

Algorithms for Adaptive Game-playing Agents

*A dissertation submitted in compliance with the requirements
for the degree of Doctor of Philosophy*

by

Niels Orsleff Justesen

Supervisor:

Sebastian Risi

IT University of Copenhagen

Submitted 31st of August, 2019

Abstract

Several games have been promoted by researchers as key challenges in the research field of Artificial Intelligence (AI) through the years, with the ultimate goal of defeating the best human players in these games. Recent developments in deep learning have enabled computers to learn strong policies for many games, where previous methods have fallen short. However, the most complex games, such as the Real-time Strategy (RTS) game *StarCraft* (Blizzard Entertainment, 1998), are still not mastered by AI. We identify three properties of adaptivity that we believe are required to fully master the most difficult games with AI. These properties are: (1) intra-game adaptivity: the ability to adapt to opponent strategies within a game, (2) inter-game adaptivity: the ability to intelligently switch strategy in-between games, and (3) generality: the ability to generalize to many different, and most likely unseen, variations (such as different levels). We analyze the shortcomings of state-of-the-art game-playing algorithms in regards to adaptation and present novel algorithmic approaches to each property. Several of the presented approaches also attempt to overcome the difficulty of learning adaptive policies in games with sparse rewards. The main contributions in this dissertation are: (a) a continual evolutionary planning algorithm that performs online adaptive build-order planning in StarCraft, (b) an imitation learning approach to intra-game adaptive build-order planning in StarCraft, resulting in the first (to the best of our knowledge) neural-network-based bot that plays the full game, (c) a novel imitation learning method for learning behavioral repertoires from demonstrations, which allows for inter-game adaptivity, (d) an automatic reward shaping technique for reinforcement learning that automatically assigns feedback values based on the temporal rarity of pre-defined events, that works as a form of curriculum learning and regularization technique to avoid overfitted behaviors in games with sparse rewards, (e) a new reinforcement learning framework that incorporates procedural content generation to generate new training levels each episode that get progressively harder as the agent improves, which is shown to overcome sparse rewards and increase the generality of the learned policy, (f) a pragmatic way of evaluating the fairness of game competitions between humans and AI that further highlights the importance of adaptation, and (g) a new challenge and competition for AI that is based on the board game Blood Bowl, which is orders of magnitude more complex than the game go and requires a high level of

generality. These contributions bring a new perspective on the AI challenge of playing complex games that has a focus on adaptation. We believe this perspective is crucial to achieving strong and robust game-playing AI. Our contributions may potentially have an impact on many important real-world problems beyond games, such as robotic tasks in changing environments with complex interactions that require a high level of adaptivity.

Resumé

Spil har været et uundværligt testmiljø indenfor kunstig intelligens siden forskningsfeltets oprindelse i 1956. Forskere i feltet har gennem årene promoveret adskillige spil som vigtige udfordringer, hvor det endelige mål er at slå de allerbedste spillere med et computerprogram. Nye fremskridt indenfor maskinlæring har for nyligt gjort det muligt at opnå menneskeligt niveau med kunstig intelligens i flere spil, hvor tidligere metoder har fejlet. Dog er de mest komplekse spil, såsom *StarCraft* (Blizzard Entertainment, 1998), stadig ikke mestret af kunstig intelligens. Vi identificerer tre tilpasningsevner, som vi mener er en forudsætning for, at kunstig intelligens kan mestre de mest komplekse spil. Disse tilpasningsevner er: (1) intra-spil-tilpasning: evnen til at justere sin strategi i løbet af et spil, (2) inter-spil-tilpasning: evnen til at skifte strategi imellem spil, og (3) generalitet: evnen til at generalisere til mange forskellige, og typisk ukendte, varianter af samme spil. Vi undersøger begrænsninger ved state-of-the-art spilalgoritmer i forhold til tilpasning og præsenterer nye algoritmer, som kan opnå de nævnte tilpasningsevner. Derudover, forbedrer flere af metoderne også læring i spil med få gevinster. De primære bidrag i denne afhandling er: (a) en *continual evolutionary planning* algoritme, der løbende planlægger den fremtidige build-order i StarCraft, (b) en *imitation learning* metode til intra-spil tilpassende build-order planlægning i StarCraft som har resulteret i den første bot, der spiller StarCraft med et neutralt netværk, (c) en ny *imitation learning* metode, som lærer et repertoire af opførsler fra demonstrationer, hvilket kan bruges til inter-spil tilpasning, (d) en automatisk *reward shaping* metode til *reinforcement learning*, der automatisk tildeler feedback-værdier baseret på frekvensen af prædefinerede hændelser, og som virker som en form for *curriculum learning* og *regularization* metode for at opnå opførsler i svære spil med forbedret generalitet, (e) et nyt *reinforcement learning* framework, som inkorporerer *procedural content generation* til at generere nye træningsbaner hver episode, som bliver gradvist sværere, når agenten forbedrer sig, hvilket forbedrer læring i spil med få gevinster, og samtidig øger agentens generalitet, (f) en pragmatisk måde at evaluere retfærdigheden af konkurrencer mellem mennesker og kunstig intelligens, som yderligere fremhæver vigtigheden af de tre nævnte tilpasningsevner, og (g) en ny udfordring indenfor kunstig intelligens som er baseret på brætspillet Blood Bowl, der i høj grad er mere komplekst end spillet go og kræver et højt niveau af generalitet. Afhandlingen bidrager

samlet set med et nyt perspektiv på udfordringerne ved at spille komplekse spil med kunstig intelligens, som fokuserer på tilpasning. Vi mener, at dette perspektiv er afgørende for at opnå stærk og robust kunstig intelligens i spil. Vores bidrag kan potentielt også få indvirkning på vigtige problemer i den virkelige verden, såsom robotopgaver i skiftende miljøer med ukendte agenter, hvor det i høj grad er nødvendigt at kunne tilpasse sin opførsel.

Acknowledgments

First and foremost, I would like to thank my supervisor Sebastian Risi for giving me advice and encouragement throughout my time as his student. I am grateful that Sebastian always provided me with valuable feedback on my work so promptly.

Special thanks also goes out to the co-authors of the papers in this dissertation, which are Sebastian Risi, Julian Togelius, Jean-Baptiste Mouret, Philip Bontrager, Ruben Rodriguez Torrado, Ahmed Khalifa, Michael S. Debus, Miguel Gonzalez Duque, Daniel Cabarcas Jaramillo, Tobias Mahlmann, Lasse Møller Uth, Christopher Jakobsen, and Peter David Moore.

I would also like to show my gratitude to Julian Togelius and Jean-Baptiste Mouret for their supervision during my two research stays abroad and to all the members of the different research groups that I have collaborated with, for their feedback, inspiring ideas, and friendship. These include the Robotics, Evolution, and Art (REAL) lab and Center for Computers Games Research (CCGR) at the IT University of Copenhagen, the Game Innovation Lab at New York University Tandon School of Engineering, and team Larsen at INRIA in Nancy. In this regard, I am also thankful for the financial support from the Elite Research travel grant from The Danish Ministry for Higher Education and Science.

Finally, I would like to thank the numerous researchers who took the time to comment on drafts of the papers in this dissertation, including Chen Tessler, Diego Pérez-Liébana, Ethan Caballero, Hal Daumé III, Jonas Busk, Kai Arulkumaran, Malcolm Heywood, Marc G. Bellemare, Marc-Philippe Huget, Mike Preuss, Nando de Freitas, Nicolas A. Barriga, Olivier Delalleau, Peter Stone, Santiago Ontañón, Tambet Matiisen, Yong Fu, Yuqing Hou, and Miguel Gonzalez Duque.

Contents

1	Introduction	1
1.1	Contributions	5
1.2	List of Papers	7
1.3	Outline	9
1.4	Notes on Style and Notation	9
2	Background	11
2.1	Neural Networks	11
2.2	Training Neural Networks with Gradient Descent	19
2.3	Reinforcement Learning	24
2.4	Evolutionary Algorithms	31
2.5	Search Algorithms	38
3	State-of-the-art in AI for Video-game Playing	45
3.1	AI for Real-Time Strategy Games	45
3.2	Playing Video Games with Deep Neural Networks	50
4	Continual Evolutionary Planning	75
4.1	Approach	76
4.2	Experiments	83
4.3	Discussion	87
4.4	Summary	88
5	Learning Build-order Planning in StarCraft from Replays	89
5.1	Approach	90
5.2	Results	95
5.3	Discussion	99
5.4	Summary	101
6	Behavioral Repertoire Imitation Learning	103
6.1	Dimensionality Reduction	106
6.2	Universal Policies	106
6.3	Approach	107
6.4	Experiments	109
6.5	Discussion	115
6.6	Summary	116
7	Rarity of Events	119
7.1	Rarity of Events	120

7.2	Experiments	121
7.3	Discussion	129
7.4	Summary	131
8	Procedural Content Generation for Reinforcement Learning	133
8.1	Related Work	134
8.2	Parameterized Level Generator	136
8.3	Procedural Level Generation for Deep Reinforcement Learning	138
8.4	Experiments	139
8.5	Exploring the Distribution of Generated Levels	143
8.6	Discussion	144
8.7	Summary	146
9	When Are We Done with Games?	147
9.1	The Blackbox Approach	149
9.2	A Prototypical Human Competition	151
9.3	Game Extrinsic and Intrinsic Factors	154
9.4	Critique of AI Achievements in Games	156
9.5	Adaptation in Fair Competitions	162
9.6	Summary	162
10	Discussion	165
10.1	Intra-game and Inter-game Adaptivity	166
10.2	Generality	168
10.3	Overcoming Sparse Rewards	168
11	Future Directions	171
11.1	MAP-Elites for Noisy Domains	171
11.2	Diverging Policies using Rarity of Events	177
11.3	Blood Bowl: A New Board Game Challenge and Competition for AI . . .	183
12	Conclusions	203
Appendix		237

List of Figures

2.1.1 A two-layered fully-connected neural network visualized as a computational graph with two input values, one hidden layer of three units using ReLU activations and one output layer with a single unit using a sigmoid activation.	13
2.1.2 A visualization of commonly used activation functions in hidden layers of artificial neural networks: sigmoid, the hyperbolic tangent (\tanh), and the rectified linear unit (ReLU).	14
2.1.3 A simple example of convolution (usually denoted by an asterisk) followed by ReLU activations and max pooling. Here, a striding of one is used, no padding, and a max pooling window size of four. The kernel thus makes four interactions with the input feature map, producing four values in the output. The first interaction between the input and the kernel is highlighted: $1 \times 1 + 0 \times 2 + 0 \times 2 + 1 \times 0 = 1$.	16
2.1.4 Two visualizations of an RNN where A is a recurrent hidden layer. (Left) in the style of a circuit diagram. (Right) as an unfolded computational graph. The illustration is from Christopher Olah's excellent visual explanation of RNNs: https://colah.github.io/posts/2015-08-Understanding-LSTMs/	17
2.1.5 A visualization an LSTM module (or cell) which has an <i>forget gate</i> , an <i>input gate</i> , and an <i>output gate</i> . The illustration is from Christopher Olah's excellent visual explanation of RNNs: https://colah.github.io/posts/2015-08-Understanding-LSTMs/	18
2.2.1 A visualization of a neural network using multi-task learning with one task-dependant output layer for each task and one general hidden layer.	24
2.4.1 A simple of example of single-point and two-point crossover operations on two parent genotypes in $\{0, 1\}^8$, each resulting in two offspring.	32
2.4.2 Final points visited in a deceptive navigation task by (a) novelty search and (b) objective-based search, starting at the green dot and the objective is the distance to the goal (the red dot). The figures are from Lehman and Stanley (2011a).	36
2.4.3 A 2D behavior-performance map of voxel-based virtual creatures found by MAP-Elites. Here, the performance metric is walking speed. The map consist of behavioral niches each with a unique solution. The map is illuminated by the fitness of these solutions. This figure is from Mouret and Clune (2015).	38
2.5.1 A search tree produced by the Minimax algorithm with the minimax values shown on each node. Here, the optimal choice is to go right, because the minimum value we can obtain is -7 and thus higher than -10 when going left. This illustration is made by Nuno Nogueira.	40

2.5.2 The four phases of one iteration in Monte Carlo Tree Search. The figure is from Chaslot et al. (2008a).	42
2.5.3 An example of uniform crossover when using evolutionary planning to evolve actions sequences for the game Hero Academy. Two action sequences (top) results in a new action sequence (bottom). This figure is from Justesen et al. (2017).	44
3.2.1 An example of a typical network architecture used in deep reinforcement learning for game-playing with pixel input. The input usually consists of a preprocessed screen image, or several stacked or concatenated images, which is followed by a couple of convolutional layers (often without pooling), and a few fully connected layers. Recurrent networks have a recurrent layer, such as LSTM or GRU, after the fully connected layers. The output typically consists of one unit for each unique combination of actions in the game, and actor-critic methods also have one for the state value $V(s)$. Examples of this architecture, without a recurrent layer and with some variations, are Mnih et al. (2013, 2015); Nair et al. (2015); Van Hasselt et al. (2016); Schaul et al. (2016); Osband et al. (2016); Mnih et al. (2016); Wang et al. (2017a); Rusu et al. (2016b); Salimans et al. (2017); Bellemare et al. (2017); Fortunato et al. (2018); Wang et al. (2016b); Hessel et al. (2018); Wu et al. (2017b); Such et al. (2017); Conti et al. (2018); Espeholt et al. (2018), and examples with a recurrent layer are Hausknecht and Stone (2015); Mnih et al. (2016); Jaderberg et al. (2017).	51
3.2.2 Influence diagram of the deep learning techniques discussed in this paper. Each node is an algorithm while the color represents the game benchmark. The distance from the center represents the date that the original paper was published on arXiv. The arrows represent how techniques are related. Each node points to all other nodes that used or modified that technique. Arrows pointing to a particular algorithm show which algorithms influenced its design. Influences are not transitive: if algorithm a influenced b and b influenced c , a did not necessarily influence c . AlphaStar and OpenAI Five are described in this section but are not on this diagram as they are very recent approaches that have not yet been peer-reviewed.	52
3.2.3 Screenshots of selected games and frameworks used as research platforms for research in deep learning.	53
3.2.4 The AlphaStar League. First, imitation learning is applied on human demonstrations. Then, several iterations of reinforcement learning is applied where several new agents are created as copies, assigned different reward functions and then matched against each other. This figure is from (Vinyals et al., 2019).	69
4.0.1 <i>Continual Online Evolutionary Planning</i> (COEP) continually evolves future build orders while UAlbertaBot executes the best one found so far.	76
4.1.1 Two-point crossover for two parent build orders and the resulting offspring. Notice that the build in the bottom right corner remains in the genotype but becomes inactive because its requirements are no longer met.	78

4.1.2 A build order with ten builds, which is manipulated by the four mutation operators. Builds are highlighted (red) if they are changed during an operation. (a) Shows the initial build order, (b) the result of a clone mutation from index 5 to 8, (c) a swap mutation on index 1 and 3, which swaps the two builds, (d) an add mutation on index 7, which adds a dragoon and recursively adds its requirements first and (e) a remove mutation at index 2 that moves the build to the end of the build order.	79
4.2.1 The average fitness over generations for Online Evolutionary Planning using a different mutation operator. Opaque coloring shows standard deviations.	85
4.2.2 A visualization of Continual Online Evolutionary Planning's (COEP) ability to perform successful intra-game adaptation by continually adjusting the Protoss build order in-game against the built-in Terran bot. The upper plot shows the number of zealots, dragoons, marines and firebats over time and the lower plot shows the highest fitness in the population. Green vertical lines indicate when the game state was updated. The four screenshots in the top show critical situations in the game. Early in the game the bot observes a group of Terran marines and continues to produce zealots to counter them. Shortly after, these zealots fight against a large group of Terran firebats and many zealots die. COEP quickly adapts its strategy to switch production to dragoons as they are superior to firebats. A video of this game can be found here: https://youtu.be/SCZbDpIaqmI	86
5.1.1 An overview of the data preprocessing that converts StarCraft replays into vectorized state-action pairs. (a) shows the process of extracting data from replay files into an SQL database, which was done by (Robertson and Watson, 2014). (b) shows our extended data processing that first extracts events from the database into files (c) containing builds, kills and observed enemy units. All events are then run through a forward model to generate vectorized state-action pairs with normalized values (d).	92
5.1.2 Neural Network Architecture. The input layer consists of a vectorized state containing normalized values representing the number of each unit, building, technology, and upgrade in the game known to the player. Only a small subset is shown on the diagram for clarity. Three inputs also describe the player's supplies. The neural network has four hidden fully-connected layers with 128 units each using the ReLU activation function. These layers are followed by an output layer using the softmax activation function and the output of the network is the prediction of each build being produced next in the given state.	94
5.2.1 The prediction of the next build being a Nexus (a base expansion) predicted by the trained neural network. Each data point corresponds to one prediction from one state. These states have only one Nexus and are taken from the test set. The small spike around 11 and 12 probes shows that the network predicts a fast expansion build order if the Protoss player has not build any gateways at this point.	97

5.2.2 The opening build order learned by the neural network when playing against the built-in Terran bot (the build order also depends on the enemy units observed). The number next to each build icon represents the probability of the build being produced next, and points on the timescale indicate when the bot requests the network for the next build. In this example the network follows the greedy strategy, always picking the build with the highest probability.	98
6.0.1 Behavioral Repertoire Imitation Learning (BRIL) trains a policy $\pi(s, b)$ supervised on a data set of state-actions pairs augmented with behavioral descriptors in \mathbb{R}^2 for each demonstration. When deployed, a system can adapt its behavior by modulating b . High-dimensional behavioral spaces can be reduced using dimensionality reduction, as low-dimensional behavioral descriptions allow for faster adaption.	104
6.4.1 Visualizations of the 2D behavioral space of Terran army unit combinations in 7,777 Terran versus Zerg replays. Each point represents a replay from the Terran player's perspective. The space was reduced using t-SNE. (a) The data points are illuminated (black is low and yellow is high) by the ratio of Marines, Marauders, Hellions, or Siege Tanks produced in each game. (b) 62 clusters found by DBSCAN. Cluster centroids are marked with a circle and the cluster number and outliers are black. The noticeable cluster 2 has no army units. (c) The similarity between the behaviors of the human players and our approach with four different feature inputs, corresponding to the coordinates of centroids of cluster 10, 11, 30, and 32. The behavior of our approach is averaged over 100 games against the easy Zerg bot and its nearest human behavior is marked with a star. The behavior of the learned policy can be efficiently manipulated to change its behavior. Additionally, we can control the behavior such that it resembles the behavior of a human demonstration.	111
6.4.2 Screenshots of typical army compositions produced by our trained BRIL policy with behavioral features corresponding to the centroids of cluster 10, 11, 30 and 32. BRIL (C10) executes early timing pushes with Hellions and Cyclones, BRIL (C11) is aggressive with Marines only, BRIL (C30) creates mixed armies with many Marines and Siege Tanks, and BRIL (C32) also creates mixed armies but with less Marines and more Widow Mines.	114
7.2.1 The five VizDoom scenarios. Scenarios with multiple spawning positions randomly select one of them at the start of an episode. The episode ends when the goal armor, which only appears in <i>My Way Home</i> and <i>Deadly Corridor</i> , is picked up. The agent periodically loses health when standing on acid floors.	123
7.2.2 From top-left to bottom-right: Screenshot from <i>Deathmatch</i> , <i>My Way Home</i> , <i>Health Gathering Supreme</i> , and <i>Deadly Corridor</i> . Notice that in some scenarios the agent cannot shoot. The scenario <i>Health Gathering</i> is similar to <i>Health Gathering Supreme</i> but without walls within the room.	124

7.2.3 The reward per episode of A2C and A2C+RoE during training in five VizDoom scenarios (smoothed). A2C is trained from the environment's extrinsic reward while A2C+RoE uses our proposed method without access to the reward. The drop in performance seen in the <i>My Way Home</i> scenario is discussed in-depth in Section 7.2.4.1.	126
7.2.4 Episodic mean occurrence during training for a subset of the event types in the five VizDoom scenarios. Notice the last spike in the <i>My Way Home</i> scenario with A2C+RoE, in which the policy ignores the final goal (armor pickup) to prioritize continuous movement around the maze.	126
7.2.5 Heat maps showing the proportional time spent at each location on the map in the <i>Deathmatch</i> scenario and its five variations. The values are based on evaluating the two trained policies 100 times each and clipped at 0.025. The heat maps show that the A2C-policy prefers to stay near the plasma gun, even in the map variations where it is not present, while the A2C+RoE-policy has learned distinct behaviors for each weapon type. The results in Table 7.2.1 shows that the A2C+RoE-policy is able to reach high scores in these variations event though it was never trained on them.	128
8.2.1 Procedurally generated levels for Solarfox, Zelda, Frogs, and Boulderdash with various difficulties between 0 and 1. For each game, human-designed levels are shown as well.	137
8.4.1 Smoothed mean scores and level difficulties during training across five repetitions of Progressive PCG in Zelda, Solarfox, Frogs, and Boulderdash. One standard deviation is shown in opaque. *Only three repetitions of PPCG and one of PCG 1 for Solarfox.	141
8.5.1 Visualization of the level distributions and how they correlate to human-designed levels (white circles). Levels were reduced to two dimensions using PCA and clustered using DBSCAN ($\epsilon = 0.5$ and a minimum of 10 samples per cluster). Outliers are black and centroids are larger.	144
9.2.1 A prototypical human vs. human competition consisting of one or more series between two teams or individuals. Each series consists of multiple game instances of the same orthogame.	152
9.4.1 A typical AI vs. human competition consisting of one series between two teams or individuals. The series often consist of identical game instances, or a limited set of game instances, of the same orthogame.	161
11.1.1 The <i>corrected</i> collection size, total normalized quality, and the mean number of elite evaluations for (a) Rastrigin with noisy performance measures and (b) feature measures, and (c) for the BipedalWalker.	175
11.1.2 Examples of <i>estimated</i> and <i>corrected</i> performance-behavior maps for (a) Rastrigin with noisy performance measures and (b) feature measures, and (c) for the BipedalWalker. When behavioral measures are noisy, we can see how solutions drifts to areas of higher performance over time. This effect is smaller when using more evaluations.	176

11.2.1Visualizations of an expanding archive with behavioral feature dimensions f_1 , f_2 , and f_3 . (a) A new solution (the green dot) must compete with neighboring elites (red dots) to enter the archive. The size of the cell (dotted lines) surrounding the new solution is determined by a constant factor of each dimension's length. (b) The length of f_2 is increased as a new solution (green dot) exceeds the previous bound. Because the size of the feature space is altered, the size of the future cells surrounding new solutions is also changed.	179
11.2.2A dimensionality-reduced behavioral space using PCA of the policies in the archive. The colors correspond to the different instances of A2C which are the same on the plots above. The size of the dots represent the fitness of each solution.	182
11.3.1The game board in FFAI after both teams have set up. The blue team just kicked the ball to the red team and assumed a defensive cover formation, while the red team is in an offensive wedge formation, protecting the wings against blitzing opponents.	186
11.3.2Effects of tackle zones on different dice rolls visualized in FFAI. (a) The red Lineman can block one of two blue Linemen. When blocking the top-most blue Lineman, two block dice are used due to the two assisting red Blitzers in the top, while it only gets one block die when blocking the bottom-most Lineman. (b) The red Catcher can move to seven adjacent free squares. Since it is already in a blue player's tackle zone, a Dodge roll is required. The player has $AG = 3$, which makes the dodge successful on a roll of 3+. However, this number is increased by one for each opponent tackle zone covering the target square. (c) A red human Thrower can attempt to pass the ball to four nearby team-mates. Two of these team-mates are in the <i>quick pass</i> range where a pass will be accurate on a roll of 3+ while the other two are in the <i>short pass</i> range requiring a roll of 4+. If the pass is accurate, the ball can be caught on 3+ with an additional modifier for each opponent tackle zone covering the catching player. Note that (b) and (c) show the required dice rolls after modifiers have been added.	187
11.3.3The 28 spatial feature layers in the FFAI Gym observation. Each layer has a name, which is shown above the visualization. Here, black squares represent a value of 1 and white squares represent a value of 0.	197
11.3.4Touchdowns per episode of A2C during training in the three smallest FFAI Gym environments: (a) FFAI-1-v1, (b) FFAI-1-v3, and (c) FFAI-1-v5, which features 1, 3, and 5 players on the pitch for each team. Simple renderings of each environment is shown above the plots. The agent plays against an agent that takes random actions. The touchdowns are smoothed over 200,000 steps. The red and green lines show the touchdowns per episode the Random and Endzone baselines. We see that A2C learns a policy that is better than the baselines in all three environments.	198

List of Tables

1.0.1 An overview of adaptive game-playing algorithms that are presented in this dissertation. RL refers to reinforcement learning and IL refers to imitation learning. ●: The method is capable of achieving this property. Note that imitation learning and evolutionary planning do not learn from environmental feedback and thus do not suffer from the difficulty of sparse rewards. (●): More investigation is needed to determine whether the method has this property. Macro-based BRIL imitates human behaviors including their intra-game adaptivity. However, when using fixed behavioral features in BRIL, that enables inter-game adaptation, the intra-game adaptivity may be reduced. Our experiments with Rarity of Events (RoE) suggests that RoE technique results in balanced behaviors with increased generality. MAP-based reinforcement learning with diverging policies is based on an intrinsic reward mechanism (in our case RoE) which have shown to overcome sparse rewards. However, it is yet unknown if the proposed divergence technique has either a positive or negative effect on learning in games with sparse rewards. COEP: Continual evolutionary planning. PCG/PPCG: (progressive) procedural content generation. DP: Diverging Policies. AS: Adaptive Sampling. DE: Drifting Elites.	4
3.2.1 Human-normalized scores reported with various deep reinforcement learning algorithms in ALE on 57 Atari games using the <i>30 no-ops</i> evaluation metric. References in the first column refer to the paper that included the reported results, while the last column references the paper that first introduced the specific algorithm. Note, that the reported scores use various amounts of training time and resources, thus not entirely comparable. Successors typically use more resources and less wall-clock time. *Hyper-parameters was tuned for every game leading to higher scores for UNREAL.	59
4.1.1 Unit matchup table that values how strong units are against each other which is a critical part of the heuristic applied. Values are in the range [0, 2]. 81	81
4.1.2 Upgrade and tech multipliers, which give units additional value in the heuristic.	81

4.2.1 Unit combinations of evolved build orders found by Online Evolutionary Planning after 100 generations. Results are averaged over 50 evolutionary runs. Some units are excluded from the results for brevity. Each row represents one scenario containing the Terran units on the left as well as a Protoss nexus, pylon and four probes. The Protoss units on the right are the average unit combination of the evolved build orders. For each unit type, the average count as well as the standard deviation is shown. The main result is that by following the implemented heuristics, Online Evolutionary Planning is able to evolve build orders that can effectively counter the opponent's strategy.	83
4.2.2 Number of wins, draws and losses by Continual Online Evolutionary Planning (COEP) against each of the three races controlled by the built-in bot in StarCraft. The bottom row shows results of COEP with a random fitness function.	85
4.2.3 Number of wins, draws and losses by Continual Online Evolutionary Planning (COEP) against three scripted Protoss opening strategies performed by UAlbertaBot.	86
5.2.1 The top-1, top-3 and top-10 error rates of trained networks (averaged over five runs) with different combinations of inputs. (a) is the player's own material, (b) is material under construction, (c) is the progress of material under construction, (d) is the opponent's material and (e) is supply. The input layer is visualized in Figure 5.1.2. <i>Probe</i> is a baseline predictor that always predicts the next build to be a probe and <i>Random</i> samples from a uniform distribution. The best results (in bold) are achieved by using all the input features.	96
5.2.2 The win percentage of UAlbertaBot with the trained neural network as a production manager against the built-in Terran bot. The probabilistic strategy selects actions with probabilities equal to the outputs of the network while the greedy network always selects the action with the highest output, and random always picks a random action. The blind probabilistic network does not receive information about the opponent's material (inputs are set to 0). UAlbertaBot playing as Protoss with the scripted dragoon rush strategy wins 100% of all games against the built-in Terran bot. . .	97
5.2.3 The average number of different unit types produced by the two different action selection strategies against the built-in Terran bot. The results show that the greedy strategy executes a very one-sided unit production while the probabilistic strategy is more varied.	98
6.4.1 Test accuracy and loss for Imitation Learning (IL), BRIL, and IL trained on clusters 10, 11, 30 and 32. Results show no significant difference between the IL and BRIL in terms of prediction accuracy. BRIL is, however, able to express multiple behaviors based on the additional input (see Table 6.4.2).	112

6.4.2 Results in StarCraft using Imitation Learning (IL) on the whole training set, IL on individual clusters (C10, C11, C30, and C32), Behavioral Repertoire Imitation Learning (BRIL) with fixed behavioral features corresponding to centroids in C10, C11, C30, and C32. Additionally, results are shown in which UCB1 selects between the four behavioral features in-between games. Each variant played 100 games against the easy Zerg bot. These results demonstrate that by using certain behavioral features, the BRIL policy outperforms the traditional IL approach as well as IL on behavioral clusters.	113
6.4.3 Results in StarCraft using Imitation Learning (IL) on the whole training set, IL on individual clusters (C10, C11, C30, and C32), Behavioral Repertoire Imitation Learning (BRIL) with fixed behavioral features corresponding to centroids in C10, C11, C30, and C32. Additionally, results are shown in which UCB1 selects between the four behavioral features in-between games. Each variant played 100 games against the easy Zerg bot. The nearest demonstration in the entire dataset was found based on the bot's mean behavior (normalized army unit combination) and the distance to each cluster centroid are shown. These results demonstrate that the behavioral features of the learned BRIL policy can be manipulated and controlled towards a desired behavior.	114
7.2.1 Shown are average scores based on evaluating the best policies found for A2C and A2C+RoE 100 times each. The best results are shown in bold. The five last rows show how the policies that were trained on the original <i>Deathmatch</i> scenario generalize to five variations where only one weapon type is available. Standard deviations are shown for each experiment and two-tailed p-values from unpaired t-tests.	127
8.4.1 Test results of A2C under different training regimens: a single human-designed level (<i>Level 0</i> and <i>Level 4</i>), several human-designed levels (<i>Level 0-3</i>), procedurally generated levels with a fixed difficulty (<i>PCG 0.5</i> and <i>PCG 1</i>), and <i>PPCG</i> that progressively adapts the difficulty of the levels to match the agent's performance. <i>Random</i> refers to results of an agent taking uniformly random actions and <i>Max</i> shows the maximum possible score. Scores are in red if the training level is the same as the test level. The best scores for a game, that is not marked red, are in bold. *Only three repetitions of PPCG and one of PCCG 1 were made for Solarfox so far.	140
11.3.1The positional players allowed on a Human team. MA=Movement Allowance, ST=Strength, AG=Agility, AV=Armour value. The Ogre is a special type of player called <i>Big Guys</i> and its skills are not explained here.	189
A3.1 Experimental configurations for A2C and A2C+RoE. 16 worker threads were used in <i>Deathmatch</i> .	238

Chapter 1

Introduction

Several games have been promoted by researchers as key challenges in the research field of Artificial Intelligence (AI) through the years, with the ultimate goal of defeating the best human players in these games. Traditional board games stood the test at first, including checkers, backgammon, and chess. Shannon (1988) wrote in his seminal paper on computational chess, that “chess is generally considered to require ‘thinking’ for skillful play; a solution of this problem will force us either to admit the possibility of a mechanized thinking or to further restrict our concept of ‘thinking’” and similarly Simon and Chase (1988) wrote “if one could devise a successful chess machine one would seem to have penetrated the core of human intellectual endeavor”. After IBM’s computer program Deep Blue defeated the chess world champion Gerry Gasparov in 1997 (Campbell et al., 2002) we do not yet seem to have “penetrated the core of human intellectual endeavor”, while we certainly have become aware of the opportunities of AI. Since Gasparov’s defeat, there has been a race toward mastering even more complex games with AI. Recent developments in machine learning, using deep artificial neural networks (LeCun et al., 2015), have allowed computer algorithms through reinforcement learning and imitation of human demonstrations to master several advanced games, including go (Silver et al., 2016) and several Atari arcade games given only the raw screen pixels as input (Mnih et al., 2015). Researchers have considered the real-time strategy (RTS) game StarCraft (Blizzard Entertainment, 1998) to be one of the hardest games for computers to play (Čertický and Churchill, 2017; Yannakakis and Togelius, 2018a), which could suggest that this game is the final grand challenge for AI in games before moving on to more difficult real-world

problems. The AI challenge of mastering RTS games, such as StarCraft, was first proposed by Buro (2003) and later, large research companies such as Facebook (Synnaeve et al., 2016), DeepMind (Vinyals et al., 2017), Samsung¹, and Tencent (Sun et al., 2018) have pursued the goal of developing algorithms that can master StarCraft. Additionally, the corporation OpenAI has attempted to master the game Dota 2 (OpenAI, 2017) that shares many of the same challenges as StarCraft.

In this dissertation, the focus lies on some of the remaining challenges of developing game-playing AI for the most difficult games, such as StarCraft and Dota 2. These games are difficult for AI to master for several reasons. They have large state and action spaces as well as sparse feedback signals (rewards), making it difficult for both search algorithms and reinforcement learning to explore them adequately. In this dissertation, I argue (in Chapter 9) that before AI can fully master a game, it must compete in a setting similar to human competitions which typically involves series of matches with different variations (different maps/levels, heroes/characters, etc.) and against different players/teams. Such a setting requires players to adapt in several ways, and I believe this focus on adaptation brings a new perspective on the challenge of mastering complex games with AI and highlights the shortcomings of current algorithms. I have identified the following three ways in which players must be able to adapt before they can master the most complex games:

- **Intra-game adaptivity:** The ability to adapt to opponent strategies within a game. Prior to the work presented in this dissertation, StarCraft bots usually followed a hard-coded strategy throughout the entire game (Ontanón et al., 2013; Churchill et al., 2016). Fixed policies like these can easily be exploited within the game by an intra-game adaptive player, e.g. a trained human player.
- **Inter-game adaptivity:** The ability to intelligently switch strategy in-between games. Complex games, including most RTS games, are usually non-transitive, meaning that if strategy A is preferred over strategy B (written $A > B$), and $B > C$, it is not necessarily true that $A > C$. The simplest example of a non-transitive game is Rock-Paper-Scissors. In non-transitive games, there are typically no dominant strategy and players must instead rely on mixed strategies, i.e. sampling from a

¹<https://github.com/TeamSAIDA/SAIDA>

distribution of known strategies. A key challenge in these games is to avoid being exploited by the opponent and at the same time to attempt to exploit the opponent’s weaknesses. Such exploitable weaknesses include being unaware of certain strategies and simply being unable to execute a strategy properly. I argue that selecting from a large repertoire of diverse and strong strategies allows for strong inter-game adaptation.

- **Generality:** The ability to generalize to many different, and most likely unseen, variations of the game. These variations include different starting positions, levels/maps, graphics, and available heroes/characters that may change in-between a series of matches. I argue that players must be able to play well in all sensible variations to fully master a game. In Chapters 7 and 8, we show how a state-of-the-art reinforcement learning algorithm overfits to small training sets of levels.

This dissertation explores new approaches to achieve artificial agents with these three properties of adaptivity in complex games with large state and action spaces and sparse rewards. The contributions are all described in the next section while I briefly highlight the main algorithmic approaches next. Table 1.0.1 gives an overview of these algorithms and their properties. I first present a variant of evolutionary planning and apply it to build-order planning as a module in a scripted StarCraft bot (Chapter 4). This system demonstrates intra-game adaptivity by continually planning high-level actions throughout the game, which are usually hard-coded in StarCraft bots. In the same setup, I present a neural-network-based approach to playing StarCraft. Here, a neural network is trained to imitate human build-order actions that naturally express human-like intra-game adaptation, while this approach alone is incapable of inter-game adaptation (Chapter 5). A third approach that I propose for the build-order task is called *Behavioral Repertoire Imitation Learning* (BRIL), which learns a large repertoire of diverse behaviors from human demonstrations (Chapter 6). Our experiments show that using BRIL, a bot can express inter-game adaptation by selecting intelligently between the learned behaviors (strategies) when playing a series of matches against the same opponent. Imitation learning alone cannot exceed human skill level or explore strategies that are not present in the demonstration set, and thus methods within reinforcement learning is explored as well. Here, I present an automatic reward shaping technique called *Rarity of Events* that

adapts the reward signals to the temporal rarity of predefined events (Chapter 7). This approach is shown to be useful to overcome sparse rewards and requires limited domain knowledge and limited tweaking. Additionally, our results suggest that the learned policies have improved generality. I also propose a new framework for reinforcement learning in games in which a new level is generated for each play-through (episode) to achieve better generalization (Chapter 8). In this framework, I show how the difficulty of the generated levels can be matched to the performance of the learning agent to overcome sparse rewards, allowing agents to master games that were otherwise too complex to learn directly. In the chapter containing ongoing work (Chapter 11), I describe a variant of MAP-Elites for noisy domains, such as games, and a new quality-diversity reinforcement learning algorithm.

Method	Properties of Adaptivity			
	Intra-game adaptivity	Inter-game adaptivity	Generality	Overcomes sparse rewards
Macro-based COEP (Chapter 4)	●			●
Macro-based IL (Chapter 5)	●			●
Macro-based BRIL (Chapter 6)	(●)	●		●
RoE (Chapter 7)			(●)	●
RL+PCG (Chapter 8)			●	
RL+PPCG (Chapter 8)			●	●
MAP-based RL + DP (Chapter 11)		●		(●)
MAP-Elites + AS + DE Chapter 11)		●		

Table 1.0.1: An overview of adaptive game-playing algorithms that are presented in this dissertation. RL refers to reinforcement learning and IL refers to imitation learning. ●: The method is capable of achieving this property. Note that imitation learning and evolutionary planning do not learn from environmental feedback and thus do not suffer from the difficulty of sparse rewards. (●): More investigation is needed to determine whether the method has this property. Macro-based BRIL imitates human behaviors including their intra-game adaptivity. However, when using fixed behavioral features in BRIL, that enables inter-game adaptation, the intra-game adaptivity may be reduced. Our experiments with Rarity of Events (RoE) suggests that RoE technique results in balanced behaviors with increased generality. MAP-based reinforcement learning with diverging policies is based on an intrinsic reward mechanism (in our case RoE) which have shown to overcome sparse rewards. However, it is yet unknown if the proposed divergence technique has either a positive or negative effect on learning in games with sparse rewards. COEP: Continual evolutionary planning. PCG/PPCG: (progressive) procedural content generation. DP: Diverging Policies. AS: Adaptive Sampling. DE: Drifting Elites.

1.1 Contributions

This dissertation takes a new perspective on the challenges of developing artificial game-playing agents in complex games. The traditional metrics used to measure the complexity of games have usually focused entirely on computational complexity, including the branching factor and the size of the state and action spaces. I suggest that several other factors are equally important, such as the league structures employed in human competitions of the game, which typically require players to adapt their playing style throughout repeated matches against multiple opponents. Another important factor is the span of different game variations, which in some games require players to compete in unseen variations of the game (e.g. on a completely new level). This dissertation investigates the challenges introduced by these factors and proposes concrete algorithmic solutions along with experimental evaluations and analyses. The list of contributions contains work from several different research collaborations, which are, however, closely connected to the three adaptive properties described in the previous section. The main contributions can be summarized as follows:

- **A continual evolutionary planning algorithm that can be used for adaptive long-term planning in real-time games.** This approach is similar to Rolling Horizon Evolutionary Algorithms with EA-shift that was independently developed by Gaina et al. (2017b). However, some algorithmic differences remain as our implementation ran the evolutionary algorithm in a separate process and received periodic state updates from a bot that used the planner's output. This algorithm demonstrates promising intra-game adaptivity in StarCraft.
- **A modular neural-network-based approach to playing StarCraft wherein a neural network takes high-level decisions that are performed by several micro-management modules.** This approach, to the best of our knowledge, is the first bot that, in collaboration with scripted modules, plays the full game of StarCraft with a neural network. This approach offers a feasible StarCraft AI approach that requires fewer computational resources compared to end-to-end learning methods. Our approach trained the neural network using imitation learning from human demonstrations while several other researchers (Tang et al., 2018; Sun et al., 2018)

later extended this approach with reinforcement learning to beat the best bot developed by Blizzard Entertainment (which is cheating in several ways).

- **A novel imitation learning method that learns a repertoire of different behaviors from a set of demonstrations.** This approach is simple and yet powerful as it allows the user to manually design a behavioral space and then train a single policy that can express multiple behaviors through a simple modulation technique. I demonstrate that this method allows for inter-game adaptation to find the optimal behavior of a modular bot against a fixed opponent. I believe this approach is also promising for automatic testing of games, as just one system could express a whole space of different players.
- **An automatic reward shaping technique that adapts to the rarity of predefined events.** Using this approach, there is no need to manually specify the reward values, something that can be difficult in many games. This technique is applied to the video game Doom and it was able to learn strong policies for scenarios with sparse rewards. Our results showed that the learned polices had mixed behaviors with superior generalization in a set of test environments.
- **An exploration of overfitting in deep reinforcement learning through the use of procedural content generation.** I show that policies learned with deep reinforcement learning, in four out of four games in the General Video Game AI framework, were unable to generalize to new levels of the same game. To deal with this problem, I present a new framework in reinforcement learning in which a new level is generated each new episode, matching the generated levels to the current training performance of the agent. This approach learned strong policies for games that could not otherwise be learned. This work shifts the focus in reinforcement learning from the learning algorithm to the challenge of procedurally generating variations of the environment to improve the generality of the agent and to overcome sparse rewards.
- **A pragmatic way of evaluating the fairness of game competitions between humans and AI that is based on a black-box view on the competitors and treats both intrinsic and extrinsic factors of the competition format.** This evaluation framework is applied to evaluate the fairness of recent game competitions

between humans and AI, including matches with AlphaStar (in StarCraft) and OpenAI Five (in Dota 2). The shortcomings, that our evaluation framework marks as violations, further highlight the importance of our focus on adaptation.

- **A new game challenge and competition for AI that is based on the board game Blood Bowl.** This board game is several orders of magnitude more complex than previously studied board games in AI as it has a large and complicated observation space, a variable action space and immensely sparse reward signals. Additionally, to play the full game in human-like league or tournament constructs, a policy would need to adapt to millions of different team configurations, requiring a very high-level of generality.

1.2 List of Papers

This dissertation is based on ten papers. Eight of the papers were published in conference or workshop proceedings, one paper was published in a scientific journal, and one pre-print paper titled *Learning a Behavioral Repertoire from Demonstrations* has not yet been submitted for peer-review. All nine published papers have been peer-reviewed following high academic standards. Pre-print versions of all ten papers are made available on my website www.njustesen.com/publications. The nine peer-reviewed and accepted papers included in this dissertation are:

- Niels Justesen, and Sebastian Risi. *Continual Online Evolutionary Planning for In-Game Build Order Adaptation in StarCraft*. Proceedings of the Genetic and Evolutionary Computation Conference. ACM, 2017.
- Niels Justesen, and Sebastian Risi. *Learning Macromanagement in StarCraft from Replays using Deep Learning*. IEEE Conference on Computational Intelligence and Games (CIG). IEEE, 2017.
- Niels Justesen, and Sebastian Risi. *Automated Curriculum Learning by Rewarding Temporally Rare Events*. 2018 IEEE Conference on Computational Intelligence and Games (CIG). IEEE, 2018.
- Niels Justesen, Sebastian Risi, and Julian Togelius. *Blood Bowl: The Next Board*

Game Challenge for AI. Foundations of Digital Games. ACM, 2018.

- Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. *Illuminating Generalization in Deep Reinforcement Learning through Procedural Level Generation.* NeurIPS Workshop on Deep Reinforcement Learning, 2018.
- Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. *Deep Learning for Video Game Playing.* IEEE Transactions on Games, 2019.
- Niels Justesen, Sebastian Risi, and Jean-Baptiste Mouret. *MAP-Elites for Noisy Domains by Adaptive Sampling.* Proceedings of the Genetic and Evolutionary Computation Conference Companion. ACM, 2019.
- Niels Justesen, Michael S. Debus, and Sebastian Risi. *When Are We Done with Games?.* 2019 IEEE Conference on Games (CoG). IEEE, 2019.
- Niels Justesen, Lasse Møller Uth, Christopher Jakobsen, Peter David Moore, Julian Togelius, and Sebastian Risi. *Blood Bowl: A New Board Game Challenge and Competition for AI* 2019 IEEE Conference on Games (CoG). IEEE, 2019.

This dissertation has a chapter that is based on work that is not yet peer-reviewed and just recently has been uploaded as pre-print to www.arXiv.org:

- Niels Justesen, Miguel Gonzalez Duque, Daniel Cabarcas Jaramillo, Jean-Baptiste Mouret, Sebastian Risi *Learning a Behavioral Repertoire from Demonstrations.* arXiv preprint arXiv:1907.03046, 2019.

It should be noted that the paper *Playing Multi-Action Adversarial Games: Online Evolution vs. Tree Search* (Justesen et al., 2017), not listed above, was published in the IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG) journal in 2017 (during my time as a Ph.D. candidate) but it has been excluded from this dissertation as it contains contributions that have already been credited in my master thesis (Justesen, 2015).

1.3 Outline

The introduction of this dissertation is followed by a background chapter that is rather inclusive to encompass the theory and methods that are applied in the main chapters. The background chapter describes neural networks, search algorithms, and optimization algorithms used for game-playing. The background is followed by a review of the state-of-the-art within AI for RTS games as well as video-game playing approaches that rely on deep neural networks. The main chapters are structured such that they each represent one publication, while their background section and most of their related work sections have been lifted into the main background chapter of this dissertation. The main chapters are followed by a discussion and a chapter describing future directions, which includes two papers and some preliminary results of ongoing projects. This dissertation finally concludes on the main contributions.

