela does not come with trained weights of the network. Part of Leela is a community effort to compute these weights.

PhoenixGo

PhoenixGo is a strong Go program by Tencent [746]. It is based on the AlphaGo Zero architecture. It can be found at Github PhoenixGo. A trained network is available as well.

Table A.3 summarizes the self-learning environments.

³<https://github.com/gcp/leela-zero>

Appendix B

Running Python

Python is a modern programming language that is quite popular in AI. Many packages have Python interfaces, such as TensorFlow and Keras. This appendix shows how to start working with Python. Python was developed by Guido van Rossum while he was a researcher at the national research center for mathematics and computer science CWI in Amsterdam [421].

Before you install Python you should check if it is already available on your computer. Start a terminal, or a command prompt, and, at the prompt \$, type `python`. Now one of two things will have happened. If you see an error message, Python is not installed correctly on your computer. If you see a response of the form

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

then python is working already on your computer. You can get out of Python by typing `Ctrl-D` or `quit()`. Interactive help is available by typing `help()`. You can skip this appendix, and start playing around with the examples and exercises. How to install TensorFlow and Keras is discussed in the chapters (Section 6.5).

To install Python, go to the [python.org](https://www.python.org) website.¹ Got to Downloads, click on it, and you should see the latest version of python being offered for your current operating systems (Linux, macOS, or Windows). Click the button to start the download, and when the download is finished after a few minutes, launch the installer. While you are waiting, browse around on the website to the Documentation section. There are great reference works and tutorials on learning Python. Please consult your library or bookstore , or the Internet. Try Google or Stackoverflow.

One of the great advantages of Python is the wealth of high quality software packages that are available, especially for data science and artificial intelligence.

¹<https://www.python.org>

Among them are popular packages such as scikit-learn, numpy, matplotlib, Tensorflow and gym. These packages can be installed easily by typing `pip install numpy` for installing the numpy numerical package.

Please go ahead and explore!

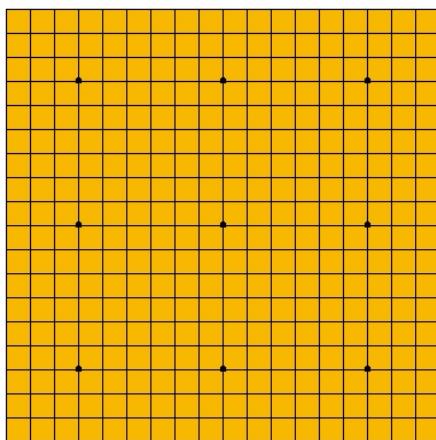
Appendix C

Tutorial for the Game of Go

This tutorial contains a brief introduction to the game of Go. It was taken, permission pending, from <https://www.kiseido.com/ff.htm>, Kiseido Publishing Company, Japan. Other tutorials can be found in your bookstore and on the Internet. As you are surfing, you may want to see if there is a Go club nearby. There is no better way to learn to play Go than from real people.

The Board

Go is usually played on a 19x19 grid, or board. Figure 1 shows an empty board. Notice the nine marked points. These points are usually referred to as the star points. They serve as reference points as well as markers on which the handicap stones are placed in handicap games.



Dia. 1

The Stones

The pieces used are black and white lens-shaped disks, called stones. Black starts out with 181 stones and White with 180. The total of 361 stones corresponds to the number of intersections on the standard 19x19 go board. The stones are usually kept in wooden bowls next to the board.

How Go is Played

At the beginning of the game, the board is empty. One player takes the black stones, the other player the white ones.

The player with the black stones, referred to as 'Black', makes the first move. The player with the white stones, referred to as 'White', makes the second move. Thereafter, they alternate making their moves.

A move is made by placing a stone on an intersection. A player can play on any unoccupied intersection he wants to.

A stone does not move after being played, unless it is captured and taken off the board.

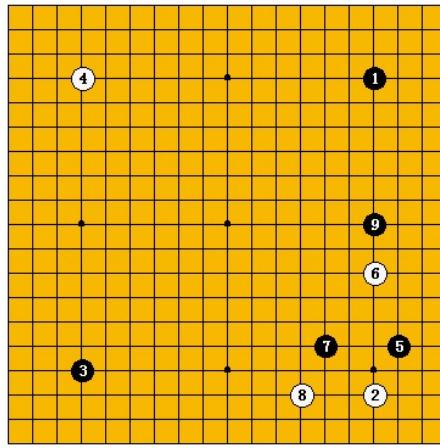


Figure 2 shows the beginning of a game. Black plays the first move in the upper right corner.

White plays 2 in the lower right corner.

Black plays 3 and White plays 4.

This is a typical opening where each player has staked out a position in two of the four corners.

Next Black approaches White 2 with 5 and White pincers 5 with 6.

Black escapes into the center with 7 and White stakes out a position in the bottom right with 8.

Next Black pincers the white stone at 6 with 9.

The Object of Go Is to Control Territory

The object of go is to control more territory than your opponent. At the end of the game, the player who controls the more territory wins the game.

We are going to show you how territory is formed in a game on a 9x9 board. Although go is usually played on a 19x19 board, it can also be played on a 9x9 board, or any size board from 5x5 up. Explaining the rules on a 9x9 board is convenient because the game is over quickly and the beginner can immediately grasp the flow of the game and how the score is counted. We also recommend that you play your first games on a 9x9 board and, when you have mastered the rules, start playing on the 19x19 board.

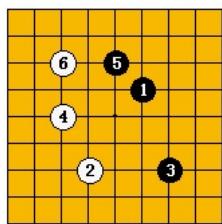


Figure 1

An Example Game

Figure 3, Black makes his first move on the 4-4 point, after which White makes his move.

Thereafter, both sides continue to alternate in making their moves. With White 6, the territories of both sides are beginning to take shape. Black has staked out the right side and White has laid claim to the left side.

Once you have mapped out your territory, there are two basic strategies to choose from.

One is to expand your own territory while reducing your opponent's territory. The other is to invade the territory your opponent has mapped out.

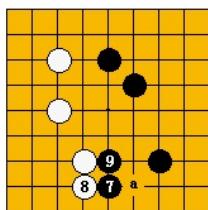


Figure 2

Black 7 in Figure 4 follows the first strategy: Black expands his territory on the lower right while preventing White from expanding his own with a move at White 'a'.

White must defend at 8 to block an incursion by Black into his territory on the left. Next, Black reinforces his territory on the right with 9.

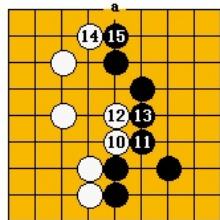


Figure 3

It is now White's turn to expand his territory. He does this by first expanding his center with 10 and 12 in Figure 5, then expanding his upper left territory with 14. Black must defend his top right territory with 15. The points around 'a' at the top and bottom must now be decided.

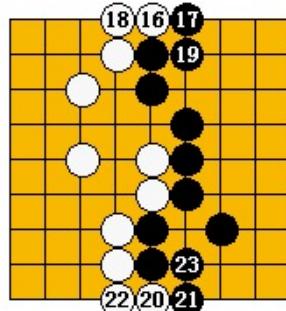


Figure 4

The moves from White 16 to Black 19 in Figure 4 are a common sequence. The same kind of sequence is next played at the bottom from White 20 to Black 23. By playing these moves, White is able to expand his territory while reducing Black's.

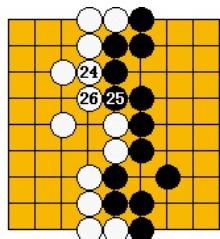


Figure 5

White 24 to White 26 in Figure 7 are the last moves of the game. It is now possible to determine the winner. In this case, counting the score is easy.

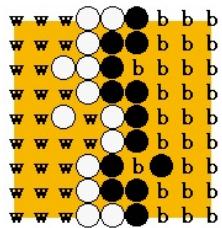


Figure 6

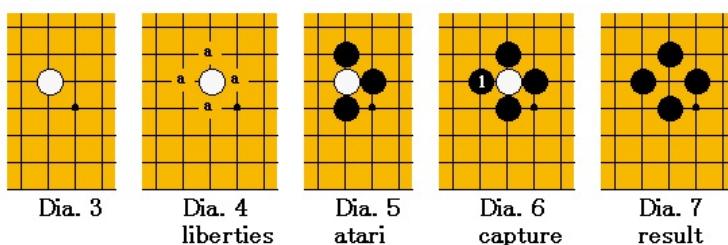
Black's territory here consists of all the vacant points he controls on the right side, while White's territory consists of all the vacant point he controls on the left. More precisely, Black's territory is all the points marked 'b' and White's territory is all the points marked 'w'. If you count these points, you will find that Black has 28 points, while White has 27. Therefore, Black wins by one point.

This was a very simple game and some of the rules did not arise. However, playing over this game will show you what Go is about.

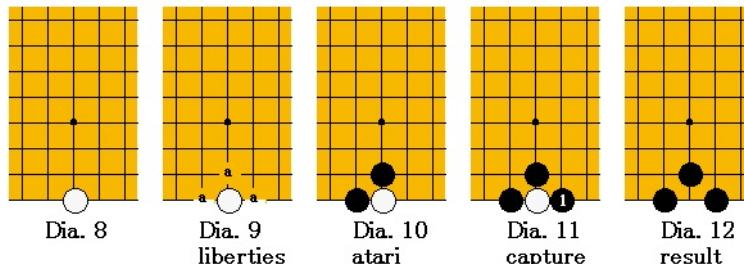
The Rule of Capture

An important rule of Go concerns the capturing of stones. We will first show you how stones are captured, then show how this occurs in a game.

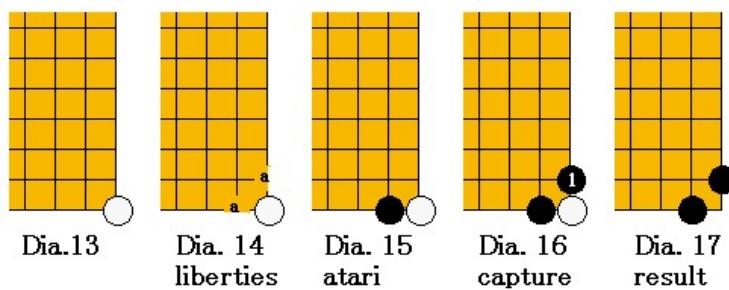
Liberties



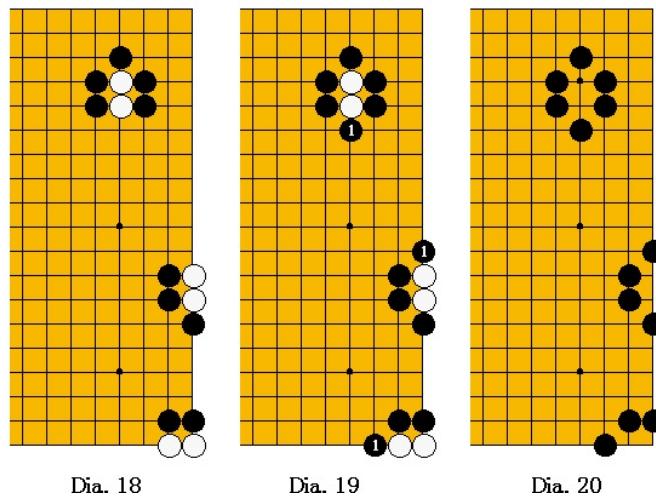
The lone white stone in Diagram 3 has four liberties. Namely, the four points 'a' in Diagram 4. If Black can occupy all four of these points, he captures the white stone. Suppose, for example, that Black occupies three of these liberties in Diagram 5. The white stone would be in Atari and Black would be able to capture it on his next move, that is with 1 in Diagram 6. Black would then remove the white stone from the board and put it in his prisoner pile. The result of this capture is shown in Diagram 7.



At the edge of the board a stone has only three liberties. The white stone in Diagram 8 is on the edge of the board; that is on the first line. Its three liberties are at 'a' in Diagram 9. If Black occupies two of these liberties, as in Diagram 10, the white stone would be in Atari. Black captures this stone with 1 in Diagram 11. The result of this capture is shown in Diagram 12.

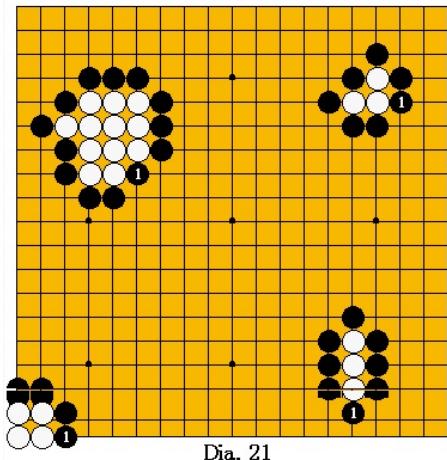


A stone in the corner has only two liberties. The white stone in Diagram 13 is on the 1-1 point. Its two liberties are at 'a' in Diagram 14. If Black occupies one of these points, as in Diagram 15, the white stone would be in Atari. Black captures this stone with 1 in Diagram 16. The result is shown in Diagram 17.



It is also possible to capture two or more stones if you occupy all their liberties. In Diagram 18, there are three positions in which two white stones are in

Atari. Black captures these stones with 1 in Diagram 19. The results are shown in Diagram 20.



Dia. 21

Any number of stones making up any kind of shape can be captured if all their liberties are occupied.

In Diagram 21, there are four different positions.

Black 1 captures twelve stones in the upper left, four stones in the lower left, three stones in the upper right and three stones in the lower right.

When you capture stones in a game, you put them in your prisoner pile.

Then, at the end of the game, these captured stones are placed inside your opponent's territory.

A Game

Let's look at a game to see how this actually works.

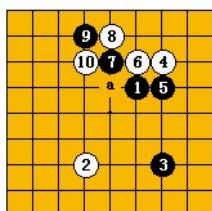


Figure 7

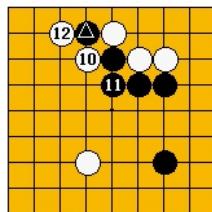


Figure 8

After Black plays 3 in Figure 7, White makes an invasion inside Black's sphere of influence with 4. White 10 Atari's the black stone at 7. This stone has only one liberty remaining at the point 'a'.

If Black doesn't play his next move at 'a', White will play on this point and capture the black stone at 7.

Therefore, black connects at 11 in Figure 8, but White Atari's again at 12. The marked stone cannot be rescued, so Black has to sacrifice it.

White's
prisoner: ●

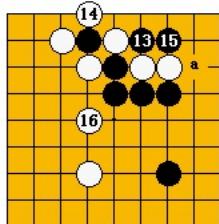


Figure 9
White's
prisoner: ●

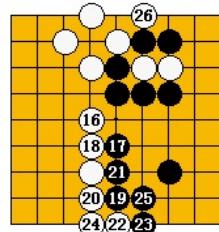


Figure 10

He plays his own Atari with 13 in Figure 9. White then captures with 14 and Black Atari's two white stones with 15. That is, he threatens to capture them by playing at 'a'.

With 16 in Figure 10, White maps out the territory on the left side, and Black expands his territory on the right side with 17 to 21.

The moves from White 22 to Black 24 are the same kind of endgame sequence we saw in Figure 4 of the first game.

White's
prisoner: ●

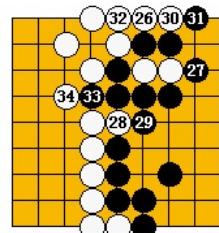


Figure 11

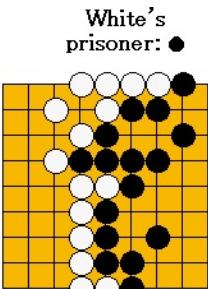


Figure 12

Black's
Prisoners ○○

White 26 forces Black to capture two white stones with 27. in Figure 11. Next, the moves at White 28 and 30 each reduce Black's territory by one point. Black 31 Atari's the two white stones at 26 and 30, so White must connect at 32 to save them.

Finally, Black 33 reduces White's territory on the left by one point. The game ends when White blocks at 34.

Figure 12 show what the board looks like at the end of this game.

White has one black stone in his prisoner pile, while Black has two white stones in his.

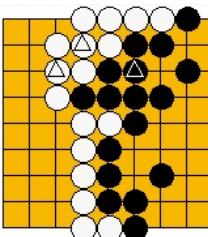


Figure 13

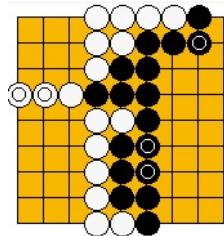


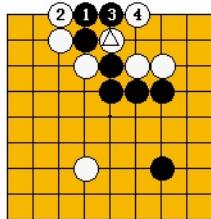
Figure 14

In Figure 13, each side places his prisoners in his opponent's territory. White places his one black prisoner (the marked black stone) inside Black's territory and Black places his two white prisoners (the two marked white stones) inside White's territory.

It is customary to rearrange the stones a bit to make the counting of territory simple and rapid. In Figure 14, the three marked black stones and the two marked

white stones were moved.

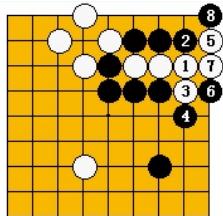
Calculation of the size of the territories can now be made at a glance. Black has 23 points; White has 24 points. White wins by one point.



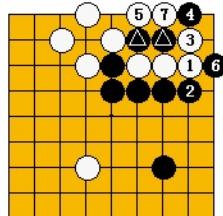
Dia. 22

Questions and Answers

After White 12 in Figure 8, why didn't Black try to escape with his marked stone?



Dia. 23

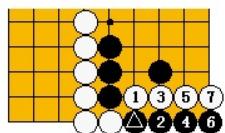


Dia. 24

Black could try to escape by playing 1 in Diagram 22, but White would pursue him and the black stones would still be in Atari. If Black persists with 3, he can Atari the marked white stone, but White captures three stones by taking Black's last liberty with 4.

After Black 15 in Figure 9, it might seem as if the two White stones in Atari could escape by extending to 'a'. Why doesn't White try this? The reason he doesn't try to escape is because he can't, unless Black blunders.

If White extends to 1 in Diagram 23, he increases his liberties to three but Black pursues him with 2 and, after 4, White is at the end of his rope: he has no way to increase his liberties. If White plays 5, Black Atari's with 6 and captures with 8.



Dia. 25

However, Black must not play 2 from the outside as in Diagram 24. White would then turn at 3 and now the two marked black stones have only two liberties, while the white group on the right has three liberties. White captures the two marked stones with 5 and 7.

Is Black 25 in Figure 10 necessary?

It certainly is. If Black omitted this move, White would Atari the marked black stone with 1 in Diagram 25. If Black tries to run away with 2 and 4, White pursues him with 3 and 5, forcing the black stones into the corner where they run out of liberties. White would then capture four black stones with 7.

These are most of the rules of go. There is one other rule: the ko rule, which prevents repetitive capture. The rule simply states: The previous board position cannot be recreated.

If you are a beginner who wishes to learn the game of Go, we recommend that you start with the book *Go: A Complete Introduction to the Game* by Cho Chikun, from which this brief introduction was taken.

Appendix D

Matches of Fan Hui, Lee Sedol and Ke Jie

This appendix contains the games of thee matches played by AlphaGo against Fan Hui, Lee Sedol, and Ke Jie.

Note that games can be played on line at DeepMind.¹ Also SGF files of the games are easily found online, allowing interactive replay.

D.1 Fan Hui

In October 2015 European Champion Fan Hui 2p played the following games against AlphaGo in London. AlphaGo won 5-0. See Figure D.1.

D.2 Lee Sedol

In March 2016 Lee Sedol 9p played the following games against AlphaGo in Seoul. AlphaGo won 4-1. See Figure D.2.

D.3 Ke Jie

In May 2017 Ke Jie 9p played the following games against AlphaGo in Wuzhen. AlphaGo won 3-0. See Figure D.3. AlphaGo played white in game 1 and black in game 2 and 3.

¹<https://deepmind.com/research/alphago/match-archive/alphago-games-english/>

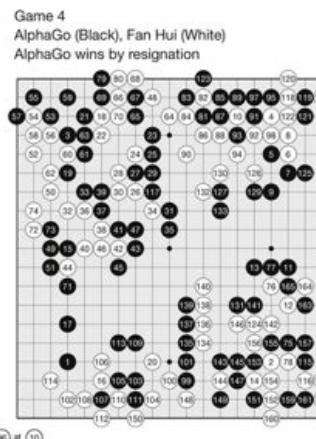
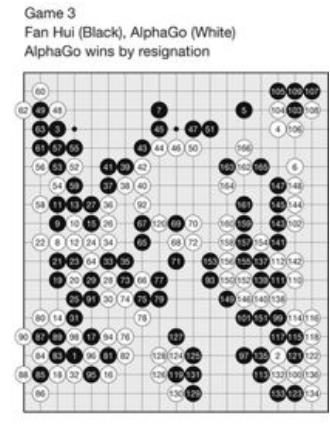
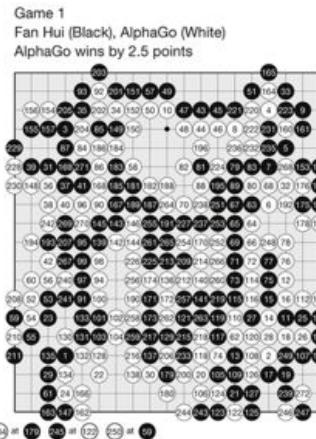


Figure D.1: Games Fan Hui vs. AlphaGo

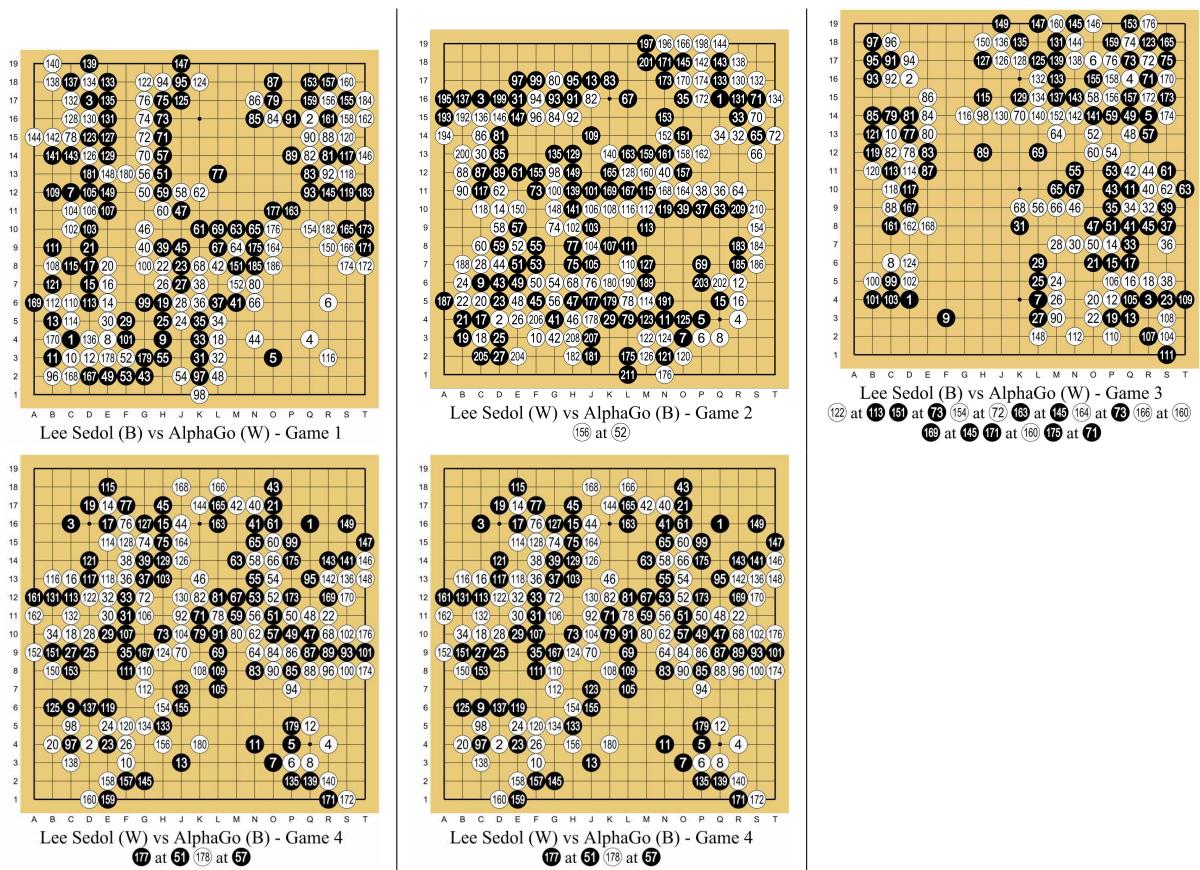


Figure D.2: Games Lee Sedol vs. AlphaGo

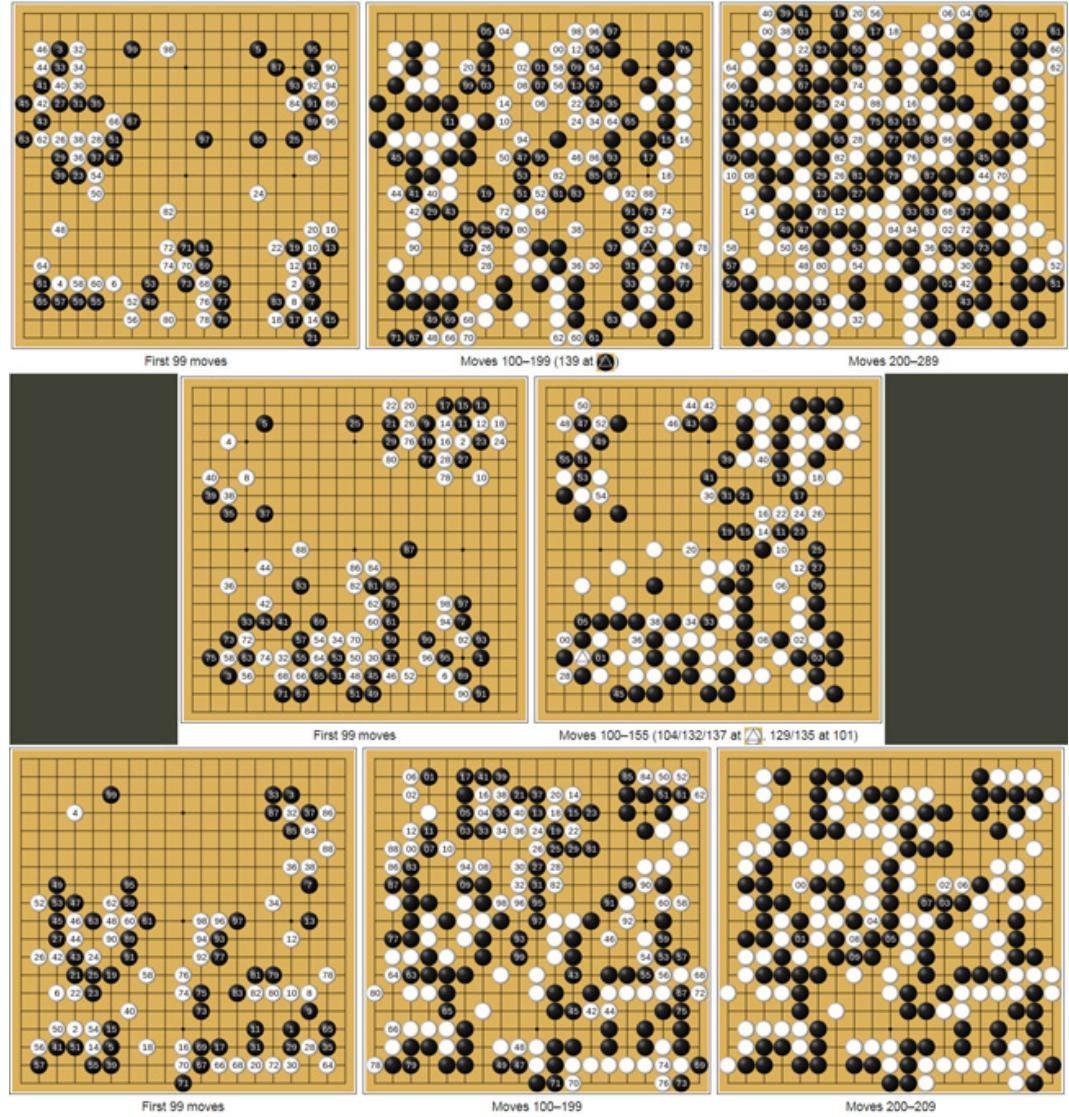


Figure D.3: Games Ke Jie vs. AlphaGo

Appendix E

*AlphaGo Technical Details

This appendix contains more technical details for the AlphaGo, AlphaGo Zero, and AlphaZero programs, that was too technical to fit in the main text of Chapter 7, but is important to truly understand how things work. All information in this appendix is from the three papers by Silver et al. [604, 607, 606].

E.1 AlphaGo

We start with AlphaGo. The AlphaGo program is a complicated program. All details are described in Silver et al. [604]. In this appendix we summarize some of the more technical details from this publication.

Search Algorithm

The search algorithm of AlphaGo is MCTS (see also Section 5.2 and [108]). The four MCTS operations are implemented as follows.

Selection For selection, a variant of P-UCT [544, 430] is used, which allows integration of prior information into the UCT selection formula. This is especially important for new, unexplored children, to be able to use the information of the policy net. With the policy net, AlphaGo does not employ the all-moves-as-first or rapid action value estimation heuristics used in most other programs, since the policy net provides priors of better quality. In addition AlphaGo does not use progressive widening [131], dynamic komi [38] or an opening book, which most other Go programs do.

The constant C_p determines the exploration/exploitation trade-off. A high value means more exploration. AlphaGo uses $C_p = 5$. This is high compared to most Go programs, where values of $C_p < 0.75$ or even $C_p \approx 0$ were used [234, 127]. AlphaGo favors a considerable amount of exploration compared to earlier Go programs. Section 5.3.1 discusses the performance sensitivity of C_p .

Expansion Children of leaves of the MCTS tree are expanded when the visit count exceeds a threshold. New children get prior probabilities from the policy

network, and are put in the queue for evaluation by the GPU with both the policy and the value network. The threshold is determined by the length of the work queue for the GPU.

Roll out Leaves in AlphaGo are evaluated both by the value network (GPU queue) and roll outs with the fast roll out network. Results are combined with a parameter $\lambda = 0.5$ (half roll out, half value network).

Backup The roll out statistics are updated as usual in MCTS (visit count and win count). AlphaGo is a tree parallel program [129], which means that multiple processes can update the MCTS tree concurrently. AlphaGo uses standard techniques for ensuring efficient parallel selection (virtual loss [596, 444]) and correctness of the tree (lock free updates [193, 446]).

Network Architecture

For the fast roll out policy network, the function approximator is actually based on a hash table. The fast roll out policy is based on small (3×3) patterns that are also found in other Go programs [234, 231]. It is based on a hash table of the *last good reply* heuristic [28].

The function approximator for the selection policy is based on a 13 layer convolutional neural network. It has 12 convolutional hidden layers (plus one input layer and one fully connected output layer). The first convolutional layers uses a kernel size of 5×5 , the other hidden layers use 3×3 . A ReLU unit is applied at each layer.

The network for the value network has one convolutional layer more than the policy network, for a total of 14 layers.

Input/Output Features

The input layer is $19 \times 19 \times 48$ with 48 feature planes. The inputs to the AlphaGo nets are more than just the stones on the board. The features of each point include stone color, how many times ago a turn was played, the number of liberties, how many opponent stones would be captured, how many own stones would be captured, the number of liberties after the move is played, ladder status, ladder escape, if a point fills an own eye, and color to play.

The output layer is a fully connected layer with a soft-max function for the policy network and a single tanh unit for the value network.

Compute Hardware

Training the networks occurs off line, before a match is played, and large amounts of computation power and time are available. During a match the time to compute is limited. Function approximation does not come cheap. Position evaluation requires evaluating multiple networks, which costs orders of magnitude more computation than traditional heuristic programs. Combining MCTS with deep neural networks has required quite some software engineering. AlphaGo uses an asynchronous multi-threaded search that executes simulations on CPUs,

and computes policy and value networks in parallel on GPUs. AlphaGo used 40 search threads, 48 CPU cores, and 8 GPUs. A distributed version was also developed, to make use of a cluster of machines. It has been run on 1,202 CPUs and 176 GPUs.

In 2016 Google developed a special kind of GPU especially optimized for tensor computations in TensorFlow, named *TPU*, for Tensor Processing Unit. DeepMind started using TPUs for AlphaGo in 2016. TPUs are designed for high throughput low precision computations (as low as 8 bit precision) whereas GPUs are typically optimized for 32 or 64 bit precision operations, and memory throughput is often a bottleneck [329].

E.2 AlphaGo Zero

A year later, AlphaGo Zero was introduced. The paper by Silver et al. [607] contains all details. We highlight some of the details here.

Detailed Self-Play Training

Let us look in more technical detail how self-play is implemented in AlphaGo Zero [607]. The neural network is trained as follows. A self-play reinforcement learning algorithm uses MCTS to play each move. First, the neural network is initialized to random weights θ_0 . At each subsequent iteration i , games of self-play, or episodes, are generated, consisting of a sequence of moves. For each move of each episode t , an MCTS search π_t is performed using the previous neural network θ_{i-1} and a move is played sampled from π_t . The game t terminates when both players pass at step T at which point the game is scored $z = \{-1, +1\}$. The data are stored in a state-policy-distribution-reward-triple (s_t, π_t, z_t) as examples in the example buffer for training the network parameters θ_i . The network $(p, v) = f_{\theta_i}(s)$ is trained by sampling uniformly over the example triples of all time steps (episodes) of the last iterations of self-play. A loss function is chosen to minimize the error between predicted value v and self-play outcome z , and to maximize the similarity of predicted move probabilities p and search probabilities π . Policy and value were used as a combined loss function that sums over the mean squared value error and cross entropy policy losses: $l = (z - v)^2 - \pi^T \log p + c\|\theta\|^2$ where c is the controlling parameter for L2 weight regularization to prevent overfitting [245, 607].

Network Architecture

The new function approximation architecture is simpler than in AlphaGo. It consists of one residual network (see Section 6.2.1) instead of a value net, a slow selection policy net, and a fast roll out policy net. The ResNet has one input and two outputs: a policy head and a value head. The ResNet is trained by self-play, the databases of positions for supervised and reinforcement learning have

disappeared. MCTS no longer does roll-outs, so the fast roll-out net has gone as well.

It is reported that combining policy and value into a single network slightly reduces move prediction accuracy, but also reduces the value error. It improved playing performance in AlphaGo Zero by 600 Elo points, in part because the dual objective regularizes the network better.

Figure E.1 shows the difference in performance between the combined dual policy/value network and the separate networks, and the convolutional and residual networks [607].

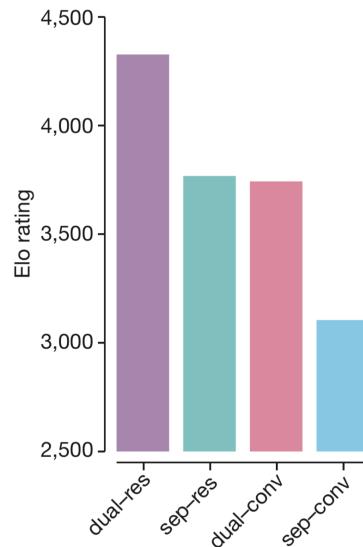


Figure E.1: Comparison of Dual and Separate, and of Convolutional and Residual Networks [607]

The ResNet consist of blocks of convolutions [275] with batch normalization [315] and ReLU units. Interestingly, after experimenting with 40 block ResNets against 20 block ResNets, the team decided to use the 20 block net in the tournament version of the player. The added complexity of 40 block nets did not increase performance enough to outweigh computational cost.

As before, neural network parameters are optimized by stochastic gradient descent with momentum and learning rate annealing. In a comparison against AlphaGo the resnet was more accurate, achieved lower error and improved performance in AlphaGo by over 600 Elo [607].

The input to the neural network is a $19 \times 19 \times 17$ image stack comprising 17 binary feature planes. The 17 feature planes contain the current board and a history of the 8 previous moves, both for black and for white, and the color to play. The input features are processed by a residual network tower. The tower consists of a single convolutional block followed by either 19 or 39 residual blocks, one

of which is shown in Figure E.2. The total network depth, in the 20 or 40 block network, is thus 39 or 79 parameterized layers, plus 2 layers for the policy head and 3 layers for the value head, consisting of a block ending in a fully connected layer. See the paper for details [607].

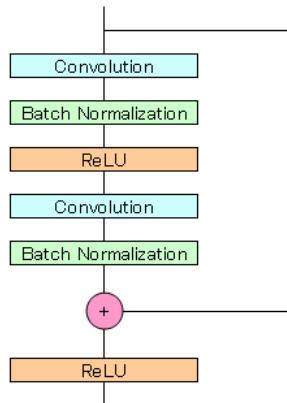


Figure E.2: AlphaGo Zero Residual Block [607]

Compute Hardware

Originally, each neural network was trained with TensorFlow, with 64 GPU workers and 19 CPU parameter servers. Later, TPUs were used. Figure E.3 shows the difference in total dissipated power consumption between the different versions of AlphaGo.

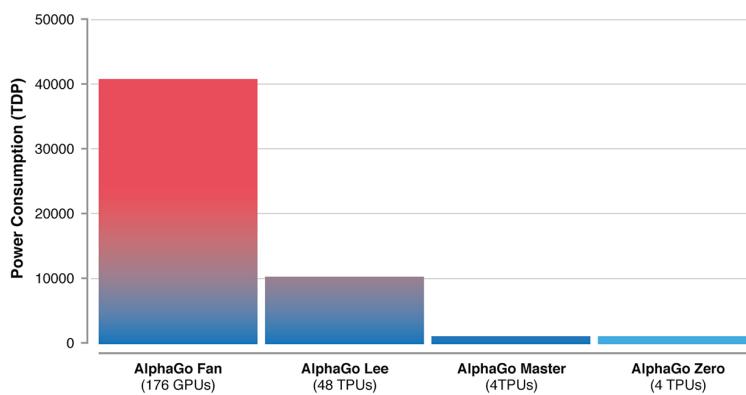


Figure E.3: Comparison of Power Consumption for Training [564]

Search Algorithm

The search algorithm of AlphaGo Zero is MCTS. For selection, the same P-UCT [544] variant was used, that incorporates priors from the policy head, and not RAVE. This MCTS did not use roll outs at the leaves of the MCTS tree, but value head lookups. In the self-play, a total of 25,000 games were played in each self-play iteration. MCTS performs 1600 simulations for each move. MCTS hyper-parameters were selected by Gaussian process optimization [599].

Input/Output Features

The input planes of AlphaGo Zero are simpler than in AlphaGo. Only the board information is used, no additional information such as ko, or liberties, is used. Even the basic heuristic of not playing moves that would fill the player's own eyes is excluded. AlphaGo Zero learns from scratch. No heuristic information is given to the program.