1.4 Notes on Style and Notation

In this introduction, I have mainly used “I” to refer to myself, while I will use “we” in the main chapters to acknowledge the collaborative work of the co-authors on the papers that this dissertation is based on. While the work presented has been lead by myself, I also want to acknowledge that it was done under the supervision of Sebastian Risi, and partly Julian Togelius and Jean-Baptiste Mouret.

In several chapters, I present algorithms in pseudo code, while I have put a great effort into explaining the algorithms in plain text and through simple equations. The pseudo code is thus only included to offer a precise description in case the reader is interested in specific details or wants to re-implement the algorithm. I have also strived to publish all the code of the presented experiments on my Github profile: <https://github.com/njustesen>.

This dissertation is not math-heavy, while I do use a few equations to explain the fundamentals of the algorithms in a precise way. Here, vectors and matrices are given a bold font, where matrices usually are capitalized. For example, linear combination of a weight matrix \mathbf{W} and input vector \mathbf{x} is written $\mathbf{y} = \mathbf{W}\mathbf{x}$. Scalar variables are not in bold,

such as the element at position (i, j) in \mathbf{W} , which is written $W_{i,j}$. Random variables are capitalized. Sets are written in this notation \mathcal{S} , except the set of real numbers \mathbb{R} .

In the notation used by Sutton and Barto (1998), both capital and lower-case letters are used for state-value and value functions in reinforcement learning algorithms: $V(s)$, $Q(s)$, $v(s)$, and $q(s)$, to denote whether they refer to a tabular value or not. To make this simpler, these functions are always capitalized.

Chapter 2

Background

This background chapter describes the methods and theory that are applied and extended in the main chapters of this dissertation, including neural networks, search algorithms, and optimization algorithms. This chapter should not be read as a comprehensive introduction to algorithms and machine learning, as the scope is limited to methods that are typically applied to game-playing AI.

2.1 Neural Networks

Machine learning, in general, deals with algorithms that learn to solve tasks from data. Two fundamental tasks studied in machine learning are classification and regression. In classification, the goal is to learn a mapping function $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$, segregating the input space into k categories (also called classes). In regression, the goal is to learn a mapping function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, predicting a scalar value for a given input. These mapping functions can be learned in various ways that can be distinguished between non-parametric and parametric methods. The k -nearest neighbors classification algorithm (k -NN) is an example of a non-parametric algorithm that assigns a class to an input value based on a plurality vote of the k nearest neighbors in a known data set. A classic example of a parametric function approximator is the linear regression model $f(x) = \mathbf{w}^\top \mathbf{x} + b$, where $\mathbf{x} \in \mathbb{R}^n$ is the input vector, $\mathbf{w} \in \mathbb{R}^n$ is a vector of parameters called weights, and b is an additional scalar parameter called bias or intercept. Linear models cannot

express non-linear relationship in the data and are thus very limiting in many situations. Multi-layered *artificial neural networks*, or just neural networks, is a widely popular class of non-linear parametric function approximators. This section describes different types of neural networks that are popular for various tasks including video game playing.

2.1.1 Feed-Forward Neural Networks

Artificial Neural Networks (ANNs) are models that are, as the name suggests, loosely inspired by biological neural networks, however, much simplified. Following this analogy, ANNs can thus be understood as a network, consisting of neurons that are connected by synapses that propagates a flow of information through the network. These connections can either be small or large, determining the strength of the signals that are passed on to the connected neurons. Strong connections thus put strong attention to the features represented by their input signal. ANNs can also be seen as either computational graphs of transformational operations, or mathematically as an extension to the linear regression model explained earlier.

A feed-forward neural network consists of multiple layers of transformational functions, each consisting of a linear combination of input values and weights followed by a non-linear activation function. A three-layered feed-forward neural network can be described as $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$, where $f^{(j)}$ refers to the neural network's j th layer. Note, that we here use the counting terminology suggested in Bishop (2006), and thus the input vector is not considered a layer as it does not involve any transformations. Each layer, except for the output layer, is called a *hidden* layer. Figure 2.1.1 shows a fully-connected feed-forward neural network with two layers.

A layer in a feed-forward neural network takes D input variables x_1, \dots, x_D equal to the output of the previous layer, where the first layer simply takes the input vector. The first step involved in forward-propagation of a neural network is to compute linear combinations of the input features and set of parameters, such that $z_j = \sum_{i=1}^D W_{ji}x_i + b_j$, where \mathbf{W} is a weight matrix and \mathbf{b} is a vector of bias parameters for this layer. The output of a layer (called *activations*) is computed by transforming z using a differentiable and non-linear function $g(\cdot)$ (called an activation function). If a linear function is used in all hidden

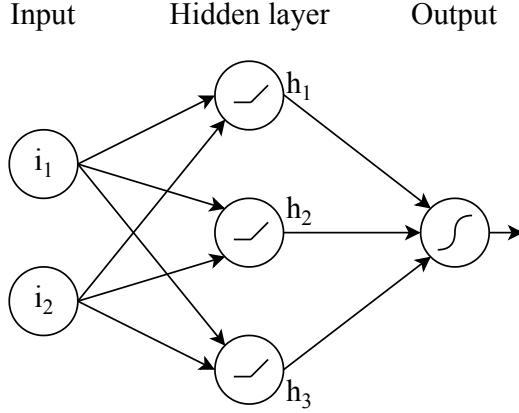


Figure 2.1.1: A two-layered fully-connected neural network visualized as a computational graph with two input values, one hidden layer of three units using ReLU activations and one output layer with a single unit using a sigmoid activation.

layers, then the neural network is a linear function, despite having multiple layers, since a function of linear functions is also linear. The forward propagation of an entire layer can also be described as a simple affine transformation between the input and the weights $f(\mathbf{x}) = g(\mathbf{W}^\top \mathbf{x} + \mathbf{b}) = [o_1, \dots, o_M]^\top$. Layers that use this form of forward propagation are called *dense* or *fully-connected*, as all input unit are densely connected to all output units.

The activation function can be any non-linear function, while simple and differentiable functions are preferred as they allow for efficient calculations of the gradient. A popular activation function is the logistic sigmoid function $S(z) = \frac{1}{1+\exp(z)}$ which squashes the value into the range $[0, 1]$. The sigmoid activation function (which is sometimes also denoted by σ) is thus useful to apply in the output layer when performing binary classification tasks. The hyperbolic tangent, $\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{\exp(2z)-1}{\exp(2z)+1}$ is another squashing function but in the range $[-1, 1]$. While logistic activation functions historically have been very popular, they run into the problem of saturation when the input is far from 0, making gradient-based learning difficult. A simpler activation function is the rectified linear unit, $\text{ReLU}(z) = \max\{0, z\}$. It is simpler because it is a piece-wise linear function with constant derivatives. ReLU thus has no saturation problem when $z > 0$ as the derivative is always 1, while gradient-based learning, for this particular activation, becomes impossible when $z < 0$, as the derivative here is 0. This issue can be mitigated by initializing biases to values larger than 0. ReLU, or variations of it, is recommended as the default activation function in hidden layers of feed-forward neural networks, while tanh, and sigmoid, for

some tasks, are used in the output layer, as well as in some recurrent neural networks (Goodfellow et al., 2016). The three activation functions that have been described are visualized in Figure 2.1.2.

The softmax activation function is a special exponential squashing function that normalizes a vector $\mathbf{z} \in \mathbb{R}^K$ into a probability distribution also in \mathbb{R}^K using $\sigma(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^K \exp(z_j)}$, such that $\sum_{i=1}^K \sigma(\mathbf{z})_i = 1$. This activation function is e.g. useful in classification tasks such that the output represents the probabilities of assigning a data example, represented by the input, to each of K classes.

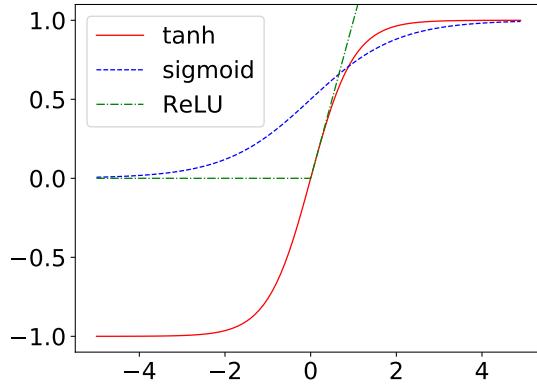


Figure 2.1.2: A visualization of commonly used activation functions in hidden layers of artificial neural networks: sigmoid, the hyperbolic tangent (tanh), and the rectified linear unit (ReLU).

It has been shown that feed-forward neural networks, with a single hidden layer with sufficiently many units and non-linear activations, are a class of universal function approximators; i.e. it has the capability of approximating any continuous function (Hornik et al., 1989). The number of units needed in the hidden layer may, however, grow exponentially with the number of inputs. Deep networks, with multiple hidden layers, allow the computations of the same units to be *reused* exponentially in terms of network depth (Montufar et al., 2014). While there exist some theoretical work on how to select the architectural hyper-parameters (Lu et al., 2017), these values are often found experimentally, e.g. using grid search or random search (Bergstra and Bengio, 2012).

2.1.2 Convolutional Neural Networks

When the data of interest is structured in a *grid*-like structure, such as 2-D images or 1-D time series, they often contain spatial relationship between values that are nearby in the grid. Here, it is thus useful to have a model that can recognize several different spatial patterns across the whole grid. For example, when the task is to detect dogs in images, it is useful to be able to recognize hair throughout the whole image. While a fully-connected neural networks can approximate any function of interest, its weights are *tied* to specific locations. Thus, a unique set of weights must be learned for every location in the grid where a local pattern might occur. *Parameter sharing* is the idea of reusing weights across the data grid and this is exactly what the convolution operation does. Convolutional neural networks (CNNs) have several layers of convolution and they have significantly improved the performance of neural networks in image classification tasks (LeCun et al., 1998; Krizhevsky et al., 2012).

A convolutional layer consist of several sparse weight matrices, called *kernels* or *filters*, that are able to detect local patterns. Both the input and output of a convolutional layer consist of several *feature maps* (grid-structured data). When convolution is applied, the kernels *stride* across the input feature maps, resulting in multiple interactions for each set of kernel weights. Each interaction computes the sum of the element-wise products between the kernel and the input area. The value in the output feature map at position (i, j) , can be formally described by the following formula (this formulation is from Goodfellow et al. (2016)):

$$S(i, j) = (\mathbf{I} * \mathbf{K})(i, j) = \sum_{m=1}^M \sum_{n=1}^N I_{i+m, j+n} K_{m,n},$$

where \mathbf{I} is two-dimensional input feature map being traversed by a $M \times N$ kernel. This process results in multiple feature layers, if multiple kernels are used. The number of kernels used, their window size, and how many values to skip when *striding*, are important hyper-parameters that can vary between layers. Additionally, one can apply padding to the input feature maps, e.g. to maintain the same dimension in the input and output.

Convolution consists of linear combinations and is thus usually followed by a non-linear

activation function. It can be useful to apply a non-parameterized pooling operation to reduce the dimensionality based on statistical summaries of values in local areas of the feature map. Max pooling express how much a pattern is represented in the input which is especially useful in classification tasks. However, the reduction also means that information about the locations is lost, which might be why it is not so common to use pooling in many control problems, such as video game-playing (see Chapter 3). Figure 2.1.3 shows a simple example of convolution followed by non-linearity and max pooling. In fact, experiments by Springenberg et al. (2014) have shown that max pooling layers can be completely replaced by convolutional layers with increased striding in many object recognition tasks. If the activation function is monotonously increasing, which most of the commonly used ones are, then the operation is commutative with max-pooling.

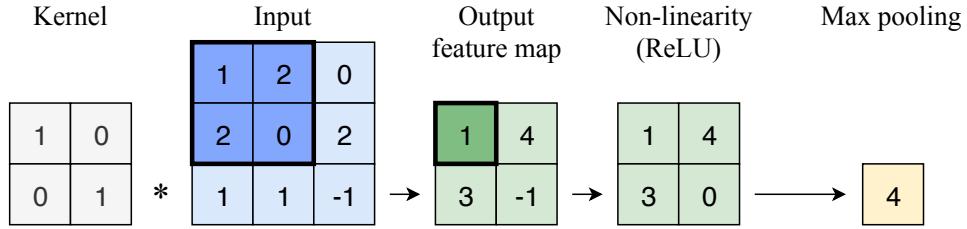


Figure 2.1.3: A simple example of convolution (usually denoted by an asterisk) followed by ReLU activations and max pooling. Here, a striding of one is used, no padding, and a max pooling window size of four. The kernel thus makes four interactions with the input feature map, producing four values in the output. The first interaction between the input and the kernel is highlighted: $1 \times 1 + 0 \times 2 + 0 \times 2 + 1 \times 0 = 1$.

2.1.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) is a class neural networks for sequential data of the form $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}$, such as text strings or sequences of images. The naive implementation to handle such data would be a fully-connected neural network that treats each element of each feature vector $\mathbf{x}^{(i)}$ as a unique input feature with its own set of parameters. This restricts the length of the sequence τ to be fixed and we run into the same issue as with fully-connected networks for spatial data as there are no parameter sharing. Humans also do not have to reconsider all past events to take decisions but instead rely on the persistent state of thoughts that are continuously updated as time goes.

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}),$$

where both $\mathbf{x}^{(t)}$ and $\mathbf{h}^{(t-1)}$ are weighted input features. This recurrent procedure allows the neural network to utilize parameter sharing and the sequence length τ is not restricted to a fixed size. This can be visualized either as a feedback loop that propagates the hidden state to itself, or the network can be *unrolled* to show the temporal propagation of hidden states in an unfolded computational graph (see Figure 2.1.4).

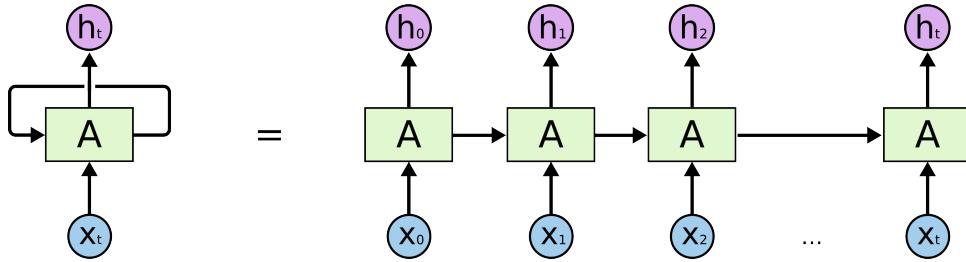


Figure 2.1.4: Two visualizations of an RNN where A is a recurrent hidden layer. (Left) in the style of a circuit diagram. (Right) as an unfolded computational graph. The illustration is from Christopher Olah’s excellent visual explanation of RNNs: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>.

Backpropagation of RNNs is basically the same as for feedforward neural networks as long as the gradient is backpropagated in the unfolded computational graph. This procedure is also referred to as *Backpropagation through time* (BPTT).

RNNs run into the problem of learning long-term dependencies as the error signal that flow through time tend to either vanish or explode exponentially. The next section will provide more details on error signals and the problem of vanishing and exploding gradients. The Long Short-Term Memory (LSTM) cell (Hochreiter and Schmidhuber, 1997) is designed to overcome these problems by introducing three sigmoid gates (visualized on Figure 2.1.5) that are parameterized by weight matrices $\mathbf{W}_f, \mathbf{W}_t, \mathbf{W}_o \in \mathbb{R}^{h \times d}$ and $\mathbf{U}_f, \mathbf{U}_t, \mathbf{U}_o \in \mathbb{R}^{h \times h}$, where d is the number of the input features and h is the number of hidden units, as well as three bias vectors $\mathbf{b}_f, \mathbf{b}_i, \mathbf{b}_o \in \mathbb{R}^h$. An LSTM layer in a neural network typically consist of multiple cells, where each cell can be treated as a hidden unit in a hidden layer. The cell operations thus operate as a layer taking a vector \mathbf{x}_t as input at time step t . A *forget gate* (f_t) determine how much of the stored cell state from the previous time step c_{t-1} to keep, an *input gate* (i_t) determines how much of the new *candidate value* (Equation 2.1.4) to add to the cell state, and an *output gate* (o_t) determines how much of the new cell

state to output and set as the cell's hidden state. Each gate is computed as follows (using the notation in (Cho et al., 2014)):

$$f_t = \sigma(\mathbf{W}_f x_t + \mathbf{U}_f h_{t-1} + \mathbf{b}_f) \quad (2.1.1)$$

$$i_t = \sigma(\mathbf{W}_i x_t + \mathbf{U}_i h_{t-1} + \mathbf{b}_i) \quad (2.1.2)$$

$$o_t = \sigma(\mathbf{W}_o x_t + \mathbf{U}_o h_{t-1} + \mathbf{b}_o) \quad (2.1.3)$$

Using these gates, a cell state c_t and a hidden state h_t can be computed:

$$c_t = f_t \circ c_{t-1} + i_t \circ \overbrace{\sigma(\mathbf{W}_c x_t + \mathbf{U}_c h_{t-1} + \mathbf{b}_c)}^{\text{Candidate value}} \quad (2.1.4)$$

$$h_t = o_t \circ \tanh(c_t) \quad (2.1.5)$$

The hidden state h_t is passed to the next layer while it is also used as recurrent input to itself at the next time step together with c_t . LSTMs thus have two states that flows through time and learns weights that determine when to store and release memory. Different variants of LSTMs exist (Gers and Schmidhuber, 2000), including the simpler variant called the Gated Recurrent Unit (GRU) that only has two gates (Cho et al., 2014).

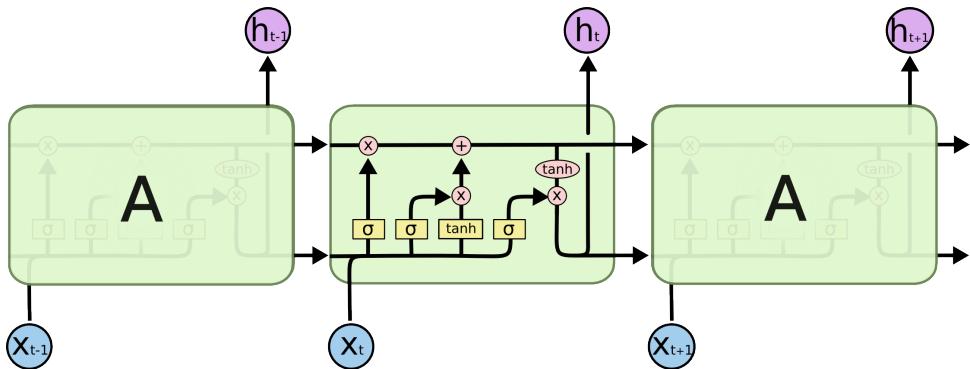


Figure 2.1.5: A visualization of an LSTM module (or cell) which has a *forget gate*, an *input gate*, and an *output gate*. The illustration is from Christopher Olah's excellent visual explanation of RNNs: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>.

2.2 Training Neural Networks with Gradient Descent

Deep learning is a large family of algorithms and techniques that allows multi-layered neural networks to be optimized efficiently (LeCun et al., 2015). Deep learning is typically based on variants of the optimization technique called gradient descent, that updates the parameters in small steps following the negative gradient of the loss function. It is called *deep* learning, because neural networks can more easily solve complex tasks when structured in multiple layers, thus becoming deep. The Backpropagation algorithm is used to efficiently compute gradients in layered neural networks and is the backbone of gradient descent. *Stochastic Gradient Descent* (SGD) is a variant that is typically used for large data sets in deep learning. Deep neural networks can of course be optimized with many other optimization algorithms, e.g. evolutionary algorithms which are described in Section 2.4.

The following section will describe backpropagation and Stochastic Gradient Descent (SGD) as well as techniques to improve generalization beyond the training set. It is not the goal here to give a broad overview to the many aspects of deep learning but rather to provide an introduction to the basic optimization techniques that are fundamental to the methods presented in the following chapters, such as deep reinforcement learning.

2.2.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a popular algorithm for optimizing the parameters of a differentiable model. Let us consider the supervised learning setting where the goal is to approximate the mapping function $f : X \rightarrow Y$ between data examples X and their labels Y . The goal of the optimization process is to reduce a loss function that determines how well the model is performing. In regression, one would typically employ the mean squared error (MSE):

$$L(f(\mathbf{x}; \theta), \mathbf{y}) = \frac{1}{N} \sum_{i=1}^N (\hat{\mathbf{y}} - \mathbf{y})_i^2,$$

between N predicted labels $\hat{\mathbf{y}}$ and true labels \mathbf{y} of the data examples \mathbf{x} . The loss function L can be seen as a fully differentiable computational graph that is based on f , where the set of partial derivatives $\frac{\partial L}{\partial w}$, for each $w \in \theta$, tells us how the loss function will change as a function of each weight. These partial derivatives form the gradient $\mathbf{g} = \frac{1}{N} \nabla_{\theta} \sum_{i=1}^n L(f(\mathbf{x}; \theta), \mathbf{y})$, which we can use to make a gradient update $\theta_{t+1} \leftarrow \theta_t - \eta \mathbf{g}$, where η is a small value called the *learning rate* or *step size*, e.g. $\eta = 0.01$. This process is called gradient descent, as the gradient is being followed in a descending direction to decrease the loss. The learning rate η controls the magnitude of each gradient update, and is an important and sometimes tricky hyper-parameter to set. If η is too low, learning becomes slow, and if η is too high, it will overshoot the minimum and fail to converge.

The *backpropagation of errors* algorithm, or just *backpropagation*, is applied to compute the gradient of each parameter in an efficient way. The algorithm computes the partial derivatives of each layer starting from the output and working its way backwards by reusing the computed derivatives along the way. This procedure is based on the chain rule of calculus, which is used to calculate the derivatives of functions that are composed of other functions. The chain rule of calculus states that if $z = f(y)$ and $y = g(x)$, then $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$. As a small example, let us consider the linear combination $z = \mathbf{W}\mathbf{x} + \mathbf{b}$ followed by an activation function $g(z) = y$. Given the partial derivative of some loss L with respect to y , we can compute $\frac{\partial L}{\partial W_{ij}}$ (the gradient of any weight W_{ij} with respect to the loss L) using the chain rule. This is done by chaining the partial derivatives:

$$\frac{\partial L}{\partial W_{ij}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial W_{ij}}.$$

Notice, that $\frac{\partial y}{\partial z}$ can be computed just once for all the weights. The strength of backpropagation is exactly this, to reuse the partial derivatives going backward through the computational graph. We can similarly also compute $\frac{\partial L}{\partial x_i}$ for all the input features and then perform the above procedure again for each unit in the preceding layer by starting a new chain from $\frac{\partial L}{\partial x_i}$. Implementations of the backpropagation algorithm follows the procedure described here, while they make use of several matrix operations to improve the efficiency.

A major issue when relying on gradients to update parameters in deep neural networks are

the *vanishing* and *exploding* gradient problems that we briefly mentioned when describing LSTMs. Because of the chain rule, the gradients of the first layers of a deep network are products of the gradients of all the proceeding layers. If these gradients are generally small, then the final computed gradient decreases exponentially with the number of layers in front of it, i.e. the gradient vanishes. In contrast, if the gradients in the proceeding layers are generally large, then the final computed gradient increases exponentially, i.e. it explodes. These effect makes it difficult to learn the parameters in the first layers. Logistic activation functions, such as sigmoid and tanh easily saturate, causing gradients to vanish. The ReLU activation function limits the effect of these problems as they only saturate in one direction. *Residual networks* (He et al., 2016b) can also overcome these issues by having layers with so-called *skip-connections*, such that layer i is connected to both layer $i + 1$ and layer $i + k$, i.e. skipping k layers.

The gradient descent algorithm follows a simple procedure. First, the gradient of the loss is computed for all data examples, and then a gradient update is performed. This procedure is then continued until some stopping criteria is met. When all data examples are used to compute the gradient, an accurate estimate of the true gradient is obtained. This can, however, be very inefficient because neural networks with many parameters require a large number of updates to converge. By the law of large numbers, it is sufficient to sample a so-called *mini-batch* of data examples instead, if we accept some inaccuracy in the estimated gradient. In fact, it is often more efficient to do many inaccurate updates rather than a few accurate updates. Stochastic Gradient Descent (SGD) follows this procedure and is much more efficient when optimizing neural networks than the naive gradient descent algorithm. Here, the batch size, i.e. the number of data examples sampled for each update, is an important hyper-parameter.

An important improvement to the basic SGD algorithm is to maintain a momentum of the gradient updates performed. The momentum is an exponentially decaying average of the gradients which limits the effect of using sampled mini-batches. A further extension is the maintenance of adaptive learning rates for each parameter in the network which perform the same functionality as momentum and thus replaces it. AdaGrad adjusts the learning rates with respect to the partial derivatives of each parameter, such that large derivatives results in rapid decreases (Duchi et al., 2011). AdaGrad is, however, designed

for convex functions as it is unable to increase the learning rates again, which can result in premature convergence in more complex optimization landscapes. Adadelta (Zeiler, 2012), RMSprop (Hinton et al., 2012), and Adam (Kingma and Ba, 2014) solves the issues of AdaGrad in various ways by maintain a momentum-like decaying average of the gradients. These variants are all considered robust optimization algorithms for neural networks, as of today.

2.2.2 Generalization

Machine learning models usually go through a training phase and are thereafter expected to perform well on new data examples that was not part of the set of training examples. If a model performs well on a training set but has poor performance on a test set, then it is said to be *overfitting*. This generally happens if the training set is too small or the model has too many parameters. In cases where a model is too simple to fit the training data adequately, it is said to be *underfitting*. In both cases, the model will have poor *generalization* and it is thus ideal to find a balance in-between the two extremes. A model's ability to generalize can be measured by setting aside a test set that is not used in training and then compare the model's performance on the two sets. When separating the data, it is important that the data points are independent and identically distributed (i.i.d.). This process is called *cross-validation* and can be repeated several times with different partitions. If data and computational resources are abundant, then generalization is generally not a problem in a supervised setting, but this is rarely the case. Instead several techniques exists to achieve good generalization, and some are described in the next sections.

2.2.2.1 Regularization

The complexity of a model is related to the number of parameters and the expressivity of the model and it has great impact on its ability to fit the training data (Hawkins, 2004). If the complexity is too large, there is a risk that the model will overfit. Regularization is a collection of techniques that aims to improve generalization, typically by maintaining a complex model while preventing it not to overfit during training.

L^2 regularization is a straightforward technique that adds a penalty term $\|\mathbf{w}\|^2$ to the loss function, such that when the model is optimized towards having few large activations values, as they typically result in an overfitted model. By using the L^2 regularization term, the model is simultaneously optimized to have a low complexity and a high performance.

Another popular and simple regularization technique is *dropout* where a portion of the activations in each layer is randomly dropped (set to zero) during training (Srivastava et al., 2014). This idea is interestingly inspired by the concept of *co-adaptation* in sexual reproduction, where genes are more robust, and thus preferred, if they can co-exist together with many different combinations of other genes. Using dropout, a neural network is thus not able to rely on particular activations for particular training examples (a common reason for overfitting), but must instead learn to utilize a random mix of multiple activations.

Other regularization techniques include *early stopping*, wherein overfitting is measured throughout training on a third held-out data set (called the validation set) and then training is stopped as soon as the validation loss had not decreased for a number of epochs. *Data augmentation* is another technique wherein new training examples are generated based on the original dataset. For image data, simple transformations such as rotation and image manipulation can be done, while this technique is tricky for some other data types.

2.2.2.2 Transfer & Multi-task Learning

Transfer learning (Pan and Yang, 2009; Tan et al., 2018) and multi-task learning (Caruana, 1997) are two techniques that are related to generalization as they both aim at improving learning in a target domain after having learned from a set of source domains. It can e.g. be beneficial in an image classification task to first learn a classifier on a large set of source objects (e.g. different animals) and then learn a classifier for the target object (e.g. a cat), especially if the number of examples in the target data set is small. This approach thus relaxes the criteria that all training data used to train a model must come the same distribution (Tan et al., 2018). A network-based deep transfer learning technique is to first train a deep neural network for task A, then reuse the first layers in the network to construct a new network model that is trained on task B (Tan et al., 2018). Usually,

the first layers represent general knowledge and deep layers represent domain specific knowledge. This procedure can be repeated over several tasks. In multi-task learning (Caruana, 1997), layers are not removed, but a new output layer is simply added for each task.

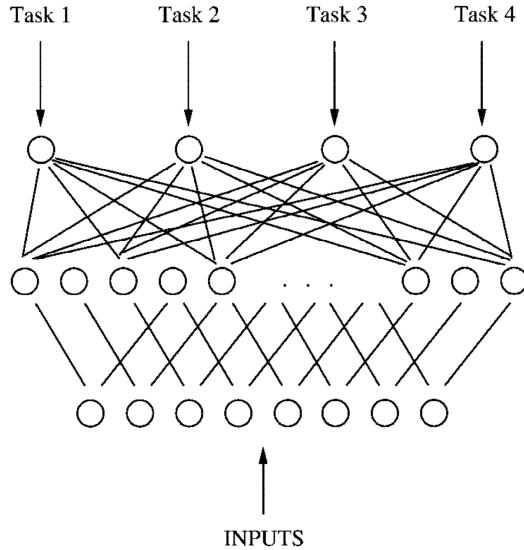


Figure 2.2.1: A visualization of a neural network using multi-task learning with one task-dependant output layer for each task and one general hidden layer.

2.3 Reinforcement Learning

The previous section described the basics of training deep neural networks using gradient descent. The training setting considered so far has been supervised learning, in which a data set of data points X and labels Y is given, and a mapping function $f : \mathcal{X} \rightarrow \mathcal{Y}$ is learned. For many interactive problems, in which a system takes actions based on the observed state of the environment, it is impractical to collect a sufficient number of demonstrational examples of a desired behavior, and thus supervised learning becomes intractable. For these problems it can be necessary to let a parameterized system interact with the environment and then gradually adjust its parameters based on the observed quality of its actions. Reinforcement Learning is a class of optimization algorithms that takes this approach. More precisely, it learns a policy (mapping between states and actions) by taking actions in an environment and adjust the policy based on feedback provided by the environment. Reinforcement learning can be combined with deep learning

to approximate the policy function with a neural network, and this combination is typically called *deep reinforcement learning*. In contrast to approximate solutions, reinforcement learning algorithms can also learn in a *tabular* setting, in which the policy function maintains a stored value for all state-action pairs. However, this approach does not scale to most interesting problems, including most games.

Reinforcement learning methods can be separated into two distinct classes; *model-based* and *model-free*. In model-based reinforcement learning, the learning algorithm has access to the dynamics of the environment; it can foresee the outcomes of taking actions in a given state. Such a model of the world is rarely available and indeed, some model-based methods attempt to learn it, e.g. the work on world models by Ha and Schmidhuber (2018). In this dissertation, only model-free reinforcement learning methods are used.

This section describes the basics of reinforcement learning, starting with an introduction to Markov Decision Processes, which provides a good framework with a set of notations used in reinforcement learning. Then, basic reinforcement learning algorithms are described, including temporal difference learning and Q-learning. Finally, modern deep reinforcement learning techniques are described, including deep Q-learning, the policy gradient algorithm REINFORCE and the actor-critic methods A3C and A2C.

2.3.1 Markov Decision Processes

Markov decision processes (MDPs) provide a formal framework and notation for describing reinforcement learning problems and algorithms in a simple and precise way. MDPs are sequential decision problems in which an *environment*, that has a *state*, is manipulated by an agent taking *actions*. At each time step t , the agent selects an action a_t to perform, which results in the environment transitioning from state s_t to a new state s_{t+1} along with a feedback signal r_{t+1} called *reward* that is sent back to the agent. This process follows an underlying distribution $p(s', r|s, a)$ for all $s', s \in \mathcal{S}$, $r \in \mathcal{R} \subset \mathbb{R}$, and $a \in \mathcal{A}$. This distribution describes all the dynamics of the environment from which we can compute both the state-transition function $p(s'|s, a)$, and the reward distributions $r(s|a)$ and $r(s, s'|a)$. The main problem of MDPs is to find a policy π that maximizes the expected future reward, which is often discounted with a factor $\gamma \in \{0, 1\}$, such that immediate

rewards are more valuable than future rewards. The discounted future reward G_t at time step t is thus defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{K-1} R_{t+k},$$

where K is the number of steps in the episode.

If states include all important information about the past that makes a difference for the future (Sutton and Barto, 1998), then the MDP has the *Markov Property*. This is not usually the case in video games, where screen images often represent a partial observation of the whole state, and thus the agent will need to remember important information about past states. If the MDP does not have the Markov Property, it is often useful to make a distinction between the true hidden states \mathcal{S} , that is unknown to the agent, and partial state *observations* that the agent has access to. It is thus common to use the term observation instead of state when solving MDPs with hidden information. An MDP is either *episodic*, if there exists a set of states that terminates the process, or *continuous*, if the process never terminates. Most games can thus be modelled as episodic MDPs.

2.3.2 Temporal Difference Learning

A central idea in many reinforcement learning algorithms is that of Temporal Difference (TD) learning which is a method for learning the state-value function $V_\pi(S_t) = \mathbb{E}_\pi[G_t | S_t = s]$ of a policy π . When following π and G_t is observed, one can compute the *TD error* λ at every step, which is the difference between the observed and expected value $\delta_t = G_t - V(S_t)$. In the tabular setting, the following update rule can then be used: $V(S_t) \leftarrow V(S_t) + \alpha \delta_t$, where α is a small learning rate. When using differentiable function approximators, such as neural networks, we can perform a gradient update on our parameters: $\theta_{t+1} \leftarrow \theta_t + \delta_t \nabla V(s_t)$.

TD-learning can be used with Monte Carlo rollouts for episodic tasks such that an entire episode is executed where after the error is computed and the update rule is applied for every step. One-step TD learning is at the other extreme, as it performs the updates at every step during an episode. Since the future rewards are not yet known when the

updates are performed, a so-called *bootstrapping* technique is applied to estimate the value of the next state. Because of the following equality:

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi[R_t + \gamma G_{t+1} | S_t = s] = \mathbb{E}_\pi[R_t + \gamma V_\pi(S_{t+1}) | S_t = s],$$

one-step TD uses the update rule $V(S) \leftarrow R_t + \gamma V_\pi(S_{t+1})$. Monte Carlo methods and one-step TD are two extremes that span a large set of intermediate update rules; usually the most efficient method is in between the two. These methods can be expressed as n -step TD, where G_t is bootstrapped after n steps, or by λ -returns, where $\lambda \in \{0, 1\}$ assigns discounted weights to future returns. The algorithm based on λ -returns is called TD(λ), where TD(0) is the same as one-step TD, and TD(1) is the same as using Monte Carlo rollouts.

2.3.3 Q-learning

Learning a state-value function is not by itself enough to learn a policy. The famous Q-learning algorithm by Watkins and Dayan (1992) is an extension to TD learning that learns a state-action value function $Q_\pi(S_t, A_t) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$ of a policy π , instead of a state-value function. Policies can be derived from Q , e.g. by greedily selecting the action with the highest value: $\max_a Q(S_t, a)$. The update step for Q-learning is similar to that of TD-learning but with $G_t = R_t + \gamma \max_a Q(S_{t+1}, a)$. The trajectories are sampled using an exploration policy, that is usually ϵ -greedy: it selects actions greedily with a probability ϵ and random otherwise.

Because the exploration policy is different from the policy of which Q is estimated (unless $\epsilon = 1$), Q-learning is an *off-policy* algorithm. An *on-policy* variant is SARSA (State-Action-Reward-State-Action), where $G_t = R_t + \gamma Q(S_{t+1}, a_{t+1})$, such that the Q-function estimates the state-action value function of the exploration policy after taking a_{t+1} in the next state s_{t+1} .

Q-learning can easily be used with function approximators such as neural networks similarly to TD learning. However, the more recent variant called Deep Q-learning by Mnih et al. (2013) improves the stability when training deep neural networks. A convolutional neural

Algorithm 1 Deep Q-learning with Experience Replay from Mnih et al. (2013)

usually express

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
    Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
    for  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$                                  $\triangleright \epsilon$ -greedy
        otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$ 

```

network trained with Deep Q-learning is called a *Deep Q-Network* (DQN). The complete Deep Q-learning algorithm is described in Algorithm 1. During training, state-transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ are appended to a *replay memory* buffer with a capacity of N tuples. Here, ϕ_t is a processed representation of state s_t , e.g. grey-scaled or down-scaled. Mini-batches are then sampled from the buffer to update Q similarly to standard supervised learning using deep neural networks. More variations of deep Q-learning and their performances on game benchmarks is covered in Chapter 3.

2.3.4 Policy Gradient

Instead of deriving a policy from a learned action value function, *policy gradient* methods learn a policy directly that is parameterized by $\boldsymbol{\theta}$, such that $\pi(a|s, \boldsymbol{\theta})$ is the probability of taking action a in s with parameters $\boldsymbol{\theta}$. The policy distribution is often expressed by having a neural network with softmax activations in the final layer. The policy π can then be used probabilistically for exploration, which reflects the policy better than ϵ -greedy. The policy is updated by performing gradient ascent based on a performance measure $J(\boldsymbol{\theta})$. The policy gradient theorem (Sutton et al., 2000) states the following proportional relationship:

$$\nabla J(\theta) \propto \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} Q_\pi(s, a) \nabla \pi(a|s, \theta),$$

where $\mu(s)$, for all $s \in \mathcal{S}$, is the state distribution of π . While $\mu(s)$ is unknown, states is expected to be encountered in this proportion as π is the exploration policy. We can thus remove it and use the following update rule:

$$\theta \leftarrow \theta + \alpha \sum_{a \in \mathcal{A}} Q_\pi(s, a) \nabla \pi(a|s, \theta)$$

Rather than computing the state-action value for all $a \in \mathcal{A}$, the REINFORCE policy gradient algorithm by Williams (1992) simplifies the rule to consider only that sampled action. To arrive at the update rule used in REINFORCE, we consider the following equality which holds as long as we sample from the same policy:

$$E_\pi[G_t] = E_\pi[Q(S_t, A_t)] = E_\pi \left[\sum_{a \in \mathcal{A}} Q_\pi(S_t, a) \pi(a|S_t, \theta) \right]$$

To use G_t instead of the unweighted sum over actions $\sum_{a \in \mathcal{A}} Q_\pi(s, a)$ (as in the previous update rule) we simply divide by the probability of sampling A_t , arriving at the following update rule:

$$\theta \leftarrow \theta + \alpha G_t \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)}.$$

This update rule can then be simplified to $\theta \leftarrow \theta + \alpha G_t \nabla \ln \pi(A_t|S_t, \theta)$.

2.3.5 Actor-Critic Methods

REINFORCE updates the policy based only on G_t , not taking into account how good this value is compared to taking the other actions. This can be a problem if all action values are high in a particular state, as the gradient update would increase the likelihood of taking all actions with little differentiation. This problem can be solved by using a

baseline $b(S_t)$, such that the gradient magnitude is determined by the error $\delta = G_t - b(S_t)$. The baseline is often a simultaneously learned parameterized state value function $\hat{v}(S_t)$, such that δ is computed similarly as in TD learning. Both the policy and the state-value function is thus updated using gradient ascent/descent. Given that $\hat{v}(S_t)$ is somewhat accurate, the policy update is now relative to the value of other actions. This relative value is also called the *advantage* $A_t = Q_\pi(S_t, A_t) - V_\pi(S_t)$; the advantage of taking an action in a state relative to the other actions.

Policy gradient methods with a state-value baseline and n -step bootstrapping are called actor-critic methods. The policy is the actor and the state-value function is the critic. An efficient actor-critic algorithm for training deep neural networks is the Asynchronous Advantage Actor-Critic (A3C) algorithm by Mnih et al. (2016) that distributes the actor and critic parameters to a number of parallel worker processes that each interact with a copy of the environment. Each worker performs asynchronous gradient updates and sends these gradients back to a master thread. The master thread performs updates on its own set of parameters based on all the accumulated gradients and periodically re-distributes them to the workers as they will diverge over time. The bottleneck of A3C lies in the inter-process communication when transferring gradients and parameters back and forth, thus limiting its ability to scale to a large number of workers or parameters. A simpler variant is the synchronous Advantage Actor-Critic (A2C) algorithm, that was first used in Wu et al. (2017b). A2C similarly maintains a global set of parameters in a master process and worker processes maintain a copy of the environment but no model. Instead, workers send their observed state and reward at every step to the master process and then receives the next action to perform. The master process queries the policy in batches upon receiving observations from all the workers, making the algorithm efficient on modern hardware. Additionally, gradient updates are only done on the master process in contrast to A3C. Both variants use the following gradient update rules on the two sets of parameters θ_π and θ_v :

$$\theta_\pi \leftarrow \theta_\pi + \alpha_\pi(G_t - V(S_t; \theta_v))\nabla_{\theta_\pi} \ln \pi(A_i | S_i; \theta_\pi)$$

$$\theta_v \leftarrow \theta_v + \alpha_v \nabla_{\theta_v} (G_t - V(S_i; \theta_v))^2$$

where G_t is the n -step bootstrapped value estimate: $G_t = \sum_{i=0}^n \gamma^i R_{t+i}$.

2.4 Evolutionary Algorithms

Evolutionary algorithms (EA) are a family of population-based optimization algorithms that are inspired by natural evolution. A population of candidate solutions (also called individuals or genomes) is maintained, wherein new solutions are constantly formed using operations such as genetic recombination and mutation to introduce new innovations. A fitness-based selection mechanism forces the least fit solutions to be replaced at every generation by more fit solutions. A solution is represented by a *genotype*; its low-level encoding, e.g. in $\{0, 1\}^n$ or \mathbb{R}^n . One can think of the genotype as the DNA of a solution. The *phenotype*, on the other hand, is the observable characteristics of a solution.

EAs differ from other classes of optimization algorithms in several aspects. First of all, they do not rely on assumptions about the search space, such as differentiability or convexity, and are therefore a class of black box optimization algorithms. Another feature that makes EAs unique is the maintenance of a population of solutions that allows for diversification, which is useful to overcome local optima.

This section will describe a few basic EAs including genetic algorithms and evolution strategies, followed by specific algorithms that makes use of diversification, including multi-objective EAs, novelty-search, and MAP-Elites.

2.4.1 Genetic Algorithms

Genetic algorithms (GAs) (Holland, 1975) are among the simplest EAs. They maintain N solutions in a population and uses three main operators in each generation: reproduction, crossover, and mutation. Reproduction involves the selection of parents in the current population that will be used for breeding offspring for the next generation. Parents are selected according to their fitness, such that the genetic material of fit solutions is maintained. Traditionally, this is done through *fitness proportionate selection* where the probability p_i of selecting solution i as parent is $p_i = f_i / \sum_{j=1}^N f_j$, where f_i is the fitness of the i th solution. *Truncation selection* is another selection mechanism that first sorts the population by fitness and then removes the least fit solutions, such that the population has been truncated to a proportion p . The remaining solutions are then

selected uniformly for reproduction or used an equal number of times. When two parents are selected, a crossover operator recombines their genotypes. This is traditionally done through single-point crossover, where a *crossover point* on the genotype strings of two parents is sampled. Two new genotypes are then created by swapping the genetic material on one side of this point. K-point crossover does the same but with k crossover points instead. Figure 2.4.1 shows an example of single-point and two-point crossover. Uniform crossover is another operator that simply selects each gene from the parents with equal probability. During the copying process, or as an additional step afterward, mutations are introduced to add new genetic material into the gene pool. Mutation can simply be done by inserting a randomly sampled gene, with some small probability, instead of the gene inherited from the parents. Algorithm 2 describes a GA with fitness proportionate selection and Algorithm 3 describes a GA with truncation selection.

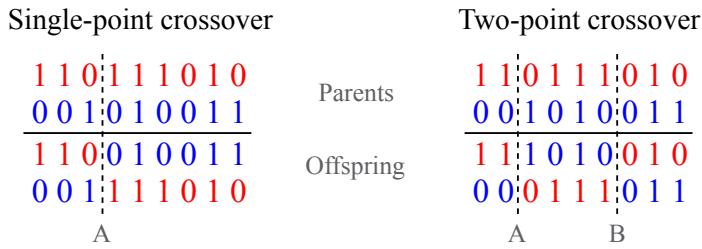


Figure 2.4.1: A simple example of single-point and two-point crossover operations on two parent genotypes in $\{0, 1\}^8$, each resulting in two offspring.

Algorithm 2 Genetic Algorithm (GA) with fitness proportionate selection

```

Initialize a population  $\mathcal{P}$  of  $N$  random solutions
Evaluate the fitness of all solutions in  $\mathcal{P}$ 
while termination criterion is not fulfilled do
   $\mathcal{P}' = \emptyset$ 
  while  $\mathcal{P}'$  contains less than  $N$  solutions do
    parent1, parent2  $\leftarrow$  SelectParent( $\mathcal{P}$ ), SelectParent( $\mathcal{P}$ )     $\triangleright$  with  $p_i = f_i / \sum_{j=1}^N f_j$ 
    child1, child2  $\leftarrow$  Crossover(parent1, parent2)
    child1, child2  $\leftarrow$  Mutate(child1), Mutate(child2)
    evaluate fitness of child1 and child2
     $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{\text{child1}, \text{child2}\}$ 
   $\mathcal{P} \leftarrow \mathcal{P}'$ 
  Evaluate the fitness for all solutions in  $\mathcal{P}$ 
  
```

GAs do not rely on assumptions about the search space, which can be seen as a weakness of the algorithm as it searches in the neighborhood of known solutions in a rather uninformed/random manner. It can thus be more useful to apply other algorithms that

do make assumptions about the search space to reduce the number of evaluations in an efficient manner. Some variants of Evolution Strategies, which are described next, does this well in continuous spaces. GAs can, however, be useful when optimizing a discrete non-differential objective function $f : \{1, \dots, K\}^n \rightarrow \mathbb{R}$.

Algorithm 3 Genetic Algorithm (GA) with truncation selection