E.3 AlphaZero

Again a year later Silver et al. [606] published their AlphaZero paper, showing how the simpler architecture of AlphaGo Zero was also a more general architecture. This appendix describes some of the more technical elements.

Input/Output Features

The input and output layers are different for each game. For Go, the same $19 \times 19 \times 17$ planes as in AlphaGo Zero were used in AlphaZero, where 8 moves of history is encoded in addition to the stones on the board. Move encoding in Go is simply a coordinate of a stone.

For Chess, a similar encoding scheme is used, although more elaborate using many more planes. The scheme encodes all possible moves from a position in different binary planes. In Chess there is a greater variety of possible moves, and the encoding scheme therefore uses more planes than in Go, $8 \times 8 \times 73$.¹

Shogi has an even greater variety of moves, leading to a stack of $9 \times 9 \times 139$ planes as input. The paper mentions that experiments with different coding schemes worked as well, although training efficiency was affected somewhat.

Compute Hardware

Both Stockfish and Elmo ran with 44 threads on 44 cores (2 Intel Xeon Broadwell CPUs with 22 cores) with 32 GB transposition tables and 3 hour per match time controls with 15 seconds per move extra. AlphaZero ran on a single machine with 44 CPU cores, and 4 first-generation TPUs.

¹Silver et al. [606] contains the details of their elaborate encoding. The encoding scheme is quite complex, although it is noted that simpler encoding schemes also work.

Bibliography

- [1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 172, 293
- [2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. 172, 293
- [3] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y Ng. An application of reinforcement learning to aerobatic helicopter flight. In *Advances in neural information processing systems*, pages 1–8, 2007. 258, 265
- [4] Bruce Abramson. Expected-outcome: A general model of static evaluation. *IEEE transactions on pattern analysis and machine intelligence*, 12(2):182–193, 1990. 129
- [5] L Victor Allis. *Searching for solutions in games and artificial intelligence*. PhD thesis, Maastricht University, 1994. 86, 87, 111, 233
- [6] L Victor Allis, Maarten van der Meulen, and H Jaap Van Den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994. 111
- [7] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2009. 67, 365
- [8] Ingo Althöfer. Root evaluation errors: How they arise and propagate. *ICGA Journal*, 11(2-3):55–63, 1988. 113
- [9] Thomas Anantharaman, Murray S Campbell, and Feng-hsiung Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–109, 1990. 112
- [10] Takahisa Ando, Noriyuki Kobayashi, and Takao Uehara. Cooperation and competition of agents in the auction of computer bridge. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 86(12):76–86, 2003. 275
- [11] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016. 249
- [12] Anonymous. Go ai strength vs time. *Reddit post*, 2017. 224, 361
- [13] Thomas Anthony, Robert Nishihara, Philipp Moritz, Tim Salimans, and John Schulman. Policy gradient search: Online planning and expert iteration without search trees. *arXiv preprint arXiv:1904.03646*, 2019. 259

- [14] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370, 2017. 240, 242, 244
- [15] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your MAML. *arXiv preprint arXiv:1810.09502*, 2018. 250
- [16] Grigoris Antoniou and Frank Van Harmelen. *A semantic web primer*. 2004. 30, 281
- [17] Oleg Arenz. Monte carlo chess. Master’s thesis, Universität Darmstadt, 2012. 264
- [18] Andreas Argyriou, Theodoros Evgeniou, and Massimiliano Pontil. Multi-task feature learning. In *Advances in neural information processing systems*, pages 41–48, 2007. 249
- [19] Broderick Arneson, Ryan B Hayward, and Philip Henderson. Monte Carlo tree search in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, 2010. 147
- [20] John Asmuth, Lihong Li, Michael L Littman, Ali Nouri, and David Wingate. A Bayesian sampling approach to exploration in reinforcement learning. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 19–26. AUAI Press, 2009. 258
- [21] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002. 66, 136, 149
- [22] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002. 130, 136, 137
- [23] Peter Auer and Ronald Ortner. UCB revisited: Improved regret bounds for the stochastic multi-armed bandit problem. *Periodica Mathematica Hungarica*, 61(1-2):55–65, 2010. 136
- [24] Tim Baarslag, Katsuhide Fujita, Enrico H Gerding, Koen Hindriks, Takayuki Ito, Nicholas R Jennings, Catholijn Jonker, Sarit Kraus, Raz Lin, Valentin Robu, et al. Evaluating practical negotiating agents: Results and analysis of the 2011 international competition. *Artificial Intelligence*, 198:73–103, 2013. 279, 288
- [25] Thomas Bäck. Evolutionary algorithms in theory and practice: evolutionary strategies, evolutionary programming, genetic algorithms, 1996. 253
- [26] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1):1–23, 1993. 30, 252, 265
- [27] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017. 257
- [28] Hendrik Baier and Peter D Drake. The power of forgetting: Improving the last-good-reply policy in Monte Carlo Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):303–309, 2010. 316
- [29] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016. 253
- [30] Bram Bakker and Jürgen Schmidhuber. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proc. of the 8-th Conf. on Intelligent Autonomous Systems*, pages 438–445, 2004. 257
- [31] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308, 2014. 279
- [32] Bruce W Ballard. The*-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350, 1983. 111
- [33] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *arXiv preprint arXiv:1710.03748*, 2017. 274
- [34] Nolan Bard, John Hawkin, Jonathan Rubin, and Martin Zinkevich. The annual computer poker competition. *AI Magazine*, 34(2):112, 2013. 275
- [35] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1-2):41–77, 2003. 255, 257

- [36] Thomas Bartz-Beielstein, Marco Chiarandini, Luís Paquete, and Mike Preuss. *Experimental methods for the analysis of optimization algorithms*. Springer, 2010. 80, 122
- [37] OpenAI Baselines. <https://openai.com/blog/openai-baselines-dqn/>, 2017. 190, 193
- [38] Petr Baudiš. Balancing mcts by dynamically adjusting the komi value. *ICGA Journal*, 34(3):131–139, 2011. 315
- [39] Petr Baudiš and Jean-loup Gailly. Pachi: State of the art open source go program. In *Advances in computer games*, pages 24–38. Springer, 2011. 130
- [40] Seth Baum. A survey of artificial general intelligence projects for ethics, risk, and policy. 2017. 28
- [41] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Knightcap: a chess program that learns by combining TD (lambda) with game-tree search. *arXiv preprint cs/9901002*, 1999. 116, 264
- [42] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000. 264
- [43] D Beal. Experiments with the null move. *Advances in Computer Chess*, 5:65–79, 1989. 113
- [44] Don F Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, 1990. 113, 117
- [45] Donald F Beal and Martin C Smith. Learning piece-square values using temporal differences. *ICGA Journal*, 22(4):223–235, 1999. 115
- [46] Donald F. Beal and Martin C. Smith. Temporal difference learning for heuristic search and game playing. *Information Sciences*, 122(1):3–21, 2000. 264
- [47] Donald Francis Beal. Recent progress in understanding minimax search. In *Proceedings of the 1983 annual conference on Computers: Extending the human resource*, pages 164–169. ACM, 1983. 103, 125
- [48] Mark F Bear, Barry W Connors, and Michael A Paradiso. *Neuroscience*, volume 2. Lippincott Williams & Wilkins, 2007. 155
- [49] Harkirat Singh Behl, Atılım Güneş Baydin, and Philip HS Torr. Alpha MAML: Adaptive Model-Agnostic Meta-Learning. *arXiv preprint arXiv:1905.07435*, 2019. 250
- [50] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine learning and the bias-variance trade-off. *arXiv preprint arXiv:1812.11118*, 2018. 198
- [51] Mikhail Belkin, Daniel Hsu, and Ji Xu. Two models of double descent for weak features. *arXiv preprint arXiv:1903.07571*, 2019. 198
- [52] Mikhail Belkin, Daniel J Hsu, and Partha Mitra. Overfitting or perfect fitting? risk bounds for classification and regression rules that interpolate. In *Advances in Neural Information Processing Systems*, pages 2300–2311, 2018. 198
- [53] Marc Bellemare, Joel Veness, and Michael Bowling. Bayesian learning of recursively factored environments. In *International Conference on Machine Learning*, pages 1211–1219, 2013. 258
- [54] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *arXiv preprint arXiv:1707.06887*, 2017. 196, 254
- [55] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013. 75, 188, 189, 197, 201, 203, 236, 294
- [56] Richard Bellman. *Dynamic programming*. Courier Corporation, 1957, 2013. 62
- [57] Richard Bellman. On the application of dynamic programing to the determination of optimal play in chess and checkers. *Proceedings of the National Academy of Sciences*, 53(2):244–247, 1965. 116
- [58] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016. 274

- [59] Yoshua Bengio. *Learning Deep Architectures for AI*. Now Publishers Inc, 2009. 159
- [60] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012. 247
- [61] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013. 159, 258, 265
- [62] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007. 158, 161
- [63] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009. 158, 161, 230, 239, 246, 247, 265, 287
- [64] David B Benson. Life in the game of go. *Information Sciences*, 10(1):17–29, 1976. 264
- [65] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. Theano: Deep learning on GPUs with Python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3, pages 1–48. Citeseer, 2011. 172, 293
- [66] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012. 252
- [67] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011. 252, 265
- [68] Elwyn Berlekamp and David Wolfe. *Mathematical Go: Chilling gets the last point*. AK Peters/CRC Press, 1994. 264
- [69] Hans J Berliner. Experiences in evaluation with BKG-a program that plays backgammon. In *IJCAI*, pages 428–433, 1977. 40
- [70] Hans J Berliner. Backgammon computer program beats world champion. *Artificial Intelligence*, 14(2):205–220, 1980. 40
- [71] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001. 30
- [72] R Bertolami, H Bunke, S Fernandez, A Graves, M Liwicki, and J Schmidhuber. A novel connectionist system for improved unconstrained handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5), 2009. 178
- [73] Dimitri P Bertsekas. Rollout algorithms for discrete optimization: A survey. In *Handbook of combinatorial optimization*, pages 2989–3013. Springer, 2013. 145
- [74] Dimitri P Bertsekas and John Tsitsiklis. *Neuro-dynamic programming*. MIT press Cambridge, 1996. 81
- [75] Shalabh Bhatnagar, Doina Precup, David Silver, Richard S Sutton, Hamid R Maei, and Csaba Szepesvári. Convergent temporal-difference learning with arbitrary smooth function approximation. In *Advances in Neural Information Processing Systems*, pages 1204–1212, 2009. 208
- [76] Wolfgang Bibel. *Automated theorem proving*. Springer Science & Business Media, 2013. 282
- [77] Darse Billings, Neil Burch, Aaron Davidson, Robert Holte, Jonathan Schaeffer, Terence Schauenberg, and Duane Szafron. Approximating game-theoretic optimal strategies for full-scale poker. In *IJCAI*, volume 3, page 661, 2003. 40
- [78] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, 2002. 275
- [79] Darse Billings, Aaron Davidson, Terence Schauenberg, Neil Burch, Michael Bowling, Robert Holte, Jonathan Schaeffer, and Duane Szafron. Game-tree search with adaptation in stochastic imperfect-information games. In *International Conference on Computers and Games*, pages 21–34. Springer, 2004. 275

- [80] Christopher M Bishop. *Pattern recognition and machine learning (information science and statistics)* springer-verlag new york. 2006. 31, 72, 152, 153, 168, 208
- [81] Christopher M Bishop et al. *Neural networks for pattern recognition*. Oxford university press, 1995. 158
- [82] Yngvi Björnsson and Hilmar Finnsson. Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009. 278
- [83] Yngvi Björnsson and Tony A Marsland. Multi-cut $\alpha\beta$ -pruning in game-tree search. *Theoretical Computer Science*, 252(1-2):177–196, 2001. 113
- [84] Eric Bonabeau, Marco Dorigo, Guy Theraulaz, et al. *Swarm intelligence: from natural to artificial systems*. Number 1. Oxford university press, 1999. 30
- [85] Édouard Bonnet, Florian Jamain, and Abdallah Saffidine. On the complexity of trick-taking card games. In *IJCAI*, pages 482–488, 2013. 86
- [86] Mark Boon. Overzicht van de ontwikkeling van een Go spelend programma. 1991. 128
- [87] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992. 207
- [88] Tibor Bosse, Catholijn M Jonker, and Jan Treur. Formalisation of Damasio's theory of emotion, feeling and core consciousness. *Consciousness and cognition*, 17(1):94–113, 2008. 282
- [89] Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005. 256
- [90] Adi Botea, Martin Müller, Jonathan Schaeffer, et al. Fast planning with iterative macros. In *IJCAI*, pages 1828–1833, 2007. 256
- [91] Bruno Bouzy and Bernard Helmstetter. Monte Carlo go developments. In *Advances in computer games*, pages 159–174. Springer, 2004. 130, 133
- [92] Bruno Bouzy, Marc Métivier, and Damien Pellier. MCTS experiments on the voronoi game. In *Advances in Computer Games*, pages 96–107. Springer, 2011. 140
- [93] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, 2015. 54, 264, 275
- [94] Michael Bowling, Nicholas Abou Risk, Nolan Bard, Darse Billings, Neil Burch, Joshua Davidson, John Hawkin, Robert Holte, Michael Johanson, Morgan Kan, et al. A demonstration of the polaris poker system. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 1391–1392. International Foundation for Autonomous Agents and Multiagent Systems, 2009. 54, 275
- [95] Ivan Bratko. *Prolog programming for artificial intelligence*. Pearson education, 2001. 281
- [96] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010. 258
- [97] Mark G Brockington. Keyano unplugged—the construction of an Othello program. Technical report, Technical Report TR 97-05, Department of Computing Science, University of Alberta, 1997. 114
- [98] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym. *arXiv preprint arXiv:1606.01540*, 2016. 197, 203, 236, 294
- [99] Joost Broekens. Emotion and reinforcement: affective facial expressions facilitate robot learning. In *Artifical intelligence for human computing*, pages 113–132. Springer, 2007. 282
- [100] Joost Broekens, Marcel Heerink, Henk Rosendal, et al. Assistive social robots in elderly care: a review. *Gerontechnology*, 8(2):94–103, 2009. 279
- [101] Rodney A Brooks. Intelligence without representation. *Artificial intelligence*, 47(1-3):139–159, 1991. 30

- [102] Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization. *arXiv preprint arXiv:1811.00164*, 2018. 276
- [103] Noam Brown and Tuomas Sandholm. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018. 40, 54, 275
- [104] Noam Brown and Tuomas Sandholm. Superhuman AI for multiplayer poker. *Science*, page eaay2400, 2019. 40, 54, 276
- [105] Noam Brown, Tuomas Sandholm, and Brandon Amos. Depth-limited solving for imperfect-information games. *arXiv preprint arXiv:1805.08195*, 2018. 54
- [106] Cameron Browne. Hex strategy. *AK Peters, Wellesley MA*, 2000. 148
- [107] Cameron Browne, Dennis JNJ Soemers, and Eric Piette. Strategic features for general games. In *KEG@ AAAI*, pages 70–75, 2019. 260
- [108] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012. 127, 130, 133, 134, 143, 147, 148, 149, 315, 365
- [109] Bernd Brügmann. Monte Carlo go. Technical report, Syracuse University, 1993. 129, 130, 133, 143
- [110] Bruno Buchberger, George E Collins, Rüdiger Loos, and Rudolph Albrecht. Computer algebra symbolic and algebraic computation. *ACM SIGSAM Bulletin*, 16(4):5–5, 1982. 30
- [111] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ennder Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013. 252
- [112] Michael Buro. Logistello: A strong learning Othello program. In *19th Annual Conference Gesellschaft für Klassifikation eV*, volume 2. Citeseer, 1995. 38, 51, 242, 264
- [113] Michael Buro. Statistical feature combination for the evaluation of game positions. *Journal of Artificial Intelligence Research*, 3:373–382, 1995. 92, 242
- [114] Michael Buro. Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. *Games in AI Research*, pages 77–96, 1997. 109, 113, 114, 125, 264
- [115] Michael Buro. The Othello match of the year: Takeshi Murakami vs. Logistello. *ICGA Journal*, 20(3):189–193, 1997. 51
- [116] Michael Buro. The evolution of strong Othello programs. In *Entertainment Computing*, pages 81–88. Springer, 2003. 51
- [117] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, And Cybernetics-Part C: Applications and Reviews*, 38 (2), 2008, 2008. 258
- [118] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep Blue. *Artificial intelligence*, 134(1-2):57–83, 2002. 49, 116, 242, 243
- [119] Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997. 228, 249
- [120] Rich Caruana, Steve Lawrence, and C Lee Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in neural information processing systems*, pages 402–408, 2001. 168
- [121] Cristiano Castelfranchi, Frank Dignum, Catholijn M Jonker, and Jan Treur. Deliberative normative agents: Principles and architecture. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 364–378. Springer, 1999. 279
- [122] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G Bellemare. Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110*, 2018. 294
- [123] Tristan Cazenave and Bernard Helmstetter. Combining tactical search and Monte-Carlo in the game of go. *CIG*, 5:171–175, 2005. 130

- [124] Gregory Chaitin. The limits of reason. *Scientific American*, 294(3):74–81, 2006. 282
- [125] Hyeong Soo Chang, Michael C Fu, Jiaqiao Hu, and Steven I Marcus. An adaptive sampling algorithm for solving Markov decision processes. *Operations Research*, 53(1):126–139, 2005. 130, 137
- [126] Henry Charlesworth. Application of self-play reinforcement learning to a four-player game of imperfect information. *arXiv preprint arXiv:1808.10442*, 2018. 258
- [127] Guillaume Chaslot. *Monte-Carlo tree search*. PhD thesis, Maastricht University, 2010. 130, 133, 143, 147, 315
- [128] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo tree search: A new framework for game AI. In *AIIDE*, 2008. 130, 139, 147
- [129] Guillaume Chaslot, Mark HM Winands, and H Jaap van den Herik. Parallel Monte-Carlo tree search. In *International Conference on Computers and Games*, pages 60–71. Springer, 2008. 144, 149, 316
- [130] Guillaume MJB Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, JWHM Uiterwijk, and H Jaap Van Den Herik. Monte-Carlo strategies for computer go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006. 132, 227, 360, 361
- [131] Guillaume MJB Chaslot, Mark HM Winands, H Jaap van den Herik, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008. 315
- [132] Kumar Chellapilla and David B Fogel. Evolving neural networks to play checkers without relying on expert knowledge. *IEEE transactions on neural networks*, 10(6):1382–1391, 1999. 187, 206, 264
- [133] Ken Chen and Zhixing Chen. Static analysis of life and death in the game of go. *Information Sciences*, 121(1-2):113–134, 1999. 128
- [134] Xinlei Chen, Hao Fang, Tsung-Yi Lin, Ramakrishna Vedantam, Saurabh Gupta, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO captions: Data collection and evaluation server. *arXiv preprint arXiv:1504.00325*, 2015. 179, 181, 361
- [135] Ping-Chung Chi and Dana S Nau. Comparison of the minimax and product back-up rules in a variety of games. In *Search in artificial intelligence*, pages 450–471. Springer, 1988. 111
- [136] Jaeyoung Choi, Jack J Dongarra, and David W Walker. PB-BLAS: a set of parallel block basic linear algebra subprograms. *Concurrency: Practice and Experience*, 8(7):517–535, 1996. 172
- [137] Francois Chollet. *Deep learning with python*. Manning Publications Co., 2017. 172, 293
- [138] François Chollet et al. Keras, 2015. 293
- [139] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. Back to basics: Benchmarking canonical evolution strategies for playing atari. *arXiv preprint arXiv:1802.08842*, 2018. 253
- [140] Dan Cireşan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *arXiv preprint arXiv:1202.2745*, 2012. 165
- [141] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010. 172
- [142] Christopher Clark and Amos Storkey. Teaching deep convolutional neural networks to play go. arxiv preprint. *arXiv preprint arXiv:1412.3409*, 1, 2014. 187, 206, 264
- [143] Christopher Clark and Amos Storkey. Training deep convolutional neural networks to play go. In *International Conference on Machine Learning*, pages 1766–1774, 2015. 206, 242, 264
- [144] Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. *AAAI/IAAI*, 1998:746–752, 1998. 258
- [145] William F Clocksin and Christopher S Mellish. *Programming in Prolog: Using the ISO standard*. Springer Science & Business Media, 2012. 30, 281

- [146] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*, 2018. 197
- [147] Simon Colton. The painting fool: Stories from building an automated painter. In *Computers and creativity*, pages 3–38. Springer, 2012. 282
- [148] Kevin Coplan. A special-purpose machine for an improved search algorithm for deep chess combinations. In *Advances in Computer Chess*, pages 25–43. Elsevier, 1982. 109
- [149] Rémi Coulom. Go ratings. <https://www.goratings.org/en/history/>. 216
- [150] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006. 127, 130, 149
- [151] Rémi Coulom. Monte-Carlo tree search in Crazy Stone. In *Proc. Game Prog. Workshop, Tokyo, Japan*, pages 74–75, 2007. 130, 133
- [152] Rémi Coulom. The Monte-Carlo revolution in go. In *The Japanese-French Frontiers of Science Symposium (JFFoS 2008), Roscoff, France*, 2009. 130
- [153] Joseph C Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998. 117, 133
- [154] Ken Currie and Austin Tate. O-plan: the open planning architecture. *Artificial intelligence*, 52(1):49–86, 1991. 255
- [155] George Cybenko. Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:183–192, 1989. 158, 159
- [156] Shubhomoy Das, Weng-Keen Wong, Thomas Dietterich, Alan Fern, and Andrew Emmott. Incorporating expert feedback into active anomaly discovery. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 853–858. IEEE, 2016. 247
- [157] Omid E David, Nathan S Netanyahu, and Lior Wolf. Deepchess: End-to-end deep neural network for automatic learning in chess. In *International Conference on Artificial Neural Networks*, pages 88–96. Springer, 2016. 116, 206, 242, 264
- [158] Omid E David, H Jaap van den Herik, Moshe Koppel, and Nathan S Netanyahu. Genetic algorithms for evolving computer chess programs. *IEEE transactions on evolutionary computation*, 18(5):779–789, 2014. 92
- [159] Omid David-Tabibi, Moshe Koppel, and Nathan S Netanyahu. Genetic algorithms for mentor-assisted evaluation function optimization. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1469–1476. ACM, 2008. 116
- [160] Morton D Davis. *Game theory: a nontechnical introduction*. Courier Corporation, 2012. 33
- [161] Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. In *Advances in neural information processing systems*, pages 271–278, 1993. 255, 361
- [162] Arie De Bruin, Wim Pijls, and Aske Plaat. Solution trees as a basis for game-tree search. *ICCA Journal*, 17(4):207–219, 1994. 95
- [163] Luis M De Campos, Juan M Fernandez-Luna, José A Gámez, and José M Puerta. Ant colony optimization for learning Bayesian networks. *International Journal of Approximate Reasoning*, 31(3):291–311, 2002. 258
- [164] Jeffrey De Fauw, Joseph R Ledsam, Bernardino Romera-Paredes, Stanislav Nikolov, Nenad Tomasev, Sam Blackwell, Harry Askham, Xavier Glorot, Brendan O’Donoghue, Daniel Vissentini, et al. Clinically applicable deep learning for diagnosis and referral in retinal disease. *Nature medicine*, 24(9):1342, 2018. 279
- [165] Dave de Jonge, Tim Baarslag, Reyhan Aydogan, Catholijn Jonker, Katsuhide Fujita, and Takayuki Ito. The challenge of negotiation in the game of Diplomacy. 33, 275
- [166] Dave De Jonge and Jordi González Sabaté. Negotiations over large agreement spaces. 2015. 275
- [167] Dave De Jonge and Carles Sierra. D-brane: a Diplomacy playing agent for automated negotiations research. *Applied Intelligence*, 47(1):158–177, 2017. 275

- [168] Dave De Jonge and Dongmo Zhang. Automated negotiations for general game playing. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 371–379. International Foundation for Autonomous Agents and Multiagent Systems, 2017. 275
- [169] Rina Dechter. *Learning while searching in constraint-satisfaction problems*. University of California, Computer Science Department, 1986. 170
- [170] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. ieee, 2009. 169, 170, 173
- [171] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI baselines. <https://github.com/openai/baselines>, 2017. 201, 202, 294, 365
- [172] Philip K Dick. Do androids dream of electric sheep? 1968. *New York: Del Rey*, 1996. 209
- [173] Thomas G Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000. 75, 257
- [174] Thang Doan, João Monteiro, Isabela Albuquerque, Bogdan Mazoure, Audrey Durand, Joelle Pineau, and R Devon Hjelm. On-line adaptative curriculum learning for GANs. 2019. 248
- [175] Samuel Dodge and Lina Karam. A study and comparison of human and deep learning recognition performance under visual distortions. In *Computer Communication and Networks (ICCCN), 2017 26th International Conference on*, pages 1–7. IEEE, 2017. 176
- [176] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2625–2634, 2015. 180
- [177] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):1–17, 1988. 172
- [178] Christian Donninger. Null move and deep search. *ICGA Journal*, 16(3):137–143, 1993. 109, 114, 125
- [179] Derek Doran, Sarah Schulz, and Tarek R Besold. What does explainable ai really mean? a new conceptualization of perspectives. *arXiv preprint arXiv:1710.00794*, 2017. 260
- [180] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66, 1997. 30
- [181] Kenji Doya. Reinforcement learning in continuous time and space. *Neural computation*, 12(1):219–245, 2000. 258
- [182] Anders Drachen, Matthew Yancey, John Maguire, Derrek Chu, Iris Yuhui Wang, Tobias Mahlmann, Matthias Schubert, and Diego Klabajan. Skill-based differences in spatio-temporal team behaviour in defence of the ancients 2 (dota 2). In *2014 IEEE Games Media Entertainment*, pages 1–8. IEEE, 2014. 274
- [183] Sacha Drost and Johannes Fürnkranz. Learning of piece values for chess variants. Technical report, Tech. Rep. TUD-KE-2008-07, Knowledge Engineering Group, TU Darmstadt, 2008. 115
- [184] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. RL²: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016. 250
- [185] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011. 159
- [186] Richard D Duke and Jac Geurts. *Policy games for strategic management*. Rozenberg Publishers, 2004. 33

- [187] Daniel James Edwards and TP Hart. The alpha-beta heuristic. 1961. 114
- [188] Jeffrey L Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48(1):71–99, 1993. 247, 286
- [189] Arpad E Elo. *The rating of chessplayers, past and present*. Arco Pub., 1978. 29
- [190] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018. 253
- [191] Herbert D Enderton. The Golem go program. 1991. 243
- [192] Markus Enzenberger. The integration of a priori knowledge into a go playing neural network. 1996. 206, 207, 242, 243
- [193] Markus Enzenberger and Martin Müller. A lock-free multithreaded Monte-Carlo tree search algorithm. In *Advances in Computer Games*, pages 14–20. Springer, 2009. 316
- [194] Markus Enzenberger, Martin Müller, Broderick Arneson, and Richard Segal. Fuego—an open-source framework for board games and go engine based on Monte Carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010. 130
- [195] Susan L Epstein. Toward a theory for well-guided search. In *Proceedings of the AAAI Fall Symposium on Games: Planning and Learning*, 1993. 251
- [196] Susan L Epstein. Toward an ideal trainer. *Machine Learning*, 15(3):251–277, 1994. 251
- [197] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010. 247
- [198] Dumitru Erhan, Pierre-Antoine Manzagol, Yoshua Bengio, Samy Bengio, and Pascal Vincent. The difficulty of training deep architectures and the effect of unsupervised pre-training. In *Artificial Intelligence and Statistics*, pages 153–160, 2009. 246
- [199] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(Apr):503–556, 2005. 208
- [200] Theodoros Evgeniou and Massimiliano Pontil. Regularized multi-task learning. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 109–117. ACM, 2004. 249
- [201] Haw-ren Fang, Tsan-sheng Hsu, and Shun-chin Hsu. Construction of chinese chess endgame databases by retrograde analysis. In *International Conference on Computers and Games*, pages 96–114. Springer, 2000. 117
- [202] Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and SA Whiteson. Treeqn and atreec: Differentiable tree planning for deep reinforcement learning. International Conference on Learning Representations, 2018. 260
- [203] Li Fei-Fei, Jia Deng, and Kai Li. Imagenet: Constructing a large-scale image database. *Journal of Vision*, 9(8):1037–1037, 2009. 169
- [204] Alan Fern and Paul Lewis. Ensemble Monte-Carlo planning: An empirical study. In *ICAPS*, 2011. 144, 149
- [205] Santiago Fernández, Alex Graves, and Jürgen Schmidhuber. An application of recurrent neural networks to discriminative keyword spotting. In *International Conference on Artificial Neural Networks*, pages 220–229. Springer, 2007. 178
- [206] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015. 253
- [207] Rebecca Anne Fiebrink. *Real-time human interaction with supervised learning algorithms for music composition and performance*. Citeseer, 2011. 282
- [208] Richard E Fikes, Peter E Hart, and Nils J Nilsson. Learning and executing generalized robot plans. *Artificial intelligence*, 3:251–288, 1972. 255

- [209] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971. 30, 255
- [210] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017. 250, 287
- [211] Chelsea Finn, Kelvin Xu, and Sergey Levine. Probabilistic Model-Agnostic Meta-Learning. In *Advances in Neural Information Processing Systems*, pages 9516–9527, 2018. 250
- [212] Hilmar Finnsson and Yngvi Björnsson. Learning simulation control in general game-playing agents. In *AAAI*, volume 10, pages 954–959, 2010. 278
- [213] John P Fishburn. Analysis of speedup in distributed algorithms. 1982. 106
- [214] Yannis Flet-Berliac. The promise of hierarchical reinforcement learning. <https://thegradient.pub/the-promise-of-hierarchical-reinforcement-learning/>, March 2019. 255
- [215] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1702.08887*, 2017. 229
- [216] David B Fogel. An introduction to simulated evolutionary optimization. *IEEE transactions on neural networks*, 5(1):3–14, 1994. 30
- [217] David B Fogel. *Blondie24: Playing at the Edge of AI*. Elsevier, 2001. 206, 242, 264
- [218] David B Fogel, Timothy J Hays, Sarah L Hahn, and James Quon. Further evolution of a self-learning chess program. In *CIG*, 2005. 116
- [219] David B Fogel, Timothy J Hays, Sarah L Hahn, and James Quon. The blondie25 chess program competes against fritz 8.0 and a human chess master. In *Computational Intelligence and Games, 2006 IEEE Symposium on*, pages 230–235. Citeseer, 2006. 116
- [220] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017. 196
- [221] Aviezri S Fraenkel and David Lichtenstein. Computing a perfect strategy for $n \times n$ chess requires time exponential in n . In *International Colloquium on Automata, Languages, and Programming*, pages 278–293. Springer, 1981. 86
- [222] Varela Francisco, Evan Thompson, and Eleanor Rosch. The embodied mind: cognitive science and human experience, 1991. 22
- [223] Vincent Fran ois-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Joelle Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018. 69
- [224] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. *arXiv preprint arXiv:1710.09767*, 2017. 250, 257, 287
- [225] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, NY, USA:, 2001. 152
- [226] Johannes F rnkranz. Machine learning in computer chess: The next generation. *ICGA Journal*, 19(3):147–161, 1996. 116
- [227] Luca M Gambardella and Marco Dorigo. Ant-q: A reinforcement learning approach to the traveling salesman problem. In *Machine Learning Proceedings 1995*, pages 252–260. Elsevier, 1995. 274
- [228] Artur d'Avila Garcez, Tarek R Besold, Luc De Raedt, Peter F ldiak, Pascal Hitzler, Thomas Icard, Kai-Uwe K hnberger, Luis C Lamb, Risto Miikkulainen, and Daniel L Silver. Neural-symbolic learning and reasoning: contributions and challenges. In *2015 AAAI Spring Symposium Series*, 2015. 31
- [229] Ralph Gasser. Solving nine men's morris. *Computational Intelligence*, 12(1):24–41, 1996. 117

- [230] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvari, and Olivier Teytaud. The grand challenge of computer go: Monte Carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012. 130, 139, 140, 149, 243, 360
- [231] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007. 143, 149, 243, 316
- [232] Sylvain Gelly and David Silver. Achieving master level play in 9 × 9 computer go. In *AAAI*, volume 8, pages 1537–1540, 2008. 143, 242, 243, 264
- [233] Sylvain Gelly and David Silver. Monte-Carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011. 143, 149
- [234] Sylvain Gelly, Yizao Wang, Olivier Teytaud, Modification Uct Patterns, and Projet Tao. Modification of UCT with patterns in Monte-Carlo go. 2006. 130, 133, 143, 149, 242, 264, 315, 316
- [235] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62, 2005. 231, 278
- [236] Aurelien Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* ” O’Reilly Media, Inc.”, 2019. 180, 201, 207, 208
- [237] Felix A Gers, Jurgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999. 181
- [238] Andrew Gilpin and Tuomas Sandholm. A competitive texas hold’em poker player via automated abstraction and real-time equilibrium computation. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1007. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006. 275
- [239] Alexios Giotis, Michael Emmerich, Boris Naujoks, Kyriakos Giannakoglou, and Thomas Back. Low-cost stochastic optimization for engineering applications. 279
- [240] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014. 176
- [241] Malcolm Gladwell. *Outliers: The story of success*. Hachette UK, 2008. 285
- [242] Peter W Glynn and Donald L Iglehart. Importance sampling for stochastic simulations. *Management Science*, 35(11):1367–1392, 1989. 72
- [243] Gordon Goetsch and Murray S Campbell. Experiments with the null-move heuristic. In *Computers, Chess, and Cognition*, pages 159–168. Springer, 1990. 114
- [244] Irving John Good. A five-year plan for automatic chess. *Machine intelligence 2*, pages 89–118, 1968. 114
- [245] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016. 152, 157, 158, 162, 164, 168, 208, 246, 317, 361
- [246] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014. 177, 204, 208
- [247] Geoffrey J Gordon. Stable function approximation in dynamic programming. In *Machine Learning Proceedings 1995*, pages 261–268. Elsevier, 1995. 208
- [248] Tobias Graf and Marco Platzner. Adaptive playouts in Monte-Carlo tree search with policy-gradient reinforcement learning. In *Advances in Computer Games*, pages 1–11. Springer, 2015. 264
- [249] Alex Graves. Supervised sequence labelling. In *Supervised sequence labelling with recurrent neural networks*, pages 5–13. Springer, 2012. 180
- [250] Alex Graves, Marc G Bellemare, Jacob Menick, Remi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. *arXiv preprint arXiv:1704.03003*, 2017. 247, 249

- [251] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Bidirectional lstm networks for improved phoneme classification and recognition. In *International Conference on Artificial Neural Networks*, pages 799–804. Springer, 2005. 181, 208
- [252] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *International Conference on Machine Learning*, pages 1764–1772, 2014. 180
- [253] Michael SA Graziano and Sabine Kastner. Human consciousness and its relationship to social neuroscience: a novel hypothesis. *Cognitive neuroscience*, 2(2):98–113, 2011. 283
- [254] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2017. 181, 208
- [255] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, et al. An investigation of model-free planning. *arXiv preprint arXiv:1901.03559*, 2019. 259
- [256] Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with mctsnets. *arXiv preprint arXiv:1802.04697*, 2018. 259
- [257] David Gunning. Explainable artificial intelligence (xai). *Defense Advanced Research Projects Agency (DARPA), nd Web*, 2, 2017. 260
- [258] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline Monte-Carlo tree search planning. In *Advances in neural information processing systems*, pages 3338–3346, 2014. 264
- [259] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187:27–48, 2016. 173, 176
- [260] Abhishek Gupta, Benjamin Eysenbach, Chelsea Finn, and Sergey Levine. Unsupervised meta-learning for reinforcement learning. *arXiv preprint arXiv:1806.04640*, 2018. 250
- [261] Abhishek Gupta, Russell Mendonca, YuXuan Liu, Pieter Abbeel, and Sergey Levine. Meta-reinforcement learning of structured exploration strategies. *arXiv preprint arXiv:1802.07245*, 2018. 250
- [262] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. *arXiv preprint arXiv:1811.04551*, 2018. 259, 260
- [263] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009. 293
- [264] Eric A Hansen and Rong Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28:267–297, 2007. 101
- [265] Moritz Hardt, Benjamin Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. *arXiv preprint arXiv:1509.01240*, 2015. 197
- [266] Mark Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007. 252
- [267] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. 237
- [268] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010. 195
- [269] Johan Håstad and Mikael Goldmann. On the power of small-depth threshold circuits. *Computational Complexity*, 1(2):113–129, 1991. 158, 161
- [270] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009. 197

- [271] Thomas Hauk, Michael Buro, and Jonathan Schaeffer. Rediscovering $*$ -minimax search. In *International Conference on Computers and Games*, pages 35–50. Springer, 2004. 111
- [272] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994. 30
- [273] Simon S Haykin et al. *Neural networks and learning machines/Simon Haykin*. New York: Prentice Hall, 2009. 158
- [274] Ryan B Hayward and Bjarne Toft. *Hex: The Full Story*. CRC Press, 2019. 148
- [275] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 175, 176, 318, 361
- [276] David Heckerman, Dan Geiger, and David M Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine learning*, 20(3):197–243, 1995. 258
- [277] Nicolas Heess, David Silver, and Yee Whye Teh. Actor-critic reinforcement learning with energy-based policies. In *EWRL*, pages 43–58, 2012. 187
- [278] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016. 229, 258, 274
- [279] Ernst A Heinz. Adaptive null-move pruning. *ICGA Journal*, 22(3):123–132, 1999. 114
- [280] Ernst A Heinz. New self-play results in computer chess. In *International Conference on Computers and Games*, pages 262–276. Springer, 2000. 264
- [281] David P Helmbold and Aleatha Parker-Wood. All-Moves-As-First heuristics in Monte-Carlo go. In *IC-AI*, pages 605–610, 2009. 143
- [282] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *arXiv preprint arXiv:1709.06560*, 2017. 203, 236, 263
- [283] Bernhard Hengst. Discovering hierarchy in reinforcement learning with hexq. In *ICML*, volume 19, pages 243–250, 2002. 257
- [284] Bernhard Hengst. Hierarchical approaches. In *Reinforcement learning*, pages 293–323. Springer, 2012. 257
- [285] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017. 194, 195, 197, 204, 208, 361
- [286] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, et al. Stable baselines, 2018. 294
- [287] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006. 154, 158, 166, 246
- [288] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006. 158
- [289] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012. 168
- [290] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1), 1991. 166
- [291] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001. 166
- [292] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 178, 181
- [293] Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001. 249