```

Initialize a population  $\mathcal{P}$  of random solutions
Evaluate the fitness of all solutions in  $\mathcal{P}$ 
while termination criterion is not fulfilled do
    Truncate  $\mathcal{P}$  to a proportion  $p$  of the fittest solutions           ▷ E.g.  $p = 1/2$ 
     $\mathcal{P}' = \emptyset$ 
    while  $\mathcal{P}'$  is not filled do
        parent1, parent2  $\leftarrow$  SelectParent( $\mathcal{P}$ ), SelectParent( $\mathcal{P}$ )           ▷ E.g. uniformly
        child1, child2  $\leftarrow$  Crossover(parent1, parent2)
        child1, child2  $\leftarrow$  Mutate(child1), Mutate(child2)
        evaluate fitness of child1 and child2
         $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{\text{child1, child2}\}$ 
     $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{P}'$ 

```

2.4.2 Evolution Strategies

Evolution Strategies (ES) (Rechenberg, 1978) is a class of EAs that usually searches for solutions in continuous spaces, i.e. optimizing a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. In the case where ES is applied to a discrete solutions space, it can be seen as a variant of a GA with truncation selection. However, most ES variants assumes a continuous space.

ES comes in two traditional variants that are described in Algorithm 4 as well as numerous newer improvements. One of the traditional variants is $(\mu + \lambda)$ -ES, wherein a population of $\mu + \lambda$ individuals is the basis for the next generation, such that solutions are selected at the end of each generation from both the parent population and the newly created set of offspring. In (μ, λ) -ES, the population contains only μ solutions, where the solutions for the next generation is selected only from the set of λ offspring. $(\mu + \lambda)$ -ES is possibly the most common of the two as it, in contrast to $\mu + \lambda$, guarantees a monotonically improvement. In both variants, the initial population is drawn from a multivariate normal distribution in \mathbb{R}^n and the mutation operator adds a random perturbation vector drawn from another multivariate normal distribution with zero mean and constant variance.

Algorithm 4 $(\mu + \lambda)$ -ES and (μ, λ) -ES

```

Initialize a population  $\mathcal{P}$  of  $\mu$  solutions
Evaluate the fitness for all solutions in  $P$ 
while termination criterion is not fulfilled do
    Reproduce and mutate a set  $\mathcal{P}'$  of  $\lambda$  offspring
    Evaluate the fitness of the solutions in  $\mathcal{P}'$ 
    if  $(\mu + \lambda)$ -selection then
        Select the  $(\mu + \lambda)$  best fit solutions in  $\mathcal{P} \cup \mathcal{P}'$  as the new population  $\mathcal{P}$ 
    else
        Select the  $\mu$  best fit solutions in  $\mathcal{P}'$  as the new population  $\mathcal{P}$ 

```

The simplest ES algorithm is the $(1, 1)$ -ES that maintains just one solution in the population and produces one new offspring each generation. This is also called a hill-climbing algorithm. When optimizing deep neural networks, it has become popular to use a variant of $(1, \lambda)$ -ES, i.e. it only maintains one set of parameters θ , with a reproduction operation consisting only of Gaussian perturbations (Salimans et al., 2017). This variant belongs to the class of Natural Evolution Strategies (NES) that estimates a gradient of the objective function over λ perturbations of θ . In Salimans et al. (2017), the gradient is approximated by computing a fitness-weighted average of the λ offspring. θ is then updated using gradient ascent. NES algorithms differ in the way they estimate the gradient and the distribution used to sample perturbations (Wierstra et al., 2014).

The covariance matrix adaptation evolution strategy (CMA-ES) is a highly-praised improvement to ES that adapts the variance of the distribution used to sample perturbations. The perturbation distribution thus becomes ellipsoid-shaped corresponding to the eigenvectors of the covariance matrix computed from the data points in the current generation, such that new points are sampled in a non-uniform direction towards the promising solutions. CMA-ES is an efficient algorithm in non-linear non-convex problems but it does not scale well with the number of dimensions n in the solutions space as the covariance matrix requires $O(n^2)$ computations. CMA-ES is thus usually not used to optimize large neural networks.

2.4.3 Multi-objective Evolutionary Algorithms

Multi-objective EAs (MOEAs) is a class of algorithms for problems with multiple objectives with the goal of finding solutions along the *optimal Pareto frontier*, such that none of the

solutions are *dominated*. In the two-objective case, with objectives f_1 and f_2 , a solution A is dominated by another known solution B if $f_1(B) > f_1(A)$ and $f_2(B) > f_2(A)$. Solutions along a Pareto front forms a *Pareto set* which are the optimally known solutions, from which the user can make a trade-off. The Non-dominated Sorting Genetic Algorithm II (NSGA-II) is an efficient MOEA (Deb et al., 2002). Multi-objective EAs is not the focus in this dissertation but they are to some degree related to the family of quality-diversity algorithms that are discussed and applied several times.

2.4.4 Quality Diversity

In *deceptive* problems, progression towards higher fitness sometimes leads to solutions that are further from the objective (Goldberg, 1987). Objective-based search are thus susceptible to converge to local optima that are far from the global optimum. Whitley (1991) argues that all challenging optimization problems are to some degree deceptive. The problem with deception is that important stepping stones toward the objective are not properly reflected by the objective function (Lehman and Stanley, 2011c). Novelty Search (Lehman and Stanley, 2011a) introduces a rather counter-intuitive approach to deceptive problems, which is to abandon the objective completely. Instead, the search is focused toward novelty; solutions that behave differently from previously found solutions receive a high fitness.

The novelty search algorithm simply replaces the normally static fitness function with a dynamically adjusting *novelty metric*. This metric relies on a distance function between two solutions in a domain-specific behavioral space, such that new solutions that are far from prior solutions receive a high fitness. The novelty of a solution is equal to the *sparseness* of the behavioral neighborhood, which is computed as the average distance (in the behavioral space) to the k -nearest neighbors. If the measured novelty of an individual is above a certain threshold, then it is permanently added to an archive that represents previously found solutions. Individuals in the both archive and the current population are then used to compute the novelty scores. Novelty search has been shown to outperform objective-based search in deceptive navigation (see Figure 2.4.2) and locomotion tasks, despite the objective being unknown to novelty search (Lehman and Stanley, 2011a)). In the navigation task, the behavioral description of a solution is equal to the 2D coordinate



Figure 2.4.2: Final points visited in a deceptive navigation task by (a) novelty search and (b) objective-based search, starting at the green dot and the objective is the distance to the goal (the red dot). The figures are from Lehman and Stanley (2011a).

of the final point visited in the maze.

Searching for novelty alone does not guarantee that high-performing solutions are found, especially if solutions with similar behavioral characteristics can differ widely in terms of performance. As soon as novelty search has found some solutions that occupy a behavioral niche, it will no longer add new solutions with the same behavioral characteristics, even if their performance is better. Novelty Search with Local Competition (NSLC) (Lehman and Stanley, 2011b) is an extension that employs a multi-objective EA with both a performance and a novelty objective. Here, the performance metric is a local competition between the new solution and the k -nearest neighbors in the behavioral space. The idea here is to focus the search on optimizing the quality of solutions in every niche instead of just the best niches. NSLC thus aims at optimizing both the quality and the diversity of solutions in the archive. This idea has given rise to a new family of EAs called Quality-Diversity (QD) algorithms (Pugh et al., 2016). While traditional optimization algorithms aim at finding the optimal solution to a problem, novelty search aims at finding a set of solutions that behave as different as possible. QD algorithms thus combine these two goals and attempt to find a set of high-performing solutions that each behaves differently (Pugh et al., 2016).

MAP-Elites (Mouret and Clune, 2015) is a QD-algorithm that differs from NSLC in several aspects. First, it does not maintain a population throughout the evolutionary run, only an archive divided into cells that reflects regions (niches) of the behavioral space. The procedure of MAP-Elites is straight-forward (see Algorithm 5. Individuals are iteratively sampled from the archive, mutated, and evaluated. If the new solution outperforms the elite in its behavioral niche (corresponding to a cell in the archive) it becomes the new

elite of that niche. MAP-Elites is also called an *illumination* algorithm as it aims to find the highest-performing solution in every niche of the behavioral space, i.e. illuminate the whole behavioral space, while NSLC merely balances the novelty and performance objectives.

Algorithm 5 MAP-Elites (Mouret and Clune, 2015)

```

( $\mathcal{P} \leftarrow \emptyset, \mathcal{X} \leftarrow \emptyset$ )       $\triangleright$  Create an archive of {solutions  $\mathcal{X}$  and their performances  $\mathcal{P}$  }
 $\text{for iter} = 1 \rightarrow I \text{ do}$                        $\triangleright$  Repeat for  $I$  iterations.
     $\text{if iter} < G \text{ then}$                    $\triangleright$  Initialize by generating  $G$  random solutions
         $\mathbf{x}' \leftarrow \text{random\_solution}()$ 
     $\text{else}$ 
         $\mathbf{x} \leftarrow \text{random\_selection}(\mathcal{X})$            $\triangleright$  Randomly select an elite  $x$  from  $\mathcal{X}$ 
         $\mathbf{x}' \leftarrow \text{random\_variation}(\mathbf{x})$            $\triangleright$  Create an offspring  $x'$  of parent  $x$ 
         $\mathbf{b}' \leftarrow \text{feature\_descriptor}(\mathbf{x}')$          $\triangleright$  Simulate  $x'$  and record its feature descriptor  $\mathbf{b}'$ 
         $p' \leftarrow \text{performance}(\mathbf{x}')$                   $\triangleright$  Record the performance  $p'$  of  $x'$ 
         $\text{if } \mathcal{P}(\mathbf{b}') = \emptyset \text{ or } \mathcal{P}(\mathbf{b}') < p' \text{ then}$            $\triangleright$  If  $x'$  is best in the cell
             $\mathcal{P}(\mathbf{b}') \leftarrow p'$                           $\triangleright$  store the solution  $x'$  in the archive
             $\mathcal{X}(\mathbf{b}') \leftarrow \mathbf{x}'$ 
    