- [294] John Holland. Adaptation in natural and artificial systems: an introductory analysis with application to biology. *Control and artificial intelligence*, 1975. 66
- [295] John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992. 30
- [296] Alec Holt, Isabelle Bichindaritz, Rainer Schmidt, and Petra Perner. Medical applications in case-based reasoning. *The Knowledge Engineering Review*, 20(3):289–292, 2005. 281
- [297] Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations and applications*. Elsevier, 2004. 252
- [298] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982. 178
- [299] Ronald A Howard. *Dynamic programming and Markov processes*. NEW YORK: JOHN-WILEY,, 1964. 58
- [300] Meng Hsuen Hsieh, Meng Ju Hsieh, Chin-Ming Chen, Chia-Chang Hsieh, Chien-Ming Chao, and Chih-Cheng Lai. Comparison of machine learning models for the prediction of mortality of patients with unplanned extubation in intensive care units. *Scientific reports*, 8(1):17116, 2018. 279
- [301] Ming Yu Hsieh and Shi-Chun Tsai. On the fairness and complexity of generalized k-in-a-row games. *Theoretical Computer Science*, 385(1-3):88–100, 2007. 86
- [302] Feng-Hsiung Hsu. *Behind Deep Blue: Building the computer that defeated the world chess champion*. Princeton University Press, 2004. 49, 81, 92, 112
- [303] Feng-hsiung Hsu, Thomas Anantharaman, Murray Campbell, and Andreas Nowatzky. A grandmaster chess machine. *Scientific American*, 263(4):44–51, 1990. 49, 242
- [304] Bojun Huang. Pruning game tree by rollouts. In *AAAI*, pages 1165–1173, 2015. 145, 146, 149, 365
- [305] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017. 176
- [306] Timothy Huang, Graeme Connell, and Bryan McQuade. Learning opening strategy in the game of go. In *FLAIRS Conference*, pages 434–438, 2003. 243
- [307] David H Hubel and TN Wiesel. Shape and arrangement of columns in cat's striate cortex. *The Journal of physiology*, 165(3):559–568, 1963. 162
- [308] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968. 162
- [309] Barbara J (Liskov) Huberman. A program to play chess end games. Technical report, Stanford University Department of Computer Science, 1968. 110, 125
- [310] Fan Hui. Fan hui commentary game 2 move 37 lee sedol. <https://deepmind.com/research/alphago/match-archive/alphago-games-english/>, 2018. 215
- [311] Johan Huizinga. *Homo ludens: proeve eener bepaling van het spel-element der cultuur. English: Homo Ludens: A Study of the Play-Element in Culture*. London: Routledge & Kegan Paul. Amsterdam University Press, 1938, 2008. 281
- [312] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011. 252, 265
- [313] Robert M Hyatt and Monty Newborn. Crafty goes deep. *ICGA Journal*, 20(2):79–86, 1997. 124
- [314] Hiroyuki Iida, Makoto Sakuta, and Jeff Rollason. Computer shogi. *Artificial Intelligence*, 134(1-2):121–144, 2002. 86, 233
- [315] Sergey Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. In *Advances in Neural Information Processing Systems*, pages 1945–1953, 2017. 167, 169, 207, 318

- [316] Riashat Islam, Peter Henderson, Maziar Gomrokchi, and Doina Precup. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arXiv preprint arXiv:1708.04133*, 2017. 236
- [317] Shigeki Iwata and Takumi Kasai. The othello game on an $n \times n$ board is PSPACE-complete. *Theoretical Computer Science*, 123(2):329–340, 1994. 86
- [318] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in first-person multiplayer games with population-based deep reinforcement learning. *arXiv preprint arXiv:1807.01281*, 2018. 272
- [319] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castañeda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019. 254, 272, 274
- [320] Max Jaderberg, Valentijn Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017. 253
- [321] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014. 172, 293
- [322] Lu Jiang, Deyu Meng, Qian Zhao, Shiguang Shan, and Alexander G Hauptmann. Self-paced curriculum learning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. 247
- [323] Nanlin Jin, Peter Flach, Tom Wilcox, Royston Sellman, Joshua Thumim, and Arno Knobbe. Subgroup discovery in smart electricity meter data. *IEEE Transactions on Industrial Informatics*, 10(2):1327–1336, 2014. 279
- [324] Michael Johanson. Measuring the size of large no-limit poker games. *arXiv preprint arXiv:1302.7008*, 2013. 40
- [325] Matthew Johnson, Jeffrey M Bradshaw, Paul J Feltovich, Robert R Hoffman, Catholijn Jonker, Birna van Riemsdijk, and Maarten Sierhuis. Beyond cooperative robotics: The central role of interdependence in coactive design. *IEEE Intelligent Systems*, 26(3):81–88, 2011. 279
- [326] Catholijn M Jonker, Reyhan Aydogan, Tim Baarslag, Katsuhide Fujita, Takayuki Ito, and Koen V Hindriks. Automated negotiating agents competition (anac). In *AAAI*, pages 5070–5072, 2017. 275
- [327] Catholijn M Jonker, Valentin Robu, and Jan Treur. An agent architecture for multi-attribute negotiation using incomplete preference information. *Autonomous Agents and Multi-Agent Systems*, 15(2):221–252, 2007. 275
- [328] Michael Irwin Jordan. *Learning in graphical models*, volume 89. Springer Science & Business Media, 1998. 260
- [329] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017. 317
- [330] Andreas Junghanns and Jonathan Schaeffer. Search versus knowledge in game-playing programs revisited. In *IJCAI (1)*, pages 692–697, 1997. 125
- [331] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251, 2001. 256
- [332] Leslie Pack Kaelbling. Hierarchical learning in stochastic domains: Preliminary results. 257
- [333] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998. 274
- [334] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996. 81

- [335] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical planning in the now. In *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010. 257
- [336] Daniel Kahneman. *Thinking, fast and slow*. Farrar, Straus and Giroux, 2011. 23, 33, 56, 122, 240, 282, 283
- [337] Hermann Kaindl. Minimaxing: Theory and practice. *AI Magazine*, 9(3):69, 1988. 113
- [338] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019. 69, 259, 260
- [339] Satwik Kansal and Brendan Martin. Learn data science webpage., 2018. 76, 77, 78, 79, 360, 365
- [340] Sergey Karakovskiy and Julian Togelius. The mario ai benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):55–67, 2012. 236
- [341] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>, 2015. 179, 180, 181, 182, 361
- [342] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of GANs for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017. 177, 204
- [343] Michael N Katehakis and Arthur F Veinott Jr. The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research*, 12(2):262–268, 1987. 137
- [344] John D Kelleher, Brian Mac Namee, and Aoife D’Arcy. *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*. MIT Press, 2015. 208
- [345] Stephen Kelly and Malcolm I Heywood. Multi-task learning in atari video games with emergent tangled program graphs. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 195–202. ACM, 2017. 249
- [346] Stephen Kelly and Malcolm I Heywood. Emergent tangled program graphs in multi-task learning. In *IJCAI*, pages 5294–5298, 2018. 249
- [347] James Kennedy. Swarm intelligence. In *Handbook of nature-inspired and innovative computing*, pages 187–219. Springer, 2006. 30
- [348] Khimya Khetarpal, Zafarali Ahmed, Andre Cianflone, Riashat Islam, and Joelle Pineau. Re-evaluate: Reproducibility in evaluating reinforcement learning algorithms. 2018. 236
- [349] Anders Kierulf, Ken Chen, and Jurg Nievergelt. Smart game board and go explorer: a study in software and knowledge engineering. *Communications of the ACM*, 33(2):152–166, 1990. 128
- [350] Hajime Kimura, Shigenobu Kobayashi, et al. An analysis of actor-critic algorithms using eligibility traces: reinforcement learning with imperfect value functions. *Journal of Japanese Society for Artificial Intelligence*, 15(2):267–275, 2000. 68
- [351] D Kinga and J Ba. Adam: method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, volume 5, 2015. 159
- [352] Hiroaki Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex systems*, 4(4):461–476, 1990. 253
- [353] Julien Kloetzer. Monte-Carlo opening books for amazons. In *International Conference on Computers and Games*, pages 124–135. Springer, 2010. 140
- [354] Craig A Knoblock. Learning abstraction hierarchies for problem solving. In *AAAI*, pages 923–928, 1990. 255
- [355] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975. 95, 106, 125
- [356] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1997. 108

- [357] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013. 81, 258
- [358] Christof Koch. *The quest for consciousness: A neurobiological approach*. Roberts and Company Englewood, CO, 2004. 283
- [359] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006. 132, 136, 149
- [360] Vincent J Koeman, Harm J Griffioen, Danny C Plenge, and Koen V Hindriks. Starcraft as a testbed for engineering complex distributed systems using cognitive agent technology. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1983–1985. International Foundation for Autonomous Agents and Multiagent Systems, 2018. 236
- [361] Teun Koetsier. On the prehistory of programmable machines: musical automata, looms, calculators. *Mechanism and Machine theory*, 36(5):589–603, 2001. 23
- [362] Daphne Koller and Avi Pfeffer. Representations and solutions for game-theoretic problems. *Artificial intelligence*, 94(1-2):167–215, 1997. 275, 289
- [363] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000. 68
- [364] Vijaymohan R Konda and Vivek S Borkar. Actor-Critic-type learning algorithms for Markov Decision Processes. *SIAM Journal on control and Optimization*, 38(1):94–123, 1999. 68
- [365] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! 2018. 274
- [366] Richard E Korf. Macro-operators: A weak method for learning. *Artificial intelligence*, 26(1):35–77, 1985. 255, 256
- [367] Richard E Korf. Recent progress in the design and analysis of admissible heuristic functions. In *International Symposium on Abstraction, Reformulation, and Approximation*, pages 45–55. Springer, 2000. 117
- [368] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1):826–830, 2017. 253
- [369] Sarit Kraus, Ethan Ephrati, and Daniel Lehmann. Negotiation in a non-cooperative environment. *Journal of Experimental & Theoretical Artificial Intelligence*, 3(4):255–281, 1994. 33, 275, 289
- [370] Sarit Kraus and Daniel Lehmann. Diplomat, an agent in a multi agent environment: An overview. In *Computers and Communications, 1988. Conference Proceedings., Seventh Annual International Phoenix Conference on*, pages 434–438. IEEE, 1988. 275, 289
- [371] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009. 169
- [372] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 152, 158, 166, 172, 174, 193, 244
- [373] Kai A Krueger and Peter Dayan. Flexible shaping: How learning in small steps helps. *Cognition*, 110(3):380–394, 2009. 247
- [374] Jan Kuipers, Aske Plaat, JAM Vermaseren, and H Jaap van den Herik. Improving multivariate Horner schemes with Monte Carlo tree search. *Computer Physics Communications*, 184(11):2391–2395, 2013. 141, 142, 147, 360
- [375] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in neural information processing systems*, pages 3675–3683, 2016. 257
- [376] M Pawan Kumar, Benjamin Packer, and Daphne Koller. Self-paced learning for latent variable models. In *Advances in Neural Information Processing Systems*, pages 1189–1197, 2010. 246, 247

- [377] Vipin Kumar, Dana S Nau, and Laveen N Kanal. A general branch-and-bound formulation for and/or graph and game tree search. In *Search in Artificial Intelligence*, pages 91–130. Springer, 1988. 124
- [378] Kamolwan Kunanusont, Simon M Lucas, and Diego Pérez-Liébana. General video game ai: learning from screen capture. In *Evolutionary Computation (CEC), 2017 IEEE Congress on*, pages 2078–2085. IEEE, 2017. 278
- [379] Michail G Lagoudakis and Ronald Parr. Least-squares policy iteration. *Journal of machine learning research*, 4(Dec):1107–1149, 2003. 208
- [380] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*, 2015. 116, 207, 242, 244
- [381] Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985. 66, 137, 138
- [382] Tze-Leung Lai and Sidney Yakowitz. Machine learning and nonparametric bandit theory. *IEEE Transactions on Automatic Control*, 40(7):1199–1209, 1995. 137
- [383] John E Laird, Allen Newell, and Paul S Rosenbloom. Soar: An architecture for general intelligence. *Artificial intelligence*, 33(1):1–64, 1987. 281
- [384] Robert Lake, Jonathan Schaeffer, and Paul Lu. *Solving large retrograde analysis problems using a network of workstations*. Department of Computing Science, University of Alberta, 1993. 117
- [385] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolut, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, et al. Openspiel: A framework for reinforcement learning in games. *arXiv preprint arXiv:1908.09453*, 2019. 55, 294
- [386] Marc Lanctot, Abdallah Saffidine, Joel Veness, Christopher Archibald, and Mark HM Winands. Monte Carlo*-minimax search. In *IJCAI*, pages 580–586, 2013. 111
- [387] Evgenii Mikhailovich Landis and Isaak Moiseevich Yaglom. About aleksandr semenovich kronrod. *Russian Mathematical Surveys*, 56(5):993–1007, 2001. 28
- [388] Raúl Lara-Cabrera, Carlos Cotta, and Antonio J Fernández-Leiva. A review of computational intelligence in rts games. In *2013 IEEE Symposium on Foundations of Computational Intelligence (FOCI)*, pages 114–121. IEEE, 2013. 274
- [389] Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl HaJJar, Hui Chen, Torbjørn S Dahl, Amine Kerkeni, and Karim Beguir. Ranked reward: Enabling self-play reinforcement learning for bin packing. 2019. 274
- [390] Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl HaJJar, Torbjørn S Dahl, Amine Kerkeni, and Karim Beguir. Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization. *arXiv preprint arXiv:1807.01672*, 2018. 262, 274
- [391] Steffen L Lauritzen. *Graphical models*, volume 17. Clarendon Press, 1996. 260
- [392] Quoc V Le, Rajat Monga, Matthieu Devin, Kai Chen, Greg S Corrado, Jeff Dean, and Andrew Y Ng. Building high-level features using large scale unsupervised learning. 2012. 158
- [393] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015. 30, 154, 158, 159, 175, 207, 258, 265
- [394] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989. 162, 163, 246
- [395] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 170, 171, 365
- [396] Chang-Shing Lee, Martin Müller, and Olivier Teytaud. Special issue on Monte Carlo techniques and computer go. *IEEE Transactions on Computational Intelligence and AI in Games*, (4):225–228, 2010. 130

- [397] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th annual international conference on machine learning*, pages 609–616. ACM, 2009.
- [398] Joel Z Leibo, Edward Hughes, Marc Lanctot, and Thore Graepel. Autocurricula and the emergence of innovation from social interaction: A manifesto for multi-agent intelligence research. *arXiv preprint arXiv:1903.00742*, 2019.
- [399] Charles Leiserson and Aske Plaat. Programming parallel applications in cilk. *SINews: SIAM News*, 31(4):6–7, 1998.
- [400] Fabien Letouzey. Fruit chess program. <http://www.fruitchess.com/about.htm>.
- [401] John Levine, Clare Bates Congdon, Marc Ebner, Graham Kendall, Simon M Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General video game playing. 2013.
- [402] Sergey Levine and Vladlen Koltun. Guided policy search. In *International Conference on Machine Learning*, pages 1–9, 2013.
- [403] Andrew Levy, Robert Platt, and Kate Saenko. Hierarchical reinforcement learning with hindsight. *arXiv preprint arXiv:1805.08180*, 2018.
- [404] David NL Levy and Monty Newborn. *How computers play chess*, volume 8. Computer Science Press New York, 1991.
- [405] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint arXiv:1712.09381*, 2017.
- [406] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.
- [407] Milton Lim. History of ai winters. <https://www.actuaries.digital/2018/09/05/history-of-ai-winters/>, 2018.
- [408] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.
- [409] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.
- [410] Raz Lin and Sarit Kraus. Can automated agents proficiently negotiate with humans? *Communications of the ACM*, 53(1):78–88, 2010.
- [411] Raz Lin, Sarit Kraus, Tim Baarslag, Dmytro Tykhonov, Koen Hindriks, and Catholijn M Jonker. Genius: An integrated environment for supporting the design of generic automated negotiators. *Computational Intelligence*, 30(1):48–70, 2014.
- [412] CL Lisetti. Affective computing, 1998.
- [413] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine Learning Proceedings 1994*, pages 157–163. Elsevier, 1994.
- [414] Hui Liu, Song Yu, Zhangxin Chen, Ben Hsieh, and Lei Shao. Sparse matrix-vector multiplication on nvidia gpu. *International Journal of Numerical Analysis & Modeling, Series B*, 3(2):185–191, 2012.
- [415] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.
- [416] Richard J Lorentz. Experiments with Monte-Carlo tree search in the game of Havannah. *ICGA Journal*, 34(3):140–149, 2011.
- [417] Richard J Lorentz. An mcts program to play einstein würfelt nicht! In *Advances in Computer Games*, pages 52–59. Springer, 2011.

- [419] Donald W Loveland. *Automated Theorem Proving: a logical basis*. Elsevier, 2016. 282
- [420] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pages 6379–6390, 2017. 258
- [421] Mark Lutz and Mark Lutz. *Programming python*, volume 8. O'Reilly, 1996. 297
- [422] Siyuan Ma, Raef Bassily, and Mikhail Belkin. The power of interpolation: Understanding the effectiveness of sgd in modern over-parametrized learning. *arXiv preprint arXiv:1712.06559*, 2017. 198
- [423] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008. 154, 259
- [424] Brooke N Macnamara, David Z Hambrick, and Frederick L Oswald. Deliberate practice and performance in music, games, sports, education, and professions: A meta-analysis. *Psychological science*, 25(8):1608–1618, 2014. 286
- [425] Chris J Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. *arXiv preprint arXiv:1412.6564*, 2014. 206, 243
- [426] A Madrigal. How checkers was solved, the story of a duel between two men, one who dies, and the nature of the quest to build artificial intelligence (2017). 48
- [427] Rajbala Makar, Sridhar Mahadevan, and Mohammad Ghavamzadeh. Hierarchical multi-agent reinforcement learning. In *Proceedings of the fifth international conference on Autonomous agents*, pages 246–253. ACM, 2001. 257
- [428] Tambet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. Teacher-student curriculum learning. *arXiv preprint arXiv:1707.00183*, 2017. 247, 265, 274
- [429] Masakazu Matsugu, Katsuhiko Mori, Yusuke Mitari, and Yuji Kaneda. Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural Networks*, 16(5-6):555–559, 2003. 162
- [430] Kiminori Matsuzaki. Empirical analysis of puct algorithm with evaluation functions of different quality. In *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 142–147. IEEE, 2018. 136, 315
- [431] David Allen McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35(3):287–310, 1988. 111
- [432] James L McClelland, Bruce L McNaughton, and Randall C O'reilly. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review*, 102(3):419, 1995. 191
- [433] Pamela McCorduck. *Machines who think: A personal inquiry into the history and prospects of artificial intelligence*. AK Peters/CRC Press, 2009. 22, 23, 24
- [434] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. 156
- [435] Amy McGovern, Richard S Sutton, and Andrew H Fagg. Roles of macro-actions in accelerating reinforcement learning. In *Grace Hopper celebration of women in computing*, volume 1317, 1997. 256, 257
- [436] Donald Michie. Game-playing and game-learning automata. In *Advances in programming and non-numerical computation*, pages 183–200. Elsevier, 1966. 111
- [437] Donald Michie. Machine learning in the next five years. In *Proceedings of the 3rd European Conference on European Working Session on Learning*, pages 107–122. Pitman Publishing, Inc., 1988. 236
- [438] Daniel Michulke and Michael Thielscher. Neural networks for state evaluation in general game playing. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 95–110. Springer, 2009. 278

- [439] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, 2019. 253
- [440] Tomáš Mikolov. Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April*, 80, 2012. 180
- [441] Jonathan K Millen. Programming the game of go. *Byte Magazine*, 1981. 52
- [442] S Ali Mirsoleimani, Aske Plaat, and Jaap van den Herik. Ensemble uct needs high exploitation. 2015. 144
- [443] S Ali Mirsoleimani, Aske Plaat, Jaap van den Herik, and Jos Vermaseren. Structured parallel programming for Monte Carlo tree search. 2017. 144
- [444] S Ali Mirsoleimani, Aske Plaat, H Jaap van den Herik, and Jos Vermaseren. An analysis of virtual loss in parallel mcts. In *ICAART (2)*, pages 648–652, 2017. 316
- [445] S Ali Mirsoleimani, Aske Plaat, Jaap Van Den Herik, and Jos Vermaseren. Scaling Monte Carlo tree search on Intel Xeon phi. In *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*, pages 666–673. IEEE, 2015. 144
- [446] S Ali Mirsoleimani, Jaap van den Herik, Aske Plaat, and Jos Vermaseren. A lock-free algorithm for parallel mcts. In *ICAART (2)*, pages 589–598, 2018. 316
- [447] S Ali Mirsoleimani, Jaap van den Herik, Aske Plaat, and Jos Vermaseren. Pipeline pattern for parallel mcts. In *ICAART (2)*, pages 614–621, 2018. 144, 149
- [448] Tom Mitchell, William Cohen, Estevam Hruschka, Partha Talukdar, Bo Yang, Justin Betteridge, Andrew Carlson, B Dalvi, Matt Gardner, Bryan Kisiel, et al. Never-ending learning. *Communications of the ACM*, 61(5):103–115, 2018. 249
- [449] Tom M Mitchell. *The need for biases in learning generalizations*. Department of Computer Science, 1980. 248
- [450] Tom Michael Mitchell. *The discipline of machine learning*, volume 9. Carnegie Mellon University, School of Computer Science, Machine Learning, 2006. 248
- [451] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016. 196, 197
- [452] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. Recurrent models of visual attention. In *Advances in neural information processing systems*, pages 2204–2212, 2014. 180
- [453] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 46, 75, 152, 183, 186, 187, 188, 189, 194, 207, 242, 244
- [454] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015. 152, 183, 188, 189, 192, 194, 200, 205, 244
- [455] Thomas M Moerland, Joost Broekens, and Catholijn M Jonker. Efficient exploration with double uncertain value networks. *arXiv preprint arXiv:1711.10789*, 2017. 196, 254
- [456] Thomas M Moerland, Joost Broekens, and Catholijn M Jonker. The potential of the return distribution for exploration in rl. *arXiv preprint arXiv:1806.04242*, 2018. 196, 254
- [457] Thomas M Moerland, Joost Broekens, Aske Plaat, and Catholijn M Jonker. A0c: Alpha zero in continuous action space. *arXiv preprint arXiv:1805.09613*, 2018. 147, 258, 274
- [458] Thomas M Moerland, Joost Broekens, Aske Plaat, and Catholijn M Jonker. Monte Carlo tree search for asymmetric trees. *arXiv preprint arXiv:1805.09218*, 2018. 147
- [459] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 73:1–15, 2018. 259

- [460] James Moor. The dartmouth college artificial intelligence conference: The next fifty years. *Ai Magazine*, 27(4):87, 2006. 24
- [461] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017. 54, 264, 275, 276
- [462] Andreas C Müller, Sarah Guido, et al. *Introduction to machine learning with Python: a guide for data scientists.* " O'Reilly Media, Inc.", 2016. 208
- [463] Martin Müller. Computer go. *Artificial Intelligence*, 134(1-2):145–179, 2002. 128
- [464] Martin Müller. Fuego at the Computer Olympiad in Pamplona 2009: A tournament report. *University of Alberta, TR*, pages 09–09, 2009. 130
- [465] Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare. Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1054–1062, 2016. 72
- [466] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2775–2785, 2017. 70
- [467] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data. arxiv, arXiv:1912.02292, 2019. 198
- [468] Preetum Nakkiran, Gal Kaplun, Dimitris Kalimeris, Tristan Yang, Benjamin L Edelman, Fred Zhang, and Boaz Barak. Sgd on neural networks learns functions of increasing complexity. *arXiv preprint arXiv:1905.11604*, 2019. 198
- [469] Dana Nau, Paul Purdom, and Chun-Hung Tzeng. Experiments on alternatives to minimax. *International Journal of Parallel Programming*, 15(2):163–183, 1986. 111
- [470] Dana S Nau. Pathology on game trees: A summary of results. In *AAAI*, pages 102–104, 1980. 103, 125
- [471] Richard E Neapolitan et al. *Learning Bayesian networks*, volume 38. Pearson Prentice Hall Upper Saddle River, NJ, 2004. 258
- [472] Joseph Needham. Science and civilisation in china, vol. 4: Physics and physical technology. part iii. *Civil Engineering and Nautics (d) Building Technology*, Cambridge University Press, pages 60–66, 1965. 23
- [473] Ulric Neisser, Gwyneth Boodoo, Thomas J Bouchard Jr, A Wade Boykin, Nathan Brody, Stephen J Ceci, Diane F Halpern, John C Loehlin, Robert Perloff, Robert J Sternberg, et al. Intelligence: Knowns and unknowns. *American psychologist*, 51(2):77, 1996. 280
- [474] Monty Newborn. *Deep Blue: an artificial intelligence milestone*. Springer Science & Business Media, 2013. 49
- [475] Allen Newell, John Calman Shaw, and Herbert A Simon. Elements of a theory of human problem solving. *Psychological review*, 65(3):151, 1958. 281
- [476] Allen Newell and Herbert Simon. Computer science as empirical enquiry: Symbols and search. *CACM*, 1976. 125
- [477] Muhammad Abdul Hakim Newton, John Levine, Maria Fox, and Derek Long. Learning macro-actions for arbitrary planners and domains. In *ICAPS*, pages 256–263, 2007. 256
- [478] Alexander E Nezhinsky and Fons J Verbeek. Pattern recognition for high throughput zebrafish imaging using genetic algorithm optimization. In *IAPR International Conference on Pattern Recognition in Bioinformatics*, pages 301–312. Springer, 2010. 279
- [479] Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004. 168
- [480] Nils J Nilsson. *The quest for artificial intelligence*. Cambridge University Press, 2009. 30

- [481] T Nitsche. A learning chess program. In *Advances in Computer Chess*, pages 113–120. Elsevier, 1982. 116
- [482] Ruth Nussinov, Chung-Jung Tsai, Amarda Shehu, and Hyunbum Jang. Computational structural biology: Successes, future directions, and challenges. *Molecules*, 24(3):637, 2019. 212
- [483] Brendan O’Donoghue, Remi Munos, Koray Kavukcuoglu, and Volodymyr Mnih. Combining policy gradient and q-learning. *arXiv preprint arXiv:1611.01626*, 2016. 68, 197
- [484] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004. 172
- [485] Chris Olah. Understanding lstm networks. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>, 2015. 179, 180, 361
- [486] Joseph O’Neill, Barty Pleydell-Bouverie, David Dupret, and Jozsef Csicsvari. Play it again: reactivation of waking experience and memory. *Trends in neurosciences*, 33(5):220–229, 2010. 191
- [487] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):293–311, 2013. 41, 54, 276, 277
- [488] OpenAI. OpenAI five. <https://blog.openai.com/openai-five/>, 2018. 274
- [489] Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepesvári, Satinder Singh, et al. Behaviour suite for reinforcement learning. *arXiv preprint arXiv:1908.03568*, 2019. 294
- [490] Christopher Painter-Wakefield and Ronald Parr. Greedy algorithms for sparse reinforcement learning. *arXiv preprint arXiv:1206.6485*, 2012. 258
- [491] Sinno Jialin Pan, Qiang Yang, et al. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010. 228, 249, 287
- [492] Liliu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous agents and multi-agent systems*, 11(3):387–434, 2005. 258
- [493] Zhen-Jia Pang, Ruo-Ze Liu, Zhou-Yu Meng, Yi Zhang, Yang Yu, and Tong Lu. On reinforcement learning for full-length game of starcraft. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4691–4698, 2019. 257
- [494] Aleksandr I Panov and Aleksey Skrynnik. Automatic formation of the structure of abstract machines in hierarchical reinforcement learning with state clustering. *arXiv preprint arXiv:1806.05292*, 2018. 257
- [495] Giuseppe Davide Paparo, Vedran Dunjko, Adi Makmal, Miguel Angel Martin-Delgado, and Hans J Briegel. Quantum speedup for active learning agents. *Physical Review X*, 4(3):031002, 2014. 279
- [496] Gisele L Pappa, Gabriela Ochoa, Matthew R Hyde, Alex A Freitas, John Woodward, and Jerry Swan. Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 15(1):3–35, 2014. 252
- [497] Ronald Parr and Stuart J Russell. Reinforcement learning with hierarchies of machines. In *Advances in neural information processing systems*, pages 1043–1049, 1998. 257
- [498] Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sébastien Racanière, David Reichert, Théophane Weber, Daan Wierstra, and Peter Battaglia. Learning model-based planning from scratch. *arXiv preprint arXiv:1707.06170*, 2017. 244
- [499] Gian-Carlo Pascutto. Leela zero, 2017. 295
- [500] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 172, 293

- [501] 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. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019. 293
- [502] Judea Pearl. Scout: A simple game-searching algorithm with proven optimal properties. In *AAAI*, pages 143–145, 1980. 106, 107, 365
- [503] Judea Pearl. On the nature of pathology in game searching. *Artificial Intelligence*, 20(4):427–453, 1983. 103, 125
- [504] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc., Reading, MA, 1984. 91, 124, 125
- [505] Judea Pearl and Dana Mackenzie. *The book of why: the new science of cause and effect*. Basic Books, 2018. 124
- [506] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011. 253
- [507] Barney Pell. Metagame: A new challenge for games and learning. 1992. 278
- [508] Barney Pell. Logic programming for general game-playing. Technical report, University of Cambridge, Computer Laboratory, 1993. 278
- [509] Barney Pell. A strategic metagame player for general chess-like games. *Computational Intelligence*, 12(1):177–198, 1996. 231, 278
- [510] Diego Perez Liebana, Jens Dieskau, Martin Hunermund, Sanaz Mostaghim, and Simon Lucas. Open loop search for general video game playing. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 337–344. ACM, 2015. 278
- [511] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M Lucas, Adrien Couëtoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):229–243, 2016. 278
- [512] Theodore J Perkins, Doina Precup, et al. Using options for knowledge transfer in reinforcement learning. *University of Massachusetts, Amherst, MA, USA, Tech. Rep*, 1999. 257
- [513] Nicola Pezzotti, Thomas Höllt, Jan Van Gemert, Boudewijn PF Lelieveldt, Elmar Eisemann, and Anna Vilanova. Deepeyes: Progressive visual analytics for designing deep neural networks. *IEEE transactions on visualization and computer graphics*, 24(1):98–108, 2017. 259
- [514] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018. 253
- [515] Wim Pijls and Arie de Bruin. Another view on the SSS* algorithm. In *Algorithms*, pages 211–220. Springer, 1990. 95, 124, 125
- [516] Aske Plaat. *Research re: search and re-search*. PhD thesis, Erasmus University Rotterdam, 1996. 108
- [517] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie De Bruin. A new paradigm for minimax search. Technical report, University of Alberta, 1994. 106
- [518] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie De Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1-2):255–293, 1996. 106, 108, 109, 124, 125, 365
- [519] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie De Bruin. Exploiting graph properties of game trees. In *AAAI/IAAI, Vol. 1*, pages 234–239, 1996. 87
- [520] Matthias Plappert. keras-rl. <https://github.com/keras-rl/keras-rl>, 2016. 201
- [521] Jonathan A Plucker and Joseph S Renzulli. Psychometric approaches to the study of human creativity. *Handbook of creativity*, pages 35–61, 1999. 282
- [522] Jordan B Pollack and Alan D Blair. Why did td-gammon work? In *Advances in Neural Information Processing Systems*, pages 10–16, 1997. 187

- [523] Lorien Y Pratt. Discriminability-based transfer between neural networks. In *Advances in neural information processing systems*, pages 204–211, 1993. 249, 265
- [524] Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998. 168
- [525] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998. 168
- [526] Mike Preuss, Nicola Beume, Holger Danielsiek, Tobias Hein, Boris Naujoks, Nico Piatkowski, Raphael Stuer, Andreas Thom, and Simon Wessing. Towards intelligent team composition and maneuvering in real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):82–98, 2010. 274
- [527] Zhiwei Qin, Weichang Li, and Firdaus Janoos. Sparse reinforcement learning via convex optimization. In *International Conference on Machine Learning*, pages 424–432, 2014. 258
- [528] J Ross Quinlan. Learning efficient classification procedures and their application to chess end games. In *Machine learning*, pages 463–482. Springer, 1983. 116
- [529] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986. 260
- [530] Jacob Rafati and David C Noelle. Learning representations in model-free hierarchical reinforcement learning. *arXiv preprint arXiv:1810.10096*, 2018. 256, 257
- [531] Rajat Raina, Alexis Battle, Honglak Lee, Benjamin Packer, and Andrew Y Ng. Self-taught learning: transfer learning from unlabeled data. In *Proceedings of the 24th international conference on Machine learning*, pages 759–766. ACM, 2007. 250
- [532] Kate Rakelly, Aurick Zhou, Deirdre Quillen, Chelsea Finn, and Sergey Levine. Efficient off-policy meta-reinforcement learning via probabilistic context variables. *arXiv preprint arXiv:1903.08254*, 2019. 250
- [533] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. On adversarial search spaces and sampling-based planning. In *ICAPS*, volume 10, pages 242–245, 2010. 232
- [534] J. Rapin and O. Teytaud. Nevergrad - A gradient-free optimization platform. <https://GitHub.com/FacebookResearch/Nevergrad>, 2018. 253
- [535] Andrew L Reibman and Bruce W Ballard. Non-minimax search strategies for use against fallible opponents. In *AAAI*, pages 338–342, 1983. 111
- [536] Alexander Reinefeld. *Spielbaum-Suchverfahren*, volume 200. Springer-Verlag, 1989. 106, 107, 365
- [537] Joao Ribeiro, Pedro Mariano, and Luís Seabra Lopes. Darkblade: A program that plays Diplomacy. In *Portuguese Conference on Artificial Intelligence*, pages 485–496. Springer, 2009. 275
- [538] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005. 208
- [539] Arpad Rimmel, Olivier Teytaud, Chang-Shing Lee, Shi-Jim Yen, Mei-Hui Wang, and Shang-Rong Tsai. Current frontiers in computer go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):229–238, 2010. 130
- [540] Ronald L Rivest. Game tree searching by min/max approximation. *Artificial Intelligence*, 34(1):77–96, 1987. 145
- [541] John W Romein and Henri E Bal. Solving awari with parallel retrograde analysis. *Computer*, 36(10):26–33, 2003. 111, 117
- [542] Tord Romstad, Marco Costalba, and Joona Kiiski. <https://stockfishchess.org> Source code at <https://github.com/official-stockfish/Stockfish>. Stockfish open source chess program. 92, 233
- [543] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. 207
- [544] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011. 136, 315, 320

- [545] Guido Rossum. Python reference manual. 1995. 293
- [546] Burkhard Rost and Chris Sander. Prediction of protein secondary structure at better than 70% accuracy. *Journal of molecular biology*, 232(2):584–599, 1993. 207
- [547] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000. 154
- [548] Neil Rubens, Mehdi Elahi, Masashi Sugiyama, and Dain Kaplan. Active learning in recommender systems. In *Recommender systems handbook*, pages 809–846. Springer, 2015. 247
- [549] Jonathan Rubin and Ian Watson. Computer poker: A review. *Artificial intelligence*, 175(5–6):958–987, 2011. 275
- [550] Ben Ruijl. Games and loop integrals. *Journal of Physics: Conference Series*, 1085:022007, Sep 2018. 144
- [551] Ben Ruijl, Jos Vermaseren, Aske Plaat, and Jaap van den Herik. Combining simulated annealing and Monte Carlo tree search for expression simplification. *arXiv preprint arXiv:1312.0841*, 2013. 144
- [552] Ben Ruijl, Jos Vermaseren, Aske Plaat, and Jaap van den Herik. Hepgame and the simplification of expressions. *arXiv preprint arXiv:1405.6369*, 2014. 144, 274, 279
- [553] David E Rumelhart, James L McClelland, PDP Research Group, et al. *Parallel distributed processing*, volume 1. MIT press Cambridge, MA, 1987. 207
- [554] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016. 22, 30, 55, 72, 124, 152, 153, 274
- [555] Matthia Sabatelli, Gilles Louppe, Pierre Geurts, and Marco A Wiering. Deep quality-value (dqv) learning. *arXiv preprint arXiv:1810.00368*, 2018. 196
- [556] Aleksander Sadikov, Ivan Bratko, and Igor Kononenko. Bias and pathology in minimax search. *Theoretical Computer Science*, 349(2):268–281, 2005. 103, 125
- [557] Sumit Saha. A comprehensive guide to convolutional neural networks — the eli5 way. *Towards Data Science*, (<https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>), 2018. 160, 165, 360, 361
- [558] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017. 253
- [559] Brian Sallans and Geoffrey E Hinton. Reinforcement learning with factored states and actions. *Journal of Machine Learning Research*, 5(Aug):1063–1088, 2004. 187
- [560] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959. 43, 92, 116, 241, 242
- [561] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010. 172
- [562] Tuomas Sandholm. The state of solving large incomplete-information games, and application to poker. *AI Magazine*, 31(4):13–32, 2010. 54, 275
- [563] Juan C Santamaría, Richard S Sutton, and Ashwin Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive behavior*, 6(2):163–217, 1997. 258
- [564] Kaz Sato, Cliff Young, and David Patterson. An in-depth look at google’s first tensor processing unit (tpu) <https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu>, 2017. 319, 362
- [565] Maarten PD Schadd, Mark HM Winands, H Jaap Van Den Herik, Guillaume MJ-B Chaslot, and Jos WHM Uiterwijk. Single-player Monte-Carlo tree search. In *International Conference on Computers and Games*, pages 1–12. Springer, 2008. 139
- [566] Steve Schaefer. Mathematical recreations. <http://www.mathrec.org/old/2002jan/solutions.html>, 2002. 86

- [567] Jonathan Schaeffer. *Experiments in search and knowledge*. PhD thesis, Department of Computing Science, University of Alberta, 1986. 110
- [568] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE transactions on pattern analysis and machine intelligence*, 11(11):1203–1212, 1989. 110, 125
- [569] Jonathan Schaeffer. *One jump ahead: computer perfection at checkers*. Springer Science & Business Media, 2008. 44, 47, 48, 81
- [570] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007. 37, 47, 86, 111, 117
- [571] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992. 47, 113, 242
- [572] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21, 1996. 47, 117, 242, 243
- [573] Jonathan Schaeffer and Aske Plaat. New advances in alpha-beta searching. In *Proceedings of the 1996 ACM 24th annual conference on Computer science*, pages 124–130. ACM, 1996. 103
- [574] Jonathan Schaeffer and Aske Plaat. Kasparov versus Deep Blue: The rematch. *ICGA Journal*, 20(2):95–101, 1997. 49
- [575] Jonathan Schaeffer, Aske Plaat, and Andreas Junghanns. Unifying single-agent and two-player search. *Information Sciences*, 135(3-4):151–175, 2001. 125, 194, 273
- [576] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International Conference on Machine Learning*, pages 1312–1320, 2015. 257
- [577] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015. 195
- [578] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *Aaai*, volume 7, pages 1191–1196, 2007. 278
- [579] Juergen Schmidhuber, Jieyu Zhao, and MA Wiering. Simple principles of metalearning. *Technical report IDSIA*, 69:1–23, 1996. 249
- [580] Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta... hook*. PhD thesis, Technische Universität München, 1987. 249
- [581] Jürgen Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992. 166, 207
- [582] Jürgen Schmidhuber. Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts. *Connection Science*, 18(2):173–187, 2006. 282
- [583] Jürgen Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development*, 2(3):230–247, 2010. 282
- [584] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015. 158, 207
- [585] Jürgen Schmidhuber, F Gers, and Douglas Eck. Learning nonregular languages: A comparison of simple recurrent networks and lstm. *Neural Computation*, 14(9):2039–2041, 2002. 181
- [586] Michael John Schofield, Timothy Joseph Cerecsei, and Michael Thielscher. Hyperplay: A solution to general game playing with imperfect information. In *AAAI*, 2012. 278
- [587] Michael John Schofield and Michael Thielscher. Lifting model sampling for general game playing to incomplete-information models. In *AAAI*, volume 15, pages 3585–3591, 2015. 278
- [588] Nicol N Schraudolph, Peter Dayan, and Terrence J Sejnowski. Temporal difference learning of position evaluation in the game of go. In *Advances in Neural Information Processing Systems*, pages 817–824, 1994. 187, 242, 243