```

A key advantage of QD-algorithms is that they produce a large set of spatially related solutions that gives a system the ability to switch between solutions when deployed. Figure 2.4.3 shows an example of this in the form of a behavior-performance map. As the solutions have spatial relationships, in a usually low-dimensional behavioral space, efficient optimization algorithms for continuous spaces can be used for this adaptation, such as the intelligent trial-and-error (IT&E) algorithm (Cully et al., 2015). IT&E relies on a pre-computed archive of solutions with their prior fitness and behavioral characteristics which are used to construct a prior distribution of the objective function, which is also called a behavior-performance map. A Bayesian optimizer is then used to sample a point in the map, record the observed performance and compute a posterior distribution of the objective function. This process is then continued until a satisfying solution is found.

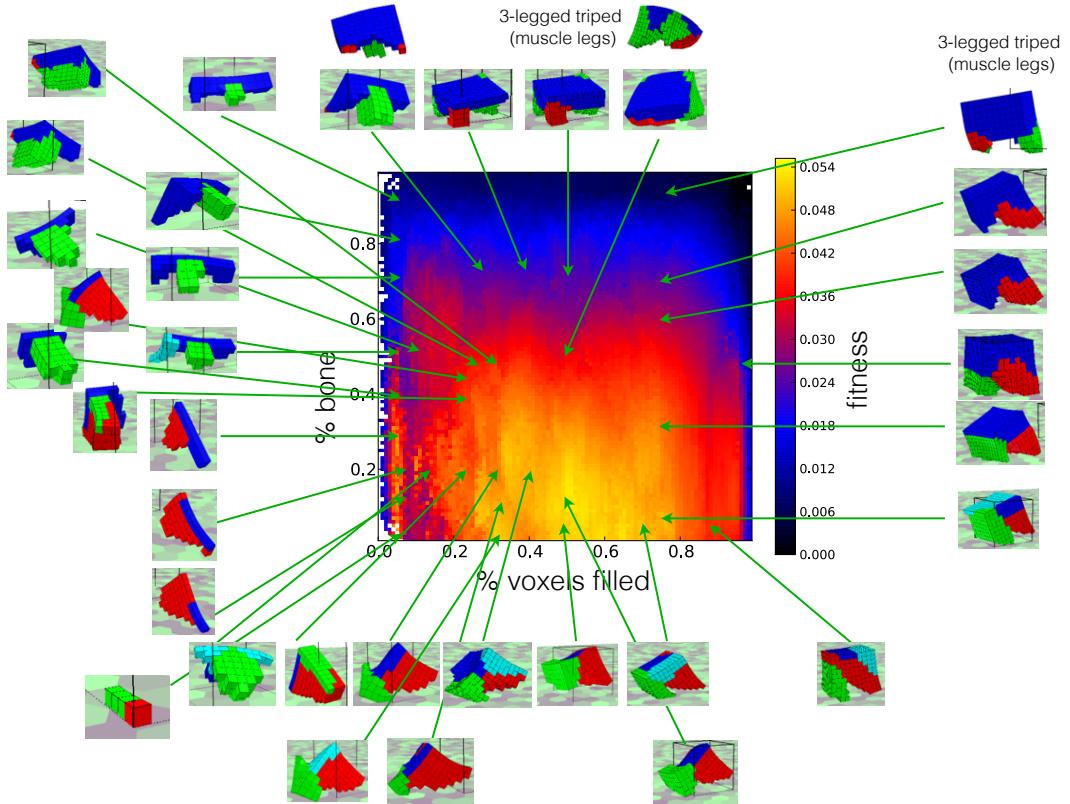


Figure 2.4.3: A 2D behavior-performance map of voxel-based virtual creatures found by MAP-Elites. Here, the performance metric is walking speed. The map consist of behavioral niches each with a unique solution. The map is illuminated by the fitness of these solutions. This figure is from Mouret and Clune (2015).

2.5 Search Algorithms

Games can be seen as a tree of sequential decisions that can be explored online (as the game is being played). Search algorithms can explore this tree of possible future outcomes to either find one promising action or to schedule a plan of several actions. This section describes such algorithms for game-playing, including A*, Minimax, Monte Carlo Tree Search (MCTS), and evolutionary planning algorithms. We distinguish these algorithms from optimization as search algorithms are applied online (during the game) in contrast to optimization that typically attempts to find solutions (e.g. policies) offline.

Fundamental to AI for game-playing are *tree search* algorithms, that start with a game state $s \in \mathcal{S}$ and creates a *search tree* by iteratively (or recursively) creating branches to possible outcomes (other game states) for each available action in \mathcal{A} . This procedure

requires a *forward model*, that simulates a state-transition function $f : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ which may follow a non-deterministic probability distribution $p(s'|s, a)$ for all $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$.

The most basic class of tree search algorithms are uninformed search, such as depth-first or breadth-first search. Best-first search algorithms, on the other hand, performs informed search and requires a heuristic to select from which node in the tree to expand from next. This heuristic evaluates the game state represented by the node, e.g. by the distance to the original game state (of the root node) or the estimated chances of winning in the game state. Such heuristics can either be manually hand-crafted or learned through data.

Tree search algorithms can either be implemented as *open loop* and *closed loop* search Perez Liebana et al. (2015). In closed loop search, nodes store the state and a branch is added for each possible outcome from a node. This is trivial for deterministic games while it can result in a high number of branches in stochastic games. In open loop search, states are not stored in the nodes, but only the statistics. Here, actions are still stored in the edges that lead to nodes in the search tree, while the encountered state can be different at each traversal. Open loop search is thus useful in stochastic games to limit the number of branches at every node.

This section will describe a few informed tree search algorithms that are useful for game-playing as well as an alternative evolutionary planning approach that instead of building a search tree evolves complete action sequences. The sections describing A*, Minimax, and MCTS are not critical prerequisites to follow the main chapters of this dissertation, while they provide a brief introduction to several game-playing algorithms that can be useful to be aware of.

2.5.1 A* Path-finding Algorithm

A well-known best-first search for path-finding is A* by Hart et al. (1968). The algorithm maintains a set of known states that have not yet been fully expanded, starting with the current state. A state is expanded by adding all reachable states to the unexpanded set. At each iteration of the algorithm the most promising state in this set is explored further. States are valued using an evaluation function $f(s) = g(s) + h(s)$, where $g(s)$ is the cost

of travelling from the starting state to s and $h(s)$ is an admissible heuristic of the minimal remaining cost before reaching the goal. The heuristic is admissible if the minimum cost of all possible paths from s to the goal is higher than $h(s)$, for all $s \in S$. A* alone can play games wherein a single character has to navigate in a simple environment, such as Super Mario Bros (Nintendo, 1985) (Togelius et al., 2010).

2.5.2 Minimax

The Minimax algorithm is useful in adversarial turn-based games as it takes into account that the opponent has agency as well. The algorithm is based on the two strategies in zero-sum games, $\max_i \max_j v_{a_i, a_j}$ (maximin) and $\max_j \max_i v_{a_i, a_j}$ (minimax) where two players select actions a_i and a_j and $v_{a_i, a_j} \in \mathbb{R}$ is the outcome that player 1 tries to maximize and player 2 tries to minimize. Player 1 thus follows the maximin strategy trying to maximize the minimum outcome after player 2 selects an action and player 2 follows the minimax strategy trying to minimize the maximum outcome after player 1 selects next.

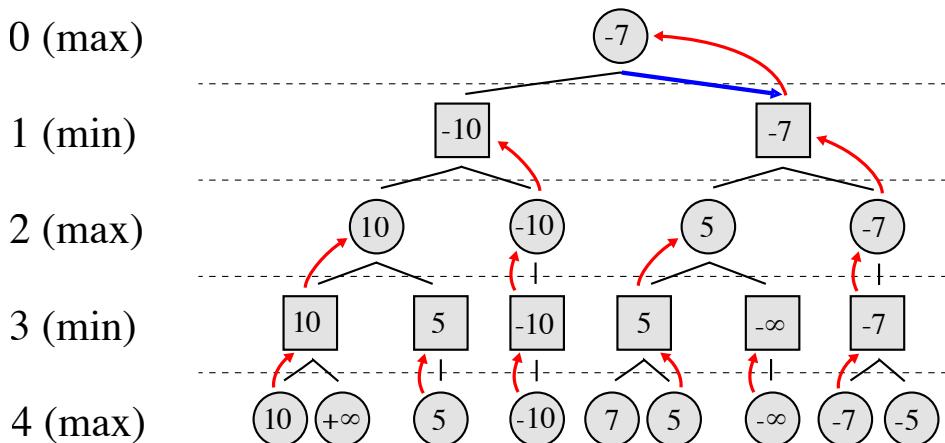


Figure 2.5.1: A search tree produced by the Minimax algorithm with the minimax values shown on each node. Here, the optimal choice is to go right, because the minimum value we can obtain is -7 and thus higher than -10 when going left. This illustration is made by Nuno Nogueira.

Alpha-beta pruning is an improvement of Minimax that safely ignores sub-trees entirely of actions that are already known to be worse than another action at the same level. This can be done when a part of the sub-tree is explored.

For most interesting games, Alpha-beta pruning is not able to reach the end of the game

and thus a heuristic is used to evaluated the game state at a certain depth. The top chess-playing programs today used Alpha-beta pruning with several additional improvements while the algorithm falls short in most games that are more complex.

2.5.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a best-first search that uses stochastic sampling as a heuristic (Chaslot et al., 2008a; Coulom, 2006), and has been successfully applied to games with large branching factors such as Civilization II (Branavan et al., 2011), Magic the Gathering (Cowling et al., 2012), and Settlers of Catan (Szita et al., 2009). The algorithm starts with a root node representing the current game state. Four phases are sequentially executed in iterations until a given time budget is used or a satisfying goal state is reached. In the selection phase, the tree is traversed from the root node using a *tree policy* until a node with unexpanded children is reached. In the expansion phase, a child node is expanded from the selected node. In the simulation phase (also called rollout), the remaining part of the game, from the expanded node's game state, is played out using a *default policy*. In the backpropagation phase, the outcome of the game is backpropageted up the tree until the root node is reached.

1. **Selection:** The tree is traversed from the root node using a *tree policy* until a node with unexpanded children is reached. Edges going from nodes represent actions available at the node's game state.
2. **Expansion:** A child node is expanded from the selected node.
3. **Simulation:** A simulation of the remaining part of the game from the expanded node's game state is played out using a *default policy*. These simulations are also called *rollouts*.
4. **Backpropagation:** The result of the game is backpropageted up the tree until the root node is reached. The visit count of each node is also incremented which affects the selection process in the following iterations.

The tree policy determines how the search balances exploration and exploitation during the selection phase. Usually the Upper Confidence Bounds (UCB) algorithm is used,

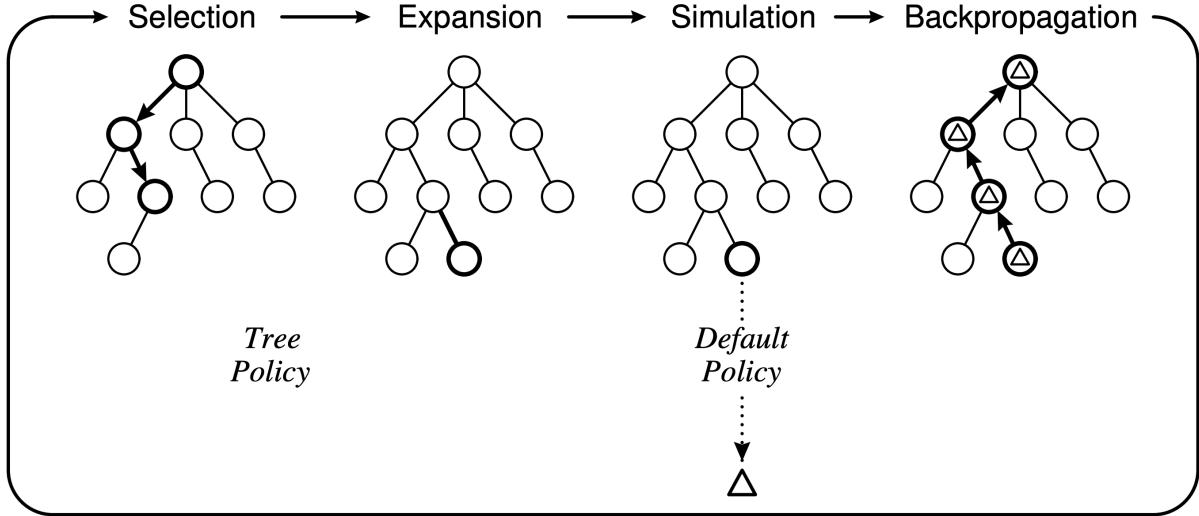


Figure 2.5.2: The four phases of one iteration in Monte Carlo Tree Search. The figure is from Chaslot et al. (2008a).

which selects the node that maximizes:

$$\text{UCB1} = \bar{X}_j + C \sqrt{\frac{2 \ln n}{n_j}},$$

where n is the visit count of the current node, n_j is the visit count of the child j , C is a constant determining the amount of exploration versus exploitation and \bar{X}_j is the normalized value of child j . The default policy is used during rollouts to select actions, which can be a complex scripted policy or one that selects random actions. An ϵ -greedy strategy can also be used to select a random action at probability ϵ and at probability $1 - \epsilon$ follows some predefined policy.

Numerous variants of MCTS have been applied to many different games (Browne et al., 2012) where other tree-search algorithms have failed. To allow MCTS to be applied to games with increasingly higher branching factors, a variety of different MCTS variations has been developed. Numerous enhancements exist for MCTS to handle large branching factors, such as First-Play Urgency (FPU) (Gelly and Wang, 2006), which encourages exploitation in the early stages by assigning a fixed score to unvisited nodes. Another enhancement that has been shown to improve MCTS in go is Rapid Action Value Estimation (RAVE) (Helmbold and Parker-Wood, 2009) which updates statistics in nodes, with a decreasing effect, when their corresponding action is selected during rollouts. Portfolio Greedy Search (Churchill and Buro, 2013) and Hierarchical Portfolio Search

(Churchill and Buro, 2015) introduced a script-based approach, which have also been applied to MCTS in StarCraft (Justesen et al., 2014). NaïveMCTS builds a tree where each node corresponds to a combination of actions, and the exploration policy is based on a naive assumption that a unit’s actions are independent of other units’ actions (Ontanón, 2013). Portfolio Greedy Search and NaïveMCTS require that actions must be tied to units, as is common in real-time strategy games. Progressive strategies have been used to limit the search space with success in go (Chaslot et al., 2008b) by focusing the search using domain knowledge and then slowly unpruning nodes. Several progressive pruning methods have shown to improve MCTS for go (Bouzy, 2005), where the idea is to prune nodes that are statistically inferior to their siblings. Sequential halving splits the time budget into a number of phases wherein exploration happens in a uniform manner and after each phase the worst half of the nodes are eliminated (Karnin et al., 2013). MCTS can use macro-actions (repeated actions) to reduce the depth of the search tree, which can be beneficial in domains that require continuous control (Powley et al., 2012).

2.5.4 Evolutionary Planning

Evolutionary algorithms have been used to evolve controllers for numerous games (Risi and Togelius, 2015) where fixed behaviors are evolved *offline* in a training phase. Perez et al. (2012) introduced *Rolling Horizon Evolutionary Algorithms* (RHEAs) that runs online while the agent is playing to evolve the future action sequences. RHEAs have been applied, with good results, to a number of real-time environments including the Physical Traveling Salesman Problem and many games in the General Video Game AI (GVG-AI) framework (Gaina et al., 2017a; Levine et al., 2013). RHEAs evolve a sequence of actions for a fixed number of steps into the future. After the agent’s time budget is used, the first action in the most fit action sequence is performed where after new actions sequences are evolved from scratch one step further into the future, i.e. the horizon is “rolling”. The evolved action sequences are evaluated by simulating these in a forward model and evaluating the outcome. Online evolutionary planning, originally called *online evolution*, (Justesen et al., 2016, 2017) is a similar formulation of RHEAs that is suited for multi-action adversarial games as the entire time budget is spent to evolve an action sequence for a complete turn, which is then executed to end. A portfolio-based version

of evolutionary planning has also been tested for small-scale battles in StarCraft (Wang et al., 2016a) using scripted behaviors instead of actions. *Evolutionary planning* will be used throughout this dissertation to refer to the class of planning algorithms that is based on evolutionary algorithms, encompassing both RHEAs, online evolutionary planning, and their variants.

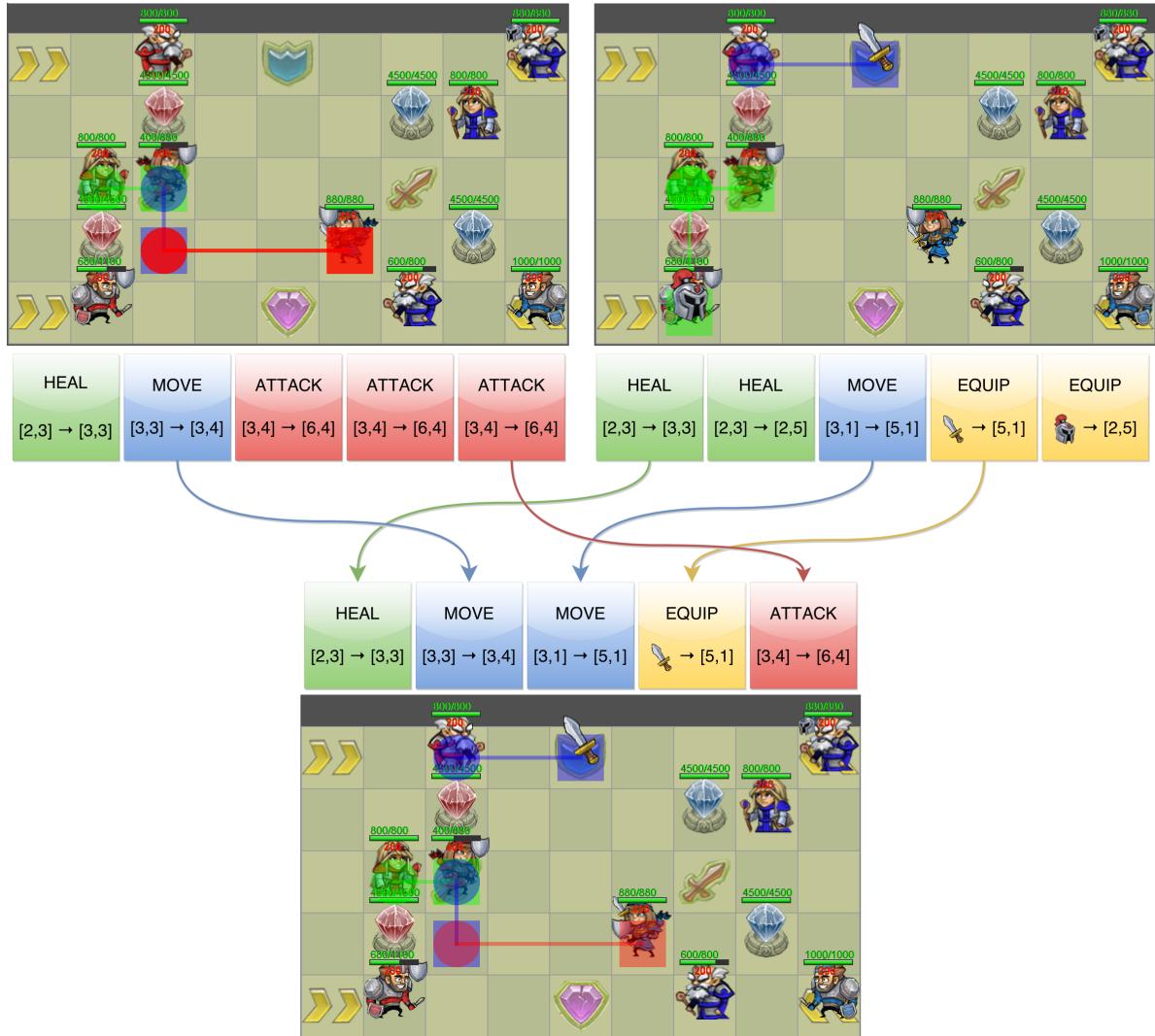


Figure 2.5.3: An example of uniform crossover when using evolutionary planning to evolve actions sequences for the game Hero Academy. Two action sequences (top) results in a new action sequence (bottom). This figure is from Justesen et al. (2017).

Chapter 3

State-of-the-art in AI for Video-game Playing

This chapter describes state-of-the-art AI methods for game-playing. This survey of methods does not include all methods for all games. Instead we focus on methods that are used to play RTS games as well as game-playing methods that employ deep neural networks, as the contributions of this dissertation build on work in these two areas. First, classic methods for playing RTS games, that are not based on deep neural networks, are reviewed. Then, a broad survey is presented on methods that do employ deep neural networks for game-playing, across many different game genres, including RTS games. The reader should be aware that many of the surveyed methods were published in parallel with, or after, the work in this dissertation was published.

3.1 AI for Real-Time Strategy Games

Strategy games are games where the player controls multiple characters or units, and the objective of the game is to prevail in some sort of conquest or conflict. Usually, but not always, the narrative and graphics reflect a military conflict, where units may be e.g. knights, tanks or battleships. The key challenge in strategy games is to lay out and execute complex plans involving multiple units. This challenge is in general significantly harder than the planning challenge in classic board games such as chess mainly because

multiple units must be moved at any time and the effective branching factor is typically enormous. The planning horizon can be extremely long, where actions that are taken at the beginning of a game impact the overall strategy. In addition, there is the challenge of predicting the moves of one or several adversaries, who have multiple units themselves. Real-time strategy games (RTS) are strategy games which are usually understood to not progress in discrete turns, but where actions can be taken at any point in time. In fact, most RTS games do implement turns, by having 24 or more steps every second where each player can take actions. RTS games add the challenge of time prioritization to the already substantial challenges of playing strategy games.

Researchers have considered RTS games, and StarCraft in particular, to be the hardest games for computers to play (Čertíký and Churchill, 2017; Yannakakis and Togelius, 2018a). The challenge of creating artificial agents for RTS games were first proposed by Buro (2003). In this dissertation, several experiments use the RTS games StarCraft as a testbed. This section first gives a brief introduction to the game, then provides a background on several different StarCraft playing bots and the techniques used. This section describes classic approaches to RTS games that does not apply neural networks, while we will return to RTS games in the next section that include approaches to game-playing using neural networks.

3.1.1 StarCraft

StarCraft is a real-time strategy (RTS) game released by Blizzard Entertainment in 1998. Its expansion set StarCraft: Brood War was released later the same year and became extremely popular as an e-sport. The sequel StarCraft II was released in 2010 with the same core gameplay but has a more modern interface as well as several new units and maps. In this dissertation, we use both StarCraft: Brood War (Blizzard Entertainment, 1998) and StarCraft II: Legacy of the Void (Blizzard Entertainment, 2015), while *StarCraft* will be used to refer to these games collectively.

Players control one of three races in StarCraft, Terran, Protoss and Zerg, each with their own strengths and weaknesses. Each player starts with four workers (even more in StarCraft II) that can gather resources and construct buildings, as well as a base that

can produce more workers. As the game progresses each player produces combat units, buildings, technologies, and upgrades (jointly referred to as *builds*) until one player is able to overrun the opponent’s base. Advanced builds require that some basic builds are produced first and these requirements form a tree structure called a *tech tree*¹. A major part of a player’s strategy is the order of builds produced, i.e. the *build order*, which determines the number and combination of units the player will have throughout the game.

StarCraft provides incomplete information about the game state, since the opponent’s base is initially hidden and must be explored by scouting units. This, combined with the fact that multiple agents must be controlled in real-time, makes it a challenging environment for decision making agents. The decision-making process can be split into *micro-management* and *macro-management* tasks. We define **micro-management** as *the tactical control of individual units and buildings*, and **macro-management** as *the strategic planning of what builds to produce and in which order..*

Bots can communicate with the StarCraft: Brood War game using the Brood War Application Programming Interface (BWAPI)². BWAPI allows other C++ programs to access the game state in StarCraft: Brood War as well as giving commands to units, and is used by all the bots in the aforementioned competitions.

3.1.2 Modular StarCraft Bots

The game has become an important benchmark in the field of game AI with several competitions such as the *AIIDE StarCraft AI Competition*³, the *CIG StarCraft RTS AI Competition*⁴ and the *Student StarCraft AI Competition*⁵. Many challenges must be overcome to succeed in these competitions, such as terrain analysis, pathfinding, and build order scheduling. Up to 2016, the most successful StarCraft bots relied mainly on hard-coded strategies (Ontanón et al., 2013; Churchill et al., 2016). Many of these

¹The tech trees for the three races in StarCraft:Brood War: https://liquipedia.net/starcraft/Technology_tree and for StarCraft II: https://liquipedia.net/starcraft2/Terran_Tech_Tree, https://liquipedia.net/starcraft2/Protoss_Tech_Tree, and https://liquipedia.net/starcraft2/Zerg_Tech_Tree.

²<http://bwapi.github.io/>

³<http://www.cs.mun.ca/~dchurchill/starcraftaicomp/>

⁴<https://liquipedia.net/starcraft/CIG>

⁵<http://sscaitournament.com/>

bots implement hard-coded build orders and are only able to adapt by following some predefined rules. The problem with hard-coded approaches is that the bot is limited to a fixed set of strategies, but more importantly the ability to adapt to what happens in the game is restricted as well. While a hard-coded approach can be successful against many other bots, it is easy for human players to counter these strategies.

Most StarCraft bots have a modular design, in which the tasks are divided into smaller sub-problems (i.e. a *divide and conquer* strategy). These modules often form hierarchy of abstraction that enables the top-level modules to perform macro-management tasks while lower level modules perform micro-management. The open source UAlbertaBot⁶ by David Churchill is an example of such an approach. The strategy manager maintains the strategy and communicates to the production manager what build order to follow. The production manager then takes care of assigning workers and buildings to produce the next builds in the queue, which happens simultaneously while the combat manager controls units in battle and the scout manager controls any scouting units. A strategy for UAlbertaBot can be described in a configuration file as a scripted build order; a hard-coded strategy followed by the bot. The modular design is described in more detail in Ontanón et al. (2013).

3.1.3 Build-order Optimization and Adaptation in StarCraft

There exist several approaches to build-order search and optimization for StarCraft. Churchill and Buro (2011) implemented a depth-first branch & bound algorithm that finds the shortest possible time span to achieve a given goal (i.e. a list of units the build order should obtain). The problem of optimizing opening build-orders has also been approached with multi-objective evolutionary algorithms (Kuchem et al., 2013; Köstler and Gmeiner, 2013; Blackford and Lamont, 2014) by encoding the genotype as a list of builds. The strength of these methods is that they do not evolve build orders to reach one goal, but several. While these approaches to build-order optimization work well, even when compared to professional players, they are only designed to find an opening

⁶<https://github.com/davechurchill/ualbertabot>

build-order and do not take the opponent’s strategy into account as the game progresses. Synnaeve and Bessiere (2011) showed promising results for adaptive build-order planning; their approach can predict the opponent’s strategy from the noisy observations in the game using a Bayesian model. However, their approach relies on hard-coded rules on top of the prediction model and it is unknown how well it will work when integrated into a bot. Another approach, by Garéa-Sánchez et al. (2015), demonstrates how a complete strategy can be evolved, including both macro-management and micro-management behaviors. The evolved strategies are however static and do not change during a game. Some attempts have been made to predict the opponent’s strategy from partial information (Cho et al., 2013; Synnaeve and Bessiere, 2011), but it has not been demonstrated how these approaches can be applied to build-order planning.

The most recent StarCraft AI competition results, at the time of writing, are from the AIIDE 2018 StarCraft AI Competition⁷. Here, a few new advanced modular bots appeared. The second place bot, *CherryPi* from Facebook AI Research used inter-game build-order adaptive optimization technique: “CherryPi has 8-13 strategies per matchup. It selects one at the start of the game based on history against the current opponent, using a bandit model with time-decaying weights on previous games. Using a pre-trained model, and given the current state of the game, it estimates the expected likelihood of winning (“value”) with each of the available strategies, and under certain conditions will switch to the strategy with the highest value. This can produce “mixed” strategies by switching back and forth”⁸. This is an interesting intra-adaptive approach that is limited by a small set of pre-defined strategies, while it also allows for inter-game adaptation.

Interestingly, the winner did not use any build-order adaptation but instead learned the optimal timing for a well executed all-in attack: “SAIDA basically plays the mechanic Terran in all games. it starts with a stable defense-first strategy and after mid stage of game, it seeks the best rush timing and win the game with a powerful one-shot attack.”⁹

⁷<http://www.cs.mun.ca/~dchurchill/starcraftaicomp/2018/>

⁸http://www.cs.mun.ca/~dchurchill/starcraftaicomp/2018/aiide/AIIDE2018_StarcraftAICompetition.pdf

⁹http://www.cs.mun.ca/~dchurchill/starcraftaicomp/2018/aiide/AIIDE2018_StarcraftAICompetition.pdf

3.2 Playing Video Games with Deep Neural Networks

In this section, recent advances in deep learning for video game playing and employed game research platforms are reviewed. This review is written from the perspective of different types of games, the challenges they pose for deep learning, and how deep neural networks can be used to play these games. A variety of review articles on deep learning exists (Goodfellow et al., 2016; LeCun et al., 2015; Schmidhuber, 2015), as well as surveys on reinforcement learning (Sutton and Barto, 1998), neuroevolution in games (Risi and Togelius, 2015) and deep reinforcement learning (Li, 2018), here we focus on these techniques applied to deep neural networks for video game playing.

In particular, we focus on game problems and environments that have been used extensively for deep-learning-based game AI, such as Atari/ALE, Doom, Minecraft, StarCraft, and car racing. Additionally, we review existing work that aim to play a particular *video game* well (in contrast to board games such as go), from pixels or feature vectors, without an existing forward model.

It is important to note that there are many uses of AI in and for games that are not covered here; Game AI is a large and diverse field (Yannakakis and Togelius, 2018b, 2015; Miikkulainen et al., 2006; Galway et al., 2008; Muñoz-Avila et al., 2013). Instead, we focus on methods that apply deep neural networks for playing video games well, while there is plenty of research on playing games in a believable, entertaining or human-like manner (Hingston, 2012). AI is also used for modeling players' behavior, experience or preferences (Yannakakis et al., 2013), or generating game content such as levels, textures or rules (Shaker et al., 2016).

Figure 3.2.2 shows an influence diagram with the reviewed methods and their relations to earlier methods. Each method in the diagram is colored to show the game benchmark. Screenshots from some of these games are shown in Figure 3.2.3.

The video games that are discussed here have to a large extent supplanted an earlier generation of simpler control problems that long served as the main reinforcement learning benchmarks but are generally too simple for modern reinforcement learning methods. In

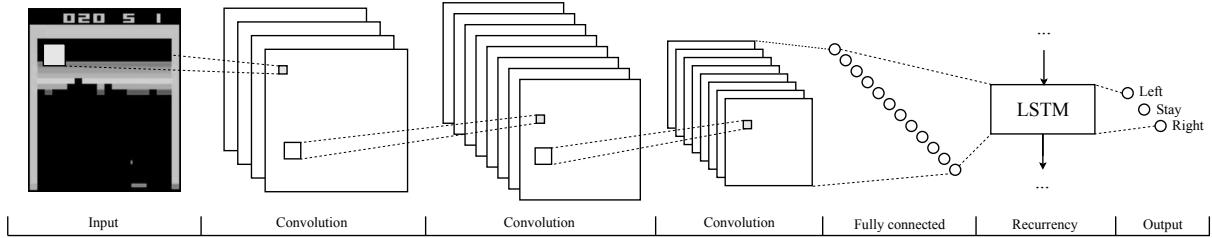


Figure 3.2.1: An example of a typical network architecture used in deep reinforcement learning for game-playing with pixel input. The input usually consists of a preprocessed screen image, or several stacked or concatenated images, which is followed by a couple of convolutional layers (often without pooling), and a few fully connected layers. Recurrent networks have a recurrent layer, such as LSTM or GRU, after the fully connected layers. The output typically consists of one unit for each unique combination of actions in the game, and actor-critic methods also have one for the state value $V(s)$. Examples of this architecture, without a recurrent layer and with some variations, are Mnih et al. (2013, 2015); Nair et al. (2015); Van Hasselt et al. (2016); Schaul et al. (2016); Osband et al. (2016); Mnih et al. (2016); Wang et al. (2017a); Rusu et al. (2016b); Salimans et al. (2017); Bellemare et al. (2017); Fortunato et al. (2018); Wang et al. (2016b); Hessel et al. (2018); Wu et al. (2017b); Such et al. (2017); Conti et al. (2018); Espeholt et al. (2018), and examples with a recurrent layer are Hausknecht and Stone (2015); Mnih et al. (2016); Jaderberg et al. (2017).

such classic control problems, the input is a simple feature vector, describing the position, velocity, and angles etc. Popular platforms for such problems are rllab (Duan et al., 2016), which includes classic problems such as pole balancing and the mountain car problem, and MuJoCo (Multi-Joint dynamics with Contact), a physics engine for complex control tasks such as the humanoid walking task (Todorov et al., 2012).

3.2.1 Arcade Games

Classic arcade games, of the type found in the late seventies' and early eighties' arcade cabinets, home video game consoles and home computers, have been commonly used as AI benchmarks within the last decade. Representative platforms for this game type are the Atari 2600, Nintendo NES, Commodore 64 and ZX Spectrum. Most classic arcade games are characterized by movement in a two-dimensional space (sometimes represented isometrically to provide the illusion of three-dimensional movement), heavy use of graphical logics (where game rules are triggered by the intersection of sprites or images), continuous-time progression, and either continuous-space or discrete-space movement. The challenges of playing such games vary by game. Most games require fast

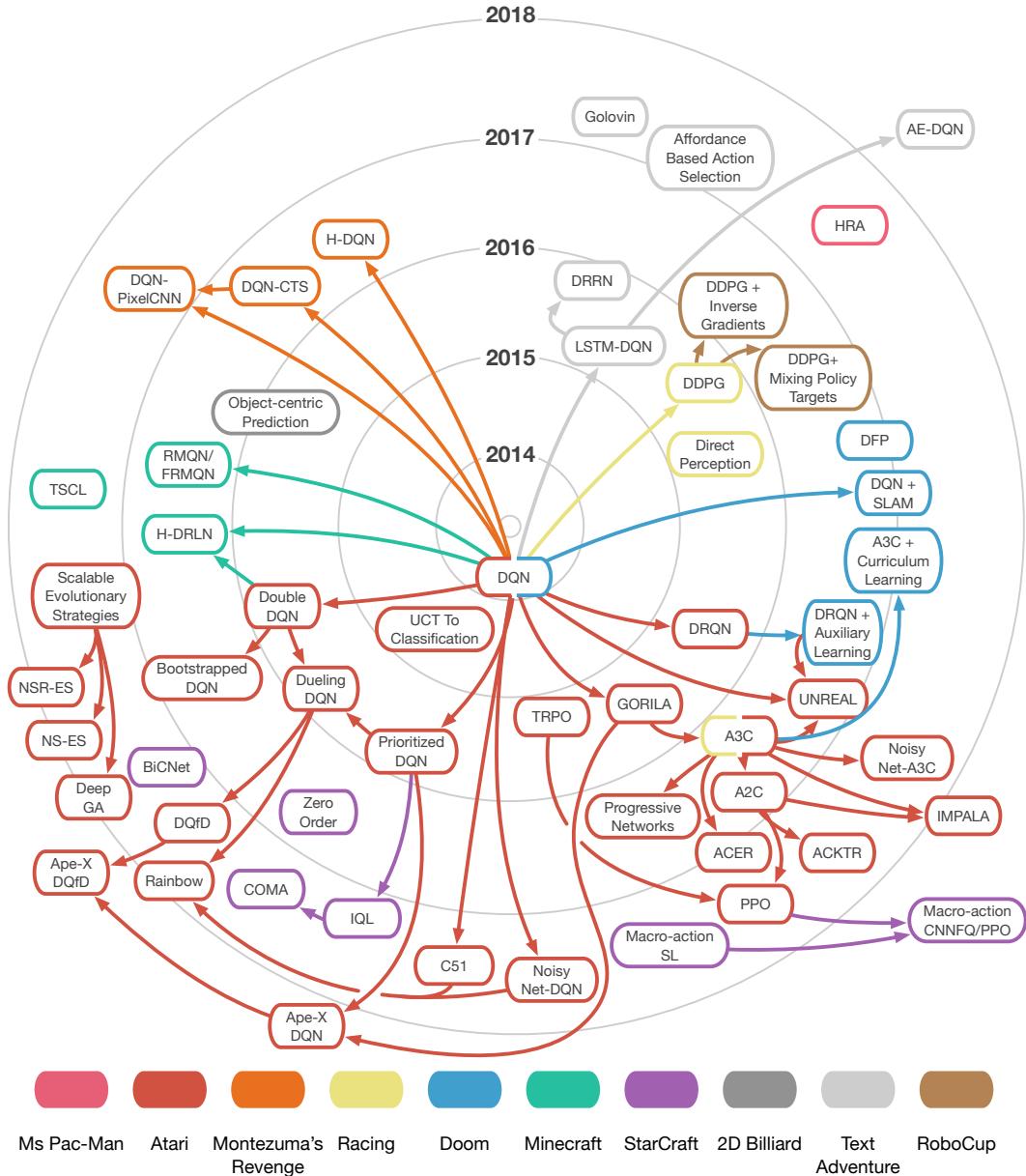


Figure 3.2.2: Influence diagram of the deep learning techniques discussed in this paper. Each node is an algorithm while the color represents the game benchmark. The distance from the center represents the date that the original paper was published on arXiv. The arrows represent how techniques are related. Each node points to all other nodes that used or modified that technique. Arrows pointing to a particular algorithm show which algorithms influenced its design. Influences are not transitive: if algorithm a influenced b and b influenced c , a did not necessarily influence c . AlphaStar and OpenAI Five are described in this section but are not on this diagram as they are very recent approaches that have not yet been peer-reviewed.

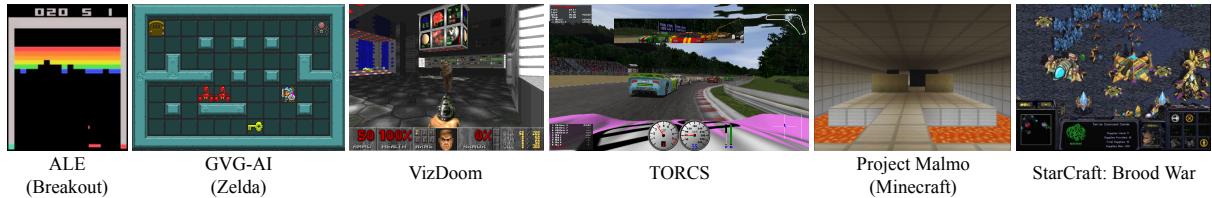


Figure 3.2.3: Screenshots of selected games and frameworks used as research platforms for research in deep learning.

reactions and precise timing, and a few games, in particular, early sports games such as *Track & Field* (Konami, 1983) rely almost exclusively on speed and reactions. Many games require prioritization of several co-occurring events, which requires some ability to predict the behavior or trajectory of other entities in the game. This challenge is explicit in e.g. *Tapper* (Bally Midway, 1983) but also in different ways part of platform games such as *Super Mario Bros* (Nintendo, 1985) and shooters such as *Missile Command* (Atari Inc., 1980). Another common requirement is navigating mazes or other complex environments, as exemplified clearly by games such as *Pac-Man* (Namco, 1980) and *Boulder Dash* (First Star Software, 1984). Some games, such as *Montezuma's Revenge* (Parker Brothers, 1984), require long-term planning involving the memorization of temporarily unobservable game states. Some games feature incomplete information and stochasticity, others are completely deterministic and fully observable.

The most notable game platform used for deep learning methods is the Arcade Learning Environment (ALE) (Bellemare et al., 2013). ALE is built on top of the Atari 2600 emulator Stella and contains more than 50 original Atari 2600 games. The framework extracts the game score, 160×210 screen pixels and the RAM content that can be used as input for game playing agents. ALE was the main environment explored in the first deep reinforcement learning papers that used raw pixels as input. By enabling agents to learn from visual input, ALE thus differs from classic control problems in the reinforcement learning literature, such as the Cart Pole and Mountain Car problems and. An longer overview and discussion of the ALE environment can be found in (Machado et al., 2018).

Another platform for classic arcade games is the Retro Learning Environment (RLE) that currently contains seven games released for the Super Nintendo Entertainment System (SNES) (Bhonker et al., 2017). Many of these games have 3D graphics and the controller allows for over 720 action combinations. SNES games are thus more complex and realistic

than Atari 2600 games but RLE has not been as popular as ALE.

The General Video Game AI (Gvg-AI) framework (Perez-Liebana et al., 2016) allows for easy creation and modification of games and levels using the Video Game Description Language (VGDL) (Schaul, 2013). There are currently over 160 games in Gvg-AI, each with five levels. The VGDL game definition specifies objects in the game and interaction rules such as rewards and effects of collisions. A level is defined as an ASCII grid where each character represents an object. This allows for quick development of games and levels making the framework ideal for research purposes (Perez-Liebana et al., 2018). The GvgAI framework has been integrated with the OpenAI Gym environment (Rodriguez Torrado et al., 2018). While Gvg-AI originally provides a forward model that allows agents to use search algorithms, the Gvg-AI Gym only provides the pixels of each frame, the incremental reward, and whether the game is won or lost.

3.2.1.1 Reinforcement Learning Methods in ALE

This section will give describe advancements of reinforcement learning methods that have been demonstrated in ALE and an overview is given in Table 3.2.1.

Deep Q-Network (DQN) was the first learning algorithm that showed human expert-level control in ALE (Mnih et al., 2013). DQN was tested in seven Atari 2600 games and outperformed previous approaches, such as SARSA with feature construction (Bellemare et al., 2015) and neuroevolution (Hausknecht et al., 2014), as well as a human expert on three of the games. DQN is based on Q-learning, where a neural network model learns to approximate $Q^\pi(s, a)$ that estimates the expected return of taking action a in state s while following a behavior policy μ . A simple network architecture consisting of two convolutional layers followed by a single fully-connected layer was used as a function approximator. A key mechanism in DQN is *experience replay* (Lin, 1993), where experiences in the form $\{s_t, a_t, r_{t+1}, s_{t+1}\}$ are stored in a replay memory and randomly sampled in batches when the network is updated. This enables the algorithm to reuse and learn from past and uncorrelated experiences, which reduces the variance of the updates. DQN was later extended with a separate target Q-network which parameters are held fixed between individual updates and was shown to achieve above human expert scores in 29 out of 49 tested games (Mnih et al., 2015).

Deep Recurrent Q-Learning (DRQN) extends the DQN architecture with a recurrent layer before the output and works well for games with partially observable states (Hausknecht and Stone, 2015).

A distributed version of DQN was shown to outperform a non-distributed version in 41 of the 49 games using the Gorila architecture (General Reinforcement Learning Architecture) (Nair et al., 2015). Gorila parallelizes actors that collect experiences into a distributed replay memory as well as parallelizing learners that train on samples from the same replay memory.

One problem with the Q-learning algorithm is that it often overestimates action values because it uses the same value function for action-selection and action-evaluation. Double DQN, based on double Q-learning (Hasselt, 2010), reduces the observed overestimation by learning two value networks with parameters θ and θ' that both use the other network for value-estimation, such that the target $Y_t = R_{t+1} + \gamma Q(S_{t+1}, \max_a Q(S_{t+1}, a; \theta_t); \theta'_t)$ (Van Hasselt et al., 2016).

Another improvement is *prioritized experience replay* from which important experiences are sampled more frequently based on the TD-error, which was shown to significantly improve both DQN and Double DQN (Schaul et al., 2016).

Dueling DQN uses a network that is split into two streams after the convolutional layers to separately estimate state-value $V^\pi(s)$ and the action-advantage $A^\pi(s, a)$, such that $Q^\pi(s, a) = V^\pi(s) + A^\pi(s, a)$ (Wang et al., 2016b). Dueling DQN improves Double DQN and can also be combined with prioritized experience replay.

Double DQN and Dueling DQN were also tested in the five more complex games in the RLE and achieved a mean score of around 50% of a human expert (Bhonker et al., 2017). The best result in these experiments was by Dueling DQN in the game Mortal Kombat (Midway, 1992) with 128%.

Bootstrapped DQN improves exploration by training multiple Q-networks. A randomly sampled network is used during each training episode and *bootstrap masks* modulate the gradients to train the networks differently (Osband et al., 2016).

Robust policies can be learned with DQN for competitive or cooperative multi-player games by training one network for each player and play them against each other in the

training process (Tampuu et al., 2017). Agents trained in multiplayer mode perform very well against novel opponents, whereas agents trained against a stationary algorithm fail to generalize their strategies to novel adversaries.

Multi-threaded asynchronous variants of DQN, SARSA and Actor-Critic methods can utilize multiple CPU threads on a single machine, reducing training roughly linear to the number of parallel threads (Mnih et al., 2016). These variants do not rely on a replay memory because the network is updated on uncorrelated experiences from parallel actors which also helps to stabilize on-policy methods. The Asynchronous Advantage Actor-Critic (A3C) algorithm is an actor-critic method that uses several parallel agents to collect experiences that all asynchronously update a global actor-critic network. A3C outperformed Prioritized Dueling DQN, which was trained for 8 days on a GPU, with just half the training time on a CPU (Mnih et al., 2016).

An actor-critic method with experience replay (ACER) implements an efficient trust region policy method that forces updates to not deviate far from a running average of past policies (Wang et al., 2017a). The performance of ACER in ALE matches Dueling DQN with prioritized experience replay and A3C without experience replay, while it is much more data efficient.

A3C with progressive neural networks (Rusu et al., 2016b) can effectively transfer learning from one game to another. The training is done by instantiating a network for every new task with connections to all the previous learned networks. This gives the new network access to knowledge already learned.

The *Advantage Actor-Critic* (A2C), a synchronous variant of A3C (Mnih et al., 2016), updates the parameters synchronously in batches and has comparable performance while only maintaining one neural network (Wu et al., 2017b). Actor-Critic using Kronecker-Factored Trust Region (ACKTR) extends A2C by approximating the natural policy gradient updates for both the actor and the critic (Wu et al., 2017b). In Atari, ACKTR has slower updates compared to A2C (at most 25% per time step) but is more sample efficient (e.g. by a factor of 10 in Atlantis) (Wu et al., 2017b). Trust Region Policy Optimization (TRPO) uses a *surrogate* objective with theoretical guarantees for monotonic policy improvement, while it practically implements an approximation called *trust region* (Schulman et al., 2015). This is done by constraining network updates with a bound on

the KL divergence between the current and the updated policy. TRPO has robust and data efficient performance in Atari games while it has high memory requirements and several restrictions. Proximal Policy Optimization (PPO) is an improvement on TRPO that uses a similar *surrogate* objective (Schulman et al., 2017), but instead uses a soft constraint (originally suggested in Schulman et al. (2015)) by adding the KL-divergence as a penalty. Instead of having a fixed penalty coefficient, it uses a clipped surrogate objective that penalizes policy updates outside some specified interval. PPO was shown to be more sample efficient than A2C and on par with ACER in Atari, while PPO does not rely on replay memory. PPO was also shown to have comparable or better performance than TRPO in continuous control tasks while being simpler and easier to parallelize.

IMPALA (Importance Weighted Actor-Learner Architecture) is an actor-critic method where multiple learners with GPU access share gradients between each other while being synchronously updated from a set of actors (Espeholt et al., 2018). This method can scale to a large number of machines and outperforms A3C. Additionally, IMPALA was trained, with one set of parameters, to play all 57 Atari games in ALE with a mean human-normalized score of 176.9% (median of 59.7%) (Espeholt et al., 2018). Experiences collected by the actors in the IMPALA setup can lack behind the learners' policy and thus result in off-policy learning. This discrepancy is mitigated through a *V-trace* algorithm that weighs the importance of experiences based on the difference between the actor's and learner's policies (Espeholt et al., 2018).

UNREAL (UNsupervised REinforcement and Auxiliary Learning) algorithm is based on A3C but uses a replay memory from which it learns auxiliary tasks and *pseudo-reward functions* concurrently (Jaderberg et al., 2017). UNREAL only shows a small improvement over vanilla A3C in ALE, but larger improvements in other domains (see Section 3.2.3). *Distributional DQN* takes a distributional perspective on reinforcement learning by treating $Q(s, a)$ as an approximate distribution of returns instead of a single approximate expectation for each action (Bellemare et al., 2017). The distribution is divided into a so-called set of atoms, which determines the granularity of the distribution. Their results show that the more fine-grained the distributions are, the better are the results, and with 51 atoms (this variant was called C51) it achieved mean scores in ALE almost comparable to UNREAL.

In *NoisyNets*, noise is added to the network parameters and a unique noise level for each parameter is learned using gradient descent (Fortunato et al., 2018). In contrast to ϵ -greedy exploration, where an agent either samples actions from the policy or from a uniform random distribution, NoisyNets use a noisy version of the policy to ensure exploration, and this was shown to improve DQN (NoisyNet-DQN) and A3C (NoisyNet-A3C).

Rainbow combines several DQN enhancements: Double DQN, Prioritized Replay, Dueling DQN, Distributional DQN, and NoisyNets, and achieved a mean score higher than any of the enhancements individually (Hessel et al., 2018).

Evolution Strategies (ES) are black-box optimization algorithms that rely on parameter-exploration through stochastic noise instead of calculating gradients and were found to be highly parallelizable with a linear speedup in training time when more CPUs are used (Salimans et al., 2017). 720 CPUs were used for one hour whereafter ES managed to outperform A3C (which ran for 4 days) in 23 out of 51 games, while ES used 3 to 10 times as much data due to its high parallelization. ES only ran a single day and thus their full potential is currently unknown. Novelty search is a popular algorithm that can overcome environments with deceptive and/or sparse rewards by guiding the search towards novel behaviors (Lehman and Stanley, 2008). ES has been extended to use Novelty Search (NS-ES) which outperforms ES on several challenging Atari games by defining novel behaviors based on the RAM states (Conti et al., 2018). A quality-diversity variant called NSR-ES that uses both novelty and the reward signal reach an even higher performance (Conti et al., 2018). NS-ES and NSR-ES reached worse results on a few games, possibly where the reward function is not sparse or deceptive.

A simple genetic algorithm with a Gaussian noise mutation operator evolves the parameters of a deep neural network (Deep GA) and can achieve surprisingly good scores across several Atari games (Such et al., 2017). Deep GA shows comparable results to DQN, A3C, and ES on 13 Atari games using up to thousands of CPUs in parallel. Additionally, random search, given roughly the same amount of computation, was shown to outperform DQN on 4 out of 13 games and A3C on 5 games (Such et al., 2017). While there has been concern that evolutionary methods do not scale as well as gradient descent-based methods, one possibility is separating the feature construction from the policy network; evolutionary algorithms can then create extremely small networks that still play well

Results	Mean	Median	Orig. paper on arXiv
DQN (Wang et al. (2016b))	228%	79%	2013, Mnih et al. (2013)
Double DQN (DDQN) (Wang et al. (2016b))	307%	118%	2015, Van Hasselt et al. (2016)
Dueling DDQN (Wang et al. (2016b))	373%	151%	2015, Wang et al. (2016b)
Prior. DDQN (Wang et al. (2016b))	435%	124%	2015, Schaul et al. (2016)
Prior. Duel DDQN (Wang et al. (2016b))	592%	172%	2015, Schaul et al. (2016)
A3C (Jaderberg et al. (2017))	853%	N/A	2016, Mnih et al. (2016)
UNREAL (Jaderberg et al. (2017))*	880%	250%	2016, Jaderberg et al. (2017)
NoisyNet-DQN (Hessel et al. (2018))	N/A	118%	2017, Fortunato et al. (2018)
Distr. DQN (C51) (Bellemare et al. (2017))	701%	178%	2017, Bellemare et al. (2017)
Rainbow (Hessel et al. (2018))	N/A	223%	2017, Hessel et al. (2018)
IMPALA (Espeholt et al. (2018))	958%	192%	2018, Espeholt et al. (2018)
Ape-X DQN (Horgan et al. (2018))	N/A	434%	2018, Horgan et al. (2018)

Table 3.2.1: Human-normalized scores reported with various deep reinforcement learning algorithms in ALE on 57 Atari games using the *30 no-ops* evaluation metric. References in the first column refer to the paper that included the reported results, while the last column references the paper that first introduced the specific algorithm. Note, that the reported scores use various amounts of training time and resources, thus not entirely comparable. Successors typically use more resources and less wall-clock time. *Hyper-parameters was tuned for every game leading to higher scores for UNREAL.

(Cuccu et al., 2018).

A few supervised learning approaches have been applied to arcade games. In Guo et al. (2014) a slow planning agent was applied offline, using Monte-Carlo Tree Search, to generate data for training a CNN via multinomial classification. This approach, called *UCTtoClassification*, was shown to outperform DQN. Policy distillation (Rusu et al., 2016a) or actor-mimic (Parisotto et al., 2016) methods can be used to train one network to mimic a set of policies (e.g. for different games). These methods can reduce the size of the network and sometimes also improve the performance. A frame prediction model can be learned from a dataset generated by a DQN agent using the encoding-transformation-decoding network architecture; the model can then be used to improve exploration in a retraining phase (Oh et al., 2015). Self-supervised tasks, such as reward prediction, validation of state-successor pairs, and mapping states and successor states to actions can define auxiliary losses used in pre-training of a policy network, which ultimately can improve learning (Shelhamer et al., 2016).

The *training objective* provides feedback to the agent while the *performance objective* specifies the target behavior. Often, a single reward function takes both roles, but for some games, the performance objective does not guide the training sufficiently. The Hybrid

Reward Architecture (HRA) splits the reward function into n different reward functions, where each of them are assigned a separate learning agent (Van Seijen et al., 2017). HRA does this by having n output streams in the network, and thus n Q-values, which are combined when actions are selected. HRA was able to achieve the maximum possible score in less than 3,000 episodes.

3.2.1.2 Montezuma’s Revenge

Environments with sparse feedback remain an open challenge for reinforcement learning. The game *Montezuma’s Revenge* is a good example of such an environment in ALE and has thus been studied in more detail and used for benchmarking learning methods based on intrinsic motivation and curiosity. The main idea of applying intrinsic motivation is to improve the exploration of the environment based on some self-rewarding system, which eventually will help the agent to obtain an extrinsic reward. DQN fails to obtain any reward in this game (receiving a score of 0) and Gorila achieves an average score of just 4.2. A human expert can achieve 4,367 points and it is clear that the methods presented so far are unable to deal with environments with such sparse rewards. A few promising methods aim to overcome these challenges.

Hierarchical-DQN (h-DQN) (Kulkarni et al., 2016) operates on two temporal scales, where one Q-value function $Q_1(s, a; g)$, the *controller*, learns a policy over actions that satisfy goals chosen by a higher-level Q-value function $Q_2(s, g)$, the *meta-controller*, which learns a policy over intrinsic goals (i.e. which goals to select). This method was able to reach an average score of around 400 in Montezuma’s Revenge where goals were defined as states in which the agent *reaches* (collides with) a certain type of object. This method, therefore, must rely on some object detection mechanism.

Pseudo-counts have been used to provide intrinsic motivation in the form of exploration bonuses when unexpected pixel configurations are observed and can be derived from CTS density models (Bellemare et al., 2016) or neural density models (Ostrovski et al., 2017). Density models assign probabilities to images, and a model’s pseudo count of an observed image is the model’s change in prediction compared to being trained one additional time on the same image. Impressive results were achieved in Montezuma’s Revenge and other hard Atari games by combining DQN with the CTS density model

(DQN-CTS) or the PixelCNN density model (DQN-PixelCNN) (Bellemare et al., 2016). Interestingly, the results were less impressive when the CTS density model was combined with A3C (A3C-CTS) (Bellemare et al., 2016). In Random Network Distillation (RND) a fixed randomly initialized *target* network produces a k -dimensional output from the agent’s training observations and a *predictor* network is trained to distil the target network (Burda et al., 2018). It is thus expected that the prediction error is high when the novelty of the observation is high, and it can thus be used as an exploration bonus to train the policy. RND achieved an average score of 8,152 in Montezuma’s Revenge which is higher than the average human score of 4,753. A potential issue with RND in games is that many dangerous states gives a high reward because they are typically rare. Go-Explore is a quite different algorithm that stores down-sampled observations in a archive (similar to MAP-Elites) together with the trajectories that lead to them. In a final step, a robust policy is learned to imitate the trajectories which can deal with stochasticity. Go-Explore reached an impressive average score above 43,000 points in Montezuma’s Revenge and above 650,000 points when using domain knowledge to construct the cell division in the archive. While Go-Explore was tested in the stochastic version Montezuma’s Revenge its state-exploration phase was done in a deterministic version of the game.

Ape-X DQN is a distributed DQN architecture similar to Gorila, as in actors are separated from the learner. Ape-X DQN was able to reach state-of-art results across the 57 Atari games using 376 cores and 1 GPU, running at 50K FPS (Horgan et al., 2018). Deep Q-learning from Demonstrations (DQfD) draw samples from an experience replay buffer that is initialized with demonstration data from a human expert and is superior to previous methods on 11 Atari games with sparse rewards (Hester et al., 2018). Ape-X DQfD combines the distributed architecture from Ape-X and the learning algorithm from DQfD using expert data and was shown to outperform all previous methods in ALE as well as beating level 1 in Montezuma’s Revenge (Pohlen et al., 2018).

To improve the performance, Kaplan et al. (2017) augmented the agent training with text instructions. An instruction-based reinforcement learning approach that uses both a CNN for visual input and RNN for text-based instruction, inputs managed to achieve a score of 3,500 points. Instructions were linked to positions in rooms and agents were rewarded when they reached those locations, demonstrating a fruitful collaboration between a

human and a learning algorithm. Experiments in Montezuma’s Revenge also showed that the network learned to generalize to unseen instructions that were similar to previous instructions.

Similar work demonstrates how an agent can execute text-based commands in a 2D maze-like environment called XWORLD, such as walking to and picking up objects, after having learned a *teacher’s* language (Yu et al., 2017). An RNN-based language module is connected to a CNN-based perception module. These two modules were then connected to an action-selection module and a recognition module that learns the teacher’s language in a question answering process.

3.2.2 Racing Games

A challenge that is common in all racing games is that the agent needs to control the position of the vehicle and adjust the acceleration or braking, using fine-tuned continuous input, so as to traverse the track as fast as possible. Doing this optimally requires at least short-term planning, one or two turns forward. If there are resources to be managed in the game, such as fuel, damage or speed boosts, this requires longer-term planning. When other vehicles are present on the track, there is an adversarial planning aspect added, in trying to manage or block overtaking; this planning is often done in the presence of hidden information (position and resources of other vehicles on different parts of the track). A popular environment for visual reinforcement learning with realistic 3D graphics is the open racing car simulator TORCS (Wymann et al., 2000). This section will review methods applied to this platform.

There are generally two paradigms for vision-based autonomous driving highlighted in Chen et al. (2015); (1) end-to-end systems that learn to map images to actions directly (behavior reflex), and (2) systems that parse the sensor data to make informed decisions (mediated perception). An approach that falls in between these paradigms is *direct perception* where a CNN learns to map from images to meaningful affordance indicators, such as the car angle and distance to lane markings, from which a simple controller can make decisions (Chen et al., 2015). Direct perception was trained on recordings of 12 hours of human driving in TORCS and the trained system was able to drive in very

diverse environments. Amazingly, the network was also able to generalize to real images. End-to-end reinforcement learning algorithms such as DQN cannot be directly applied to continuous environments such as racing games because the action space must be discrete and with relatively low dimensionality. Instead, policy gradient methods, such as actor-critic (Degris et al., 2012) and Deterministic Policy Gradient (DPG) (Silver et al., 2014) can learn policies in high-dimensional and continuous action spaces. Deep DPG (DDPG) is a policy gradient method that implements both experience replay and a separate target network and was used to train a CNN end-to-end in TORCS from images (Lillicrap et al., 2016).

The aforementioned A3C methods have also been applied to the racing game TORCS using only pixels as input (Mnih et al., 2016). In those experiments, rewards were shaped as the agent’s velocity on the track, and after 12 hours of training, A3C reached a score between roughly 75% and 90% of a human tester in tracks with and without opponent bots, respectively.

While most approaches to training deep networks from high-dimensional input in video games are based on gradient descent, a notable exception is an approach by Koutník et al. (2013), where Fourier-type coefficients were evolved that encoded a recurrent network with over 1 million weights. Here, evolution was able to find a high-performing controller for TORCS that only relied on high-dimensional visual input.

An imitation learning approach called Generative Adversarial Imitation Learning (GAIL) uses a Generative Adversarial Network (GAN), wherein a policy network (the generator) is trained to fool a discriminator by producing state-action samples that imitates human behaviors (Ho and Ermon, 2016). InfoGAIL (Li et al., 2017) extends GAIL by learning salient semantic features in an unsupervised manner in the style of InfoGAN (Chen et al., 2016), which has been used to find semantic features in images of hand-written letters and digit, as well as photos of faces. InfoGAN (and InfoGAIL) achieves this by adding an additional term to the generator’s loss, such that it attempts to maximize the mutual information between a subset of the noise variables and the observations, such that the latent space becomes less entangled. InfoGAIL was capable of learning a variety of car driving behaviors in TORCS (Li et al., 2017).

3.2.3 First-Person Shooters

More advanced game environments have recently emerged for visual reinforcement learning agents in a First-Person Shooters (FPS). In contrast to classic arcade games such as those in the ALE benchmark, FPSes have 3D graphics with partially observable states and are thus a more realistic environment to study. Usually, the viewpoint is that of the player-controlled character, though some games that are broadly in the FPS categories adopt an over-the-shoulder viewpoint. The design of FPS games is such that part of the challenge is simply fast perception and reaction, in particular, spotting enemies and quickly aiming at them. But there are other cognitive challenges as well, including orientation and movement in a complex three-dimensional environment, predicting actions and locations of multiple adversaries, and in some game modes also team-based collaboration. If visual inputs are used, there is the challenge of extracting relevant information from pixels. Among FPS platforms are *ViZDoom*, a framework that allows agents to play the classic first-person shooter Doom (id Software, 1993–2017) using the screen buffer as input (Kempka et al., 2016). *DeepMind Lab* is a platform for 3D navigation and puzzle-solving tasks based on the *Quake III Arena* (id Software, 1999) engine (Beattie et al., 2016).

Kempka et al. (2016) demonstrated that a CNN with max-pooling and fully connected layers trained with DQN can achieve human-like behaviors in basic scenarios. In the Visual Doom AI Competition 2016¹⁰, a number of participants submitted pre-trained neural network-based agents that competed in a multi-player deathmatch setting. Both a *limited* competition was held, in which bots competed in known levels, and a *full* competition that included bots competing in unseen levels. The winner of the limited track used a CNN trained with A3C using reward shaping and curriculum learning (Wu and Tian, 2017). Reward shaping tackled the problem of sparse and delayed rewards, giving artificial positive rewards for picking up items and negative rewards for using ammunition and losing health. Curriculum learning attempts to speed up learning by training on a set of progressively harder environments (Bengio et al., 2009). The second-place entry in the limited track used a modified DRQN network architecture with an additional stream of fully connected layers to learn supervised auxiliary tasks such as enemy detection, with the purpose of speeding up the training of the convolutional layers (Lample and

¹⁰<http://vizdoom.cs.put.edu.pl/competitions/vdaic-2016-cig>

Chaplot, 2017a). Position inference and object mapping from pixels and depth-buffers using Simultaneous Localization and Mapping (SLAM) also improve DQN in Doom (Bhatti et al., 2016).

The winner of the full deathmatch competition implemented a *Direct Future Prediction* (DFP) approach that was shown to outperform DQN and A3C (Dosovitskiy and Koltun, 2017). The architecture used in DFP has three streams: one for the screen pixels, one for lower-dimensional measurements describing the agent’s current state, and one for describing the agent’s goal, which is a linear combination of prioritized measurements. DFP collects experiences in a memory and is trained with supervised learning techniques to predict the future measurements based on the current state, goal and selected action. During training, actions are selected that yield the best-predicted outcome, based on the current goal. This method can be trained on various goals and generalizes to unseen goals at test time.

Navigation in 3D environments is one of the important skills required for FPS games and has been studied extensively. A CNN+LSTM network was trained with A3C extended with additional outputs predicting the pixel depths and loop closure, showing significant improvements (Mirowski et al., 2016).

The *UNREAL* algorithm, based on A3C, implements an auxiliary task that trains the network to predict the immediate subsequent future reward from a sequence of consecutive observations. UNREAL was tested on fruit gathering and exploration tasks in OpenArena and achieved a mean human-normalized score of 87%, where A3C only achieved 53% (Jaderberg et al., 2017).

The ability to transfer knowledge to new environments can reduce the learning time and can in some cases be crucial for some challenging tasks. Transfer learning can be achieved by pre-training a network in similar environments with simpler tasks or by using random textures during training (Chaplot et al., 2016). The *Distill and Transfer Learning* (Distral) method trains several worker policies (one for each task) concurrently and shares a distilled policy (Teh et al., 2017). The worker policies are then regularized to stay close to the shared policy. Distral was applied to 3D navigation tasks in the DeepMind Lab.

The *Intrinsic Curiosity Module* (ICM), consisting of several neural networks, computes

an intrinsic reward each time step based on the agent’s inability to predict the outcome of taking actions. It was shown to learn to navigate in complex Doom and Super Mario levels only relying on intrinsic rewards (Pathak et al., 2017a).

3.2.4 Open-World Games

Open-world games such as *Minecraft* (Mojang, 2011) or the *Grand Theft Auto* (Rockstar Games, 1997–2013) series are characterized by very non-linear gameplay, with a large game world to explore, either no set goals or many goals with unclear internal ordering, and large freedom of action at any given time. Key challenges for agents are exploring the world and setting goals which are realistic and meaningful. As this is a very complex challenge, most research use these open environments to explore reinforcement learning methods that can reuse and transfer learned knowledge to new tasks. Project Malmo is a platform built on top of the open-world game Minecraft, which can be used to define many diverse and complex problems (Johnson et al., 2016).

The *Hierarchical Deep Reinforcement Learning Network* (H-DRLN) architecture implements a lifelong learning framework, which is shown to be able to transfer knowledge between simple tasks in Minecraft such as navigation, item collection, and placement tasks (Tessler et al., 2017). H-DRLN uses a variation of policy distillation (Rusu et al., 2016a) to retain and encapsulate learned knowledge into a single network.

Neural Turing Machines (NTMs) are fully differentiable neural networks coupled with an external memory resource, which can learn to solve simple algorithmic problems such as copying and sorting (Graves et al., 2014). Two memory-based variations, inspired by NTM, called *Recurrent Memory Q-Network* (RMQN) and *Feedback Recurrent Memory Q-Network* (FRMQN) were able to solve complex navigation tasks that require memory and active perception (Oh et al., 2016).

The Teacher-Student Curriculum Learning (TSCL) framework incorporates a teacher that prioritizes tasks wherein the student’s performance is either increasing (learning) or decreasing (forgetting) (Matiisen et al., 2017). TSCL enabled a policy gradient learning method to solve mazes that were otherwise not possible with a uniform sampling of subtasks.

3.2.5 Real-Time Strategy Games

In this section, we will return to RTS games and survey recent approaches that employ deep neural networks to play these games. Most of the work using neural networks for RTS games are in StarCraft. Here, TorchCraft is a commonly used library built on top of BWAPI that connects the scientific computing framework Torch to StarCraft to enable machine learning research for this game (Synnaeve et al., 2016). Additionally, DeepMind and Blizzard (the developers of StarCraft) have developed a machine learning API to support research in StarCraft II with features such as simplified visuals designed for convolutional networks (Vinyals et al., 2017). This API contains several mini-challenges while it also supports the full game setting. μ RTS (Ontanón, 2013) and ELF (Tian et al., 2017) are two minimalistic RTS game engines that implement some of the features that are present in RTS games.

The previous sections described methods that learn to play games *end-to-end*, i.e. a neural network is trained to map states directly to actions. Real-Time Strategy (RTS) games, however, offer much more complex environments, in which players have to control multiple agents simultaneously in real-time on a partially observable map. Additionally, RTS games have no in-game scoring and thus the reward is determined by who wins the game. For these reasons, learning to play RTS games end-to-end have been deemed infeasible in the foreseeable future by some researchers and instead, sub-problems and smaller RTS variants were used at first.

For the simplistic RTS platform μ RTS, a CNN was trained as a state evaluator using supervised learning on a generated data set and used in combination with Monte Carlo Tree Search (Stanescu et al., 2016; Barriga et al., 2017). This approach performed significantly better than previous evaluation methods.

Deep learning methods for StarCraft have mainly focused on micromanagement (unit control) or build-order planning and has ignored other aspects of the game. The problem of delayed rewards in StarCraft can be circumvented in combat scenarios; here rewards can be shaped as the difference between damage inflicted and damage incurred (Usunier et al., 2016; Foerster et al., 2017; Peng et al., 2017; Foerster et al., 2018). States and actions are often described locally relative to units, which is extracted from the game

engine. If agents are trained individually it is difficult to know which agents contributed to the global reward (Chang et al., 2003), a problem known as the *multi-agent credit assignment problem*. One approach is to train a generic network that controls each unit separately and search in policy space using Zero-Order optimization based on the reward accrued in each episode (Usunier et al., 2016). This strategy was able to learn successful policies for armies of up to 15 units.

Independent Q-learning (IQL) simplifies the multi-agent reinforcement learning problem by controlling units individually while treating other agents as if they were part of the environment (Tan, 1993). This enables Q-learning to scale well to a large number of agents. However, when combining IQL with recent techniques such as experience replay, agents tend to optimize their policies based on experiences with obsolete policies. This problem is overcome by applying *fingerprints* to experiences and by applying an importance-weighted loss function that naturally decays obsolete data, which has shown improvements for some small combat scenarios (Foerster et al., 2017).

The *Multiagent Bidirectionally-Coordinated Network* (BiCNet) implements a vectorized actor-critic framework based on a bi-directional RNN, with one dimension for every agent, and outputs a sequence of actions (Peng et al., 2017). This network architecture is unique to the other approaches as it can handle an arbitrary number of units of different types.

Counterfactual multi-agent (COMA) policy gradients is an actor-critic method with a centralized critic and decentralized actors that address the multi-agent credit assignment problem with a counterfactual baseline computed by the critic network (Foerster et al., 2018). COMA achieves state-of-the-art results, for decentralized methods, in small combat scenarios with up to ten units on each side.

Chapter 5 presents a deep learning approach to build-order planning in StarCraft using macro-based supervised learning approach to imitate human strategies. In this approach, a trained network is integrated as a high-level module used in an existing bot that is capable of playing the full game with an otherwise hand-crafted behavior. Another macro-based approach, here using reinforcement learning instead of supervised learning, called Convolutional Neural Network Fitted Q-Learning (CNNFQ), was trained with Double DQN for build-order planning in StarCraft II and was able to win against medium-level scripted bots on small maps (Tang et al., 2018). Another macro-based reinforcement

learning method that uses Proximal Policy Optimization (PPO) for build order planning and high-level attack planning was able to outperform the built-in bot in StarCraft II at level 10 (Sun et al., 2018). This is particularly impressive as the level 10 bot cheats by having full vision of the map and faster resource harvesting. The results were obtained using 1920 parallel actors on 3840 CPUs across 80 machines and only for one matchup on one map. This system won a few games against Platinum-level human players but lost all games against Diamond-level players. The authors report that the learned policy “lacks strategy diversity in order to consistently beat human players” (Sun et al., 2018).

The first neural network that successfully could play StarCraft II end-to-end (from raw observations to raw actions) was AlphaStar (Vinyals et al., 2019). Here, an advanced LSTM-based neural network model is trained on a single map and only for the Protoss versus Protoss matchup. First, imitation learning was applied to learn from a large set of human demonstrations. Then, several agents, with their own reward function and network parameters, participated in a long tournament structure (called AlphaStar League) in a reinforcement learning phase to learn a diverse set of robust strategies. Each agent played up to 200 years of real-time StarCraft in the training phase, each using 16 TPUs (Vinyals, 2019). The successes of AlphaStar are discussed further in Chapter 9.

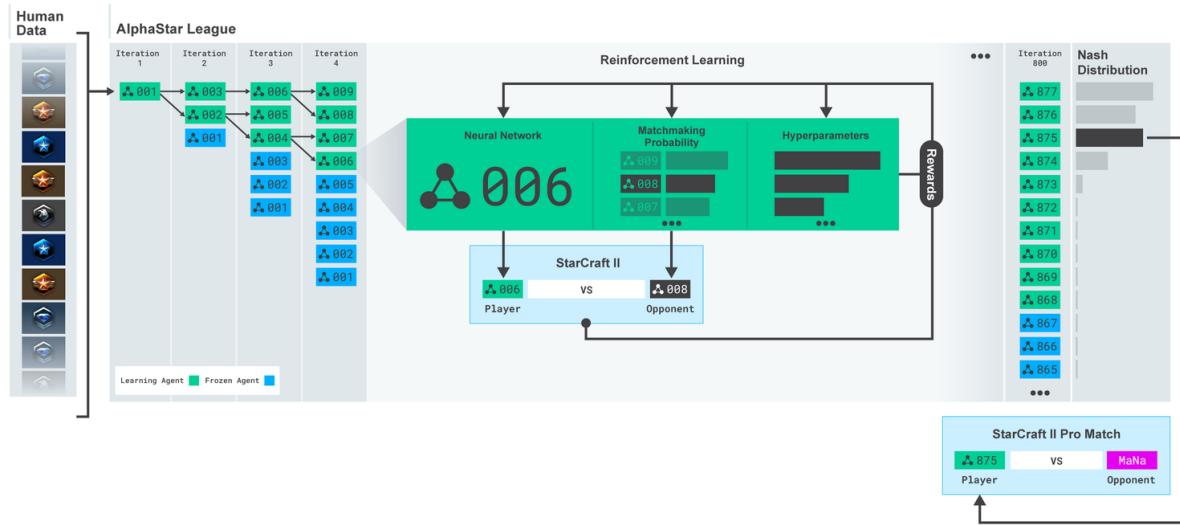


Figure 3.2.4: The AlphaStar League. First, imitation learning is applied on human demonstrations. Then, several iterations of reinforcement learning is applied where several new agents are created as copies, assigned different reward functions and then matched against each other. This figure is from (Vinyals et al., 2019).

Defense of the Ancients (DotA) is originally a mod to WarCraft III (Blizzard Entertainment,

2002) while it is now more common to play Dota 2 (Valve Corporation, 2013). DotA and Dota 2 are multiplayer online battle arena (MOBA) games, a genre similar to RTS games, but with two teams of players, and each player controls an individual hero rather than an entire army. OpenAI Five (OpenAI, 2017) is a system that controls a team of five heroes in Dota 2, with one neural network policy for each player. While Dota 2 is much more complex than e.g. Atari games, the developers found that the same techniques (here the Proximal Policy Optimization algorithm) with enormous amounts of computation was enough to reach human-level performance. OpenAI Five played around 180 years of real-time Dota 2 per day in the training phase, using 256 GPUs and 128,000 CPU cores (OpenAI, 2018a). The successes of OpenAI Five will also be discussed further in Chapter 9.

3.2.6 Team Sports Games

Popular sports games are typically based on team-based sports such as soccer, basketball, and football. These games aim to be as realistic as possible with life-like animations and 3D graphics. Several soccer-like environments have been used extensively as research platforms, both with physical robots and 2D/3D simulations, in the annual Robot World Cup Soccer Games (RoboCup) (Asada et al., 2000). *Keepaway Soccer* is a simplistic soccer-like environment where one team of agents try to maintain control of the ball while another team tries to gain control of it (Stone and Sutton, 2001). A similar environment for multi-agent learning is *RoboCup 2D Half-Field-Offense (HFO)* where teams of 2-3 players either take the role as offense or defense on one half of a soccer field (Hausknecht et al., 2016).

Deep Deterministic Policy Gradients (DDPG) was applied to RoboCup 2D Half-Field-Offense (HFO) (Hausknecht and Stone, 2015). The actor network used two output streams, one for the selection of discrete action types (dash, turn, tackle, and kick) and one for each action type's 1-2 continuously-valued parameters (power and direction). The *Inverting Gradients* bounding approach downscales the gradients as the output approaches its boundaries and inverts the gradients if the parameter exceeds them. This approach outperformed both SARSA and the best agent in the 2012 RoboCup. DDPG was also applied to HFO by mixing on-policy updates with 1-step Q-Learning updates (Hausknecht

and Stone, 2016) and outperformed a hand-coded agent with expert knowledge with one player on each team.

3.2.7 Physics Games

As video games are usually a reflection or simplification of the real world, it can be fruitful to learn an intuition about the physical laws in an environment. A predictive neural network using an object-centered approach (also called fixations) learned to run simulations of a billiards game after being trained on random interactions (Fragkiadaki et al., 2016). This predictive model could then be used for planning actions in the game.

A similar predictive approach was tested in a 3D game-like environment, using the Unreal Engine, where ResNet-34 (He et al., 2016b) (a deep residual network used for image classification) was extended and trained to predict the visual outcome of blocks that were stacked such that they would usually fall (Lerer et al., 2016).

3.2.8 Text Adventure Games

A classic text adventure game is a form of interactive fiction where players are given descriptions and instructions in text, rather than graphics, and interact with the storyline through text-based commands (Sweetser, 2008). These commands are usually used to query the system about the state, interact with characters in the story, collect and use items, or navigate the space in the fictional world.

These games typically implement one of three text-based interfaces: parser-based, choice-Based, and hyperlink-based (He et al., 2016a). Choice-based and hyperlink-based interfaces provide the possible actions to the player at a given state as a list, out of context, or as links in the state description. Parser-Based interfaces are, on the other hand, open to any input and the player has to learn what words the game understands. This is interesting for computers as it is much more akin to natural language, where you have to know what actions should exist based on your understanding of language and the given state.

Unlike some other game genres, like arcade games, text adventure games have not had a standard benchmark of games that everyone can compare against. This makes a lot

of results hard to directly compare. A lot of research has focused on games that run on Infocom’s Z-Machine game engine, an engine that can play a lot of the early, classic games. Recently, Microsoft has introduced the environment TextWorld to help create a standardized text adventure environment (Côté et al., 2018).

Text adventure games, in which both states and actions are presented as text only, are a special video game genre. A network architecture called LSTM-DQN (Narasimhan et al., 2015) was designed specifically to play these games and is implemented using LSTM networks that convert text from the world state into a vector representation, which estimates Q-values for all possible state-action pairs. LSTM-DQN was able to complete between 96% and 100% of the quests on average in two different text adventure games.

To be able to improve on these results, researchers have moved toward learning language models and word embeddings to augment the neural network. An approach that combines reinforcement learning with explicit language understanding is Deep Reinforcement Relevance Net (DRRN) (He et al., 2016a). This approach has two networks that learn word embeddings. One embeds the state description, the other embeds the action description. Relevance between the two embedding vectors is calculated with an interaction function such as the inner product of the vectors or a bilinear operation. The Relevance is then used as the Q-Value and the whole process is trained end-to-end with Deep Q-Learning. This approach allows the network to generalize to phrases not seen during training which is an improvement for very large text games. The approach was tested on the text games Saving John and Machine of Death, both choice-based games.

Taking language modeling further, Fulda et. al. explicitly modeled language affordances to assist in action selection (Fulda et al., 2017). A word embedding is first learned from a Wikipedia Corpus via unsupervised learning (Mikolov et al., 2013) and this embedding is then used to calculate analogies such as *song is to sing* as *bike is to x*, where *x* can then be calculated in the embedding space (Mikolov et al., 2013). The authors build a dictionary of verbs, noun pairs, and another one of object manipulation pairs. Using the learned affordances, the model can suggest a small set of actions for a state description. Policies were learned with Q-Learning and tested on 50 Z-Machine games.

The Golovin Agent focuses exclusively on language models (Kostka et al., 2017) that are pre-trained from a corpus of books in the fantasy genre. Using word embeddings,

the agent can replace synonyms with known words. Golovin is built of five command generators: General, Movement, Battle, Gather, and Inventory. These are generated by analyzing the state description, using the language models to calculate and sample from a number of features for each command. Golovin uses no reinforcement learning and scores comparable to the affordance method.

Most recently, Zahavy et al. (2018) proposed another DQN method. This method uses a type of attention mechanism called Action Elimination Network (AEN). In parser-based games, the actions space is very large. The AEN learns, while playing, to predict which actions that will have no effect for a given state description. The AEN is then used to eliminate most of the available actions for a given state and after which the remaining actions are evaluated with the Q-network. The whole process is trained end-to-end and achieves similar performance to DQN with a manually constrained actions space. Despite the progress made for text adventure games, current techniques are still far from matching human performance.

Outside of text adventure games, natural language processing has been used for other text-based games as well. To facilitate communication, a deep distributed recurrent Q-network (DDRQN) architecture was used to train several agents to learn a communication protocol to solve the multi-agent *Hats* and *Switch* riddles (Foerster et al., 2016). One of the novel modifications in DDRQN is that agents use shared network weights that are conditioned on their unique ID, which enables faster learning while retaining diversity between agents.

3.2.9 OpenAI Gym & Universe

OpenAI Gym is a large platform for comparing reinforcement learning algorithms with a single interface to a suite of different environments including ALE, GVG-AI, MuJoCo, Malmo, ViZDoom and more (Brockman et al., 2016a). OpenAI Universe is an extension to OpenAI Gym that currently interfaces with more than a thousand Flash games and aims to add many modern video games in the future¹¹.

¹¹<https://universe.openai.com/>

Chapter 4

Continual Evolutionary Planning

This chapter looks at intra-game adaptivity in StarCraft and presents a new evolutionary-based approach called *Continual Online Evolutionary Planning* (COEP) and apply it to build-order planning in StarCraft. Evolutionary algorithms have previously been applied to the problem of optimizing build orders offline (Kuchem et al., 2013; Köstler and Gmeiner, 2013; Blackford and Lamont, 2014) to find static opening build orders. In contrast, COEP runs continually during the game (i.e. online) to search for build-order plans that adapts to the opponent’s strategy (see Figure 4.0.1). To our knowledge, no system, prior to the publication of this work (Justesen and Risi, 2017a), could perform in-game adaptive build-order planning in StarCraft. Ontanón et al. (2013) has previously highlighted this shortcoming as well: “No bot is capable of observing the opponent and autonomously synthesize a good plan from scratch to counter the opponent strategy”. COEP is unique as it runs continually to optimize the future build-order plan while the game is being played, taking available information about the opponent’s strategy into account. The experiments presented in this chapter builds on the modular UAlbertaBot, by replacing the module that is responsible for build-order planning (i.e. which unit/building/upgrade to produce and in which order) with our evolutionary planner. Tasks such as controlling units in combat are performed by other modules in UAlbertaBot and are in themselves an activate research area (Churchill and Buro, 2013; Justesen et al., 2014; Wang et al., 2016a). A series of experiments demonstrate that COEP can outperform the game’s built-in bot as well as some scripted opening build-orders performed by UAlbertaBot.

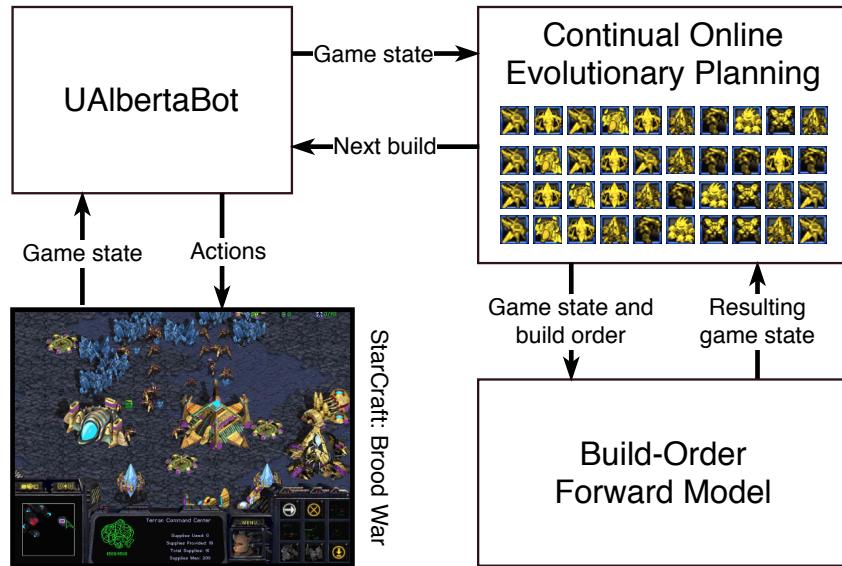


Figure 4.0.1: *Continual Online Evolutionary Planning* (COEP) continually evolves future build orders while UAlbertaBot executes the best one found so far.

4.1 Approach

We present a new adaptive planning algorithm called *Continual Online Evolutionary Planning* (COEP) which extends traditional evolutionary planning. We apply this to the StarCraft build-order tasks such that each individual in the population represents a candidate build order with a fixed length. To evaluate the fitness of individuals, a *build-order forward model* simulates the resulting game state after executing a build order (Section 4.1.1). The fitness function (Section 4.1.2) takes into account the unit composition of the resulting game state and current available information about the opponent’s units.

The most prominent difference to traditional evolutionary planning is that COEP runs *continually* and in *parallel* with the rest of the bot, forming a communication line between the two parts. When the bot is ready to produce a new build, it requests the planner which take the first build from the build order of the currently most fit candidate in the population. Simultaneously, the game state (*state* in Algorithm 6) is updated such that build orders are generated and evaluated based on new information of the game state. Furthermore, if builds have gone into production since the last update, individuals are updated such that the first build in each build-order is discarded. This idea is similar to EA-shift that was independently developed and published later by Gaina et al. (2017b).

Algorithm 6 Continual Online Evolutionary Planning (COEP)