- [589] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019. 259
- [590] Günther Schräfer. Presence and absence of pathology on game trees. In *Advances in Computer Chess*, pages 101–112. Pergamon Press, Inc., 1986. 113
- [591] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015. 197
- [592] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 197
- [593] Nicolas Schweighofer and Kenji Doya. Meta-learning in reinforcement learning. *Neural Networks*, 16(1):5–9, 2003. 250
- [594] Marco Scutari et al. Learning Bayesian networks with the bnlearn R package. *Journal of Statistical Software*, 35(i03), 2010. 258
- [595] John R Searle. *Mind, language and society: Philosophy in the real world*. Basic books, 2008. 28
- [596] Richard B Segal. On the scalability of parallel UCT. In *International Conference on Computers and Games*, pages 36–47. Springer, 2010. 316
- [597] Marwin HS Segler, Mike Preuss, and Mark P Waller. Planning chemical syntheses with deep neural networks and symbolic ai. *Nature*, 555(7698):604, 2018. 279
- [598] Burr Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009. 247
- [599] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016. 320
- [600] Claude E Shannon. Programming a computer for playing chess. In *Computer chess compendium*, pages 2–13. Springer, 1988. 42, 85, 86, 87, 117
- [601] Stuart C Shapiro. The turing test and the economist. *ACM SIGART Bulletin*, 3(4):10–11, 1992. 27
- [602] Rishi Sharma, Shane Barratt, Stefano Ermon, and Vijay Pande. Improved training with curriculum GANs. *arXiv preprint arXiv:1807.09295*, 2018. 248
- [603] David Silver. *Reinforcement learning and simulation based search in the game of Go*. PhD thesis, University of Alberta, 2009. 133, 243
- [604] David Silver, Aja Huang, Chris 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(7587):484, 2016. 46, 69, 92, 212, 216, 217, 218, 219, 220, 221, 222, 234, 242, 264, 295, 315, 361
- [605] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017. 231, 295
- [606] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018. 92, 212, 231, 232, 234, 242, 264, 315, 320, 361

- [607] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017. 46, 92, 116, 212, 216, 221, 222, 229, 230, 234, 242, 251, 252, 264, 295, 315, 317, 318, 319, 361, 362, 363
- [608] David Silver, Richard S Sutton, and Martin Müller. Reinforcement learning of local shape in the game of go. In *IJCAI*, volume 7, pages 1053–1058, 2007. 243, 264
- [609] David Silver, Richard S Sutton, and Martin Müller. Temporal-difference search in computer Go. *Machine learning*, 87(2):183–219, 2012. 207, 243
- [610] David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David Reichert, Neil Rabinowitz, Andre Barreto, et al. The predictron: End-to-end learning and planning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3191–3199. JMLR.org, 2017. 181, 260
- [611] David Silver and Joel Veness. Monte-Carlo planning in large POMDPs. In *Advances in neural information processing systems*, pages 2164–2172, 2010. 147
- [612] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 174
- [613] Satinder P Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the National Conference on Artificial Intelligence*, number 10, page 202. JOHN WILEY & SONS LTD, 1992. 257
- [614] Chiara F Sironi and Mark HM Winands. Comparison of rapid action value estimation variants for general game playing. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8. IEEE, 2016. 278
- [615] Chiara F Sironi and Mark HM Winands. On-line parameter tuning for Monte-Carlo tree search in General Game Playing. In *Workshop on Computer Games*, pages 75–95. Springer, 2017. 278
- [616] Kate A Smith. Neural networks for combinatorial optimization: a review of more than a decade of research. *INFORMS Journal on Computing*, 11(1):15–34, 1999. 274
- [617] Stephen J Smith, Dana Nau, and Tom Throop. Computer bridge: A big win for AI planning. *Ai magazine*, 19(2):93, 1998. 275
- [618] Alex J Smola and Bernhard Schölkopf. *Learning with kernels*, volume 4. Citeseer, 1998. 207
- [619] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012. 252
- [620] Deepak Soekhoe, Peter Van Der Putten, and Aske Plaat. On the impact of data set size in transfer learning using deep neural networks. In *International Symposium on Intelligent Data Analysis*, pages 50–60. Springer, 2016. 250
- [621] Edward J Sondik. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations research*, 26(2):282–304, 1978. 274
- [622] Fengguang Song and Jack Dongarra. Scaling up matrix computations on shared-memory many-core systems with 1000 cpu cores. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 333–342. ACM, 2014. 172
- [623] Mei Song, A Montanari, and P Nguyen. A mean field view of the landscape of two-layers neural networks. In *Proceedings of the National Academy of Sciences*, volume 115, pages E7665–E7671, 2018. 157
- [624] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 168
- [625] David Stern, Ralf Herbrich, and Thore Graepel. Bayesian pattern ranking for move prediction in the game of go. In *Proceedings of the 23rd international conference on Machine learning*, pages 873–880. ACM, 2006. 264

- [626] George C Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2):179–196, 1979. 124, 125
- [627] Lise Stork, Katherine Wolstencroft, Andreas Weber, Fons Verbeek, and Aske Plaat. Priming digitisation: Learning the textual structure in field books. 2018. 279
- [628] David Stoutamire. Machine learning applied to go. *MS thesis*, 1991. 243
- [629] Christopher S Strachey. Logical or non-mathematical programmes. In *Proceedings of the 1952 ACM national meeting (Toronto)*, pages 46–49. ACM, 1952. 43
- [630] Freek Stulp and Stefan Schaal. Hierarchical reinforcement learning with movement primitives. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 231–238. IEEE, 2011. 257
- [631] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 2019. 177
- [632] Sainbayar Sukhbaatar, Emily Denton, Arthur Szlam, and Rob Fergus. Learning goal embeddings via self-play for hierarchical reinforcement learning. *arXiv preprint arXiv:1811.09083*, 2018. 257
- [633] Ilya Sutskever. *Training recurrent neural networks*. University of Toronto Toronto, Ontario, Canada, 2013. 180
- [634] Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024, 2011. 180
- [635] Ilya Sutskever and Vinod Nair. Mimicking go experts with convolutional neural networks. In *International Conference on Artificial Neural Networks*, pages 101–110. Springer, 2008. 187, 206
- [636] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014. 180
- [637] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988. 44, 66, 196
- [638] Richard S Sutton and Andrew G Barto. *Reinforcement learning, an Introduction, Second Edition*. MIT press, Cambridge, Mass, 2018. 31, 44, 58, 59, 60, 64, 67, 68, 69, 78, 81, 184, 185, 196, 209, 218, 259, 360
- [639] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999. 255, 256, 361
- [640] Richard Stuart Sutton. Temporal credit assignment in reinforcement learning. 1984. 196
- [641] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017. 172
- [642] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015. 175
- [643] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016. 175
- [644] 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. 177
- [645] István Szita, Guillaume Chaslot, and Pieter Spronck. Monte-Carlo tree search in settlers of catan. In *Advances in Computer Games*, pages 21–32. Springer, 2009. 140

- [646] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. In *Advances in Neural Information Processing Systems*, pages 2154–2162, 2016. 260
- [647] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993. 258
- [648] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009. 249
- [649] Shoshannah Tekofsky, Pieter Spronck, Martijn Goudbeek, Aske Plaat, and Jaap van den Herik. Past our prime: A study of age and play style development in battlefield 3. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(3):292–303, 2015. 274
- [650] Joshua B Tenenbaum, Vin De Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000. 154
- [651] Gerald Tesauro. Neurogammon wins computer olympiad. *Neural Computation*, 1(3):321–323, 1989. 44, 92, 242
- [652] Gerald Tesauro. Neurogammon: A neural-network backgammon program. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 33–39. IEEE, 1990. 206, 207, 243
- [653] Gerald Tesauro. TD-gammon: A self-teaching backgammon program. In *Applications of Neural Networks*, pages 267–285. Springer, 1995. 40, 116, 187, 206, 242
- [654] Gerald Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3):58–68, 1995. 45, 66, 206, 243
- [655] Gerald Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2):181–199, 2002. 44, 243, 251
- [656] Marc Teyssier and Daphne Koller. Ordering-based search: A simple and effective algorithm for learning Bayesian networks. *arXiv preprint arXiv:1207.1429*, 2012. 258
- [657] Shantanu Thakoor, Surag Nair, and Megha Jhunjhunwala. Learning to play othello without human knowledge. Stanford University CS238 Final Project Report, 2017. 295
- [658] Richard H Thaler and Cass R Sunstein. *Nudge: Improving decisions about health, wealth, and happiness*. Penguin, 2009. 33, 281
- [659] Michael Thielscher. Answer set programming for single-player games in general game playing. In *International Conference on Logic Programming*, pages 327–341. Springer, 2009. 278
- [660] Michael Thielscher. A general game description language for incomplete information games. In *AAAI*, volume 10, pages 994–999, 2010. 278
- [661] Michael Thielscher. The general game playing description language is universal. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1107, 2011. 278
- [662] Ken Thompson. Retrograde analysis of certain endgames. *ICCA journal*, 9(3):131–139, 1986. 116, 117
- [663] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM, 2013. 252
- [664] Sebastian Thrun. Learning to play the game of chess. In *Advances in neural information processing systems*, pages 1069–1076, 1995. 116, 242, 243
- [665] Sebastian Thrun. *Explanation-based neural network learning: A lifelong learning approach*, volume 357. Springer Science & Business Media, 2012. 249, 287
- [666] Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012. 249, 265
- [667] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. In *Advances in Neural Information Processing Systems*, pages 2659–2669, 2017. 295

- [668] Yuandong Tian, Jerry Ma, Qucheng Gong, Shubho Sengupta, Zhuoyuan Chen, and C. Lawrence Zitnick. Elf opengo. <https://github.com/pytorch/ELF>, 2018. 236, 295
- [669] Yuandong Tian and Yan Zhu. Better computer go player with neural network and long-term prediction. *arXiv preprint arXiv:1511.06410*, 2015. 242, 243, 295
- [670] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012. 203
- [671] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, and Georgios N Yannakakis. Multiobjective exploration of the starcraft map space. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 265–272. IEEE, 2010. 54, 274
- [672] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation. In *European Conference on the Applications of Evolutionary Computation*, pages 141–150. Springer, 2010. 55
- [673] Armon Toubman, Jan Joris Roessingh, Pieter Spronck, Aske Plaat, and Jaap van den Herik. Transfer learning of air combat behavior. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 226–231. IEEE, 2015. 250
- [674] Luan Tran, Xi Yin, and Xiaoming Liu. Disentangled representation learning gan for pose-invariant face recognition. In *CVPR*, volume 3, page 7, 2017. 204
- [675] John Tromp. Number of legal go states, 2016. 87
- [676] John Tromp and Gunnar Farnebäck. Combinatorics of go. In *International Conference on Computers and Games*, pages 84–99, <https://tromp.github.io/go/gostate.pdf>, 2006. Springer. 86, 87
- [677] John N Tsitsiklis and Benjamin Van Roy. Analysis of temporal-difference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081, 1997. 184
- [678] Alan M Turing. Computing machinery and intelligence. In *Parsing the Turing Test*, pages 23–65. Springer, 1950, 2009. 26
- [679] Alan M Turing. Digital computers applied to games. *Faster than thought*, 1953. 43, 85, 117
- [680] George Tzanetakis and Perry Cook. Musical genre classification of audio signals. *IEEE Transactions on speech and audio processing*, 10(5):293–302, 2002. 282
- [681] Alberto Uriarte and Santiago Ontanón. A benchmark for starcraft intelligent agents. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015. 236, 274
- [682] Computer Shogi Association. http://www2.computer-shogi.org/index_e.html. Website. 233
- [683] Leslie G Valiant. Knowledge infusion: In pursuit of robustness in artificial intelligence. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008. 31
- [684] Kyriacos G Vamvoudakis and Frank L Lewis. Online actor-critic algorithm to solve the continuous-time infinite horizon optimal control problem. *Automatica*, 46(5):878–888, 2010. 68
- [685] Michiel Van Der Ree and Marco Wiering. Reinforcement learning in the game of othello: learning against a fixed opponent and learning from self-play. In *2013 IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL)*, pages 108–115. IEEE, 2013. 264
- [686] Erik Van Der Werf. *AI techniques for the game of Go*. PhD thesis, Maastricht University, 2004. 243
- [687] Erik Van Der Werf, Jos WHM Uiterwijk, Eric Postma, and Jaap Van Den Herik. Local move prediction in go. In *International Conference on Computers and Games*, pages 393–412. Springer, 2002. 243

- [688] Hado Van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. Deep reinforcement learning and the deadly triad. *arXiv preprint arXiv:1812.02648*, 2018. 192
- [689] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ, 2016. 195
- [690] Matthijs Van Leeuwen and Arno Knobbe. Non-redundant subgroup discovery in large and complex data. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 459–474. Springer, 2011. 279
- [691] Bas van Stein, Hao Wang, and Thomas Bäck. Automatic configuration of deep neural networks with ego. *arXiv preprint arXiv:1810.05526*, 2018. 253
- [692] Bas van Stein, Hao Wang, Wojtek Kowalczyk, Thomas Bäck, and Michael Emmerich. Optimally weighted cluster kriging for big data regression. In *International Symposium on Intelligent Data Analysis*, pages 310–321. Springer, 2015. 252
- [693] Gerard JP Van Westen, Jörg K Wegner, Peggy Geluykens, Leen Kwanten, Inge Vereycken, Anik Peeters, Adriaan P IJzerman, Herman WT van Vlijmen, and Andreas Bender. Which compound to select in lead optimization? prospectively validated proteochemometric models guide pre-clinical development. *PLoS one*, 6(11):e27518, 2011. 279
- [694] Joel Veness, David Silver, Alan Blair, and William Uther. Bootstrapping from game tree search. In *Advances in neural information processing systems*, pages 1937–1945, 2009. 264
- [695] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence-video to text. In *Proceedings of the IEEE international conference on computer vision*, pages 4534–4542, 2015. 179
- [696] Jos AM Vermaseren. New features of form. *arXiv preprint math-ph/0010025*, 2000. 141
- [697] Jos AM Vermaseren, Aske Plaat, Jan Kuiper, and Jaap van den Herik. Investigations with Monte Carlo tree search for finding better multivariate horner schemes. In *Agents and Artificial Intelligence: 5th International Conference, ICAART 2013, Barcelona, Spain, February 15-18, 2013. Revised Selected Papers*, volume 449, page 3. Springer, 2014. 141, 142
- [698] Alexander Vezhnevets, Volodymyr Mnih, Simon Osindero, Alex Graves, Oriol Vinyals, John Agapiou, et al. Strategic attentive writer for learning macro-actions. In *Advances in neural information processing systems*, pages 3486–3494, 2016. 257
- [699] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017. 257
- [700] Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18(2):77–95, 2002. 250
- [701] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the real-time strategy game starcraft ii. <https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/>, 2018. 54, 212, 277
- [702] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, pages 1–5, 2019. 276, 277
- [703] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. In *Advances in neural information processing systems*, pages 3630–3638, 2016. 250

- [704] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittweiser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017. 54, 236, 274, 276
- [705] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015. 274
- [706] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015. 179
- [707] J Von Neumann and O Morgenstern. Theory of games and economic behavior. 1944. 33
- [708] John Von Neumann, Oskar Morgenstern, and Harold William Kuhn. *Theory of games and economic behavior (commemorative edition)*. Princeton university press, 2007. 33
- [709] Loc Vu-Quoc. Neuron and myelinated axon, 2018. 155, 360
- [710] Niklas Wahlström, Thomas B Schön, and Marc Peter Deisenroth. From pixels to torques: Policy learning with deep dynamical models. *arXiv preprint arXiv:1502.02251*, 2015. 244
- [711] Matthieu Walraet and John Tromp. A googolplex of go games. In *International Conference on Computers and Games*, pages 191–201. Springer, 2016. 87
- [712] Hao Wang, Thomas Bäck, Aske Plaat, Michael Emmerich, and Mike Preuss. On the potential of evolutionary algorithms for replacing backpropagation for network weight optimization. In *GECCO*, 2019. 253
- [713] Hui Wang, Michael Emmerich, and Aske Plaat. Assessing the potential of classical q-learning in general game playing. *arXiv preprint arXiv:1810.06078*, 2018. 278
- [714] Hui Wang, Michael Emmerich, Mike Preuss, and Aske Plaat. Hyper-parameter sweep on alphazero general. *arxiv*, *arXiv:1903.08129*, 2019. 253
- [715] Hui Wang, Michael Emmerich, Mike Preuss, and Aske Plaat. Policy or value? loss function and playing strength in alphazero-like self-play. Technical report, *arXiv*, 2019. 253
- [716] Hui Wang, Yanni Tang, Jiamou Liu, and Wu Chen. A search optimization method for rule learning in board games. In Xin Geng and Byeong-Ho Kang, editors, *PRICAI 2018: Trends in Artificial Intelligence*, pages 174–181, Cham, 2018. Springer International Publishing. 278
- [717] Jane X Wang, Zeb Kurth-Nelson, Dhruba Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016. 250
- [718] Panqu Wang and Garrison W Cottrell. Basic level categorization facilitates visual object recognition. *arXiv preprint arXiv:1511.04103*, 2015. 248
- [719] Ziyu Wang, Frank Hutter, Masrour Zoghi, David Matheson, and Nando de Freitas. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, 55:361–387, 2016. 252
- [720] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015. 72, 195
- [721] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989. 71, 81
- [722] Manuel Watter, Jost Springenberg, Joschka Boedecker, and Martin Riedmiller. Embed to control: A locally linear latent dynamics model for control from raw images. In *Advances in neural information processing systems*, pages 2746–2754, 2015. 244
- [723] Théophane Weber, Sébastien Racanière, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. In *Advances in neural information processing systems*, pages 5690–5701, 2017. 244, 260

- [724] Eddie Weill. LeNet in Keras on Github. <https://github.com/eweill/keras-deepcv/tree/master/models/classification>. 171, 365
- [725] Jean-Christophe Weill. The NegaC* search. *ICGA Journal*, 15(1):3–7, 1992. 109
- [726] Daphna Weinshall, Gad Cohen, and Dan Amir. Curriculum learning by transfer learning: Theory and experiments with deep networks. *arXiv preprint arXiv:1802.03796*, 2018. 247, 248
- [727] Joseph Weizenbaum et al. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966. 26
- [728] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974. 156, 207
- [729] Marco A Wiering. Self-play and using an expert to learn to play backgammon with temporal difference learning. *JILSA*, 2(2):57–68, 2010. 264
- [730] Daan Wierstra, Tom Schaul, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 3381–3387. IEEE, 2008. 253
- [731] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. 68
- [732] Brandon Wilson, Austin Parker, and DS Nau. Error minimizing minimax: Avoiding search pathology in game trees. In *Proceedings of International Symposium on Combinatorial Search (SoCS-09)*, 2009. 103, 125
- [733] Ian H Witten. The apparent conflict between estimation and control—a survey of the two-armed bandit problem. *Journal of the Franklin Institute*, 301(1-2):161–189, 1976. 66
- [734] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016. 293
- [735] Anson Wong. Solving the multi-armed bandit problem. *Towards Data Science*, 2017 <https://towardsdatascience.com/solving-the-multi-armed-bandit-problem-b72de40db97c>. 137, 360
- [736] Larry Wos, Ross Overbeck, Ewing Lusk, and Jim Boyle. Automated reasoning: introduction and applications. 1984. 281, 282
- [737] Dominik Wujastyk. *The roots of ayurveda: Selections from Sanskrit medical writings*. Penguin, 2003. 23
- [738] Ruiyang Xu and Karl Lieberherr. Learning self-game-play agents for combinatorial optimization problems. 2019. 262, 274
- [739] Xin Yang, Yuezun Li, and Siwei Lyu. Exposing deep fakes using inconsistent head poses. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8261–8265. IEEE, 2019. 177, 361
- [740] Georgios N Yannakakis and Julian Togelius. *Artificial intelligence and games*, volume 2. Springer, 2018. 55
- [741] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999. 253
- [742] Nozomu Yoshinari, Kento Uchida, Shota Saito, Shinichi Shirakawa, and Youhei Akimoto. Probabilistic model-based dynamic architecture search. 2018. 253
- [743] Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. *arXiv preprint arXiv:1910.10897*, 2019. 250, 294, 295
- [744] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014. 249
- [745] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014. 174, 175, 265, 361

- [746] Qinsong Zeng, Jianchang Zhang, Zhanpeng Zeng, Yongsheng Li, Ming Chen, and Sifan Liu. Phoenixgo. <https://github.com/Tencent/PhoenixGo>, 2018. 295, 296
- [747] Amy Zhang, Nicolas Ballas, and Joelle Pineau. A dissection of overfitting and generalization in continuous reinforcement learning. *arXiv preprint arXiv:1806.07937*, 2018. 197, 198
- [748] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018. 197, 198
- [749] Ruixiang Zhang, Tong Che, Zoubin Ghahramani, Yoshua Bengio, and Yangqiu Song. Metagan: An adversarial approach to few-shot learning. In *Advances in Neural Information Processing Systems*, pages 2365–2374, 2018. 250
- [750] Shangtong Zhang and Richard S Sutton. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*, 2017. 192
- [751] Yuan Zhou. *Rethinking Opening Strategy: AlphaGo’s Impact on Pro Play*. CreateSpace, 2018. 13, 235
- [752] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In *Advances in neural information processing systems*, pages 1729–1736, 2008. 40, 276
- [753] Albert L Zobrist. A new hashing method with application for game playing by. Technical report, University of Wisconsin, Madison, 1970. 102
- [754] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 253

List of Figures

1.1	AlphaGo versus Lee Sedol	14
1.2	Stack of Chapters	17
1.3	Chapter Overview	18
2.1	Dartmouth AI Workshop 1956	24
2.2	AI Winters [408]	25
2.3	Geoffrey Hinton, Yann LeCun and Yoshua Bengio	26
2.4	Turing Test	27
2.5	Arpad Elo	29
2.6	15 Puzzle	32
2.7	Chess	35
2.8	Checkers	36
2.9	Shogi	37
2.10	Othello	38
2.11	Go	39
2.12	Backgammon	39
2.13	Poker	40
2.14	StarCraft	41
2.15	Claude Shannon	42
2.16	Ferranti Mark 1	43
2.17	Alan Turing	44
2.18	Christopher Strachey	45
2.19	Arthur Samuel	45
2.20	Gerald Tesauro	46
2.21	Jonathan Schaeffer	46
2.22	Marion Tinsley	47
2.23	Feng-hsiung Hsu not playing Chess	48
2.24	Garry Kasparov	49
2.25	Deep Blue Team	50
2.26	Game 6 Caro Kahn	51
2.27	Michael Buro	52
2.28	David Silver	52
2.29	Michael Bowling	53
2.30	Tuomas Sandholm	53

3.1 Reinforcement Learning [638]	58
3.2 Tic Tac Toe Game Tree	59
3.3 A Tree and a Graph; Node j is a Transposition	60
3.4 Richard Bellman	63
3.5 Recursion: Droste-effect	64
3.6 Value vs Policy in Actor-Critic	68
3.7 Model-free and Model-based methods	69
3.8 Taxi World [339]	76
3.9 Richard Sutton and Andrew Barto	82
3.10 Leslie Kaelbling	82
4.1 Search-Eval Architecture in a Tree	84
4.2 Tic Tac Toe Game Tree (part)	86
4.3 Tic Tac Toe Game Tree with values	89
4.4 Minimax Tree of Listing 4.2	89
4.5 Judea Pearl	91
4.6 Critical Tree	94
4.7 Max Solution Tree	95
4.8 Min Solution Tree	96
4.9 Alpha-beta example	100
4.10 Transposition: c4,d5,d4 or d4,d5,c4?	103
4.11 Narrow-window example	105
4.12 Barbara Liskov	110
4.13 Fabien Letouzey	115
4.14 Ken Thompson	116
4.15 Search–Eval Architecture	118
4.16 John McCarthy	124
5.1 Searching a Tree vs a Path	129
5.2 Rémi Coulom and Crazy Stone (left)	129
5.3 Sylvain Gelly	130
5.4 Monte Carlo Tree Search [130]	132
5.5 MCTS example	135
5.6 Levente Kocsis and Csaba Szepesvári	136
5.7 A Multi-Armed Bandit	137
5.8 Explore or Exploit? [735]	137
5.9 Go Boards 19×19 , 13×13 , 9×9	139
5.10 Performance of MCTS programs [230]	140
5.11 Adaptive MCTS tree [374]	141
5.12 Four Scatter Plots	142
6.1 Biological Neuron [709]	155
6.2 A Shallow Neural Network	156
6.3 Layers of Features of Increasing Complexity [397]	158
6.4 Some MNIST images	159
6.5 Three Fully Connected Hidden Layers [557]	160

6.6 Over-fitting	161
6.7 Convolutional Filters [245]	162
6.8 Max and Average 2×2 Pooling [245]	164
6.9 Convolutional Network Example Architecture [557]	165
6.10 Sigmoid and ReLU	166
6.11 Fei-Fei Li	169
6.12 Rina Dechter	170
6.13 ZFnet layers [745]	175
6.14 Residual Cell [275]	176
6.15 Deep Fake [739]	177
6.16 Deep Dream	178
6.17 RNN	179
6.18 Time-unrolled RNN	179
6.19 RNN unrolled in time [485]	180
6.20 Captioning Tasks [134]	181
6.21 RNN configurations [341]	182
6.22 LSTM [341]	182
6.23 Atari 2600 Console	188
6.24 Atari Games	190
6.25 DQN Architecture	191
6.26 Rainbow Graph: Performance over 57 Atari Games [285]	194
6.27 Cartpole Experiment	201
6.28 Jürgen Schmidhuber	207
6.29 Andrew Ng	208
6.30 Ian Goodfellow	209
7.1 DeepMind Team and Go Champions	213
7.2 "Speechless" Move 37 by AlphaGo in Game 2	214
7.3 AlphaGo Networks: [604]	217
7.4 AlphaGo Elo Rating (pale red is 4 handicap stones) [604]	220
7.5 AlphaGo Networks [604]	221
7.6 Elo Rating of AlphaGo Distributed Versions [604]	222
7.7 CPU, GPU, and TPU	223
7.8 Go Playing Strength over the Years [12]	224
7.9 A Diagram of Self-Learning	225
7.10 Levels in Self-Play	226
7.11 Monte Carlo Tree Search [130]	227
7.12 Supervised Learning and Self-Play	229
7.13 Performance of AlphaGo Zero [607]	229
7.14 AlphaGo Zero is Learning Joseki [607]	230
7.15 Elo Rating of AlphaZero in Chess, Shogi, and Go [606]	234
7.16 Reinforcement Learning	240
7.17 Chelsea Finn	250
7.18 Supervised Learning and Self-Play	251
7.19 Hierarchy in Feudal Reinforcement Learning [161]	255
7.20 A Maze with Rooms [639]	256

7.21 Sokoban	256
7.22 Model-free and Model-based methods	259
7.23 Pieter Abbeel	265
8.1 Stack of Chapters	268
8.2 Simplified Self-Play Loop	270
8.3 Intelligence in Games as Layers of Optimization	272
8.4 Michael Genesereth	279
8.5 Daniel Kahneman	280
8.6 Sarit Kraus	289
8.7 Daphne Koller	290
D.1 Games Fan Hui vs. AlphaGo	312
D.2 Games Lee Sedol vs. AlphaGo	313
D.3 Games Ke Jie vs. AlphaGo	314
E.1 Dual/Separate, Convolution/Residual	318
E.2 AlphaGo Zero Residual Block [607]	319
E.3 Comparison of Power Consumption for Training [564]	319

List of Tables

1.1	Heuristic-Sample-Generalize	16
2.1	Games	35
3.1	MDP Tuple and Reinforcement Learning in the Chapters	73
4.1	Solution Trees and Bounds	93
4.2	Critical Tree and Path	93
4.3	Search Enhancements	97
4.4	Eval Enhancements	112
4.5	Heuristic-Sample-Generalize	118
5.1	Heuristic-Sample-Generalize	130
5.2	MDP Tuple and Reinforcement Learning in the Chapters	131
5.3	MCTS Enhancements	140
6.1	Heuristic-Sample-Generalize	153
6.2	Supervised versus Reinforcement learning	184
7.1	AlphaGo, AlphaGo Zero, AlphaZero	212
7.2	Search-Eval Architecture	224
7.3	Input Planes	232
7.4	Evolution of Planning, Learning and Self-play in Game Playing . .	242
7.5	AlphaGo Hyper-parameters [607]	252
8.1	Method Matrix	270
8.2	Fast and Slow, Eval and Search	283
A.1	General Learning Environments	293
A.2	Deep Reinforcement Learning Environments	294
A.3	Self-Learning Environments	295

List of Algorithms

3.1	Value Iteration pseudo code (based on [7])	67
3.2	SARSA Taxi Example, after [339]	77
3.3	Evaluate the Optimal SARSA Taxi Result (based on [339])	78
3.4	Q-learning Taxi Example, after [339]	79
4.1	Small Tree code	87
4.2	Minimax code	88
4.3	Depth Limited Minimax	90
4.4	Alpha-Beta	98
4.5	Iterative Deepening	102
4.6	Use of Transposition Table (best action and depth not shown)	104
4.7	Scout [502, 536]	107
4.8	MTD(f) [518]	108
5.1	MCTS main loop [108]	134
5.2	Roll out MCTS in pseudo-Python [304]	146
5.3	Roll out alpha-beta in pseudo-Python [304]	146
6.1	LeNet-5 code [395, 724]	171
6.2	Cartpole code [171]	202
7.1	Self-Play Loop	225

Index

- C_p , 141, 144, 315
- α , TD learning rate, 66
- ϵ -greedy, 65
- γ , TD discount rate, 66
- λ , AlphaGo mixing parameter, 316
- *-minimax, 111
- 15-puzzle, 32
- A0G, 295
- A3C, 196
- action, 59
- active learning, 247
- actor-critic, 68, 196
- agent, 57
- Aja Huang, 213
- Alan Turing, 42
- ALE, 189, 203, 236
- Alex Krizhevsky, 172
- AlexNet, 172, 174
- alpha-beta, 97, 101, 145
- alpha-beta cut-off, 98
- alpha-beta post-condition, 106
- alpha-beta window, 97
- AlphaGo, 52, 213
 - fast roll out policy network, 218
 - slow selection policy network, 218
 - value network, 218
- AlphaGo networks, 218
- AlphaGo Zero, 221
- AlphaStar, 54, 276
- AlphaZero, 231
- AlphaZero General, 295
- Andrew Barto, 81
- Andrew Ng, 207
- Arcade Learning Environment, 189, 203, 236
- Arpad Elo, 29, 121
- Arthur Samuel, 43
- Atari, 189
- Atari 2600, 203
- auto-sklearn, 253
- AutoML, 253
- AutoWeka, 253
- backgammon, 206
- backpropagation, 156
- backtrack, 89
- bandit theory, 66
- Barbara Liskov, 110
- batch normalization, 167, 175
- Bellman, 63
- benchmarks, 203
- best-first search, 237
- Bridge, 275
- Caffe, 172
- captioning challenge, 178
- CFR, 276
- Checkers, 47
- Chess, 49
- Chinook, 47, 243
- ChipTest, 49
- Christopher Strachey, 43
- CIFAR, 169
- Claude Shannon, 42
- CNN, 161, 170
- connectionist AI, 30
- continuous action space, 258
- convergence, 193
- convolutional network, 161, 170
- CrazyStone, 130
- critical path, 93
- critical tree, 95
- Csaba Szepesvári, 136

- curriculum learning, 230
curriculum learning, 245, 246
- dan (Go-rank), 140
Daphne Koller, 289
darkforest, 243
data augmentation, 167
David Silver, 51, 213
DDQN, 195
deadly triad (approximation, bootstrapping, off-policy learning), 184
decision trees, 208
Deep Blue, 48, 49
deep dreaming, 178
deep fake, 177, 178
deep learning, 158
DeepChess, 264
DeepStack, 54
Demis Hassabis, 213
distributional DQN, 196
Donald Michie, 236
DOTA 2, 274
double descent phenomenon, 197
DQN, 182
DQV learning, 196
dropouts, 167
Drosophila Melanogaster, 28, 274, 277
dueling DDQN, 195
dynamic programming, 63
- early stopping, 168
ELF, 236, 295
Elmo, 232
Elo, 29, 121
Elo rating, 121
end-game database, 116
end-to-end learning, 154
environment, 57
epoch, 156
evolutionary strategies, 253
ExIt, 244
Expectimax, 111
experience replay, 191
Explainable AI, 260
explainable AI, 236, 258
exploitation, 65
- exploration, 65
- Fabien Letouzey, 114
Fan Hui, 213
feature discovery, 154
Fei-Fei Li, 169
Feng-hsiung Hsu, 48
Ferranti Mark 1, 42
Feudal reinforcement learning, 255
few-shot learning, 249
forward pruning, 109
Fruit (Letouzey's Chess program), 114
function approximation, 152
- GAN, 177, 248
Garry Kasparov, 48, 49
General Game Playing, 277
generalization, 197
generative adversarial net, 177, 248
Geoffrey Hinton, 25, 158, 172
Gerald Tesauro, 44
GGP, 222, 277
Giraffe, 244
GNU Go, 52
Go, 52, 299
GoogLeNet, 175
GPI, 68
GPU, 172
GVGP, 278
Gym, 74, 203, 236
- heuristic evaluation, 91
hidden Markov model, 208
hierarchical reinforcement learning, 257
history heuristic, 110
HMM, 208
holdout validation set, 168
Homo Economicus, 281
Homo Ludens, 281
hyper-parameter optimization, 252
- Ian Goodfellow, 208
ImageNet, 169
imperfect information, 274
implicit planning, 259
inception, 175

- interpolation, 197
- irace, 253
- iterative deepening, 101
- Jürgen Schmidhuber, 206, 207
- Johan Huizinga, 281
- John McCarthy, 125
- Jonathan Schaeffer, 46
- k-means, 208
- Kalman filters, 208
- Ke Jie, 213
- Ken Thompson, 116
- keras, 170, 172, 198
- killer move, 110
- Knightcap, 264
- knowledge representation, 258
- kyu (Go-rank), 140
- Lee Sedol, 213
- Leela, 296
- LeNet-5, 170
- Leslie Kaelbling, 81
- Levente Kocsis, 136
- Libratus, 54
- lifelong learning, 246
- lock free, 316
- Logistello, 51, 264
- London, England, 311
- loss function AlphaGo Zero, 317
- LSTM, 178, 181
- Macros, 256
- MAML, 249
- Mario, 236
- Marion Tinsley, 46
- Markov, 58
- max pooling, 164
- max solution tree, 124
- MAXQ, 257
- MCTS, 130, 131
 - Backup, 133
 - Expand, 133
 - Playout, 133
 - Select, 132
- meta-learning, 249
- Michael Bowling, 53
- Michael Buro, 51
- Michael Genesereth, 278
- min solution tree, 95
- minimax, 89
- MNIST, 169, 170
- model-based, 69, 259
- model-free, 69, 259
- MoGo, 130
- Monte Carlo Tree Search, 131
- Montezuma's Revenge, 189, 203, 256
- MTD(f), 108, 122
- MTD-bi, 109
- MuJoCo, 203
- multi agent reinforcement learning, 258
- multi armed bandit, 137
- multi-task learning, 228, 249
- MuZero, 259
- neural network, 152
- NeuroChess, 243
- Neurogammon, 206, 243
- NeuroGo, 206, 243
- nevergrad, 253
- noisy DQN, 196
- nudging, 281
- null window, 104
- null-move, 113
- odd even effect, 114
- off-policy learning, 70
- on-policy learning, 70
- one-shot learning, 249
- OpenAI Gym, 74
- OpenGo, 295
- OpenSpiel, 55
- Options, 256
- Othello, 51, 264
- over-fitting, 153, 167, 197
- P-UCT, 315
- Pachi, 218
- parallelism, 254
- pathology, 103, 114
- pattern database, 116, 133
- PCA, 208

- PhoenixGo, 296
piece-square tables, 114
Pieter Abbeel, 265
PlaNet, 259
Pluribus, 54, 276
Poker, 53, 264, 275
policy function, 60
policy iteration, 67
Policy-function, 60
principal component analysis, 208
prioritized DDQN, 195
Proof Number Search, 111
Python, 297
- Q-function, 61
Q-learning, 74, 78
quiescence search, 113
- R-CNN, 176
Rémi Coulom, 130
Rainbow paper, 194
random forests, 208
RAVE, 143
recurrent neural network, 178
regularization, 167
REINFORCE, 68, 218
reinforcement learning, 57
ReLU, 166
reproducibility, 236
residual net, 175
ResNet, 175, 228
reward, 59
Richard Sutton, 81
Rina Dechter, 170
RNN, 178
robotics, 258
roll out, 145
- sample efficiency, 71, 254
Sarit Kraus, 289
SARSA, 74, 76
scikit-learn, 253
Scout, 106
search tree, 90
search-eval architecture, 42, 85
self-learning, 223, 224
- self-paced learning, 246
self-play, 224
Seoul, Korea, 311
SGD, 156
shared weights, 163
Shogi, 232
SimPLe, 259
SMAC, 253
Soft-max, 165
solution tree, 124
sparse reward, 258
SSS*, 125, 145
stable baselines, 204
StarCraft, 40, 54, 236, 274, 276
state, 59
state space complexity, 87
Stockfish, 51, 232
strategic game, 39
strong AI, 236
Sunfish, 120
supervised learning, 153, 243
support vector machine, 208
SVM, 208
Sylvain Gelly, 130
symbolic AI, 30
- tablebase, 116
Tabula Rasa, 221
tactical game, 36
TD, 66
TD-Gammon, 44, 187, 206, 243
temporal difference, 66, 243
TensorFlow, 172
test set, 153
Theano, 172
Tic Tac Toe, 86
Torch, 172
TPU, 216, 317
training set, 153
transfer learning, 248, 249
transposition table, 102
Tuomas Sandholm, 53
Turing test, 25
- UCI, 169
UCT, 136

under-fitting, 153
value function, 60
value iteration, 67
Value-function, 61
vanishing gradients, 165
VGG, 174
virtual loss, 316
Volodymyr Mnih, 152
weight-sharing, 163
Wuzhen, China, 311
XAI, 236, 260
Yann LeCun, 25, 158
Yoshua Bengio, 25, 158
zero sum game, 34
ZFnet, 174
Zugzwang, 113, 114