```

1: COEP continually creates a new population and runs evolution for number of
   generations. State is updated by the bot as soon as new information is obtained
   and the best found build order can be retrieved from the champion (the most fit
   individual).
2: procedure COEP(GameState s)                                 $\triangleright s$  is the initial game state.
3:   champion = NULL                                          $\triangleright$  Accessible by bot
4:   state = s                                               $\triangleright$  Accessible by bot
5:   while game is not over do
6:     pop =  $\emptyset$                                           $\triangleright$  Create new population
7:     if champion is not NULL then pop.Push(champion)
8:     for i = Size(pop) to POP_SIZE do
9:       g = Genome(s)
10:      g.buildOrder = legal build order from s
11:      g.fitness = FITNESS(s, genome.buildOrder)
12:      pop.Push(genome)
13:     for i = 1 to GENERATIONS do
14:       Reduce pop based on elitism rules
15:       Reproduce offspring using crossover
16:       Mutate some offspring
17:       Evaluate fitness of offspring
18:       Add offspring to pop
19:     champion = most fit individual

```

COEP runs a fixed number of generations after which it restarts using a new population. These restarts are intended to prevent the evolution getting stuck in local optima, which would prevent the bot from adapting to the continuously changing game state. When populations are restarted, the most fit individual of the previous population is transferred to the new population but excluded from the reproduction phase. In this way, COEP can attempt to evolve new build orders while keeping the best from the last population until a superior solution is found. This idea is similar to case-injected genetic algorithms, in which solutions to previously solved problems are periodically injected into the population (Louis and McDonnell, 2004; Louis and Miles, 2005).

Crossover is applied directly to build orders from two randomly sampled parents (Figure 4.1.1). If some builds within a build order of an offspring are illegal, the forward model simply ignores them.

Four different mutation operators are tested to ensure that existing build orders can be reorganized effectively (Figure 4.1.2): **Clone:** Two indices *a* and *b* are randomly selected. Build at position *a* becomes the same as the build at position *b*. **Swap:** Two random

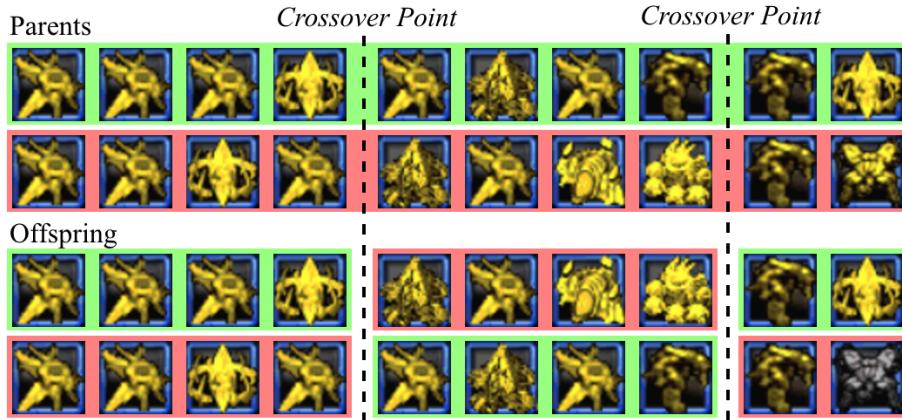


Figure 4.1.1: Two-point crossover for two parent build orders and the resulting offspring. Notice that the build in the bottom right corner remains in the genotype but becomes inactive because its requirements are no longer met.

builds swap position. **Add:** One random build is randomly inserted. For each unmet requirement, each required build is recursively added in front, such that the first build is moved backwards and eventually out of the build order. **Remove:** One random build is moved to the end of the build order and all builds after the moved build's initial position slide one step forward.

4.1.1 Forward Model

A forward model can predict the outcome of taking some actions in a given game state, which we use to evaluate the fitness of build orders. The forward model used here does not need to implement all the game rules since we are only concerned with build-order planning and we are not concerned with how units move and attack. Such a *build-order forward model* was implemented for the Protoss and Terran race and the source code is available online¹. The forward model (Appendix A1) iterates the given build order and tries to add each build to the given game state in order; if the requirements of a build are not satisfied it is simply ignored.

A few constants are used by the forward model: The average minerals and gas collection speed per frame by workers were estimated to 0.05 and 0.07 respectively. Similar values of 0.045 and 0.07 were used by (Churchill and Buro, 2011). The amount of minerals gathered decreases if more than ten workers mine at each base, such that workers 11–20

¹<https://github.com/njustesen/coep-starcraft/>

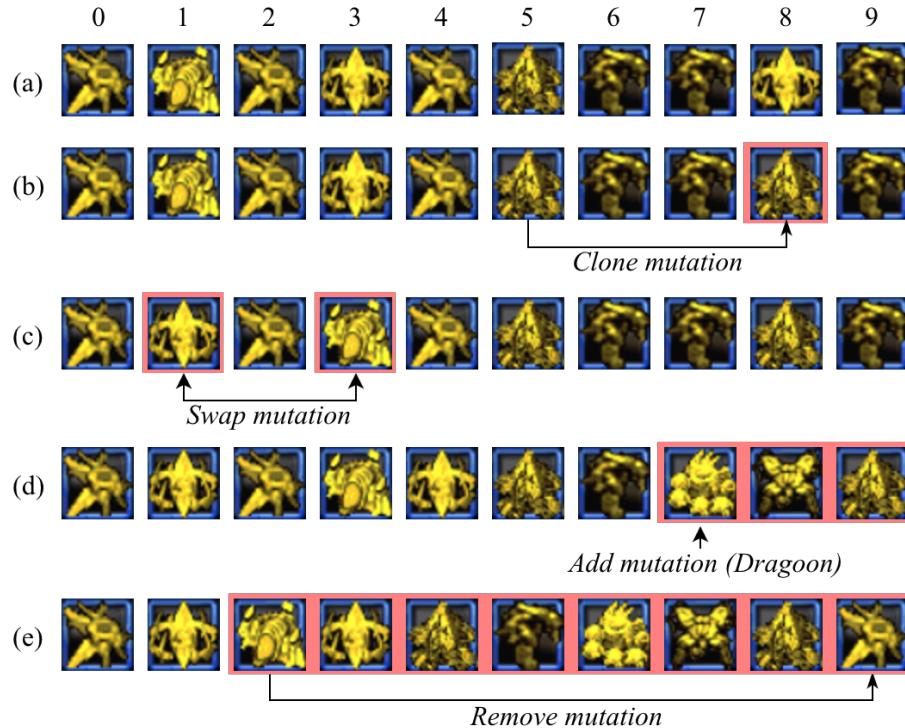


Figure 4.1.2: A build order with ten builds, which is manipulated by the four mutation operators. Builds are highlighted (red) if they are changed during an operation. (a) Shows the initial build order, (b) the result of a clone mutation from index 5 to 8, (c) a swap mutation on index 1 and 3, which swaps the two builds, (d) an add mutation on index 7, which adds a dragoon and recursively adds its requirements first and (e) a remove mutation at index 2 that moves the build to the end of the build order.

only gather half as many minerals, 21–30 a third, etc.

COEP receives information available about the current game state from the bot, which includes the number of all known friendly and enemy units, buildings, technologies and upgrades as well as the current frame number. The technologies and upgrades researched by the opponent are not known and are thus excluded. Also note that the game state only includes the partial knowledge about the enemy units that the player has obtained. Additionally, the game state includes a list of friendly builds that are in production as well as the frame number in which they are completed.

4.1.2 Fitness

Building on the forward model, which predicts the resulting game state after applying a build order, the fitness of a build order is determined by how desirable this future game

Algorithm 7 Discounted Accumulated Fitness

```

1: Determines the fitness of a genome by calculating the discounted accumulated fitness
   of several steps of stepSize frames for a total of horizon frames. A discount factor in
    $\gamma \leq 1$  is used to prioritize short-term army production.
2: procedure FITNESS(Genome g, GameState s, int horizon, int stepSize)
3:   state = CLONE(s)
4:   step = 0
5:   while state.frame < s.frame + horizon do
6:     next = Min(s.frame + horizon, state.frame + stepSize)
7:     build = next unbuilt build in g.buildOrder
8:     state = PREDICT(state, [build], next)
9:     g.fitness += HEURISTIC(state)  $\times^{step}$ 
10:    step += 1

```

state is for the player. A challenge with this naive approach is that, at least in real-time games, the longer one tries to predict into the future the more uncertain the outcome becomes. For example, a build order with a very strong economy in the beginning and a large unit production in the end would give a high fitness, even though the player has no army and is defenseless during most of the evaluated period.

Therefore, the fitness function introduced here performs an evaluation several times during the time span of the build order in addition to incorporating a discount factor (similarly to discounted returns in MDPs) that values short-term rewards higher than long-term rewards. In other words, instead of applying the forward model one time on the entire build order, it is applied in several steps on subsets. Within each step, the heuristic of the intermediate game state is accumulated into the final fitness of the build order (Algorithm 7).

The fitness function is based on a heuristic that can evaluate how desirable game states are. The game state in StarCraft is seen from a player perspective and is thus only partially visible (see Section 4.1.1). Designing an optimal heuristic for StarCraft is extremely challenging and highly dependent on the micro-management modules of the bot. The simple heuristic used here, evaluates both players' unit composition based on basic knowledge about combat units in StarCraft. Some units are superior against particular units while inferior against others (e.g. the powerful Zerg ultralisk, a ground melee unit, is defenseless against a Protoss scout, a flying ranged unit). To express how strong each unit type is against any other unit type a *unit matchup table* is introduced (Table 4.1.1). For example, the Terran firebat (short-ranged unit) is valued 0.4 against a Protoss dragoon

	Zealot	Dragoon	Dark Templar	Scout
Marine	0.7	0.5	0.6	1.6
Firebat	1.3	0.1	0.8	0.7
Vulture	1.6	0.7	1.6	0.7
Goliath	0.9	0.7	0.9	1.5
Siege Tank	0.8	1.4	0.5	0.9

Table 4.1.1: Unit matchup table that values how strong units are against each other which is a critical part of the heuristic applied. Values are in the range [0, 2].

	Zealot	Dragoon	Scout
Ground Armor	1.02	1.02	-
Plasma Shields	1.02	1.02	1.02
Air Armor	-	-	1.02
Singularity Charge	-	1.25	-

Table 4.1.2: Upgrade and tech multipliers, which give units additional value in the heuristic.

(long-ranged unit) to express its weakness in this matchup. The value of a dragoon against a firebat is the same, but inverted: $2 - 0.4 = 1.6$. Attributes such as damage types, unit size, and whether they are invisible or can detect invisible units are also considered.

Upgrades and technologies can improve the strength of some units, which is reflected in Table 4.1.2. The values in this table are multiplied with the matchup value from Table 4.1.1 to determine the final values. For example, a Protoss dragoon has a final value against a Terran firebat of $1.6 \times 1.25 = 2$, if the *Singularity Charge* upgrade has been researched. Note that upgrade bonuses are not added to enemy units. We define a function matchup that performs these calculations given a friendly unit type x and enemy unit type y (e.g. $\text{matchup}(\text{dragoon}, \text{firebat}) = 1.6$; $\text{matchup}(\text{firebat}, \text{dragoon}) = 0.4$). The value for player p of a unit matchup of friendly units of type x and enemy units of type y is:

$$\text{value}(p, x, y) = \text{matchup}(x, y) \times n(x) \times n(y) \times \left(1 - \frac{n(x)}{N(p)}\right),$$

where $\text{matchup}(p, x, y)$ refers to the unit matchup table, $n(y)$ and $n(x)$ is the number of units of type y and x , and $N(p)$ is the number of all units controlled by player p . The idea is that a player should strive to optimize all four components of this function to achieve a good unit combination. This heuristic prefers a balanced unit composition, in which units individually have high unit matchup values against the enemy units. The first three

components increase if the player has many units that counter the enemy units while the last component ($1 - \frac{n(x)}{N(p)}$) increases if the player has a balanced mix of unit types. It should be noted that $n(x)$ in the last component is further divided by 2 if x is a worker. The heuristic used to evaluate a state s with players p1 and p2 is the sum of matchup values for all permutations of the unit types u_{p1} and u_{p2} controlled by the two players:

$$\text{heuristic}(s) = \sum_x \sum_y^{\substack{u_{p1} \\ u_{p2}}} \text{value}(\text{p1}, x, y) - \text{value}(\text{p2}, y, x).$$

After prior experimentation with this heuristic we found it necessary to penalize expansions while having few workers as well as not expanding while having many workers. The expansion penalty in state s is equal to $\text{numOfBases} \times 14 - \text{MineralWorkers}(s)$. Likewise, a penalty for having too many supply buildings was found necessary. A complete implementation of the heuristic can be found in the source code².

4.1.3 Integration with UAlbertaBot

Because of UAlbertaBot’s modular design it is simple to replace the existing production manager module with the presented COEP approach; all other modules in the bot are kept unchanged. UAlbertaBot is an open source StarCraft bot developed by David Churchill³ that won the annual AIIDE StarCraft AI Competition in 2013. The bot consists of numerous hierarchical modules, such as an information manager, building manager and production manager. The production manager is responsible for managing the build order queue, i.e. the order in which the bot produces new builds. This architecture enables us to replace the production manager with our neural network, such that whenever the bot is deciding what to produce next, the network predicts what a human player would produce. The modular design of UAlbertaBot is described in more detail in (Ontanón et al., 2013). The new production manager requests the COEP planner for a build whenever a new build is being produced in the game or if 600 frames have passed.

²<https://github.com/njustesen/coep-starcraft>

³<https://github.com/davechurchill/ualbertabot>

4.2 Experiments

The experiments are split into two parts, where the first part consists of experiments that tests the ability of Online Evolutionary Planning (OEP), without the continual extension, to evolve strong build orders for static game states in StarCraft. Without the continual extension, the algorithm runs normally for a fixed number of generations using the same game state and then terminates. In the second part, COEP is applied to UAlbertaBot and is then tested in a total of 900 StarCraft games against the game’s built-in bot as well as UAlbertaBot with four scripted opening build orders. The games were played on the two-player map Astral Balance and all the game replays are made available⁴.

The configuration of COEP can be seen in Appendix A2. It takes on average 156 ± 18 ms. for the algorithm to run one generation on a regular laptop (2,6 GHz Intel Core i5).

Terran units								Average unit combinations of evolved build orders.						
10	10	0	0	0	0	0	0	7.5 \pm 2.8	8.2 \pm 4.2	1.2 \pm 2.0	2.3 \pm 2.3	0.0 \pm 0.0	0.1 \pm 0.4	
10	0	10	0	0	0	0	0	2.2 \pm 2.0	11.8 \pm 5.0	0.5 \pm 0.9	2.3 \pm 2.6	0.4 \pm 0.7	0.3 \pm 0.9	
10	0	0	8	0	0	0	0	1.0 \pm 1.0	7.5 \pm 3.5	0.0 \pm 0.1	1.6 \pm 2.1	1.2 \pm 1.5	1.4 \pm 1.6	
10	0	0	4	0	4	0	0	2.9 \pm 1.9	3.5 \pm 2.7	0.3 \pm 0.1	2.6 \pm 2.9	0.6 \pm 1.1	1.4 \pm 1.6	
10	0	0	0	4	4	0	0	6.1 \pm 3.0	2.8 \pm 2.8	0.6 \pm 1.8	3.2 \pm 2.7	0.1 \pm 0.4	0.6 \pm 1.0	
10	0	0	0	0	0	4	2	0.8 \pm 1.0	9.7 \pm 3.7	0.0 \pm 0.3	1.6 \pm 1.6	0.4 \pm 0.7	0.6 \pm 1.0	

Table 4.2.1: Unit combinations of evolved build orders found by Online Evolutionary Planning after 100 generations. Results are averaged over 50 evolutionary runs. Some units are excluded from the results for brevity. Each row represents one scenario containing the Terran units on the left as well as a Protoss nexus, pylon and four probes. The Protoss units on the right are the average unit combination of the evolved build orders. For each unit type, the average count as well as the standard deviation is shown. The main result is that by following the implemented heuristics, Online Evolutionary Planning is able to evolve build orders that can effectively counter the opponent’s strategy.

4.2.1 Results: Online Evolutionary Planning

Online Evolutionary Planning (OEP) was tested, without the continual extension, on its ability to evolve build orders to counter different enemy unit combinations. More specifically, OEP had to find effective build orders for a Protoss player against a Terran player. Six different scenarios were created (Table 4.2.1) all with one nexus (main base),

⁴<http://bit.ly/2omfT5G>

four probes (workers) and one pylon (supply building) for the Protoss player, each with a different set of units for the Terran player. For each of the six scenarios, OEP ran for 100 generations with a horizon of 12 minutes. Table 4.2.1 shows the unit combination of the best evolved build orders averaged over 50 independent evolutionary runs. The results demonstrate that OEP is capable of evolving diverse unit combinations that clearly depend on the combination of enemy units. For example, in the scenario shown in row six, the algorithm avoids zealots and dark templars (both ground melee units) against wraiths and battlecruisers (both flying units). In scenario 2 the algorithm prefers dragoons (long-ranged units) against firebats (short-ranged units). The reason why zealots and dragoons are so dominant in the evolved build orders is that they are cheaper units that can be produced early in the game. Referencing the values in the unit matchup table (Table 4.1.1) shows that the evolved build orders produce matching unit combinations.

To determine the importance of the introduced mutation operators, we ran 50 independent evolutionary runs for 100 generations with only one of the four mutation operators enabled, compared to all of them enabled (Figure 4.2.1). Interestingly, the clone and swap operators were the most efficient, but significantly less effective than as all four operators together ($p < .01$; two-tailed Mann-Whitney U Test). The algorithm was also tested with uniform crossover, single-point crossover and two-point crossover, but no significant change in performance was detected.

4.2.2 Results: Continual Online Evolutionary Planning

In the previous experiments, it was shown that OEP is capable of evolving build orders to counter the opponent’s strategy. In the following experiments, COEP is applied to UAlbertaBot to perform in-game build-order planning, playing as the Protoss race. COEP uses the same configuration as in the previous experiment as well as 100 generations in each loop. The bot played a total of 300 games against the built-in bots in StarCraft, 100 against each of the three races. Our bot won 275 games (91.7%) with 5 games (1.7%) ending in a draw. A summary of the results can be seen in Table 4.2.2. Each iteration of COEP, which consists of initializing a new population and running 100 generations, takes on average 9.96 ± 0.8 seconds. COEP was also tested with a random fitness function, which performs significantly worse, corroborating the heuristic chosen in this paper. In most

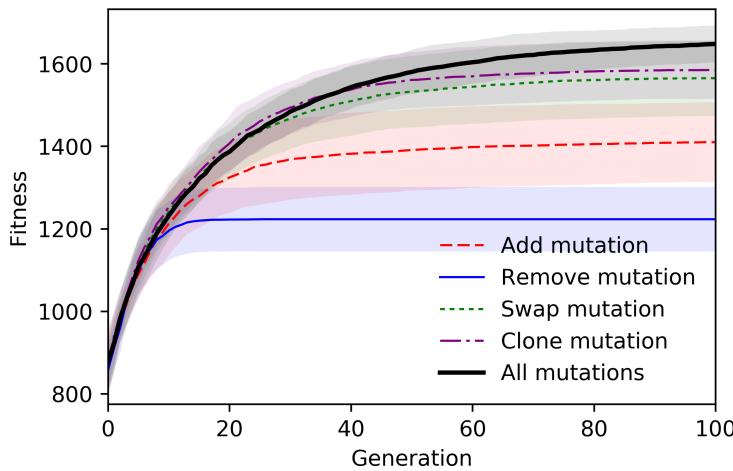


Figure 4.2.1: The average fitness over generations for Online Evolutionary Planning using a different mutation operator. Opaque coloring shows standard deviations.

games, the bot demonstrated the ability to adapt to the opponent’s strategy efficiently enough to win. An example of such adaption is shown in Figure 4.2.2. The upper plot displays the number of zealots, dragoons, marines and firebats in the game. It is clear that our system (controlling the Protoss units) prefers zealots against the enemy marines but switches to a unit combination dominated by dragoons when firebats are spotted. This adaption rule can be seen in Tables 4.1.1 and 4.2.1. The bottom plot shows the highest fitness in the population over time as well as the times the COEP’s game state was updated.

	Protoss	Terran	Zerg
COEP	83/4/13	96/0/4	96/1/3
COEP Random Fitness	1/0/99	0/0/100	4/0/96

Table 4.2.2: Number of wins, draws and losses by Continual Online Evolutionary Planning (COEP) against each of the three races controlled by the built-in bot in StarCraft. The bottom row shows results of COEP with a random fitness function.

The final experiment compared our adaptive approach with four scripted protoss strategies played by UAlbertaBot. This test is more challenging as these scripts employ established opening strategies, optimized to destroy the enemy early in the game. Zealot, dragoon, and dark templar rushes are aggressive strategies, in which the player tries to obtain an army of only one type of unit as fast as possible to surprise the opponent. Still COEP

was competitive against these challenging openings, winning 52% of all games (draws counted as half a win). The most challenging for COEP were the very fast zealot rush, which does not allow much time to adapt. These results are summarized in Table 4.2.3. Being competitive to established opening rushes is quite impressive, given that COEP finds build-order plans online with no scripted starting point.

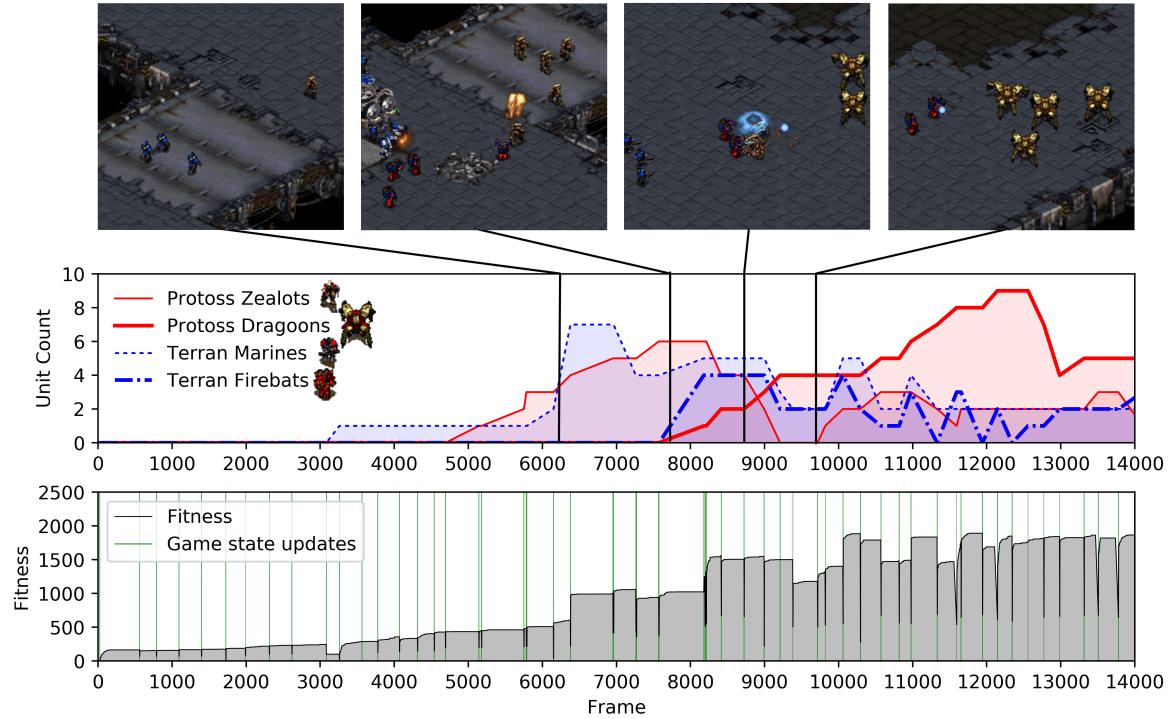


Figure 4.2.2: A visualization of Continual Online Evolutionary Planning’s (COEP) ability to perform successful intra-game adaptation by continually adjusting the Protoss build order in-game against the built-in Terran bot. The upper plot shows the number of zealots, dragoons, marines and firebats over time and the lower plot shows the highest fitness in the population. Green vertical lines indicate when the game state was updated. The four screenshots in the top show critical situations in the game. Early in the game the bot observes a group of Terran marines and continues to produce zealots to counter them. Shortly after, these zealots fight against a large group of Terran firebats and many zealots die. COEP quickly adapts its strategy to switch production to dragoons as they are superior to firebats. A video of this game can be found here: <https://youtu.be/SCZbDpIaqmI>.

	Zealot rush	Dragoon Rush	Dark Templar Rush
COEP	19/0/81	60/0/40	73/7/20

Table 4.2.3: Number of wins, draws and losses by Continual Online Evolutionary Planning (COEP) against three scripted Protoss opening strategies performed by UAlbertaBot.

4.3 Discussion

In some cases, COEP can struggle, such as when it has to adapt to the very aggressive zealot rush. Since our heuristic only takes the enemy units, and not production buildings into account, COEP’s ability to adapt is slightly delayed, which is devastating during rushes. In the future we plan to extend COEP to also take buildings into account.

Designing the heuristic has been challenging as it needs to correlate with the playing style of the underlying bot. UAlbertaBot implements a specific behavior, which has its own quirks when it comes to controlling larger groups of units or when it expands to new bases. The strategies preferred by our implementation involve large armies with various unit types which require more advanced micro-management compared to the simpler rush strategies. UAlbertaBot also displayed difficulties using more advanced units such as reavers, high templars and shuttles, which limits the range of possible unit combinations for our approach. UAlbertaBot was probably not designed to have an adaptive build order module which requires a great deal of generality in its implementation. Developing a more advanced and general StarCraft bot, or improving upon an existing bot, as well as fully incorporating COEP are important next steps.

A promising direction would be to improve evolutionary planning such that it learns a heuristic in a training phase. Work in this direction has recently been published by Tong et al. (2019), where a policy and value network is trained while evolutionary planning (here Rolling Horizon Evolution) performs rollouts assisted by the policy network. This approach could perhaps learn the quirks of UAlbertaBot, while it would require a diverse set of opponent players to achieve an inter-game adaptive policy.

After the publication of the work presented in this chapter, Churchill et al. (2019) published a continual extension of their earlier work where they used a recursive depth-first search algorithm for build-order planning (Churchill and Buro, 2011). In their extension, an *Army Integral Evaluation* method is introduced that attempts to minimize the integral between the future army value curve and the curve for the maximum possible army value for any build-order. This approach is similar to our discounted accumulated fitness function and it would be interesting to compare the effect of these two methods. Their army value heuristic is only based on resources spent on army units and does not consider

enemy units, making it unable to adapt to the opponent’s strategy.

Instead of having a complete reactive approach it might be fruitful to imagine what the opponent is doing along with our own planning. Introducing co-evolution by also evolving build-orders for the opponent player, could perhaps provide a more preventive behavior.

4.4 Summary

This chapter presented a variation of Evolutionary Planning called *Continual Online Evolutionary Planning* (COEP) that can perform intra-game adaptive build-order planning in StarCraft. COEP implements a discounted accumulated fitness function that favors short-term army production over long-term rewards. COEP was applied to an existing StarCraft bot called UAlbertaBot, where it replaced the existing macro-management module. The results demonstrate that COEP is capable of intra-game adaptive build-order planning, continually adapting to the changes in the game. While COEP still struggles against some very aggressive rushes, it outperforms the built-in bot in StarCraft: Brood War with a 91.7% win rate and can compete with a number of scripted opening build orders performed by UAlbertaBot.

Chapter 5

Learning Build-order Planning in StarCraft from Replays

In this chapter, we will explore a neural network based approach to the build-order planning task in StarCraft. In the previous chapter, build-order plans were evolved online to adapt to the opponent. Here, we train a neural network using imitation learning to predict human build-order decision from demonstrational data extracted from replay files (i.e. game logs) of highly skilled human players. The trained neural network is combined with the existing StarCraft bot UAlbertaBot, as in the previous chapter, and is responsible for deciding what unit, building, technology, or upgrade to produce next, given the current state of the game. While our approach does not achieve state-of-the-art results on its own, it is a promising first step towards neural network based methods for build-order planning in RTS games, as the same system setup could be used with reinforcement learning as well. The approach and results presented here, are to our knowledge the first published approach using deep learning in a system that plays the full StarCraft game.

By imitation human strategic decisions based on game state descriptions that include knowledge about the opponent units and buildings, we expect the learned neural network to be capable of inter-game adaptation. Additionally, by sampling from the learned action distribution, instead of sampling deterministically, the build order planner will express the wide range of behaviors that is expressed in the data set. While this does not qualify as an intelligent inter-game adaptive behavior, it makes it less exploitable by an adaptive opponent.

5.1 Approach

This section describes the presented approach in four parts. First, the data set and extraction procedure is described. Then, the employed neural network architecture is presented, followed by the training procedure. Finally, the integration with UAlbertaBot are detailed.

5.1.1 Dataset

This section gives an overview of the dataset used for training and how it has been created from replay files. A replay file for StarCraft contains every action performed throughout the game by each player, and the StarCraft engine can recreate the game by executing these actions in the correct order. To train a neural network to predict the build-order decisions made by players, state-action pairs are extracted from replay files, where a state describes the current game situation and an action corresponds to the next build produced by the player. Additionally, states are encoded as a vector of normalized values to be processed by our neural network.

Replay files are in a binary format and require preprocessing before knowledge can be extracted. The dataset used in this paper is extracted from an existing dataset. Synnaeve and Bessiere (2012) collected a repository of 7,649 replays by scraping the three StarCraft community websites GosuGamers, ICCup and TeamLiquid, which are mainly for highly skilled players including professionals. A large amount of information was extracted from the repository and stored in an SQL database by Robertson and Watson (2014). This database contained state changes, including unit attributes, for every 24 frames in the games. Our dataset is extracted from this database, and an overview of the preprocessing steps is shown in Figure 5.1.1.

From this database, we extract all events describing material changes throughout every Protoss versus Terran game, including when (1) builds are produced by the player, (2) units and buildings are destroyed and (3) enemy units and buildings are observed. These events take the perspective of one player and thus maintain the concept of partially observable states in StarCraft. The set of events thus represent a more abstract version

of the game only containing information about material changes and actions that relate to build-order decisions. The events are then used to simulate abstract StarCraft games via the build-order forward model presented in Section 4.1.1. Whenever the player takes an action in these abstract games, i.e. produces something, the action and state pair is added to our dataset. This simulation step using the forward model can be skipped if the original replays are simulated in the actual game engine. The recorded state describes the player’s own material in the game: the number of each unit, building, technology, and upgrade present and under construction, as well as enemy material observed by the player.

The entire state vector consists of a few sub-vectors described here in order, in which the numbers represent the indexes in the vector:

1. **0-31:** The number of units/buildings of each type present in the game controlled by the player.
2. **32-38:** The number of each technology type researched in the game by the player.
3. **39-57:** The number of each upgrade type researched in the game by the player. For simplicity, upgrades are treated as a one-time build and our state description thus ignores level 2 and 3 upgrades.
4. **58-115:** The number of each build in production by the player.
5. **116-173:** The progress of each build in production by the player. If a build type is not in production it has a value of 0. If several builds of the same type are under construction, the value represents the progress of the build that will be completed first.
6. **174-206:** The number of enemy units/buildings of each type observed.
7. **207-209:** The number of supplies used by the player and the maximum number of supplies available. Another value is added which is the supply left, i.e. the difference between supply used and maximum supplies available.

All values are normalized into the interval [0, 1]. The preprocessed dataset contains 2,005 state-action files with a total of 789,571 state-action pairs. Six replays were excluded because the Protoss player used the rare *mind control* spell on a Terran SCV that allows

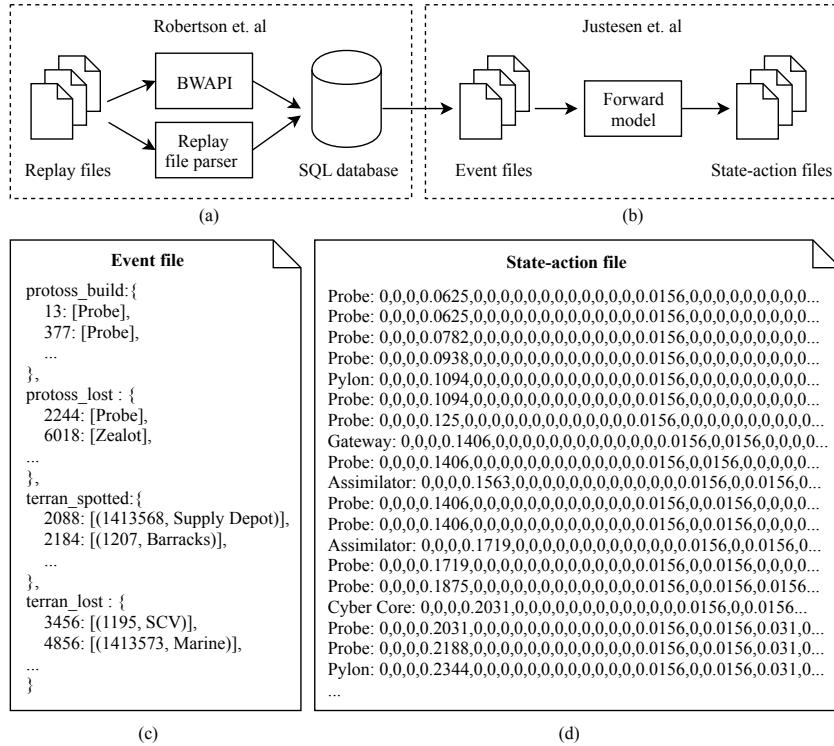


Figure 5.1.1: An overview of the data preprocessing that converts StarCraft replays into vectorized state-action pairs. (a) shows the process of extracting data from replay files into an SQL database, which was done by (Robertson and Watson, 2014). (b) shows our extended data processing that first extracts events from the database into files (c) containing builds, kills and observed enemy units. All events are then run through a forward model to generate vectorized state-action pairs with normalized values (d).

the Protoss player to produce Terran builds. While the data preprocessing required for training is a relatively long process, the same data can be gathered directly by a playing (or observing) bot during a game.

5.1.2 Network Architecture

Since our dataset contains neither images nor sequential data, a simple multi-layered network architecture with fully-connected layers is used. Our game state contains all the material produced and observed by the player throughout the game, unless it has been destroyed, and thus there is no need for recurrent connections in our model. The network that obtained the best results has four hidden layers. The input layer has 210 units, based on the state vector described in Section 5.1.1, which is followed by four hidden layers of 128 units with the ReLU activation function. The output layer has one output neuron for each of the 58 build types a Protoss player can produce and uses the softmax activation

function. The output of the network is thus the probability of producing each build in the given state.

5.1.3 Training

The dataset of 789,571 state-action pairs is split into a training set of 631,657 pairs (80%) and a test set of 157,914 pairs (20%). The training set is exclusively used for training the network, while the test set is used to evaluate the trained network. The state-action pairs, which come from 2,005 different Protoss versus Terran games, are not shuffled prior to the division of the data to avoid that actions from the same game end up in both the training and test set.

The network is trained on the training set, which is shuffled before each epoch. Xavier initialization is used for all weights in the hidden layers and biases are initialized to zero. The learning rate is 0.0001 with the Adam optimization algorithm (Kingma and Ba, 2014) and a batch size of 100. The optimization algorithm uses the cross entropy loss function $-\sum_i y'_i \log(y_i)$, where y is the output vector of the network and y' is the one-hot target vector. The problem is thus treated as a classification problem, in which the network tries to predict the next build given a game state. In contrast to classical classification problems, identical data examples (states) in our dataset can have different labels (builds), as human players execute different strategies and also make mistakes while playing. Also, there is no correct build for any state in StarCraft, but some builds are much more likely to be performed by players. The network could also be trained to predict whether the player is going to win the game, but how to best incorporate this in the decision-making process is an open question. Instead here we focus on predicting actions made by human players, similarly to the supervised learning step in AlphaGo (Silver et al., 2016).

5.1.4 Integration with a StarCraft Bot

We build on the UAlbertaBot, which has a production manager that manages a queue of builds that the bots must produce in order. The production manager, which normally uses a goal-based search, is modified to use the network trained on replays instead. Whenever

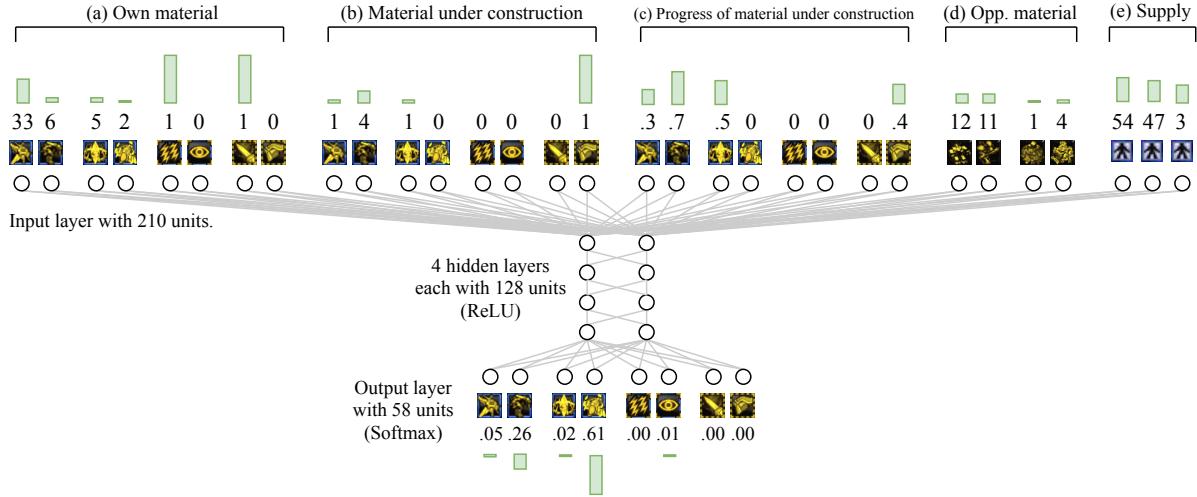


Figure 5.1.2: Neural Network Architecture. The input layer consists of a vectorized state containing normalized values representing the number of each unit, building, technology, and upgrade in the game known to the player. Only a small subset is shown on the diagram for clarity. Three inputs also describe the player's supplies. The neural network has four hidden fully-connected layers with 128 units each using the ReLU activation function. These layers are followed by an output layer using the softmax activation function and the output of the network is the prediction of each build being produced next in the given state.

the module is requested (by other modules) for the next build to produce, the request is forwarded along, with a description of the current game state, to the neural network which then returns a probability distribution of the build prediction. Since the network is only trained on Protoss versus Terran games, it is only tested in this matchup. Our approach can, however, easily be applied to the other matchups as well. UAlbertaBot does not handle some of the advanced units well, so these were simply excluded from the output signals of the network. The excluded units are: archons, carriers, dark archons, high templars, reavers and shuttles.

Two action selection policies were tested:

Greedy action selection: The build with the highest probability is always sampled. This approach creates a deterministic behavior with a low variation in the units produced. A major issue of this approach is that rare builds such as upgrades will likely never be sampled.

Probabilistic action selection: Builds are sampled with the probabilities of the softmax output units of the network. In the example in Figure 5.1.2, a probe will be selected with a 5% probability and a zealot with 26% probability. With a low probability, this approach

will also select some of the rare builds, and can express a wide range of strategies. Another interesting feature is that it is stochastic and harder to predict, and thus exploit, by the opponent.

5.2 Results

5.2.1 Build Prediction

The best network architecture we found managed to reach a top-1 error rate of 54.6% on the test set (averaged over five training runs), which means that it is able to guess the next build around half the time, and with top-3 and top-10 error rates of 22.92% and 4.03%. For a simple comparison to a few baselines, an approach that always predicts the next build to be a probe, which is the most common build in the game for Protoss, has a top-1 error rate of 73.9% and thus performs significantly worse. Predicting randomly with uniform probabilities achieves a top-1 error rate of 98.28%. Some initial experiments with different input layers show that we obtain worse error rates by omitting parts of the state vector that we described in Section 5.1.1. For example, when opponent material is omitted, the top-1 error increases to an average of 58.17%. Similarly, omitting the material under construction (together with the progress) increases the average top-1 error rate to 58.01%. The results of these experiments are summarized in Table 5.2.1 with error rates averaged over five training runs for each input layer design. The top-1, top-3 and top-10 error rates in the table show the networks' ability to predict using one, three and ten guesses respectively, determined by their output. All networks were trained for 50 epochs as the error rates stagnated prior to this point. Overfitting is minimal with a difference less than 1% between the top-1 training and test errors.

To gain further insights into the learned model, the predictions of building a new base given a varying number of probes (workers) is plotted in Figure 5.2.1. States are taken from the test set in which the player has only one base. The network successfully learned that humans usually create a base expansion when they have around 20-30 probes, which is aligned our understanding of expansion timings for Protoss.

Input	Top-1 error	Top-3 error	Top-10 error
a+b+c+d+e	54.60% ± 0.12%	22.92% ± 0.09%	4.03% ± 0.14%
a+b+c+e	58.17% ± 0.16%	24.92% ± 0.10%	4.23% ± 0.04%
a+d	58.01% ± 0.42%	24.95% ± 0.31%	4.51% ± 0.16%
a	60.81% ± 0.09%	26.64% ± 0.11%	4.65% ± 0.21%
Probe	73.90% ± 0.00%	73.90% ± 0.00%	73.90% ± 0.00%
Random	98.28% ± 0.04%	94.87% ± 0.05%	82.73% ± 0.08%

Table 5.2.1: The top-1, top-3 and top-10 error rates of trained networks (averaged over five runs) with different combinations of inputs. (a) is the player’s own material, (b) is material under construction, (c) is the progress of material under construction, (d) is the opponent’s material and (e) is supply. The input layer is visualized in Figure 5.1.2. *Probe* is a baseline predictor that always predicts the next build to be a probe and *Random* samples from a uniform distribution. The best results (in bold) are achieved by using all the input features.

5.2.2 Playing StarCraft

UAlbertaBot is tested playing the Protoss race against the built-in Terran bot, with the trained network as production manager. Both the greedy and probabilistic actions selection strategies are tested in 100 games in the two-player map Astral Balance. The results, summarized in Table 8.4.1, demonstrates that the probabilistic strategy is clearly superior, winning 68% of all games. This is significant at $p \leq 0.05$ according to the two-tailed Wilcoxon Signed-Rank. The greedy approach, which always selects the action with the highest probability, does not perform as well. While the probabilistic strategy is promising, it is important to note that when UAlbertaBot plays as Protoss and follows the hand-designed dragoon rush strategy, it wins 100% of all games against the built-in Terran bot. However, the dragoon rush is fixed and is thus highly exploitable by adaptive players.

To further understand the difference between the two approaches, the builds selected by each selection strategy are analyzed. A subset of these builds are shown in Table 5.2.3. The probabilistic strategy clearly expresses a more varied strategy than the greedy one. Protoss players often prefer a good mix of zealots and dragoons as it creates a good dynamic army, and the greedy strategy clearly fails to achieve this. Additionally, with the greedy approach, the bot never produces any upgrades, because they are too rare in a game to ever become the most probable build. The blind probabilistic approach (which

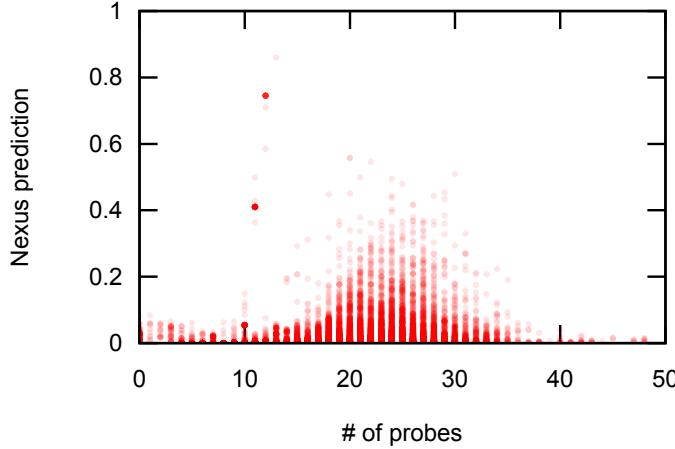


Figure 5.2.1: The prediction of the next build being a Nexus (a base expansion) predicted by the trained neural network. Each data point corresponds to one prediction from one state. These states have only one Nexus and are taken from the test set. The small spike around 11 and 12 probes shows that the network predicts a fast expansion build order if the Protoss player has not build any gateways at this point.

Action selection	Wins vs. Built-in Terran
Probabilistic	68/100
Probabilistic (blind)	59/100
Greedy	53/100
Random	0/100
UAlbertaBot (dragoon rush)	100/100

Table 5.2.2: The win percentage of UAlbertaBot with the trained neural network as a production manager against the built-in Terran bot. The probabilistic strategy selects actions with probabilities equal to the outputs of the network while the greedy network always selects the action with the highest output, and random always picks a random action. The blind probabilistic network does not receive information about the opponent’s material (inputs are set to 0). UAlbertaBot playing as Protoss with the scripted dragoon rush strategy wins 100% of all games against the built-in Terran bot.

ignores knowledge about the opponent by setting these inputs to zero) reached a lower win rate of just 59%, further corroborating that the opponent’s units and buildings are important for build-order decision making. We also tested the probabilistic approach against UAlbertaBot with the original production manager configured to follow a fixed marine rush strategy, which was the best opening strategy for UAlbertaBot when playing Terran. Our approach won 45% of 100 games, demonstrating that it can play competitively against this aggressive rush strategy, learning from human replays alone.

Figure 5.2.2 visualizes the learned opening strategy with greedy action selection. While the probabilistic strategy shows a better performance in general (Table 8.4.1), the strategy

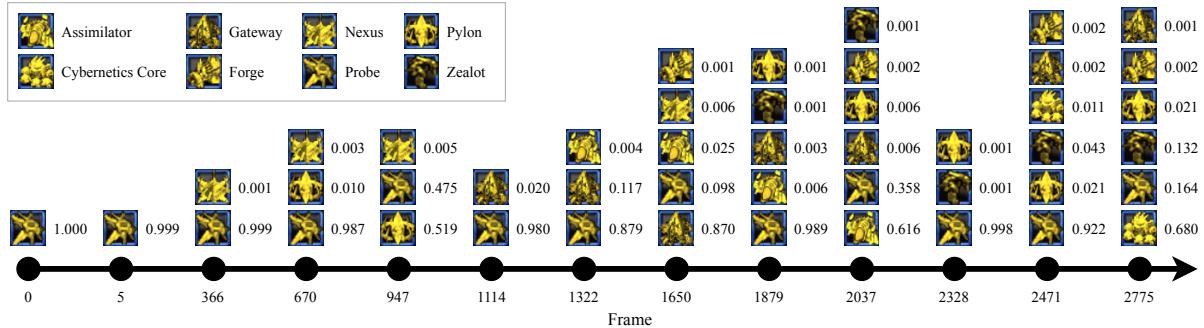


Figure 5.2.2: The opening build order learned by the neural network when playing against the built-in Terran bot (the build order also depends on the enemy units observed). The number next to each build icon represents the probability of the build being produced next, and points on the timescale indicate when the bot requests the network for the next build. In this example the network follows the greedy strategy, always picking the build with the highest probability.

Action selection	Units produced							Upgrades researched			
Probabilistic	50.84	14.62	17.3	1.00	3.56	0.11	0.13	0.32	0.03	0.07	0.01
Greedy	70.12	1.46	32.75	0.00	2.40	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.2.3: The average number of different unit types produced by the two different action selection strategies against the built-in Terran bot. The results show that the greedy strategy executes a very one-sided unit production while the probabilistic strategy is more varied.

performed by the greedy action selection is easier to analyze because it is deterministic and has a one-sided unit production. The learned build order shown in Figure 5.2.2 is a *One Gate Cybernetics Core* opening with no zealots before the cybernetics core. This opening was performed regularly against the built-in Terran bot, which does not vary much in its strategy. The opening is followed by a heavy production of dragoons and a few observers. A base expansion usually follows the first successful confrontation. Some losses of the greedy approach were caused by UAlbertaBot not being able to produce more buildings, possibly because there was no more space left in the main base. A few losses were also directly caused by some weird behavior in the late game, where the bot (ordered by the neural network) produces around 20 pylons directly after each other. Generally, the neural network expresses a behavior that often prolongs the game, as it prefers expanding bases when leading the game. This is something human players also tend to do, but since UAlbertaBot does not handle the late game very well, it is not a good strategy for this particular bot.

The behavior of the probabilistic strategy is more difficult to analyze, as it is stochastic. It usually follows the same opening as the greedy approach, with small variations, but then later in the game, it begins to mix its unit production between zealots, dragoons and dark templars. The timings of base expansions are very different from game to game as well as the use of upgrades.

5.3 Discussion

This chapter demonstrated that build-order tasks in StarCraft can be learned from replays using deep learning, and that the learned policy can be used to outperform the built-in bot in StarCraft. In this section, we discuss the short-comings of this approach and give suggestions for future research that could lead to strong StarCraft bots.

The built-in StarCraft bot is usually seen as a weak player compared to humans. It gives a sufficient amount of competition for new players but only until they begin to learn established opening strategies. A reasonable expectation would be that UAlbertaBot, using our trained network, would defeat the built-in bot almost every time. By analyzing the games played, it becomes apparent that the performance of UAlbertaBot decrease in the late game. It simply begins to make mistakes as it takes weird micromanagement decisions when it controls several bases and groups of units. The strategy learned by our network further enforces this faulty behavior, as it prefers base expansions and heavy unit production (very similar to skilled human players) over early and risky aggressions. The trained network was also observed to make a few faulty decisions, but rarely and only in the very late game. The reason for these faults might be because some actions were excluded, since UAlbertaBot does not handle these builds well, and because there are fewer data examples for rare late game situations.

Despite the presented approach not achieving a skill level on par with humans, it should be fairly straightforward to extend it further with reinforcement learning. Supervised learning on replays can be applied to pre-train networks, ensuring that the initial exploration during reinforcement learning is sensible, which proved to be a critical step to surpass humans in the game go (Silver et al., 2016). Reinforcement learning is especially promising for a modular-based bot as it could optimize the build-order policy to fit the fixed

micromanagement policy. Additionally, learning a build-order policy to specifically beat other bots that are competing in a tournament is a promising future direction. After this work was published (Justesen and Risi, 2017b), Tang et al. (2018) applied reinforcement learning to a similar setup as described here, achieving high win rates against the level 7 (out of 10) built-in bot in StarCraft II. Sun et al. (2018) further extended the approach of Tang et al. (2018) in a bot called TStarBots which is able to outperform the built-in level 10 bot that actually cheats by harvesting resources faster and by having access to information that is normally hidden. The downside of these extensions are that they learn very particular rush strategies that are likely to be exploitable by other players than the bot they trained against, as it only plays one particular strategy. The authors report that the learned policy “lacks strategy diversity in order to consistently beat human players” (Sun et al., 2018).

This chapter also introduces a new benchmark for machine learning, where the goal is to *predict the next unit, building, technology or upgrade that is produced by a human player given a game state in StarCraft*. An interesting extension to the presented approach, which could potentially improve the results, could involve including positional information as features for the neural network. The features could be graphical and similar to the minimap in the game that gives an abstract overview of where units and buildings are located on the map. Regularization techniques such as dropout (Srivastava et al., 2014) or L2 regularization (Nowlan and Hinton, 1992) could perhaps reduce the error rate of deeper networks and ultimately improve the playing bot. We test some of these ideas in the next chapter.

Finally, it would be interesting to apply our trained network to a more sophisticated StarCraft bot that is able to manage several bases well and can control advanced units such as spell casters and shuttles. This is currently among our future goals, and hopefully this bot will participate in the coming StarCraft competitions. Since this work was published, we have developed our own bot and done some preliminary testing against two skilled human players. While the bot played well in two rather long games, they eventually lost. A summary with videos is available online¹.

¹<https://njustesen.com/2017/10/18/playing-starcraft-against-humans-with-a-neural-network/>

5.4 Summary

This chapter presented an approach that learns from StarCraft replays to predict the next build produced by human players. 789,571 state-action pairs were extracted from 2,005 replays of highly skilled players. We trained a neural network with supervised learning on this dataset, with the best network achieving top-1 and top-3 error rates of 54.6% and 22.9%. To demonstrate the usefulness of this approach, the open source StarCraft bot UAlbertaBot was extended to use such a neural network as a production manager, thereby allowing the bot to produce builds based on the networks predictions. Two action selection strategies were introduced: A greedy approach that always selects the action with the highest probability, and a probabilistic approach that selects actions corresponding to the probabilities of the network’s softmax output. The probabilistic strategy proved to be the most successful and managed to achieve a win rate of 68% against the games built-in Terran bot. Additionally, we demonstrated that the presented approach was able to play competitively against UAlbertaBot with a fixed rush strategy. The presented approach stands in contrast to AlphaStar (Vinyals et al., 2019) that was announced two years later. AlphaStar learns the game end-to-end and achieves a high skill level. Our approach requires less computational resources and remain a practical alternative to AlphaStar, and we believe that this line of work deserves to be explored further.

Chapter 6

Behavioral Repertoire Imitation Learning

Reinforcement learning has critical shortcomings when reward signals are sparse or interaction with the environment is expensive. There are several attempts to mitigate these shortcomings, including curriculum learning (Bengio et al., 2009; Graves et al., 2016; Matiisen et al., 2017), reward shaping (Ng et al., 1999), curiosity-driven exploration (Pathak et al., 2017b), diversification (Conti et al., 2018; Eysenbach et al., 2018), and Imitation Learning (Bakker and Kuniyoshi, 1996).

In this chapter, we focus on imitation learning, extending on the work from the previous chapter. Traditional imitation learning techniques result in a single policy, which usually expresses an “averaged” behavior among all the behaviors present in the dataset. We see this as a major limitation of imitation learning. It would be more desirable to instead learn a diverse set of policies, expressing all the different types of behaviors present in the dataset. Additionally, having a repertoire of different behaviors allows for inter-game adaptation while the system is deployed.

An imitation learning approach called Generative Adversarial Imitation Learning (GAIL) use a Generative Adversarial Network (GAN), wherein a policy network (the generator) is trained to fool a discriminator by producing state-action samples that imitates human behaviors Ho and Ermon (2016). Two extensions of GAIL can learn diverse behaviors similarly to our work. Wang et al. (2017b) employs a variational auto-encoder to map

demonstration sequences to an embedding vector (behavioral features) that is used to condition the policy network and the discriminator. This approach was applied to a robot arm for reaching tasks where different behaviors (reached positions) could be interpolated after training. InfoGAIL is another extension to GAIL that also learns a repertoire of behaviors (Li et al., 2017). InfoGAIL learns salient semantic features in an unsupervised manner in the style of InfoGAN (Chen et al., 2016), which has been used to find semantic features in images of hand-written letters and digit, as well as photos of faces. While our approach requires a manually designed behavioral dimensions, this can give the user more control over the learned policy; different behavioral spaces can be beneficial for different purposes. InfoGAN was capable of learning a variety of car driving behaviors in TORCS (Chen et al., 2016). A latent space that has been learned in an unsupervised manner does, however, not necessarily reflect that space behaviors that is useful for strategic adaptation, in contrast to a manually defined behavioral space. The approach presented in this chapter is not based on GANs but instead extends traditional supervised imitation learning conditioned with a low-dimensional behavioral description that is suitable for fast adaptation in strategy games.

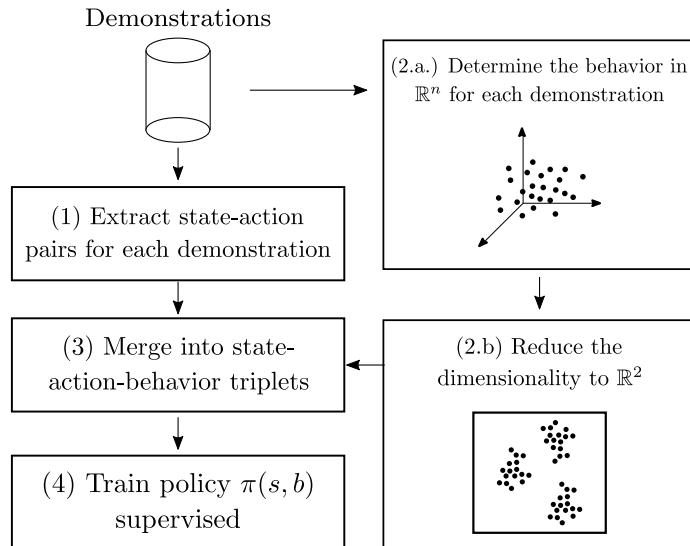


Figure 6.0.1: Behavioral Repertoire Imitation Learning (BRIL) trains a policy $\pi(s, b)$ supervised on a data set of state-actions pairs augmented with behavioral descriptors in \mathbb{R}^2 for each demonstration. When deployed, a system can adapt its behavior by modulating b . High-dimensional behavioral spaces can be reduced using dimensionality reduction, as low-dimensional behavioral descriptions allow for faster adaption.

Addressing the limitations of current imitation learning methods, we present a new IL approach called *Behavioral Repertoire Imitation Learning* (BRIL), which is inspired by Quality-Diversity (QD) algorithms (Pugh et al., 2016; Mouret and Clune, 2015) and reinforcement learning methods that learn multiple different behaviors. In contrast to traditional optimization techniques, QD-algorithms attempt to find a diverse set of high-quality solutions rather than a single optimal solution. When QD-algorithms search in policy space, they typically discover hundreds or thousands of different policies controlled by different neural networks. BRIL instead learns a behavioral repertoire using *a single model* that can be manipulated to express multiple behaviors, similarly to reinforcement learning algorithms that learn a single policy for multiple goals (Schaul et al., 2015; Andrychowicz et al., 2017). BRIL consists of a multi-step process (see Figure 6.0.1) wherein the experimenter: (1) extracts state-action pairs (similarly to many imitation learning approaches), (2) designs a set of behavioral dimensions to form a behavioral space (similarly to many QD algorithms) and determines the behavioral description (coordinates in the space) for each demonstration, (3) merges the data to form a dataset of state-action-behavior triplets, and (4) trains a model to predict actions from state-behavior pairs through supervised learning. When deployed, the model can act as a policy and the behavior of the model can be manipulated by changing its behavioral input features.

BRIL is tested on the build-order planning problem in StarCraft from Chapter 5. The results in this chapter demonstrate that the behavior of the learned policy can be efficiently manipulated without further training. Additionally, the behavior of the learned policy can be optimized online with the Upper Confidence Bounds (UCB1) algorithm, outperforming a traditional imitation learning approach when tested against one of the built-in bots. We believe this approach can be particularly useful when modeling human players in a game, where the policy should express the entire range of distinct behaviors instead of the average of all. We hypothesize that this property can allow a system to be more robust to exploitation, which is a concern for AI system in many games. Furthermore, BRIL could be useful in applications beyond games, e.g. in adaptive and resilient robotics.

6.1 Dimensionality Reduction

Principal Component Analysis (PCA) is a popular dimensionality reduction method that finds a new set of dimensions, also called the *principal components*, from a set of data points. The number of dimensions can then be reduced by selecting only a subset of the principal components that explain the maximal variance of the data points. Reducing the dimensionality of a dataset from n to p dimensions, where $n \geq p$, using PCA consists of several fairly simple steps. First, the data is standardized, a covariance matrix is computed, and the eigenvectors and eigenvalues of the covariance matrix are found. These eigenvectors correspond to the principal components and the eigenvalues represent how well they each explain the variance in the data. The eigenvectors are thus sorted in descending order by their eigenvalues and the first p eigenvectors are selected to form a new matrix called a *feature vector* with columns equal to the eigenvectors. The original dataset, structured as a matrix with the data points as columns, can then be transformed into the reduced space by multiplying it with the transpose of the feature matrix.

PCA can, however, not capture non-linear relationships in the data, in contrast to more advanced methods such as t-Distributed Stochastic Neighbor Embedding (t-SNE) (Maaten and Hinton, 2008) or UMAP (McInnes et al., 2018). In the case where only the similarities (or dissimilarities) are known, rather than coordinates in the behavioral space, Multidimensional Scaling (MDS) can be a useful method (Jaworska and Chupetlovska-Anastasova, 2009). For example, if behaviors are described as sequences of actions, which cannot easily be transformed into coordinates, it is possible to compute their similarities instead.

6.2 Universal Policies

In value-based reinforcement learning, one typically learns a state value function $V_\pi(s)$ or a state-action value function $Q_\pi(s, a)$ for a policy π . Universal Value Function Approximators (UVFA) instead learn a joint distribution $V_\pi(s, g)$ or $Q_\pi(s, a, g)$ over all goals G (Schaul et al., 2015). UVFA can be learned using supervised learning from a training set of optimal values such as $V_g^*(s)$ or $Q_g^*(s, a)$, or alternatively, it can be learned through

reinforcement learning by switching between goals both when generating trajectories and when computing gradients. Hindsight Experience Replay is an extension to UVFAs, which performs an additional gradient update with the goal being replaced by the terminal state; this modification can give further improvements when it is infeasible to reach the goals (Andrychowicz et al., 2017). An extension to Generative Adversarial Imitation Learning (GAIL) augments each trajectory with a *context* (Merel et al., 2017), which specifies the agent’s sub-goals that can be modulated at test-time.

In our approach, we are not considering goals, but rather behaviors, with the aim of learning a universal policy $\pi(s, b)$ over states $s \in S$ and behaviors $b \in B$ in a particular behavioral space. We are thus combining the QD approach of designing a behavioral space with the idea of learning a universal policy to express behaviors in this space.

6.3 Approach

This section describes two approaches to learning behavioral repertoires using imitation learning. We first describe how a behavioral space can be formed from demonstrations. Then we introduce a naive imitation learning approach that first clusters the demonstrations based on their coordinates in the behavioral space, and then applies traditional imitation learning on the clusters. Finally, BRIL is introduced, which learns a single policy augmented with a behavioral feature input rather than learning multiple policies for each behavioral cluster. BRIL consists of a number of steps which are depicted in Fig. 6.0.1.

6.3.1 Behavioral Spaces from Demonstrations

A behavioral space consists of a number of behavioral dimensions that are typically determined by the experimenter. For example, in StarCraft, behavioral dimensions can correspond to the ratio of each army unit produced throughout the game to express the strategic characteristics of the player. A behavioral space can require numerous dimensions to be able to express meaningful behavioral relationships between interesting solutions for a problem. Intuitively, if the problem is complex, more dimensions can give a finer

granularity in the diversity of solutions. However, there is a trade-off between granularity and adaptation, as low-dimensional spaces are easier to search in. We thus propose the idea of first designing a high-dimensional behavioral space and then reducing the number of dimensions through dimensionality reduction techniques. In our preliminary experiments, it has shown beneficial to reduce the space to two dimensions, as it allows for easy visualization of the data distribution and it also seems to be a good trade-off between granularity and adaptation speed. In preliminary experiments with one-dimensional behavioral spaces, we noticed that nearby solutions could be wildly different.

6.3.2 Imitation Learning on Behavioral Clusters

A naive imitation learning approach, that learns behavioral repertoires, trains n policies on n behaviorally diverse subsets of the demonstrations. This idea is similar to the state-space clustering by Thurau et al. (2004), but here we cluster data points in a behavioral space instead. When a behavioral space is defined, each demonstration can be defined by a particular behavioral description (a coordinate in the \mathbb{R}^n dimensional space), where afterwards a clustering algorithm can split the dataset into several subsets. Hereafter, traditional imitation learning can be applied to each subset with the goal of learning one policy for each behavioral cluster. This approach creates a discrete set of policies similarly to current QD algorithms. However, it introduces a difficult dilemma: if the clusters are small, there is a risk of overfitting to these reduced training sets. On the other hand, if the clusters are large but few, the granularity of behaviors is lost.

6.3.3 Learning Behavioral Repertoires

QD algorithms typically fill an archive with diverse and high-quality solutions, sometimes resulting in thousands of policies stored in a single run, which increases the storage requirements in training as well as in deployment. To reduce the storage requirement, one can decrease the size of the archive, with the trade-off of losing granularity in the behavioral space. The main approach introduced here, called Behavioral Repertoire Imitation Learning (BRIL), solves these issues and reduces overfitting by employing a universal policy instead, in which a single policy is conditioned on a behavioral description.

In contrast to QD algorithms, the goal of BRIL is not to optimize neither quality nor diversity directly. Instead, BRIL attempts to imitate and express the diverse range of behaviors and the quality that exists in a given set of demonstrations. Additionally, BRIL produces a continuous space of policies which is potentially more expressive than a discrete set.

BRIL extends the traditional imitation learning setting through the following approach. First, behavioral characteristics of each demonstration are determined. If the dimensionality of these descriptions is large, it can be useful to reduce the space as described in the earlier section. A training set of state-action-behavior triplets is then constructed, such that the behavior is equal to the behavioral description of the corresponding demonstration. Then, a policy $\pi(s, b)$ is trained supervised on this dataset to map states and behaviors to actions. Using this approach, the training set is not reduced to small behavioral clusters.

When the trained policy is deployed, the behavioral feature input can be modulated with the goal of manipulating its behavior. The simplest approach is to fix the behavioral features throughout an episode, evaluate the episodic return, and then consider new behavioral features for the next episode. This approach should allow for inter-game adaptation, which is explored in our experiments. One could also manipulate the behavioral features during an episode e.g. by learning a meta-policy.

6.4 Experiments

This section presents the experimental results of applying BRIL to the game of StarCraft. Policies are trained to control the build order planning module of a relatively simple scripted StarCraft bot¹ that plays the Terran race. While the policy is trained off-line, our experiments attempt to optimize the playing strength of this bot online, in-between games, by manipulating its behavior.

¹<https://github.com/njustesen/sc2bot>

6.4.1 Behavioral Feature Space

The behavioral space for a StarCraft build-order policy can be designed in many ways. Inspired by the AlphaStar League Strategy Map (Vinyals et al., 2019), the behavioral features are constructed from the army composition, such that the dimensions represent the ratios of each unit type. We achieve this by traversing all demonstrations in the data set, counting all the army unit creation events, and computing the relative ratios. Each demonstration thus has an n -dimensional behavioral feature description, where $n = 15$ is the number of army unit types for Terran.

To form a 2D behavioral space, which allows for easier online search and analysis, we apply the T-distributed Stochastic Neighbor Embedding (t-SNE). We have also experimented with the PCA, but for this dataset, the separation of points was less distinguishable. Fig. 6.4.1 visualizes the points of all the demonstrations in this 2D space, and Fig. 6.4.1a shows four plots where the points are colored to show the ratios of Marines, Marauders, Hellions, and Siege Tanks that was produced in the games.

6.4.2 Clustering

For the baseline approach that applies imitation learning on behavioral clusters, we use density-based spatial clustering of applications with noise (DBSCAN) with $\epsilon = 0.02$ and a minimum number of samples per cluster of 30. We performed a grid-search on these two parameters to find the most meaningful data separation; however, the clustering is not perfect due to the large number of outliers. The clusters are visualized in Fig. 6.4.1b, where outliers are colored black.

6.4.3 Prediction Accuracy

Data from StarCraft 2 replays are extracted with sc2reaper², a tool built using the StarCraft II Learning Environment inspired on the MSC Database by Wu et al. (2017a).

²<https://github.com/miguelgondu/sc2reaper>

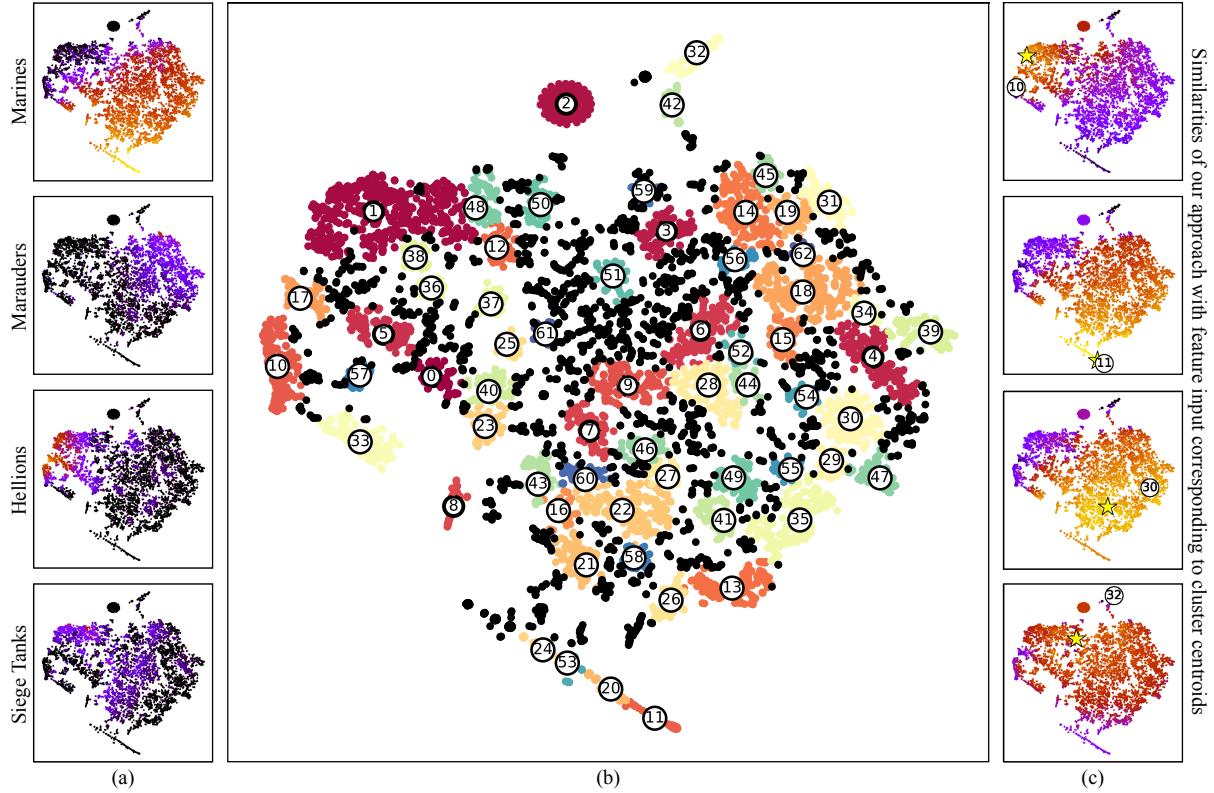


Figure 6.4.1: Visualizations of the 2D behavioral space of Terran army unit combinations in 7,777 Terran versus Zerg replays. Each point represents a replay from the Terran player’s perspective. The space was reduced using t-SNE. (a) The data points are illuminated (black is low and yellow is high) by the ratio of Marines, Marauders, Hellions, or Siege Tanks produced in each game. (b) 62 clusters found by DBSCAN. Cluster centroids are marked with a circle and the cluster number and outliers are black. The noticeable cluster 2 has no army units. (c) The similarity between the behaviors of the human players and our approach with four different feature inputs, corresponding to the coordinates of centroids of cluster 10, 11, 30, and 32. The behavior of our approach is averaged over 100 games against the easy Zerg bot and its nearest human behavior is marked with a star. The behavior of the learned policy can be efficiently manipulated to change its behavior. Additionally, we can control the behavior such that it resembles the behavior of a human demonstration.

7,777 replays of Terran vs. Zerg were processed, extracting state-action pairs every half a second resulting in a dataset of 1,625,671 state-action pairs.

These states contain an abstraction of the game state similar to the work in Chapter 5. The abstraction includes: 1) the agent’s resources, supply, units and technologies, 2) a tally of the enemy’s units that have been observed, and 3) the agent’s units and technologies in progress, including how far they are from being completed.

Once the replays were post-processed for clustering, the dataset was split into training/test/validation following a 60% / 10% / 30% split per cluster. Three groups

Method	Mean test accuracy	Mean test loss	# of replays
IL	48.090 ± 0.080	1.775 ± 0.003	7777
BRIL	48.167 ± 0.083	1.768 ± 0.003	7777
IL (C10)	32.608 ± 0.417	2.096 ± 0.013	189
IL (C11)	71.326 ± 0.316	1.088 ± 0.008	74
IL (C30)	45.709 ± 0.499	1.976 ± 0.020	149
IL (C32)	45.855 ± 0.681	1.770 ± 0.006	97

Table 6.4.1: Test accuracy and loss for Imitation Learning (IL), BRIL, and IL trained on clusters 10, 11, 30 and 32. Results show no significant difference between the IL and BRIL in terms of prediction accuracy. BRIL is, however, able to express multiple behaviors based on the additional input (see Table 6.4.2).

of neural networks were trained, all with three hidden layers and 256 hidden nodes per layer: (1) One baseline model trained on the whole dataset with no augmentation of behavioral features, (2) a BRIL model on the whole dataset with two extra input nodes for the augmented behavioral features (i.e. the coordinates in Fig. 6.4.1b), and (3) several cluster baseline models trained on demonstrations from their respective clusters without the augmented behavioral features.

These experiments were carried out ten times per model. Table 6.4.1 shows the mean test accuracy and mean test loss over these ten models for the baseline imitation learning approach, the novel BRIL approach, and four different cluster baselines (Clusters 10, 11, 30 and 32), which were selected for their wildly different behaviors. The results show that augmenting by behavioral features has no significant effect on the test accuracy or loss. However, the next section shows how our new approach is able to express different behaviors with a single neural network.

6.4.4 Performance in StarCraft

We applied the trained policy models as build order modules in a scripted StarCraft II Terran bot called sc2bot. It is important to note that this is a very simplistic bot with several flaws and limitations. Therefore, the main goal of our experiments here is not to achieve human-level performance in StarCraft, but rather to test if BRIL allows us to do manipulate its behavior and enables inter-game adaptation. The build order module, here controlled by one of our policies, is queried with a state description and returns a build order action, i.e. which building, research, or unit to produce next. The worker and building modules of the bot perform these actions accordingly, while assault, scout, and

army modules control the army units. Importantly, policies we test act in a system that consists of both the bot, the opponent bot, and the game world. When we want to utilize our method for adaptation, we are thus not only adapting to the opponent but also the peculiarities of the bot itself.

Method	Wins	Distance to cluster centroid			
		C10	C11	C30	C32
IL	41/100	0.58	0.22	0.39	0.75
IL (C10)	3/100	0.05	0.76	0.81	0.71
IL (C11)	7/100	0.74	0.00	0.52	0.96
IL (C30)	18/100	0.76	0.21	0.31	0.79
IL (C32)	0/100	0.71	0.94	0.57	0.04
BRIL (C10)	27/100	0.21	0.85	0.81	0.60
BRIL (C11)	76/100	0.70	0.05	0.53	0.95
BRIL (C30)	47/100	0.60	0.31	0.29	0.65
BRIL (C32)	16/100	0.42	0.72	0.53	0.36
Wins for each option					
Method	Wins	C10	C11	C30	C32
BRIL (UCB1)	61/100	5/14	47/59	8/18	1/9

Table 6.4.2: Results in StarCraft using Imitation Learning (IL) on the whole training set, IL on individual clusters (C10, C11, C30, and C32), Behavioral Repertoire Imitation Learning (BRIL) with fixed behavioral features corresponding to centroids in C10, C11, C30, and C32. Additionally, results are shown in which UCB1 selects between the four behavioral features in-between games. Each variant played 100 games against the easy Zerg bot. These results demonstrate that by using certain behavioral features, the BRIL policy outperforms the traditional IL approach as well as IL on behavioral clusters.

We will first focus on the results of the traditional IL approach. Table 6.4.2 shows the number of wins in 100 games on the two-player map CatalystLE as well as the corresponding average behaviors (i.e. the army unit ratios). Our bot played as Terran against the built-in Easy Zerg bot. The traditional IL approach won 41/100 games and IL on behavioral clusters showed very poor performance with a maximum of 18/100 wins by the model trained on C30. We hypothesize that the poor win rate of this naive approach are due to their training sets being too small such that the policies do not generalize to many of the states explored in the test environment. Besides the number of wins, we compute the nearest demonstration in the entire data set from the average behavior, and use it as an estimate of the policy’s position in the 2D behavioral space. From the estimated point, we calculate the distance to each of the four cluster centroids. This analysis revealed that the policies trained on behavioral clusters express behaviors close to the clusters they were trained on (see the distances to the cluster centroids in Table 6.4.3).

Method	Wins	Combat units produced				
		Marines	Marauders	Hellions	S. Tanks	Reapers
IL	41/100	44.1 ± 50.5	0.7 ± 3.2	2.6 ± 7.6	1.7 ± 6.5	0.3 ± 1.1
IL (C10)	3/100	1.1 ± 2.3	0.1 ± 0.3	3.11 ± 6.1	0.1 ± 0.4	0.1 ± 0.33
IL (C11)	7/100	18.8 ± 38.4	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
IL (C30)	18/100	43.5 ± 62.6	0.9 ± 5.4	0.2 ± 1.3	0.0 ± 0.2	0.2 ± 0.8
IL (C32)	0/100	0.1 ± 0.2	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	9.9 ± 18.5
BRIL (C10)	27/100	2.4 ± 4.9	0.0 ± 0.0	14.6 ± 18.9	4.0 ± 5.2	0.2 ± 0.6
BRIL (C11)	76/100	81.4 ± 50.1	0.0 ± 0.1	0.2 ± 0.1	0.9 ± 2.4	0.3 ± 0.6
BRIL (C30)	47/100	41.6 ± 36.4	2.4 ± 6.7	0.7 ± 2.5	4.1 ± 7.8	0.5 ± 1.2
BRIL (C32)	16/100	7.1 ± 11.4	1.7 ± 6.5	3.2 ± 8.3	6.7 ± 9.7	0.8 ± 1.5
Method	Wins	Combat units produced				
		Marines	Marauders	Hellions	Siege Tanks	Reapers
BRIL (UCB1)	61/100	52.1 ± 47.7	0.5 ± 3.2	4.3 ± 12.5	2.5 ± 6.2	0.3 ± 1.3

Table 6.4.3: Results in StarCraft using Imitation Learning (IL) on the whole training set, IL on individual clusters (C10, C11, C30, and C32), Behavioral Repertoire Imitation Learning (BRIL) with fixed behavioral features corresponding to centroids in C10, C11, C30, and C32. Additionally, results are shown in which UCB1 selects between the four behavioral features in-between games. Each variant played 100 games against the easy Zerg bot. The nearest demonstration in the entire dataset was found based on the bot’s mean behavior (normalized army unit combination) and the distance to each cluster centroid are shown. These results demonstrate that the behavioral features of the learned BRIL policy can be manipulated and controlled towards a desired behavior.

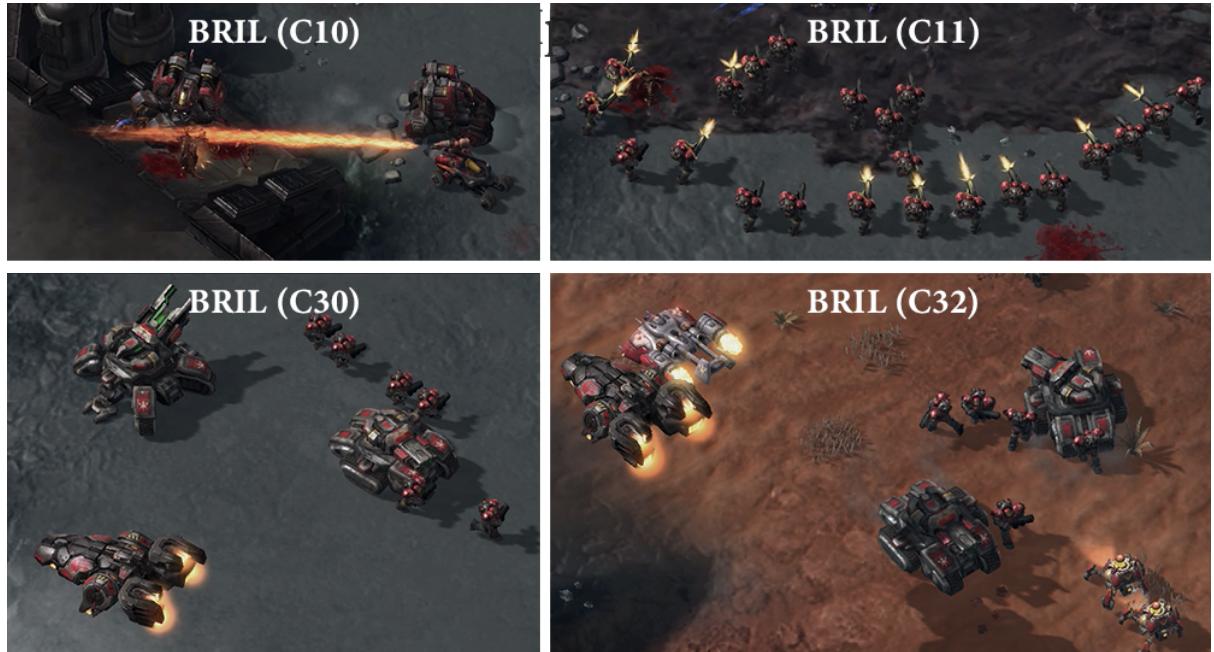


Figure 6.4.2: Screenshots of typical army compositions produced by our trained BRIL policy with behavioral features corresponding to the centroids of cluster 10, 11, 30 and 32. BRIL (C10) executes early timing pushes with Hellions and Cyclones, BRIL (C11) is aggressive with Marines only, BRIL (C30) creates mixed armies with many Marines and Siege Tanks, and BRIL (C32) also creates mixed armies but with less Marines and more Widow Mines.

Table 6.4.2 and 6.4.3 also shows the results for BRIL with the coordinates of the four cluster centroids as behavioral features. BRIL (C11) achieves a win rate of 76/100, thus outperforming traditional IL. These results demonstrate that the model can be tuned to achieve a higher performance than traditional IL, at least in this particular test environment. Analyzing the behavior of the bot with the behavioral features of C11 reveals that it performs an all-in Marine push, similarly to the behavior of the demonstrations in C11 (notice the position of C11 on Fig. 6.4.1.b and the illumination of Marines on Fig. 6.4.1.a). With the behavioral features of C30, the approach reached a higher win rate than traditional IL; however, this difference was not significant. We also notice that for both BRIL and IL on behavioral clusters, the average expressed behavior is closest to the cluster centroid that it was modulated to behave as, among the four clusters we selected. The results show that the behavior of the learned BRIL policy can be successfully controlled. However, the distances are on average larger than for IL on behavioral clusters. Figure 6.4.2 shows screenshots of typical army compositions produced by the BRIL policy with the four different behavioral features used.

6.4.5 Inter-game Adaptation

The final test aims to verify that we can indeed use BRIL for inter-game adaptation. We thus apply the UCB1 algorithm to select behavioral features from the discrete set of four options: {C10, C11, C30, C32} (i.e. the two-dimensional feature descriptions of these cluster centroids). This approach enables the algorithm to switch between behavioral features in-between games based on the return of the previous one, which is 1 for a win and 0 otherwise. The adaptive approach achieves 61/100 wins by identifying the behavior of C11 as the best option. Not surprisingly, the win rate is lower than when having the behavioral features of C11 fixed, but it outperforms traditional IL.

6.5 Discussion

We proposed two new IL methods in this chapter, one which learns a policy that is trained on only one behavioral cluster of data points and one which learns a single modifiable

policy on the whole dataset. Our results suggest that policies trained on small behavioral clusters overfit and are thus unable to generalize beyond the states available in the cluster. This drawback might be solved with fewer and larger clusters at the cost of losing granularity in the repertoire of policies. If data is abundant, this approach may also work better while we still suspect the same overfitting would occur. BRIL, on the other hand, is simple to implement and results in a continuous distribution of policies by adjusting the behavioral features. Additionally, the results suggest that BRIL generalizes better, most likely because it learns from the whole training set. However, that generality potentially comes with the cost of higher divergence between the expected behavior (corresponding to the behavioral input features) and the resulting behavior when tested. While an important concern, a divergence is somewhat expected since the test environment is very different from that of the training set (different maps and opponents). Further investigation is also required to analyze if the inter-game adaptive properties of BRIL comes with a cost of poorer intra-game adaptivity, as there is a chance, that the behaviors are inflexible and does not react to the opponent in-game.

Previous work showed how IL can kick-start learning before applying reinforcement learning (Silver et al., 2016; Vinyals et al., 2019). With BRIL, one can easily form a population of diverse solutions instead of just one, which may be a promising approach for domains with a plethora of strategic choices like StarCraft. Promising future work could thus combine BRIL with ideas from AlphaStar to automatically form the initial population of policies used in the AlphaStar League.

6.6 Summary

We introduced a new method called Behavioral Repertoire Imitation Learning (BRIL) and our results demonstrate its usefulness for inter-game adaptation. By labeling each demonstration $d \in D$ with a behavior descriptor confined within a pre-defined behavioral space, BRIL can learn a policy $\pi(s, b)$ over states $s \in S$ and behaviors $b \in B$. In our experiments, a low-dimensional representation of the behavioral space was obtained through dimensionality reduction. The results here demonstrate that BRIL can learn a policy that, when deployed, can be manipulated by conditioning it with a behavioral

feature input b , to express a wide variety of behaviors. Additionally, the observed behavior of the policy resembles the behavior characterized by b . Furthermore, a BRIL trained policy can be optimized online by searching for optimal behavioral features in a given setting. In our experiments, a policy trained with BRIL was optimized online beyond the performance reached by traditional IL, using UCB1 to select among a set of discrete behavioral features. Our results suggest that BRIL is useful learning method for inter-game adaptive game-playing agents. Future work will show if InfoGAIL (Li et al., 2017) can achieve the same property in strategy games and explore the advantages of each method.

Chapter 7

Rarity of Events

Deep reinforcement learning and deep neuroevolution have achieved impressive results learning to play video games (Chapter 3) and controlling both simulated and physical robots (Chebotar et al., 2017; Mirowski et al., 2016; Andrychowicz et al., 2017; Gu et al., 2016). These approaches, however, struggle to learn in environments where feedback signals (also called rewards) are sparse and/or delayed. A popular way to overcome this issue is to shape the reward function with prior knowledge such that the agent receives additional rewards to guide its learning process (Ng, 2003; Laud, 2004; Lample and Chaplot, 2017b). This approach can be time-consuming and requires substantial domain knowledge. Additionally, it is especially difficult to apply reward shaping for complex environments, as the particular reward values are difficult to determine. However, it is typically easy to define the events that should result in a positive reward. In this chapter, we propose a simple method that automatically shapes the reward values of pre-defined events during the training phase with the goal of performing a form of curriculum learning that adapts the reward values to the agent’s performance. The only required domain knowledge is the specification of a set of positive events that can happen in the environment (e.g. picking up items, moving, winning etc.), which is typically easy to implement, especially for games. The method introduced here rewards a reinforcement learning agent by the rarity of experienced events such that rare events have a higher value than frequent events. In our experiments, we completely discard the extrinsic reward and instead motivate the agent intrinsically to explore rare (pre-defined) events. As the agent first experiences certain types of events that are relatively easy to learn (e.g. moving around and picking

up common items) they will slowly become less rewarding, pushing the agent to explore less common events that are potentially harder to achieve. The hope is that by rewarding events for their rarity, the system performs a form of automated curriculum learning, by scaling the reward values based on how obtainable each event is.

The goal of this approach is to learn through a process of *curiosity* rather than optimizing toward a difficult pre-defined goal. We apply our method, called *Rarity of Events* (RoE), to learn agent behaviors from raw pixels in the VizDoom framework (Kempka et al., 2016). While our approach could be applied to any reward-based learning method and possibly also fitness-based evolutionary methods, we train deep convolutional networks through the actor-critic algorithm A2C Mnih et al. (2016). In the future, RoE could offer a new way to learn versatile behaviors in increasingly complex environments such as StarCraft Vinyals et al. (2017).

This chapter does not focus directly on adaptation but rather the issue of enabling reinforcement learning to learn in complex games with sparse rewards. However, as a side effect, our results suggest that RoE can be a useful method to avoid overfitted behaviors that are unable to generalize to changes in the environment.

7.1 Rarity of Events

This section describes *Rarity of Events* (RoE) and its integration with A2C in VizDoom.

The reward function in RoE adapts throughout training to the policy’s ability to explore the environment. By rewarding events based on how often they occur during training, the agent is intrinsically motivated toward exploring new parts of the environment rather than aiming for a single goal that might be difficult to obtain directly. In effect, the approach performs a form of curriculum learning since events are rewarded based on the agent’s current ability to obtain them. As the agent learns, it becomes less interested in events that are frequent and *curious* about newly discovered events.

Our method requires a set of pre-defined events, and the reward $R_t(\epsilon_i)$ for experiencing one of these events ϵ_i at time t is determined by its temporal rarity $\frac{1}{\mu_t(\epsilon_i)}$, where $\mu_t(\epsilon_i)$ is the temporal episodic mean occurrence of ϵ_i at time t , i.e. how often ϵ_i occurs per episode

at the moment. The mean occurrences of events are clipped to be above a lower threshold τ (we used 0.01 such that the maximum reward for any event is 100). For a vector of event occurrences \mathbf{x} , such that x_i is the number of times ϵ_i occurred in a game step, the reward is the sum of all event rewards:

$$R_t(\mathbf{x}) = \sum_{i=1}^{|x|} x_i \frac{1}{\max(\mu_t(\epsilon_i), \tau)}. \quad (7.1.1)$$

The rarity measure $\frac{1}{\mu_t(\epsilon_i)}$ is not arbitrary but is designed such that all events have equal importance. If any event ϵ_i is experienced n times during an episode, and $n = \mu_t(\epsilon_i)$ (which is the expected amount), then the accumulated reward for ϵ_i is 1 regardless of the rarity. This means that in theory all events have equal importance. In practice, the policy might learn that some events have a negative or positive influence on the occurrence of others.

There are arguably many ways to determine the temporal episodic mean occurrence $\mu_t(\epsilon_i)$; here we employ a simple approach that nevertheless achieves the desired outcome. Whenever an episode during training reaches a terminal state, a vector ϵ containing the occurrence of events in this episode is added to a buffer of size N . The size of the buffer determines the adaptability of the reward function. If N is small, the agent quickly becomes *bored* of new events as it easily forgets their rarity in the past. If N is large, the agent will stay *curious* for a longer period of time. The temporal episodic mean occurrence $\mu_t(\epsilon)$ is then determined as the mean of all records in the buffer, i.e. the episodic mean of the last N episodes.

7.2 Experiments

7.2.1 Policy

The presented reward shaping approach can be applied to most (if not all) reinforcement learning methods that learn from a reward signal. It could potentially also be applied to evolutionary approaches such as Evolution Strategies by defining fitness as the sum of

rewards in an episode. A standard policy network is employed that has three convolutional layers followed by a fully connected layer of 512 units, and a policy and value output. We use filter sizes of [32, 64, 32] with strides [4, 2, 1], ReLU activations for hidden layers, and softmax for the policy output.

The input is a single frame of 160×120 pixels in grayscale, cropped by removing 10 pixels on top/bottom and 30 pixels on the sides and then resized to 80×80 . In most of the scenarios, the agent can perform four actions: attack, move forward, turn left, and turn right. In this case, the policy output has $2^4 = 16$ values to allow any combination of the four actions. The event buffer is updated whenever a worker reaches a terminal state. The rewards from VizDoom, which vary between -100 and 100, are normalized to $[0, 1]$. Rewards based on our approach are not normalized and vary between $[0, 100]$ (due to $\tau = 0.01$), while for all events where $\mu_t(\epsilon_i) \geq 1$ the reward will be between 0 and 1 (following Equation 7.1.1 in Section 7.1).

7.2.2 Events in Doom

We track 26 event types in VizDoom by implementing a function that determines which events occur in every state transition (i.e. in each time step). The event types include movement (one unit), shooting (decrease in ammo), picking up an item (one event for each item type; health pack, armor, ammo, and weapons 0–9), killing (one for each weapon type 0–9 as well as one regardless of weapon type). Movement events are triggered when the agent has traveled one unit from the position of the last movement event (or the initial position if the agent has not yet moved).

7.2.3 VizDoom Test Scenarios

This section describes the five VizDoom scenarios used in our experiments. They all have sparse and/or delayed rewards and are therefore a good test domain for our approach. The scenarios are from the original VizDoom repository¹ from Kempka et al. (2016).

For each scenario we also detail the extrinsic reward from the environment, which is used

¹<https://github.com/mwydmuch/VizDoom/tree/master/scenarios>

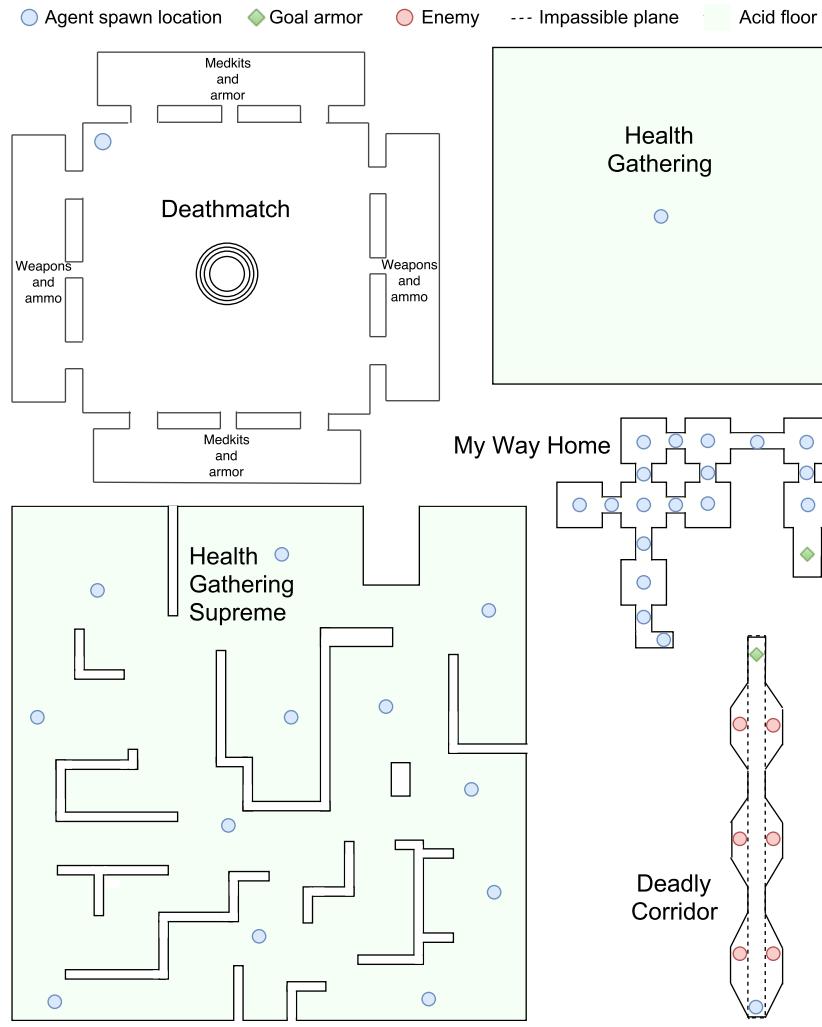


Figure 7.2.1: The five VizDoom scenarios. Scenarios with multiple spawning positions randomly select one of them at the start of an episode. The episode ends when the goal armor, which only appears in *My Way Home* and *Deadly Corridor*, is picked up. The agent periodically loses health when standing on acid floors.

when training models without RoE. Some of these extrinsic rewards were rescaled to be coherent across scenarios. If not stated otherwise, the agent can move forward, turn left, turn right, and shoot. Screenshots from these scenarios are shown in Figure 7.2.2, with top-down views in Figure 7.2.1.

Health Gathering: The goal is to survive as long as possible in a square room with an acid floor that deals damage periodically. Medkits spawn randomly in the room and can help the agent to survive as they heal when picked up. The agent is rewarded 1 for every time step it is alive, and -100 for dying. The maximum episode length is 2,100 time steps. The agent cannot shoot.

Health Gathering Supreme: Same as *Health Gathering* but within a maze.

My Way Home: The goal is to pick up an armor, which gives a reward of 100 and ends the scenario immediately. The agent cannot shoot and is rewarded -0.1 for every time step it is alive. The agent starts an episode at one of the randomly chosen spawn locations with a random rotation.

Deadly Corridor: Similarly to *My Way Home*, the goal is to pick up an armor, which gives a reward of 100 and ends the scenario immediately. The armor is located at the end of a corridor, which is guarded by enemies on both sides. The agent must kill most, if not all of the enemies to reach it, and receives a -100 reward if it dies. The original reward shaping function (the distance to the armor) has been removed to make it harder and to compare RoE with a baselines that does not use any reward shaping. The maximum episode length is 2,100 time steps.



Figure 7.2.2: From top-left to bottom-right: Screenshot from *Deathmatch*, *My Way Home*, *Health Gathering Supreme*, and *Deadly Corridor*. Notice that in some scenarios the agent cannot shoot. The scenario *Health Gathering* is similar to *Health Gathering Supreme* but without walls within the room.

Deathmatch: The agent spawns in a large battle arena with an open area in the middle and four rooms, one in each direction that contain either medkits and armor, or weapons

(chainsaw, super shotgun, chaingun, rocket launcher, and plasma gun) and ammunition for each weapon. The maximum episode length is 4,200 time steps. The agent is rewarded the following amounts when killing an enemy: Zombieman (100), ShotgunGuy (300), MarineChainsawVzd (300), Demon (300), ChaingunGuy (400), HellKnight (1,000). These enemies spawn randomly on the map when the scenario starts.

To test how well the approach can adapt to new scenarios, five variations of *Deathmatch* were also created that only include a certain weapon type. These scenarios are called *Deathmatch Chainsaw*, *Deathmatch Chaingun*, *Deathmatch Shotgun*, *Deathmatch Plasma*, and *Deathmatch Rocket* to denote which weapon that remains on the map. The ammunition for the other weapons was also removed.

7.2.4 Results

We tested A2C with *Rarity of Events* (A2C+RoE) on the five VizDoom scenarios described in Section 7.2.3. The *Deathmatch* variations were not used for training. As a comparison baseline, A2C was also trained using the extrinsic reward from the environment as described in Section 7.2.3. Due to computational constraints we only trained each method once on each scenario.

When training with A2C+RoE, the agent did not have access to the extrinsic reward throughout training but only the intrinsic reward based on the temporal rarity of the pre-defined events. The algorithms ran for 10^7 time steps for each scenario and 7.5×10^7 for the Deathmatch scenario. For both A2C and A2C+RoE we save a copy of the model parameters whenever the mean extrinsic reward across all workers improves. The last copy is considered to be the final model that we use in our tests. The complete configurations for A2C and A2C+RoE are described in Appendix A3 and the code for the experiments and trained models are available on GitHub². Videos of the learned policies are available on YouTube³.

²<https://github.com/njustesen/rarity-of-events>

³<https://youtu.be/YG-lf732a0U>

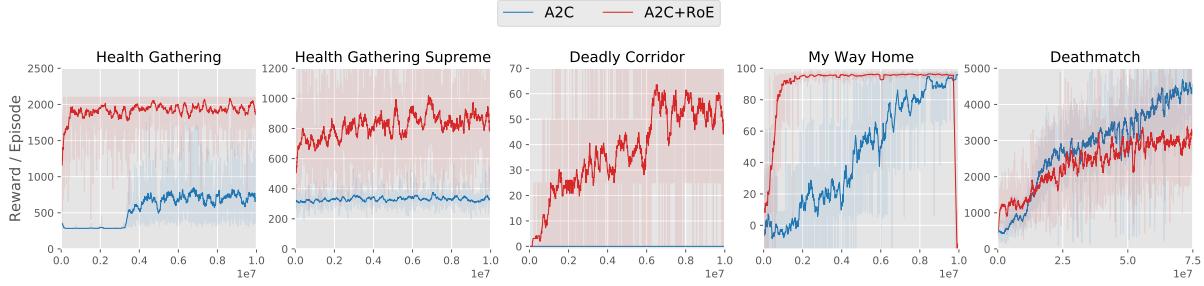


Figure 7.2.3: The reward per episode of A2C and A2C+RoE during training in five VizDoom scenarios (smoothed). A2C is trained from the environment’s extrinsic reward while A2C+RoE uses our proposed method without access to the reward. The drop in performance seen in the *My Way Home* scenario is discussed in-depth in Section 7.2.4.1.

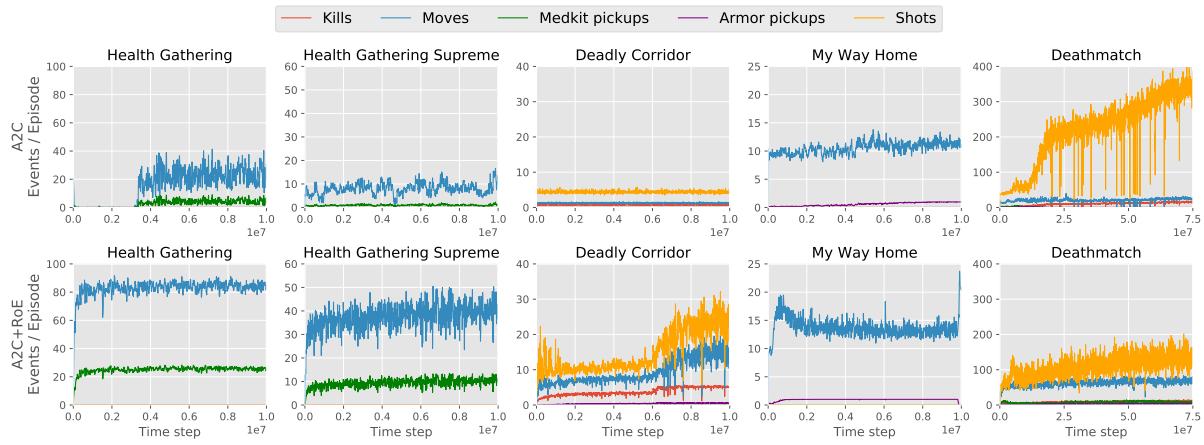


Figure 7.2.4: Episodic mean occurrence during training for a subset of the event types in the five VizDoom scenarios. Notice the last spike in the *My Way Home* scenario with A2C+RoE, in which the policy ignores the final goal (armor pickup) to prioritize continuous movement around the maze.

7.2.4.1 Learned Policies

The A2C baseline did not learn a good policy in *Health Gathering Supreme* and *Deadly Corridor*, and only improved slightly in *Health Gathering* (Figure 7.2.3). A2C learned a weak policy in three out of five scenarios, which demonstrates that they are indeed difficult to master guided by the extrinsic rewards alone. In *My Way Home*, A2C does learn a strong behavior that consistently locates and picks up the armor but only after 8–9 million training steps. In *Deathmatch*, A2C learned a very high-performing behavior that directly walks to the plasma gun (the most powerful weapon in this scenario) and shoots from cover toward the center of the map. The behavior is simple but effective until it runs out of ammunition, after which it attempts to find more ammunition and sometimes fails.

Our approach A2C+RoE learns effective behaviors in all five scenarios. The learned behavior in *Deathmatch* does not exclusively use the powerful plasma gun, which results in a slightly but not significantly worse performance than A2C ($p = 0.125$ using two-tailed t-test). The policy is still effective with over 10 kills per episode. These kills are spread across all weapons that are available, resulting in a behavior that is more varied (and interesting to watch). As we will show in Section 7.2.4.2, the versatile behavior learned by A2C+RoE allows it to adapt to critical changes in *Deathmatch* in contrast to policies trained through A2C.

Scenario	A2C	A2C+RoE	t-test
Health Gathering	399 ± 107	1261 ± 533	$p < 0.0001$
Health Gathering Supr.	305 ± 60	1427 ± 645	$p < 0.0001$
Deadly Corridor	0.00 ± 0.0	40 ± 49	$p < 0.0001$
My Way Home	96.69 ± 0.12	97.89 ± 0.01	$p < 0.0001$
Deathmatch	4611 ± 2595	4062 ± 2442	$p = 0.1250$
Deathmatch Chainsaw	1025 ± 809	3750 ± 3130	$p < 0.0001$
Deathmatch Chaingun	1487 ± 1189	2852 ± 2038	$p < 0.0001$
Deathmatch Shotgun	1375 ± 941	1832 ± 1752	$p = 0.0226$
Deathmatch Plasma	4538 ± 1537	3248 ± 2701	$p < 0.0001$
Deathmatch Rocket	616 ± 583	1463 ± 1449	$p < 0.0001$

Table 7.2.1: Shown are average scores based on evaluating the best policies found for A2C and A2C+RoE 100 times each. The best results are shown in bold. The five last rows show how the policies that were trained on the original *Deathmatch* scenario generalize to five variations where only one weapon type is available. Standard deviations are shown for each experiment and two-tailed p-values from unpaired t-tests.

The episodic mean occurrence of events (Figure 7.2.4) allows us to analyze how the policies change over time. In *Health Gathering* and *Health Gathering Supreme*, A2C+RoE quickly learns to move ~ 80 and ~ 30 units per episode, respectively. This behavior might explain why the agent also quickly learns to pick up medkits. A2C, on the other hand, learns the relationship between movement, medkits, and survival at a much slower pace, at least in the *Health Gathering* scenario. In *Deadly Corridor* A2C+RoE discovers an interesting behavior. After the agent learns to kill all six enemies (the red line) and to pick up armor (purple line), it still manages to increase the movement and the shooting events; the agent learned to walk back to its initial position while shooting and then afterwards to return to pick up the armor. This result is not unexpected as the agent is intrinsically motivated to experience as many events as possible during an episode.

In *My Way Home*, after the A2C+RoE policy has learned to routinely pick up the armor,

it shifts into a different behavior toward the end of training. The agent learned to avoid the armor to instead continuously move around in the maze. We suspect that the policy would shift back to the previous behavior if training was continued, as the movement reward is now decreasing and the armor reward is increasing. Since our rarity measure is temporal, loops between these two behaviors could emerge as well. As policies with the highest extrinsic reward are saved during training, these sudden changes do not affect the final policy. In fact, one might argue that this is a useful feature of RoE: a network that has converged to some optimum can escape it to find other interesting behaviors.

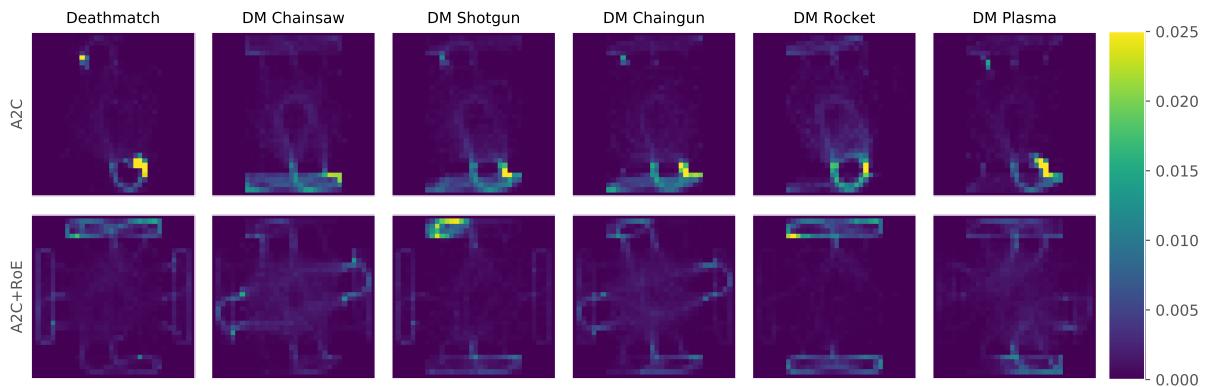


Figure 7.2.5: Heat maps showing the proportional time spent at each location on the map in the *Deathmatch* scenario and its five variations. The values are based on evaluating the two trained policies 100 times each and clipped at 0.025. The heat maps show that the A2C-policy prefers to stay near the plasma gun, even in the map variations where it is not present, while the A2C+RoE-policy has learned distinct behaviors for each weapon type. The results in Table 7.2.1 shows that the A2C+RoE-policy is able to reach high scores in these variations event though it was never trained on them.

7.2.4.2 Ability to Generalize

A2C+RoE motivates the agent intrinsically to learn a balanced policy that strives to experience a good mix of events. Reinforcement learning algorithms that exclude pre-training or proper reward shaping, including our A2C baseline, can easily converge into local optima with very *narrow* behaviors. In this context, *narrow* refers to behaviors that act in a very particular way, only utilizing a small subset of the features in the environment. This handicap prevents the learned policies from adapting to critical changes in the environment as they only know one way of behaving.

To test for such adaptivity, the learned policies are evaluated on five *Deathmatch* variations in which critical weapons and ammunition packs have been removed. Note that the policies were not directly trained on these variations. The results in Table 7.2.1 show that A2C+RoE learned a policy that significantly outperforms A2C ($p < 0.0001$ using two-tailed t-test) in four out of five *Deathmatch* variations. A2C+RoE learned a policy that is more versatile, capable of using all the weapons in the map, which is the reason it can easily adapt. Figure 7.2.5 shows heat maps (i.e. the proportional time spent at each map location) during the evaluations of the two policies on *Deathmatch* and its variations. The A2C+RoE policy expresses different strategies depending on the weapon available on the map, while the A2C policy mostly circles around the plasma gun location, regardless of it actually being there. However, if the plasma gun is present, A2C alone does execute a fairly effective strategy, shooting toward enemies in the middle of the map.

The heat maps show that the A2C policy has learned to stay at only one location on the map from which it can pick up the powerful plasma gun and thereafter shoot efficiently toward enemies in the middle of the map (see the video of the learned policies). In the *Deathmatch* variations, in which the map only contains two weapons of the same type, the A2C-policy fails to adapt to use the other weapons and instead walks around the area where the plasma gun would have been located.

The A2C+RoE policy has learned to explore a larger part of the maps in a more uniform way (Figure 7.2.5, bottom). In the different *Deathmatch* variations, a clear change in behavior can be observed when only a certain type of weapon is available. For example, in the *DM Rocket* scenario, the agent lures enemies into the map’s top and bottom room while efficiently using the rocket’s splash damage.

7.3 Discussion

While the presented approach worked well in VizDoom it will be important to test its generality in other domains in the future. RoE is designed to work well in challenging environments that have a plethora of known events and sparse and/or delayed rewards. Video games are thus a very suitable domain and we plan to test RoE in Montezuma’s Revenge and StarCraft in future work. For domains in which reward shaping is not

necessary, i.e. the extrinsic reward smoothly leads to an optimal behavior, our approach might add less value. We imagine that RoE should also work well in domains with deceptive reward structures, just as novelty search outperformed traditional evolutionary algorithms in mazes with dead ends (Lehman and Stanley, 2011a) or deceptive meta-learning tasks (Risi et al., 2010). Novelty search and RoE have the ability to learn interesting behaviors without the need for a goal. In the future, our approach could also be extended to reward the agent for both the rarity of events as well as the environment’s original objective, inspired by Quality-Diversity algorithms.

The specification of adequate events is intimately tied to the success of our approach; events that lead to direct negative performance should be avoided. For example, if the extrinsic reward is negative when the agent wastes ammunition, it should not be intrinsically rewarded for shooting event. A benefit of the presented method is that events that contribute to the occurrence of other events (e.g. such as movement leads to medkit pickups), can lead to a system that performs automated curriculum learning. However, it is not guaranteed that this effect will occur, and it might require a bit of trial and error during the specification of events. Some events can also be contradicting, such as killing with the chainsaw and killing with the plasma gun, as the agent cannot do both at the same time. Our approach is designed to learn a policy that can balance their occurrences which results in a more versatile behavior. Important future work will test how RoE scales to hundreds or even thousands of events. A promising testbed for such experimentation is StarCraft, for which events can easily be defined as the production of each unit and building type, as well as killing different opposing unit types. We believe that reinforcement learning methods that are guided by intrinsic motivation are key to solving these challenging environments.

The A2C baseline reached the best performance in the original *Deathmatch*. However, it can be argued whether it learned to actually play Doom, or just learned to follow a fixed sequence of actions that lead to the same behavior every time. While it can be useful to find a niche behavior with high performance, learning a rich and versatile behavior has particular relevance for video games. Here, behaviors that explore the game’s features could potentially help for automatic game testing and also lead to more human-like behaviors for NPCs.

Regarding our implementation of the RoE approach, future work will also explore other variations in determining the episodic mean occurrence of events, such as discounting the mean occurrences over time. With this modification, event occurrences older than N episodes (the event buffer only holds N event occurrences) would still effect the intrinsic reward.

It is important to note that since we save the best model based on the mean extrinsic reward across all worker threads, increasing the number of threads should make the evaluation less noisy by reducing the chances of accidentally overriding the best model with a worse performing one. This hypothesis still needs to be confirmed, but the number of threads was already increased from 4 to 16 in the longer *Deathmatch* scenario to speed up learning.

7.4 Summary

We introduced *Rarity of Events* (RoE), a simple reinforcement learning approach that determines reward based on the temporal rarity of pre-defined events. This approach was able to reach high-performing scores in five challenging VizDoom scenarios with sparse and/or delayed rewards. Compared to a traditional A2C baseline, the results are significantly better in four of the five scenarios. Importantly, the presented approach is able to not only receive a high final reward, but also discovers versatile behavior that can adapt to critical changes in the environment, which is challenging for the baseline A2C approach. In our experiments, the extrinsically motivated baseline either fails to learn anything useful or learns a behavior that is unable to adapt to changes in the environments it has been trained on. In the future, the presented RoE approach could allow more complex scenarios to be solved, for which it is infeasible to learn from extrinsic rewards without manual reward shaping and curriculum learning.

Chapter 8

Procedural Content Generation for Reinforcement Learning

The results in the previous chapter suggest that deep reinforcement learning algorithms easily overfit to their training environment, resulting in policies that do not generalize well to related problems or even different instances of the same problem. Even small game modifications can often lead to dramatically reduced performance, leading to the suspicion that the networks only learn to react to a small subset of the relevant state space, rather than general strategies (Kansky et al., 2017; Zhang et al., 2018). This chapter presents four contributions in an attempt to illuminate and overcome the issues of overfitting in deep reinforcement learning.

First, we show that deep reinforcement learning overfits to a large degree on 2D arcade games when trained on a fixed set of levels. These results are important because similar setups are particularly popular to use as benchmarks in deep reinforcement learning research (e.g. the Arcade Learning Environment (Bellemare et al., 2015)). Our findings suggest that policies trained in such settings merely memorize certain action sequences rather than learning general strategies to solve the game.

Second, we show that it is possible to improve generality by introducing Procedural Content Generation (PCG) (Shaker et al., 2016), more specifically procedurally generated levels, in the training loop. However, we found that this can lead to overfitting on a higher level, to the distribution of generated levels presented during training. Our experiments

investigate both types of overfitting and the effect of several level generators for multiple games.

Third, we introduce a particular form of PCG-based reinforcement learning, which we call *Progressive PCG*, where the difficulty of levels is increased gradually to match the agent’s performance. While similar techniques of increasing difficulty do exist, they have not been combined with a PCG-based approach in which agents are *evaluated on a completely new level every time a new episode begins*. Our approach applies constructive level generation techniques, rather than pure randomization, and we study the effect of several level generation methods.

Fourth, we analyze distributions of procedurally generated levels using dimensionality reduction and clustering to understand their resemblance to human-designed levels and how this impacts generalization.

It is important to note that the primary goal of this chapter is not to achieve strong results on human levels, but rather to gain a deeper understanding of overfitting and generalization in deep reinforcement learning, which is an important and neglected area in AI research. We believe the work presented in this chapter makes a valuable contribution in this regard, suggesting that a PCG-based approach could be an effective tool to study these questions from a fresh perspective.

8.1 Related Work

Randomization of objects in simulated environments has shown to improve generality for robotic grasping to such a degree that the robotic arm could generalize to realistic settings as well (Tobin et al., 2017). Low-fidelity texture randomization during training in a simulated environment has allowed for autonomous indoor flight in the real world (Sadeghi and Levine, 2016). Random level generation has been applied to video games to enable generalization of reinforcement learning agents (Beattie et al., 2016; Graves et al., 2016; Groshev et al., 2017; Klimov, 2016). Several reinforcement learning approaches exist that manipulate the reward function instead of the structure of the environment to ease learning and ultimately improve generality, such as Hindsight Experience Replay

(Andrychowicz et al., 2017).

The idea of training agents on a set of progressively harder tasks is an old one and has been rediscovered several times within the wider machine learning context. Within evolutionary computation, this practice is known as incremental evolution (Gomez and Miikkulainen, 1997; Togelius and Lucas, 2006). For example, it has been shown that while evolving neural networks to drive a simulated car around a particular race track works well, the resulting network has learned only to drive that particular track; but by gradually including more difficult levels in the fitness evaluation, a network can be evolved to drive many tracks well, even hard tracks that could not be learned from scratch (Togelius and Lucas, 2006). Essentially the same idea has later been independently invented as curriculum learning (Bengio et al., 2009). Similar ideas have been formulated within a coevolutionary framework as well (Brant and Stanley, 2017).

Several machine learning algorithms also gradually scale the difficulty of the problem. Automated curriculum learning includes intelligent sampling of training samples to optimize the learning progress (Graves et al., 2017). Intelligent task selection through asymmetric self-play with two agents can be used for unsupervised pre-training (Sukhbaatar et al., 2017). The POWERPLAY algorithm continually searches for new tasks and new problem solvers concurrently (Schmidhuber, 2013) and in Teacher-Student Curriculum Learning (Matiisen et al., 2017) the teacher tries to select sub-tasks for which the slope of the learning curve of the student is highest. Reverse curriculum generation automatically generates a curriculum of start states, further and further away from the goal, that adapts to the agent’s performance (Florensa et al., 2017).

A protocol for training reinforcement learning algorithms and evaluate generalization and overfitting, by having large training and test sets, was proposed in (Zhang et al., 2018). Their experiments show that training on thousands of levels in a simple video game enables the agent to generalize to unseen levels. Our (contemporaneous) work here differs by implementing an adaptive difficulty progression along with near endless content generation for several complex video games.

8.2 Parameterized Level Generator

Constructive level generators were designed for four hard games in GVG-AI: Boulderdash, Frogs, Solarfox and Zelda. These were picked because tree-search algorithms do not perform well in these, and we thus consider them hard (Bontrager et al., 2016). Constructive level generators are popular in game development because they are relatively fast to develop and easy to debug (Shaker et al., 2016). They incorporate game knowledge to make sure the output level is directly playable without additional testing. Alternatively, answer set programming could allow for automatic generation of levels following a set of restrictions (Neufeld et al., 2015). Generative adversarial networks could perhaps also be used to expand a small training set of levels (Volz et al., 2018). Our level generators are designed after analyzing the core components in the human-designed levels for each game and include a controllable difficulty parameter.

Boulderdash Level Generator: This game is a GVG-AI port of “Boulder Dash” (First Star Software, 1984). Here the player tries to collect at least ten gems and then exit through the door while avoiding falling boulders and attacking enemies. The level generation in Boulderdash works as follows: (1) Generate the layout of the map using Cellular Automata (Johnson et al., 2010). (2) Add the player to the map at a random location. (3) Add a door at a random location. (4) Add at least ten gems to the map at random locations. (5) Add enemies to the map at random locations in a similar manner to the third step.

Frogs Level Generator: Frogs is a GVG-AI port of “Frogger” (Konami, 1981). In Frogs, the player tries to move upwards towards the goal without drowning in the water or getting run over by cars. The level generation in Frogs follow these steps: (1) Add the player at the lowest empty row in the level. (2) Add the goal at the highest row in the level. (3) Assign the intermediate rows either as roads, water, or forest. (4) Add cars to rows with a road and wood logs rows with water.

Solarfox Level Generator: Solarfox is a GVG-AI port of “Solar Fox” (Midway Games, 1981). In Solarfox, the player is continuously moving in one of four directions (North, South, East, and West). The goal is to collect all the gems while avoiding the borders of the level as well as bullets from enemies in the north and the south. The level generation for Solarfox follow these steps: (1) Add the player in the middle of the map. (2) Add

some gems either in the upper half, left half, or upper left quarter. (3) Replicate the same pattern of gems on the remaining parts of the map.

Zelda Level Generator: Zelda is a GVG-AI port of the dungeon system in “The Legend of Zelda” (Nintendo, 1986). In Zelda, the goal is to grab a key and exit through a door without getting killed by enemies. The player can use their sword to kill enemies for higher scores. The level generation in Zelda works as follows: (1) Generate the map layout as a maze using Prim’s Algorithm (Buck, 2015). (2) Remove some of the solid walls in the maze at random locations. (3) Add the player to a random empty tile. (4) Add the key and exit door at random locations far from the player. (5) Add enemies in the maze at random locations far away from the player.

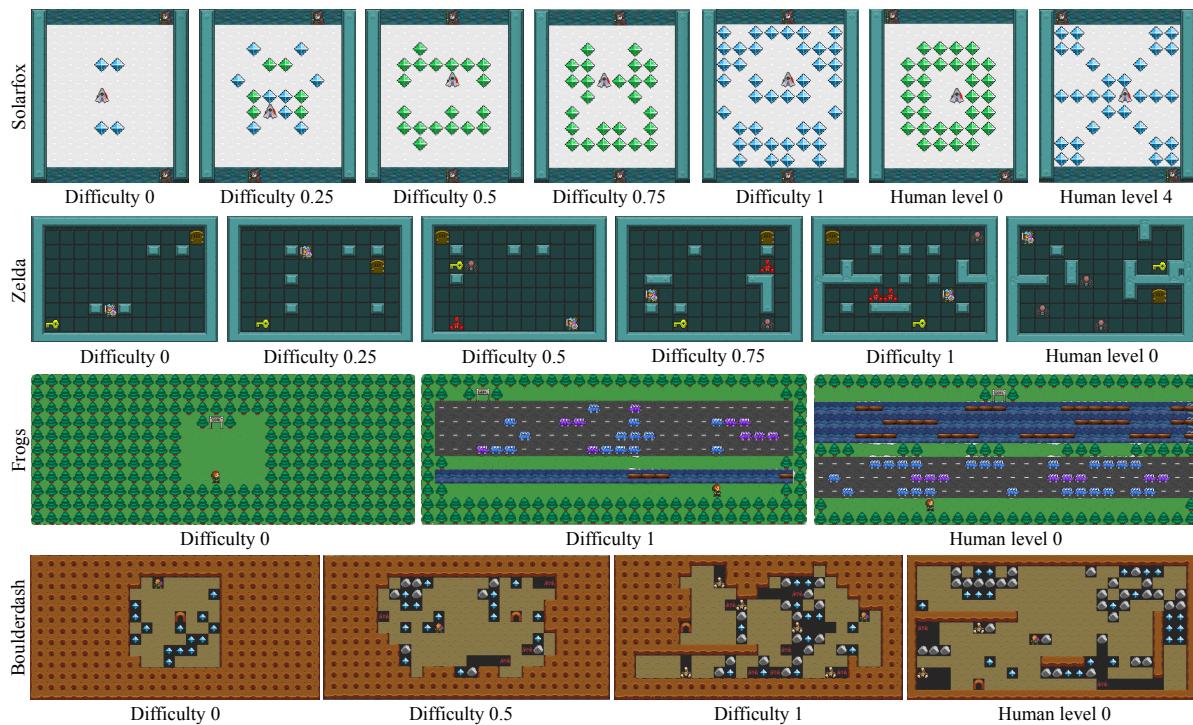


Figure 8.2.1: Procedurally generated levels for Solarfox, Zelda, Frogs, and Boulderdash with various difficulties between 0 and 1. For each game, human-designed levels are shown as well.

The difficulty of the levels created by the generator can be controlled with a *difficulty parameter* that is in the interval $[0, 1]$. Figure 8.2.1 shows the effect of the difficulty parameter in the four games. Increasing the difficulty has three effects: First, the area in the level where the player can move through (active level size) increases, except in Zelda and Solarfox where the level size is fixed. Second, the number of objects that can kill the player and/or the number of objects that the player can collect is increased. Third,

the layout of the level gets more complex to navigate. The space of possible levels for each game, using our generators, is around 10^8 at low difficulty to 10^{24} at high difficulties. Difficult levels have more possible configurations as they typically have more elements.

8.3 Procedural Level Generation for Deep Reinforcement Learning

In a supervised learning setting, generality is obtained by training a model on a large dataset, typically with thousands of examples. Similarly, our hypothesis is that reinforcement learning algorithms should achieve generality if many variations of the environment are used during training, rather than just one.

We present a novel reinforcement learning framework wherein a new level is generated *whenever a new episode begins*, which allows us to algorithmically design the new level to match the agent’s current performance. This framework also enables the use of search-based PCG techniques, that e.g. learn from existing level distributions (Volz et al., 2018), which could in the future reduce the dependency on domain knowledge. However, only constructive PCG is explored here.

When the learning algorithm is presented with new levels continuously during training, it must learn general strategies to improve. Learning a policy this way is more difficult than learning one for just a single level and it may be infeasible if the game rules and/or generated levels have sparse rewards. To ease the learning, we also introduce **Progressive PCG** (PPCG), an approach where the difficulty of the generated levels is controlled by the learning algorithm itself. In this way, the level generator will initially create easy levels and progressively increase the difficulty as the agent learns. In the implementation of PPCG, levels are initially created with the lowest difficulty of 0. If the agent wins an episode, the difficulty is incremented such that future levels during training become harder. The difficulty is increased by α for a win and decreased by the same amount for a loss. In our experiments, we use $\alpha = 0.01$. For distributed learning algorithms, the difficulty setting is shared across all processes such that the outcome of all episodes influences the difficulty of future training levels. We compare PPCG to a simpler method,

also using procedurally generated levels, but with a constant difficulty level. We refer to this approach as **PCG X**, where X refers to the fixed difficulty setting.

8.4 Experiments

To evaluate the presented approach, we employ the reinforcement learning algorithm *Advantage Actor-Critic* (A2C) (Mnih et al., 2016), specifically the implementation from the Open AI Baselines, together with the GVG-AI Gym framework. The neural network has the same architecture as in (Mnih et al., 2016) with three convolutional layers and a single fully-connected layer. A2C is configured to use 12 parallel workers, a step size of $t_{max} = 5$, no frame skipping following (Rodriguez Torrado et al., 2018), and a constant learning rate of 0.007 with the RMS optimizer (Ruder, 2016). The code for our experiments is available online¹.

We compare four different training approaches. **Lv X**: Training level is one of the five human-designed levels. **Lv 0-3**: Several human-designed levels (level 0, 1, 2, and 3) are sampled randomly during training. **PCG X**: Procedurally generated training levels with a constant difficulty X . **Progressive PCG (PPCG)**: Procedurally generated training levels where the difficulty is adjusted to fit the performance of the agent.

Each training setting was repeated four times and tested on two sets of 30 pre-generated levels with either difficulty 0.5 and 1 as well as the five human-designed levels. The training plots on Figure 8.4.1 and the test results in Table 8.4.1 are averaged across the four trained models where each model was tested 30 times on each test setup (thus a total of 120 test episodes per test set for each training setup). All four training approaches were tested on Zelda. Only PCG 1 and PPCG were tested on Solarfox, Frogs, and Boulderdash due to computational constraints. The trained agents are also compared to an agent taking uniformly random actions and the maximum possible score for each test set is shown as well.

¹https://github.com/njustesen/a2c_gvgai

8.4.1 Training on a few Human-Designed Levels

Zelda							
Training	PCG 0.5	PCG 1	Lv 0	Lv 1	Lv 2	Lv 3	Lv 4
Max.	4.40	6.87	8.00	8.00	8.00	10.00	8.00
Random	0.38	0.22	0.26	0.17	-0.11	-0.07	0.18
60M steps:							
Level 0	0.28	0.51	6.97	-0.45	-0.53	0.07	-0.58
Level 4	0.56	0.07	-0.51	0.99	0.04	-0.35	5.93
Level 0-3	1.98	2.37	6.95	7.17	7.20	8.17	1.91
PCG 0.5	3.45	4.00	2.21	2.28	0.92	2.27	0.15
PCG 1	0.27	3.56	2.40	1.37	1.49	2.88	-0.62
PPCG	3.44	4.28	2.67	3.35	2.43	1.89	0.96
100M steps:							
PCG 1	3.05	4.38	2.49	1.54	1.18	2.04	-0.29
PPCG	3.82	4.51	2.71	3.74	2.84	1.90	0.88
Solarfox*							
Max.	30.83	51.83	32.00	32.00	34.00	70.00	62.00
Random	-3.68	-4.55	-5.49	-4.80 ,	-5.41	2.03	1.13
40M steps:							
PCG 1	20.70	32.43	22.00	21.83	26.00	43.96	28.16
PPCG	16.08	21.40	16.87	10.26	12.02	27.37	20.00
Frogs							
Max.	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Random	0.01	0.00	0.00	0.00	0.00	0.00	0.00
40M steps:							
PCG 1	0.01	0.00	0.00	0.00	0.00	0.00	0.00
PPCG	0.81	0.57	0.00	0.00	0.00	0.00	0.00
Boulderdash							
Max.	31.50	29.80	48.00	52.00	58.00	48.00	44.00
Random	6.29	3.71	0.85	2.58	3.5	0.65	2.66
60M steps:							
PCG 1	14.63	8.32	5.39	10.28	5.85	5.08	8.27
PPCG	11.78	4.86	3.44	0.98	0.68	0.41	3.32

Table 8.4.1: Test results of A2C under different training regimens: a single human-designed level (*Level 0* and *Level 4*), several human-designed levels (*Level 0-3*), procedurally generated levels with a fixed difficulty (*PCG 0.5* and *PCG 1*), and *PPCG* that progressively adapts the difficulty of the levels to match the agent’s performance. *Random* refers to results of an agent taking uniformly random actions and *Max* shows the maximum possible score. Scores are in red if the training level is the same as the test level. The best scores for a game, that is not marked red, are in bold. *Only three repetitions of PPCG and one of PCG 1 were made for Solarfox so far.

Policies trained on just one level in Zelda (*Lv 0* and *Lv 4* in Table 8.4.1) reach high scores on the training level but have poor performance on all test levels (human-designed and procedurally generated). It is clear that these are prone to memorization and cannot

adapt well to play new levels. The scores on the training levels are close to the maximum scores achievable while the scores on the test levels are often lower than the random policy, a clear indication of overfitting in reinforcement learning. Policies trained on four human-designed levels in Zelda also achieve high scores on all four training levels. The testing scores are marginally higher than when trained on a single level, on both the human-designed level 4 and the PCG levels.

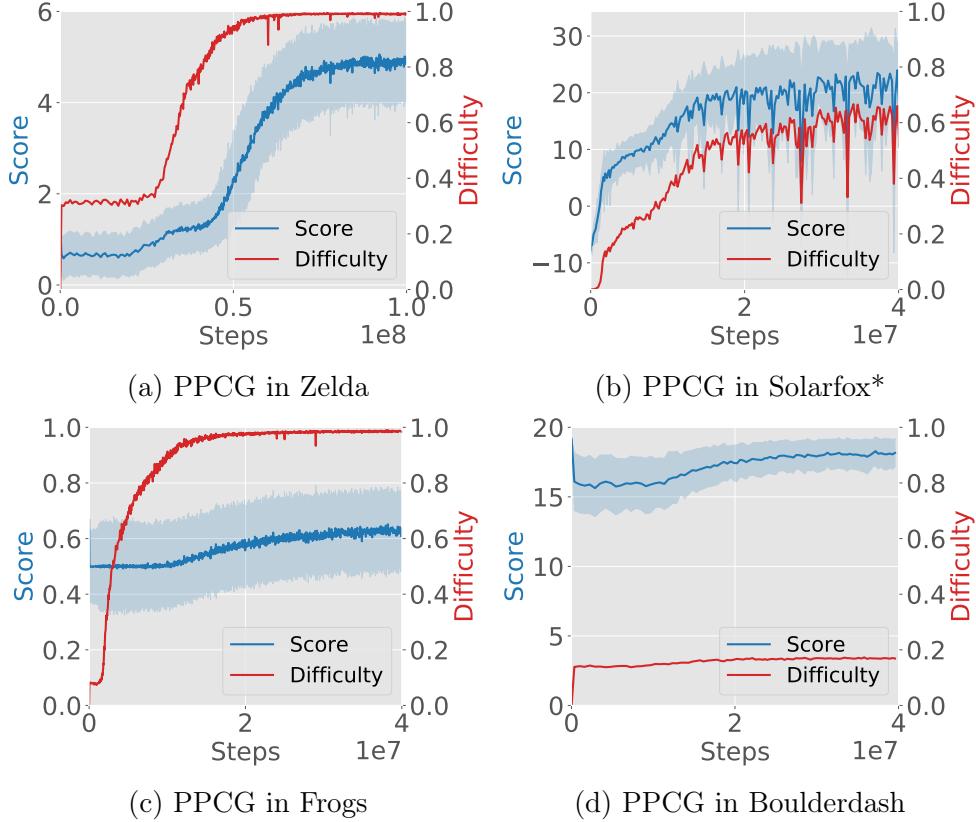


Figure 8.4.1: Smoothed mean scores and level difficulties during training across five repetitions of Progressive PCG in Zelda, Solarfox, Frogs, and Boulderdash. One standard deviation is shown in opaque. *Only three repetitions of PPCG and one of PCG 1 for Solarfox.

8.4.2 Training on Procedurally Generated Levels

Agents trained on procedurally generated levels with a fixed difficulty learned a general behavior within the distribution of procedurally generated levels, with mediocre scores in Zelda, Solarfox, and Boulderdash, while no progress was observed in Frogs. These results match similar observations by Rodriguez Torrado et al. (2018), in which DQN and A2C fail to learn anything on just one level in Frogs after 1 million training steps. While PCG

1, here with 40 million steps, also fails to learn Frogs, PPCG achieves a score of 0.57 (57% win rate) in the test set of procedurally generated levels with difficulty 1 (comparable to human levels in difficulty - see Figure 8.2.1). In Zelda, PCG 1 was able to achieve strong scores while PPCG is slightly better. Interestingly, for the two cases where PPCG is able to reach difficulty 1 during training (Frogs and Zelda), it outperforms PCG 0.5 on PCG 1. As PPCG never reaches the most difficult levels during training in Boulderdash and Solarfox, this is to be expected. In Boulderdash, the agents trained with PCG 1 reach decent scores (8.34 on average) on levels with difficulty 1. PPCG reached high scores during training but failed to win as the difficulty reached 0.2 and thus trained only on easy levels.

8.4.3 Generalization on Human-designed Levels

The results demonstrate that introducing procedurally generated levels allows the trained behaviors to generalize to unseen levels within the training distribution. It is, however, interesting whether they also generalize to the five human-designed levels in GVG-AI.

In Zelda, PCG and PPCG are decent in the human-designed levels while best in the procedurally generated levels. In Frogs, PCG and PPCG are unable to win in the human-designed levels indicating a clear discrepancy between the two level distributions. In Boulderdash, PCG 1 achieved on average 5.08–10.28 points (out of 20) in the human-designed levels compared to 8.32–14.63 in the procedurally generated levels. PPCG performs worse in this game since it never reached a difficulty level similar to the human-designed levels. Similarly, in Solarfox, PCG 1 achieved on average a higher score than PPCG on the five human-designed levels. PCG 1, however, shows remarkable generalization in Solarfox with similar scores in human-designed and procedurally generated levels.

8.4.4 Qualitative Analysis of Agent Replays

In Zelda, PPCG has learned to reliably strike down and avoid enemies but only sometimes collects the key and exits through the door. Whether this is due to the difficulties of navigating in tricky mazes or a lack of motivation towards the key and door is currently

unclear. In Solarfox, PCG 1 has learned to effectively pick up the diamonds and avoid fireballs, occasionally getting hit while trying to avoid them. This behavior is remarkably human-like. Sometimes the agent wins in the human-designed levels, which is quite impressive. PPCG jiggles a lot around the starting location to collect nearby diamonds, most likely because the easy procedurally generated levels have diamonds near the starting location, and it never reached the hard levels during training. In Frogs, PPCG always moves towards the goal while it sometimes dies when crossing the water with only a few logs being available. We suspect that navigation in this game is learned more easily than in other games as the goal in Frogs is always at the top of the screen. In Boulderdash, PCG 1 learned to fight and pick up nearby diamonds, also under boulders, while it does not seem to be capable of long-term planning. It often dies fighting enemies or moving boulders and thus dies rather quickly in most level. Often dying from boulders and enemies can explain why PPCG never reached a difficulty higher than 0.2; it simply gets killed early when these entities are introduced in the levels.

8.5 Exploring the Distribution of Generated Levels

We do not expect agents to play well on levels that are dissimilar from their training distribution. To investigate the distribution of the procedurally generated levels, and how the structure of these correlate with human-designed levels, we generated 1000 levels with difficulty 1 for each game. The high-dimensional structure of levels was compressed to two dimensions using principal component analysis (PCA), and afterward clustered with the density-based spatial clustering of applications with noise (DBSCAN). The transformed space of levels is visualized in Figure 8.5.1. For PCA to work on GVG-AI levels, they have been transformed into a binary 3D array of shape (tile_type, height, width) and then reshaped into a 1D array. The human-designed levels were included in both the transformation and clustering processes.

The generated levels for Solarfox are clustered in three wide groups: (1) levels with only green diamonds, (2) levels with both green and blue diamonds, and (3) levels with only

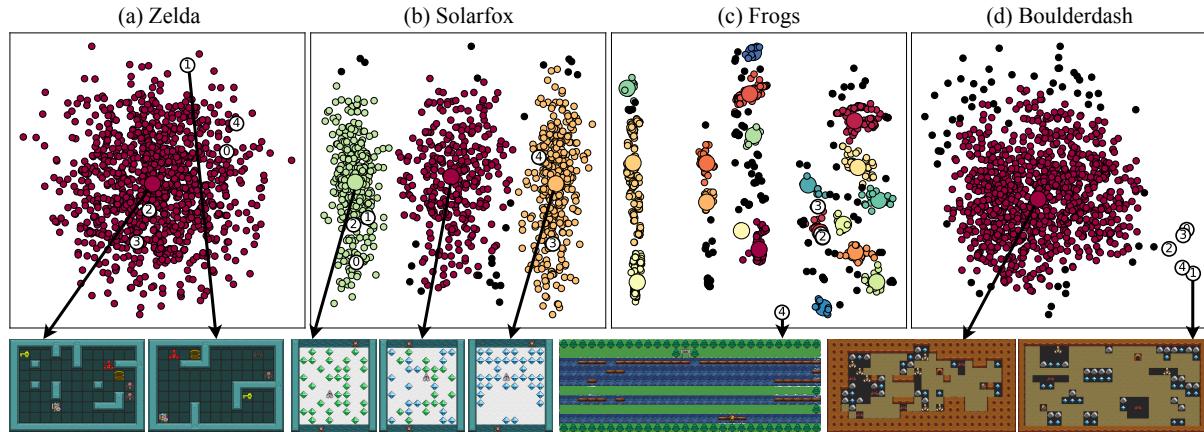


Figure 8.5.1: Visualization of the level distributions and how they correlate to human-designed levels (white circles). Levels were reduced to two dimensions using PCA and clustered using DBSCAN ($\epsilon = 0.5$ and a minimum of 10 samples per cluster). Outliers are black and centroids are larger.

blue diamonds. None of the human-designed levels use both types of diamonds and thus only belong to two of the clusters. For Zelda, only one cluster is discovered without outliers. The generated levels in Frogs have been clustered into 19 groups. This is due to the high structural effect of roads and rivers that goes across the level. It is noticeable how level 4 is the most distant outlier. This is because level 4 has a river on the starting row which is a level variation not captured by the level generator for Frogs. Level 0–3 are near the same small cluster while the generated levels are spread across many isolated clusters. It is not exactly clear why PCG 1 and PPCG fail to play on all the human-designed Frogs levels while the level distribution is remarkably different from the other games. In Boulderdash, similarly to Zelda, only one cluster emerges, but here, all human-designed levels are distant outliers. This effect is most likely a result of the fixed amount of open space in the human-designed levels with padding of only one tile while the generated levels are more varied and cave-like.

8.6 Discussion

The results of our experiments affirm the original concern with the way reinforcement learning research is often evaluated and reported. When it is reported that an algorithm has learned a policy that can play a game, it may simply mean that this policy has found optimal actions for a very small subspace of the possible observations the game offers.

This boils down to the network mapping observations in this subspace to actions without learning general concepts of the game. Table 8.4.1 shows this with the huge disparity between the performance on the training levels compared to the test levels. If the goal of the agent is to learn how to play a game, then this work shows that it must be evaluated in several variations of the game.

Incorporating procedurally generated levels in the training loop also presents a variety of new and interesting challenges. One such challenge is how to scale the difficulty of the levels to smoothen the learning curve in PPCG. In Frogs, it was very effective to apply padding to easy levels, creating smaller levels in the beginning, while it was not sufficient for Boulderdash. Another challenge is how to ensure that the distribution of procedurally generated levels matches another distribution, in this case human-designed levels. We have provided a tool using dimensionality reduction and clustering, which can be used to improve the design of constructive level generators or perhaps guide search-based level generators in future work. While the results vary across the four games, analyzing when the PCG-based approach works and when it fails gave valuable insights into the generalization abilities of these reinforcement learning algorithms. We believe that search-based PCG is an interesting area for future work that could ultimately lead to reinforcement learning agents with more general policies. We believe that this study is also relevant for robotics; learning to generalize from simulation to real-world scenarios where pure randomization of the environment is insufficient.

We have demonstrated the need to focus on generality when applying reinforcement learning to games. However, obtaining generality across multiple levels in a game can be harder on some games than in others. We suggest two new dimensions in game complexity: (1) the number of different initial states and (2) the diversity of initial states. If a game has many initial states, and these are structurally different, the game has a high complexity. Chess and go have two initial states (starting as black or white) that are almost identical, and thus have a low complexity along these two dimensions. The GVG-AI games used in this chapter, when played in an unknown level, are all very complex along these dimensions, as there are millions of different initial states. Interestingly, StarCraft is usually played on a pool of around six to eight different maps, each with a few different starting positions. StarCraft thus have a relatively low number of initial states, while many of them are

widely different.

8.7 Summary

We explored how policies learned with deep reinforcement learning generalize to levels that were not used during training. The results demonstrate that agents trained on just one or a handful of levels often fail to generalize to new levels. We presented a new approach that incorporates a procedural level generator into the reinforcement learning framework, in which a new level is generated for each episode. A variant of this approach, *Progressive PCG* (PPCG), shows that by dynamically adapting the difficulty of the generated levels during training, the agent gradually learns to solve more complex levels than when trained directly on the most difficult levels. This technique was able to achieve a win rate of 57% in difficult Frogs levels, compared to 0% for the non-progressive approach. Additionally, in Zelda, PPCG reached superior performance across procedurally generated levels and human-designed levels. In Solarfox and Boulderdash, the level difficulty of PPCG never reached the maximum during training and here training on procedurally generated levels with a fixed difficulty setting resulted in the highest performance. The results presented in this chapter also highlight the important challenge of ensuring that the training distribution resembles the test distribution. We have provided a tool that can assist with the second challenge, using dimensionality reduction and clustering to visualize the difference between two distributions of video game levels.

Chapter 9

When Are We Done with Games?

In this chapter, we take a close look at recent successes AI in complex games and make an attempt to evaluate the fairness of the employed human vs. AI competitions. With DeepMind's 'AlphaGo' beating the world's best go player Lee Sedol, 'OpenAI Five' (OpenAI, 2017) beating a team of professionals in a restricted game of Dota 2, and DeepMind's 'AlphaStar' (Vinyals et al., 2019) beating the professional StarCraft II player Grzegorz 'MaNa' Komincz, it may seem that AI has now achieved all of these long-standing goals that were set forth by the research community, which are also the foundation of the problems explored in this dissertation. So where does this leave us? Are we, as AI researchers, done with games? It should be noted here, that the results of AlphaStar and OpenAI Five were announced after most of the work in this dissertation were submitted and the technical details and results of both approaches are not yet published.

This chapter provides a discussion of designing and evaluating fairness in human vs. AI game competitions. Ultimately, we argue that a claim of superiority of AIs over humans is unfounded, until AIs compete with and beat humans in competitions that are structurally the same as common human vs. human competitions. These competitions are, after all, designed to erase particular elements of unfairness within the game, the players, or their environments. We take a black-box approach that ignores some dimensions of fairness such as learning speed and prior knowledge, focusing only on perceptual and motoric fairness. Additionally, we introduce the notions of game *extrinsic* factors, such as the competition format and rules, and game *intrinsic* factors, such as different mechanical systems and configurations within one game. We apply these terms to critically review

the aforementioned AI achievements and observe that game extrinsic factors are rarely discussed in this context, and that game intrinsic factors are significantly limited in AI vs. human competitions in digital games. These highlight critical limitations of current game-playing methods that we have addressed in this dissertation.

Claimed AI achievements in games were also reviewed by Canaan et al. (2019a), focusing on how researchers and the media have portrayed the achieved results. Additionally, they proposed six dimensions of fairness in human vs. AI game competitions: 1) Perceptual: do the two competitors use the same input space? 2) Motoric: do the two competitors use the same output space? 3) Historic: did the two systems spend the same amount of time on training? 4) Knowledge: do the two systems have the same access to declarative knowledge about the game? 5) Compute: do the two systems have the same computational power? 6) Common-sense: do the agents have the same knowledge about other things? Based on their six dimensions of fairness, they concluded that “a completely fair competition can only be achieved against an artificial system that is essentially equivalent to a flesh and blood human”. We will return to their evaluation in a later paragraph.

Our main critique of current evaluation procedures is twofold. AI superiority in games cannot be claimed without carefully treating the game extrinsic and intrinsic factors, such as the competition’s structure and rules, and the game’s configurations. First of all, it is necessary that AIs compete in a game’s extrinsic tournament structures, as already employed in human vs. human competitions. Thus, to formulate a proper competition between humans and AI systems, we argue that we must first study the extrinsic game factors that are in play when humans are competing and then formulate an experimental setup that imitates them, without limiting the game’s intrinsic variables, such as maps, races, heroes, etc., as is currently be done.

Through the discussion of two areas (the competitors perceptual and motoric abilities; and the game’s extrinsic and intrinsic factors) we hope to show that, so far, no fair competition between AIs and human has been won by an AI in Dota 2 and StarCraft II. We argue that, if these factors are accounted for in the future, and we ignore the competitors individual characteristics of knowledge acquisition (considering them as black boxes), we can construct a competition that is capable of producing a fair evaluation of the competitors’ *output*. This output can then form the basis for future discussions on AI

vs. human intelligence. For instance, if the AI wins in our hypothetically fair competition, does this mean it is more intelligent? If the human wins, what are the areas in which the AI has to improve? Are there still other factors that we did not observe and account for? In this manner, our current approach does not claim ultimate correctness, but constitutes a step forward in the area of human vs. AI competition, by critically evaluating the current state and proposing areas of consideration for future competitions. Thus, our approach is different from Canaan et al. (2019a) in that they claim the impossibility of a fair competition due to differences in the competitors, which we simply treat as black boxes in order to focus on the competition setup instead.

While we focuses on claims of super-human performance in games, there are other claims made in this context that are worth discussing; for example, whether a system has learned from *tabula rasa* (Marcus, 2018). Due to our black-box approach, these questions will not be discussed here.

In particular, we aim to address the game AI community and its discussion of how to create a competition that enables us to claim AI superiority. The implications for players, communities, game designers and game studies researchers outside of AI are outside the scope of this discussion. Furthermore, we limit the scope to only consider AI systems in the role of playing a game competitively. It is, however, important to not neglect that AI/CI has many other roles for games such as procedural content generation, player modeling, and data mining (Yannakakis, 2012). When we ask the question *are we done with games?*, we are thus only concerned with the traditional branch of AI in games where the goal is to create a system that plays competitively.

9.1 The Blackbox Approach

We propose a pragmatic way of evaluating AI against human intelligence in game competitions. To be able to do this, some obvious differences have to be pointed out and disregarded. First of all, for the purpose of comparability, we consider AI systems as black boxes. Their training, knowledge, common sense and idiosyncratic function will not be considered. Treating an AI as a black box thus disregards four of the six dimensions of fairness introduced by Canaan et al. (2019a). This leaves us with perceptual

and motoric fairness as they deal with the system's interaction with the game. In fact, human vs. human competitions also take this approach: we ignore how contestants have prepared for the competition and their IQ scores are not considered important. Only in few cases (e.g. weight in boxing or gender in physical sports) is it deemed necessary to impose further restrictions and limitations. In the case of electronic sports (eSports), gender segregation is a topic of an ongoing debate, which to cover would exceed our scope. However, to explicate the 'obvious differences' that will be disregarded, a discussion of types of potential superintelligences by Nick Bostrom is useful. While his position on the emergence of superintelligences and its consequences are arguable, we deem his threefold distinction as useful here, to describe and understand ways in which machines are simply and obviously different from humans.

In the book *Superintelligence* Bostrom (2014) distinguishes three different forms of possible superintelligences (Bostrom, 2014, p.63) that we will use to explicate the aforementioned differences. *Speed intelligence* describes a superintelligence that “[...] can do all that a human intellect can do, but much faster” (Bostrom, 2014, p. 64). AI systems can usually speed up matches and play them much faster than a human player can. It is thus an obvious difference that AIs can train/evolve *faster* than humans if the number of games over time (not the learning outcome) is the measure of speed. *Collective intelligence* describes “a system composed of a large number of smaller intellects such that the system’s overall performance across many very general domains vastly outstrips that of any current cognitive system” (Bostrom, 2014, p. 65). In the context of learning, many machine learning algorithms fit this description, as AI systems usually are trained in several parallel instances to ultimately combine the gathered information into one system – an option that a human player does not have. In fact, AlphaGo, OpenAI Five, and AlphaStar all relied heavily on these advantages. AlphaGo played 1,280,000 games against itself using 50 GPUs (Silver et al., 2016), OpenAI Five played around 180 years of real-time Dota per day on 256 GPUs and 128,000 CPU cores (OpenAI, 2018a), and for AlphaStar, a league of agents each agent played up to 200 years of real-time StarCraft, each using 16 TPUs (Vinyals, 2019).

Finally, Bostrom describes *quality superintelligence* as one system that is “[...] at least as fast as a human mind and vastly qualitatively smarter” (Bostrom, 2014, p. 68). Simply put,

this resembles the difference between two humans with different IQs; or more theoretically, the encounter with an alien race that “thinks on a different level”, incomprehensible to us. This quality super intelligence is one potential interpretation of the outcomes of a fair competition between AIs and humans.

Especially when it comes to learning and training, the AIs have advantages over humans in what Bostrom called speed and collective intelligence. During training, developers can speed up games to a degree which exceed human capabilities. In addition to that, one version of a system can play hundreds of games simultaneously and gather the gained information afterward. This is a strength of artificial intelligence that we should embrace and not handicap. This factor is thus ignored by our black box approach.

As discussed here, AI systems and humans are different in many regards, and we agree with the conclusion of Canaan et al. (2019a) that a final, holistic comparison of the two is nonsensical. However, we want to discuss what happens if we exclude the obvious differences (as partially done in human vs. human competitions as well). Our interest lies in an evaluation of whether, or to what extent, the competitions between AI systems and humans were actually carried out on equal grounds. In other words, we exclude the characteristics of the participants, to evaluate the characteristics of the competition.

To be able to discuss an AI vs. human competition within games, we need a prototypical example of a human vs. human competition and a rudimentary distinction of how such competitions regulate game extrinsic and intrinsic factors. The purpose of this is to establish certain terms that enable an in-depth discussion and comparison of AI vs. human competitions and their idiosyncratic structures.

9.2 A Prototypical Human Competition

For the following discussion, some definitions of terms are necessary. First of all, the questions *what a game is* and what should be considered parts of games have been considered in many publications (Arjoranta, 2014; Suits, 2014; Avedon and Sutton-Smith, 1971; Caillois, 2001; Juul, 2011; Stenros, 2017) as well as some heated debates (Frasca, 2003; Murray, 2005; Aarseth, 2014a; Keogh, 2014). The authors behind these discussions accept

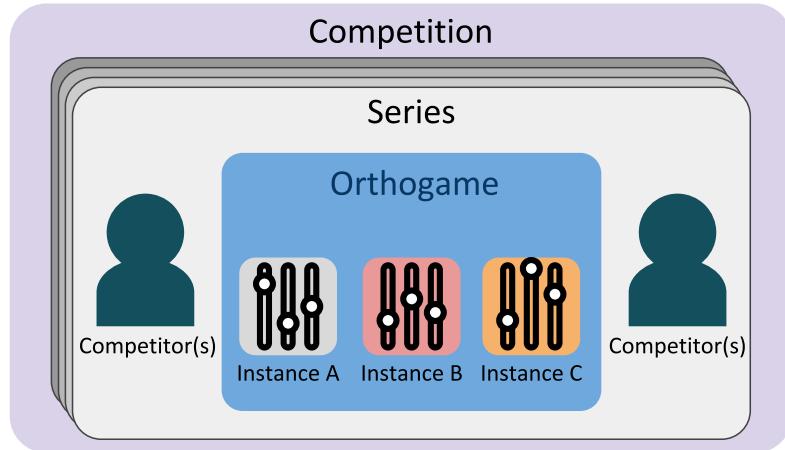


Figure 9.2.1: A prototypical human vs. human competition consisting of one or more series between two teams or individuals. Each series consists of multiple game instances of the same orthogame.

that ultimately defining games might be an impossible task that bears normative and discriminatory potential. However, for the current purpose of structuring our argument and observations, we will develop a makeshift model of game intrinsic and extrinsic factors. This model is based on previous research on the ontology of games (Aarseth and Calleja, 2015), and metagames (Carter et al., 2012; Debus, 2017).

The cybermedia model of Aarseth and Calleja (2015) constitutes a descriptive model that covers games but intentionally also other phenomena. They describe games as a player's perspective on a cybermedia object, which consists of a *materiality*, a *sign system*, and a *mechanical system*. Especially in the case of the mechanical system, it is possible that one cybermedia object contains several systems. Their example is *World of Warcraft* (Blizzard Entertainment, 2004), which contains multiple mechanical systems, such as questing, raiding, PvP arena, and PvP battlegrounds.

To avoid the confusion that the term *game* brings with it and to avoid the processual perspective on games that Aarseth (2014b) take¹, we will use the term “orthogame” by Carter et al. (2012) instead of “cybermedia objects” (Aarseth and Calleja, 2015). The orthogame describes “[...] what players collectively consider to be the ‘right and correct game’” (Carter et al., 2012). We understand the orthogame as the digital artifact installed on a computer or physical artifact as used for play (including its rules). This explicitly

¹One of the unsolved problems regarding games is the question whether they are objects of processes. The ontological commitment of a process perspective onto games is that the player constitutes an element of the game itself; a perspective that the authors do not share in the current endeavor.

excludes the player from the object itself. It is important to note that especially digital orthogames have various different 'starting configurations', i.e. maps or chosen races. These starting configurations determine individual *game instances*: subsets of the orthogame with one particular configuration. A *series* occurs between two teams or individuals across multiple instances of the orthogame, i.e. a best-of-five series. Finally, all of these concepts are encompassed by an "added metagame" (Debus, 2017, p. 5). Added metagames are structures regulating leagues, ladders, tournaments, competitions, etc.

We will now put these terms into work in an example of a StarCraft II instance. It must be noted that it is a prototypical example and a more detailed model could be drawn (as discussed at the end of this section). However, the developed terms will still be applicable in those cases, even though the structure could be expanded. Figure 9.2.1 illustrates this prototypical version of a human vs. human competition.

In our example, two players are competing for the world championship in StarCraft II. Over the course of the last months, they both proceeded through an *added metagame*: a ladder, as well as a KO system in the finals, which are held at a physical location. Now they face off in the grand final. The grand final is constituted by a best-of-five series. This means that the players face each other at least three times, playing the same orthogame (StarCraft II), but different instances of it. These instances (circle, triangle, and diamond shapes) are usually determined by one of two processes. On the one hand, there exists a "material metagame" (Debus, 2017, p. 5), which encompasses drafting armies or heroes in some games. In our StarCraft II example, there is a map selection procedure before a series, where players can veto maps that will be removed from the map pool. On the other hand, the particular configurations can be regulated by the added metagame beforehand, such as 1 vs. 1 competitions in Dota 2. These limit the orthogame to a particular player composition (two players) and a spatial layout (middle lane only). Thus, the example actually constitutes a combination of both processes, through the limitation of a map pool in StarCraft II (added metagame restriction), and the subsequent selection of maps from the pool by the players (material metagame process).

We can expand the model and include the whole added metagame of the world championships by adding more series to Figure 9.2.1, as indicated by the additional 'series' frame. These additional series, in turn, consist of (potentially) differently configured

instances of the orthogame, played by different players. Another possible constellation is an added metagame between the same players but within different orthogames. However, we further argue that the function of added metagames is to regulate game extrinsic and intrinsic factors within the added metagame, with the purpose of balancing and fairness. We will elaborate on these concepts in the following sections.

9.3 Game Extrinsic and Intrinsic Factors

To reiterate the previous section, we can split a competition into three general areas: the added metagame, series within the added metagame and instances of an orthogame played within the series. We argue that the function of the added metagame is to regulate both, the format of the series (extrinsic factors), as well as the particular configurations of the orthogame (intrinsic factors). The extrinsic and intrinsic factors of regulation is exemplified in the following sections.

9.3.1 Fairness: Game Extrinsic Factors

Added metagames generally regulate two game extrinsic areas in a competition: the structure or format of the competition and the competitors participating in it. An example of an added metagame are ladder system, as implemented in many contemporary online multiplayer games, such as League of Legends (Riot Games, 2009), Heroes of the Storm (Blizzard Entertainment, 2015) or StarCraft II. In the world championship of soccer, a mini-league system, in which teams play in discrete groups, and a KO round, in which teams eliminate each other in best-of-one series, are combined. Other sports require the finalists of a competition to face each other in several instances, for example, a best-of-five-series. The aim of this is to minimize arbitrary factors that could predetermine a victory as much as possible. In chess, for example, the white player starts the game with a slight advantage. To balance white's advantage, an extrinsic system (the series) is employed, which guarantees that both players will start with white. Another potential reason for the implementation of series is to negate factors such as day to day performance, weather, or home turf advantage. In other words, these regulations aim to make sure the

best player wins and not a player that was lucky to gain a game extrinsic advantage.

Yet in other sports, as well as eSports, it is common to divide participants into groups, depending on their physical attributes. In boxing, for example, the added metagame divides participants into groups by weight. In nearly every sport it is also common to distinguish between youth- and adult-leagues. The intention here is not only to make the individual instances more interesting for the spectators but also to prevent injuries. A similar, but a more controversial division is related to the participants' gender (Teetzel, 2006). This division is also employed in eSports, where its usefulness is a topic of ongoing debate. We will discuss the necessity of limiting 'physical' capabilities of participants in AI vs. human competitions in the section "Critique of AI Achievements in Games."

9.3.2 Fairness: Game Intrinsic Factors

As discussed earlier, contemporary digital game artifacts contain multiple mechanical systems. Aarseth and Calleja (2015) mention raiding, questing and player vs. player (PvP) as examples in World of Warcraft. Different models identify different (types of) elements of games for different purposes (Bjork and Holopainen, 2004; Wolf, 1997; Aarseth et al., 2003). Elverdam and Aarseth (2007), for example, identify dimensions of games for the purpose of classification. These include the players' relation, virtual and physical space, struggle (including the game's goals), and more. The individual categories are less important than the observation that contemporary video game can rarely be described in only one manner. While it is easy to argue that chess particularly requires two opposed players, it is impossible to make the same general statement about the digital artifact StarCraft II, which contains the potential player compositions of single-player, two-player, multi-player, and multi-team (Elverdam and Aarseth, 2007).

Another game intrinsic factor is different maps (e.g. in StarCraft II), which would be comparable to different boards in chess or go. These maps influence the individual strategies, but also the available player numbers. While StarCraft II also allows playing three different races, it is common for a player to stick to one race throughout their 'career'. In Dota 2, however, players need to be able to play a range of different heroes, as each series includes the drafting and banning of heroes from a pool of (as of writing)

around 115 heroes. Selecting heroes can be considered a “material metagame” (Debus, 2017) to determine the configuration of the game instance. Added metagames can also regulate clear orthogame intrinsic content, such as available items inside of Dota 2. In some cases, items that heroes use to increase their strength of abilities were banned by leagues, as they were considered overpowered or were simply bugged.

When comparing AI and human intelligence in games, the here discussed multifacetedness of games must be considered and both participants’ capabilities of engaging with the *artifact as a whole* need to be examined. This insight leads us to the next section, in which we examine particular games and competitions within them, using the here developed distinction of game extrinsic and intrinsic factors, as well as the prototypical human vs. human added metagame.

9.4 Critique of AI Achievements in Games

In this section we will review a few selected milestone achievements of AI in games, focusing on particular systems and their comparisons to human professionals. We will discuss the fairness of these competitions and whether it is valid to claim that the AI systems are super-human using the black box approach and the coherence of game extrinsic and intrinsic factors. The goal is to identify to what extent the presented milestones adhere to or deviate from the prototypical human vs. human added metagame. Any deviations, we argue, indicates an unfairness in the competition (for either side) and a claim of superiority must be treated carefully in these cases.

9.4.1 AlphaGo

AlphaGo, developed by DeepMind, is the first go-playing system to win against professional go players without handicap on a full-sized board. The first of these games were against the 2-dan European go champion Fan Hui in 2015, where AlphaGo won 5-0 (Silver et al., 2016). The 9-dan 18-time world champion Lee Sedol was the next target for AlphaGo, and a five-game match was scheduled in 2016 with a one million dollar prize and following the official rules: Chinese ruleset, a 7.5-point komi, and two-hour time limit for each player.

Prior to the match, the Fan Hui games were published allowing Lee Sedol to prepare against AlphaGo.² AlphaGo won 4-1 with a loss in game four, showing a weakness in the system that might be exploitable. A new version of AlphaGo called 'Master' was anonymously registered to the 'Tygem' and 'FoxGo' go servers, playing a total of 50 game instances, with a shorter time limit than usual game instances, against professional and top players, winning all of them³. AlphaGo's last game instances were at the Future of Go Summit, where AlphaGo won against several top players including the highest ranked player Ke Jie and a team of five human players, without losing a single game. After this event, AlphaGo was retired⁴.

We argue that AlphaGo ultimately competed fairly in non-restricted matches against numerous top professional players both online and in settings similar to human competitions both in terms of game intrinsic and extrinsic factors.

9.4.2 OpenAI Five

In 2016, the company OpenAI decided to pursue the challenge of beating human professionals in the multiplayer online battle arena (MOBA) game Dota 2 (OpenAI, 2018b). Dota 2 is a fast-paced real-time game, has partially observable states, high-dimensional observation and action spaces, and has long time horizons (OpenAI, 2018a). Normally in Dota 2, two teams of five players play against each other while there also is a one vs. one (1v1) variant. In 2017, OpenAI developed a bot capable of playing the 1v1 version that beat the former professional player 'Blitz' 3-0, the professional players 'Pajkatt' 2-1, 'CC&C' 3-0, the top 1v1 player 'Sumail' 6-0, and 'Dendi' 2-0. The standard (or most popular) variant of Dota 2 is played in teams of five players. However, because there exist serious 1v1 Dota competitions, the added metagame does adhere to at least an existing version of the added human vs. human metagame, mimicking a termination tournament with five players. The bot was updated by the developers between each series (OpenAI, 2017); possibly bugs were fixed and control parameters were tuned. 'Sumail' also played against the previous version of the bot and won this time 2-1. It can be argued

²<https://web.archive.org/web/20160214135238/https://gogameguru.com/an-younggils-pro-go-videos-alphago-vs-fan-hui-game-2/>

³<https://www.nature.com/news/google-reveals-secret-test-of-ai-bot-to-beat-top-go-players-1.21253>

⁴<https://deepmind.com/research/alphago/alphago-vs-alphago-self-play-games/>

that altering the bot in-between game instances is a violation of the black-box approach, as it effectively becomes a new system. However, human players usually have the ability to discuss strategies with a coach in-between game instances, so perhaps an AI should also be allowed to be influenced by a 'coach'. In any case, it depends on whether the developers that are modifying it, are considered part of the entity that is competing, which we would argue, they are not. Human modification of the AI system should thus not take place within a series of individual game instances, or even whole added metagames. Because both positions (for and against human intervention) are arguable, it appears necessary to develop an explicit regulation in this area for future human vs. AI competitions.

These series were played under standard 1v1 tournament rules. The bot had direct access to the features of the game artifact from the API, instead of being presented to the visual representation of the game artifact. The bot could only access the same information that would have been available to a human player but it was structured differently. For instance, humans have to infer the position of heroes and thus estimate the distances between units, which are important for ranged attacks, while the bot can access the exact positions and thus calculate the exact distances, instantly. This arguably goes against perceptual fairness because the input space should be the same. Here, the input space includes the same information for both the AI system and the human, but by perceiving the game state in a different way than humans, the AI system might have an advantage or disadvantage. The bot had access to the same actions as human players and they were performed at similar frequencies but with a quicker reaction time of 80ms (OpenAI, 2018c). The reaction time was, however, reduced in later competitions.

After the 1v1 win, OpenAI let the bot play thousands of games against various players, where several exploits were found to overcome the bot (OpenAI, 2017). This setup mimics a human ladder where we would expect experienced human players not to have trivial exploits. The AI, however, would quickly descend the ladder due to these discovered exploits.

In 2018, a newer version of the bot named *OpenAI Five* was able to beat a team consisting of 99.95th percentile players 'Blitz', 'Cap', 'Fogged', 'Merlini', and 'MoonMeander' (some are former professionals) in a restricted version of the 5v5 game (OpenAI, 2018c). This series was named the *OpenAI Five Benchmark*. Some of the restrictions include a fixed

hero pool of 18 heroes (instead of 117) resulting in 11 million possible game instances, no summons/illusions, and no Scan. The reaction time was increased from 80ms to 200ms in an attempt to match that of humans. The bot won the first two game instances where it did the hero drafting itself and lost the third game where the audience did the draft. The restricted hero pool is a significant limitation of the game intrinsic factors, effectively reducing the possible game instances to a much smaller subset than are usual in human vs. human competitions.

'OpenAI Five' later played two show series against the two teams of top professionals 'paiN Gaming' and 'Big God' and lost.⁵ In 2019, 'OpenAI Five' won a best-of-three series 2-0 (the OpenAI Five Finals) against the Dota team OG, which consists of top-professional players. In this series, the hero pool was further restricted to just 17 heroes (OpenAI, 2019). Playing against just one team mimics the final series of a tournament but not an entire tournament, ignoring the game extrinsic factors of complete added metagame structures.

After the win against OG, the *OpenAI Five Arena* allowed anyone to play against OpenAI Five. These games had the same restrictions as earlier. OpenAI Five won 7,215 games and lost 42 (99.4% win rate) against a total of 15,019 players⁶. One team, mainly consisting of the players 'ainodehna', 'backtoashes', 'CANYGODXXX', '.tv/juniorclanwar', and 'gazeezy', was able to reach a ten game winning streak.

The OpenAI Arena is basically an extensive ladder setup cohering to game extrinsic factors. A ladder challenges the bot to be robust to many different strategies and playing styles. The fact that it won almost every game but one team was able to beat it repeatedly is interesting. If this result was due to a trivial exploit, then most teams, knowing about the exploit, would be able to beat it; for a human opponent this would not be the case. However, the bot won 99.4% of the games in an extrinsically fair setup, which we would not expect even from human world champions. The criteria for being the best Dota 2 team on a ladder is not to have a 100% win rate, and we thus should not impose that expectation on the bot.

⁵https://liquipedia.net/dota2/The_International/2018/OpenAI_Showmatches

⁶<https://arena.openai.com/#/results>

9.4.3 AlphaStar

In 2019, DeepMind played their bot *AlphaStar* against the two top professional players Dario 'TLO' Wünsch and Grzegorz 'MaNa' Komincz and won both series 5-0 (Vinyals, 2019). All games in these two series were Protoss vs. Protoss on the standard medium-sized map *CatalystLE*.

It was claimed that these series adhered to professional match conditions (Vinyals, 2019), while this is in fact not the case. Tournaments never use just a single map for a whole series but instead a predefined map pool, thus the competition did not adhere to the game extrinsic factors. Additionally, in professional tournaments, players face multiple players controlling any of the three races, and not just Protoss.

After the match 'MaNa' also mentioned that he made a few mistakes because they played an earlier version of StarCraft II than the one he was used to; he also did not warm up, which he would usually do (Komincz, 19:50). The actions per minute (APM) count of AlphaStar was around 280, which is lower than professional players, and with a reaction time of 350ms on average. AlphaStar had only access to visual information from the game, similarly, but not exactly identical, to the screen pixels presented to human players (Vinyals et al., 2017). This is arguably a violation of the game intrinsic factors, similarly to OpenAI Five, since AlphaStar has a different input space than human players have. It is, however, a weak violation since AlphaStar's representation of the game state has the same information, while it is just structured differently.

Importantly, AlphaStar was not restricted to the limited view of the camera, which a human player has to control manually. As DeepMind puts it: "it could observe the attributes of its own and its opponent's visible units on the map directly, without having to move the camera - effectively playing with a zoomed out view of the game" (Vinyals, 2019). 'MaNa' expressed this as being "very unfair" (Komincz, 1:17:15). This is, however, a clear advantage of AlphaStar on both the levels of perceptual capabilities and motoric necessities. Furthermore, it could even be argued that this alters the "perspective dimension" (Elverdam and Aarseth, 2007) of StarCraft II from vagrant to omnipresent, which is arguably an alteration of the orthogame itself.

Later, the professional player 'MaNa' played against a prototype of AlphaStar that

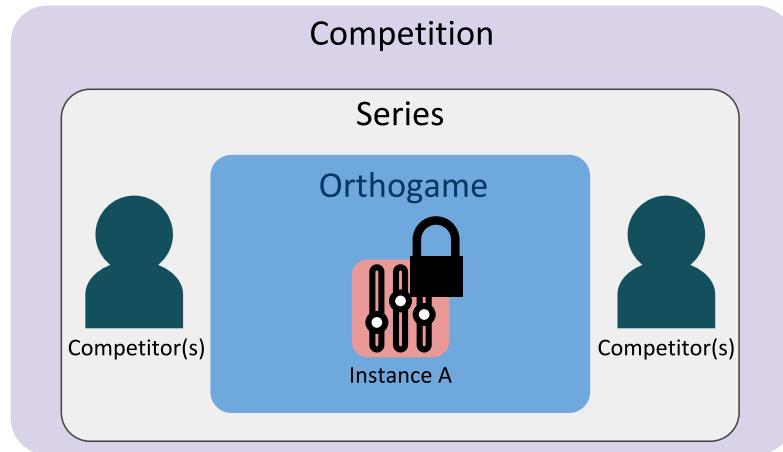


Figure 9.4.1: A typical AI vs. human competition consisting of one series between two teams or individuals. The series often consist of identical game instances, or a limited set of game instances, of the same orthogame.

controlled the camera as well, in a single-game series and won. He found a weakness in AlphaStar during a Warp Prism harassment with Immortals, continuously warping in units, picking them up, and escaping. Whether this weakness was due to the camera control or if it was a critical exploit of AlphaStar is not known. It may, however, seem that he won the last game because it was played at a later date than the others and he had time to prepare against its style. Specifically, he said that “We (‘TLO’ and ‘MaNa’) noticed that the agent sticks to the basic units a lot. It’s very confident in its micro, and it should be, it’s great micro, but it doesn’t really transition out of it.” (Komincz, 1:19:15). ’MaNa’ said his new plan was to “... defeat AlphaStar with simply better unit composition rather than unit control” (Komincz, 1:20:20). We notice here, that in the two 5-0 wins against ’TLO’ and ’MaNa’, they did not have a chance to scrutinize any recorded games played by AlphaStar, which professional player typically can do before important human vs. human series. In contrast, the developers of AlphaStar picked and knew the opponent in advance. ’MaNa’ said, commenting on his first series against AlphaStar: “I was completely in the dark ... I don’t know what to expect. If you are a StarCraft player you are familiar with people you are playing on the ladder ... you know what their styles are.” (Komincz, 18:40). Compared to a prototypical human vs. human competition, this is an unusual setup of the competition, as professional players know each others’ play styles before playing. To observe the problem from another angle: human vs. human added metagames never keep their participants secret from their participants, as was the case in ’MaNa’ vs. AlphaStar.

9.5 Adaptation in Fair Competitions

Fair human vs. AI game competitions, as we have formulated them here, undoubtedly require players to be adaptive. When competitions consists of multiple series of matches against multiple different opponents using a wide range of game variations, players cannot have fixed behaviors. Successful players must change their strategy in-between games (inter-game adaptation) to counter opponent strategies in series of matches. The ability to generalize to a large set of game variations also seem to be a requirement in fair competitions. In StarCraft, there are three different race matchups for each race (excluding random), and usually a pool of around six maps. In Dota 2, only one map is used but with 115 different heroes, players need to generalize their playing style to team combinations that they have not yet played or played against.

As we claim that adaptivity and generality are required to master fair competitions in complex games, it highlights the importance of the work in this dissertation. Learning to generalize on multiple levels/maps (Chapter 8) and learning a diverse set of behaviors (Chapter 6) seem to be particularly important.

9.6 Summary

We introduced a black box approach that can be used when designing and evaluating human vs. AI game competitions as well as the notions of game extrinsic and intrinsic factors. We applied these to discuss the fairness of recent AI achievements of AlphaGo, OpenAI Five, and AlphaStar. It appears that the added metagame's role in an AI vs. human competition has a different focus than in the human vs. human competitions. The added metagame in a human vs. human competition regulates mostly the bigger structure of extrinsic factors, such as a sequence of series, number of instances in a series and groupings due to a physical difference between the competitors. The added metagame's role in the AI vs. human competitions, however, is focused on the regulation of game intrinsic factors. This means, in competitions between AIs and humans in digital games, the orthogame is so far always limited to either one particular configuration or a very small number of possible configurations, compared to human vs. human competitions. A

visualization of this setup is shown in Figure 9.4.1. OpenAI Five, for example, is capable of playing only 17 out of (approximately) 115 heroes of Dota. Thus, the orthogame (Dota 2) in the competition between OpenAI Five and humans had to be limited to these 17 heroes.

OpenAI Five had direct access to game state variables while humans must infer positions, attack ranges, and health from a raw visual representation, which ultimately leads to 'educated guesses' more than factual knowledge. AlphaStar, in fact, used a visual representation but a different one than what is presented to humans. Furthermore, OpenAI Five and AlphaStar are incapable of 'misclicking', which is the act of giving a command unintended by the player. These two factors constitute imbalances in the perceptual and motoric capabilities of the competitors, which must be accounted for in the future. To have a fair competition, the AIs must be handicapped through their interaction with the game to imitate how humans are interacting with it, i.e. if humans have imprecise and slow means of interacting with the game it should be the same for the AI system. It can be argued that this is something that naturally occurs in human vs. human competition, as every human participant is implicitly restricted to human capabilities. Thus, the addition of a handicap to the AI simply constitutes an explicit correction through game extrinsic factors.

We thus conclude that we are not done with games. The games proposed as ultimate AI challenges, Dota 2 and StarCraft II, are not yet mastered by AI. As we identified, so far AI vs. human competitions are different in that (1) the AIs do not compete in a tournament structure, but are simply matched with the best (available) human player and (2) they limit the orthogame in particular ways, such as range of maps, heroes or races. To be able to claim that 'We are done with games' the AI has to engage in a fair competition with humans that is constituted by the same external factors, such as several matches against the same opponent, as well as different opponents. Additionally, it should not limit game internal factors, such as only allowing certain playable heroes or maps. Only then, the claim that AI is superior to humans in games is justified, given that StarCraft II and Dota 2 are and remain the most complex games to beat.

We are, however, not neglecting the significant progress made towards achieving the goals, as no system before OpenAI Five and AlphaStar could win against professionals in any

competition in these games. The fact that game intrinsic factors have been severely limited suggests that current AI methods cannot cope with the complexities introduced by varying game configurations. This limitation was explored in Chapter 8. Game extrinsic factors have also been largely ignored, pointing at a new area to consider when comparing humans and AI systems in games.

An ultimate goal that would demonstrate that an AI system can fully master a game, beyond the extrinsic factors of human vs. human competitions, would be to allow anyone to play against it over a long period of time. This setup would be similar to OpenAI Arena, without restricting any intrinsic game factors. This goal was to some degree achieved by AlphaGo when it played on the 'Tygem' and 'FoxGo' go servers without losing, and without restricting the intrinsic factors. It should be noted that AlphaStar has later been tested (after this paper was published on arXiv) on the StarCraft ladder (anonymously against top human players) on multiple maps and against multiple races, similarly to the OpenAI Arena setup. We are not aware of the exact details of these experiments and their results but it is clear that these tests pay more attention to both intrinsic and extrinsic factors. Interestingly, when playing anonymously on the ladder, players do not need to perform inter-game adaptation, as in offline StarCraft tournaments. Here, it is interesting to note, that offline tournament structures with multiple series of matches are always employed in the StarCraft II World Championship Series⁷ and The International Dota 2 championships⁸. It thus seems that these tournament structures are preferred when determining the best human player, suggesting that they should also be employed when evaluating super-human AI performance in these two games.

⁷<https://wcs.starcraft2.com/en-us/>

⁸<http://www.dota2.com/international/overview/>

Chapter 10

Discussion

This dissertation contains several contributions toward the goal of surpassing human level game-playing in the most complex games such as StarCraft. This ambitious goal has not yet been reached (in fair competitions) while the work presented here takes small steps in promising new directions with a focus on adaptation. This focus was highly disregarded prior to this dissertation, where the performance of human-level game-playing AI was mainly evaluated on restricted settings where adaptation is less needed. Without evaluating the adaptivity of a game-playing AI system, the AI system might outperform humans merely due to fast and precise execution, which is a trait that machines naturally excel in. Strategic reasoning, which includes adaptation, is, however, an intellectual trait that is much more valuable to achieve. An AI system that can outplay humans in StarCraft with super-human micromanagement control is a major achievement due to the complexity of the control problem while it would be an absolute astonishing achievement to have an AI system that can continuously outmaneuver human players in such games through strategic adaptation. The contributions of this dissertation take small steps in this direction.

x It is worth discussing AlphaStar (Vinyals et al., 2019) and how it compares to our contributions. AlphaStar was announced after most of the work in this dissertation was published, and the technical details and results of AlphaStar still remain to be peer-reviewed. The currently announced version of AlphaStar can only play one matchup and on one map. The StarCraft systems presented in Chapters 4, 5, and 6 can play on most maps, while they do not adapt their playing style to the different map features. AlphaStar was, however, capable of winning against a human player in

a restricted setup, which is still a truly amazing achievement. AlphaStar performs a weak variant of inter-game adaptation where a different policy (that was trained with a unique objective) is sampled at every game. Simple adaptation mechanisms could be applied to intelligently select between these policies, similarly to our approach in Chapter 6. AlphaStar’s ability to perform intra-game adaptation is hard to analyze with the currently available information, while it is clear that many of the games are won through superior micromanagement (see Chapter 9). In the shadow of AlphaStar’s success, the contributions in this dissertation may seem pointless, as the systems we developed did not win against human players in StarCraft. Here, it is important to consider that AlphaStar was developed by a team of more than 30 researchers and engineers with easy access to computational resources (Churchill et al., 2019), which undoubtedly have an impact on the quality of their experimental results. Our approaches may as well be able to achieve better results if they are scaled to use more computational resources together with a higher quality of engineering. AlphaStar and OpenAI Five both rely on fundamental machine learning techniques such as traditional imitation learning and reinforcement learning algorithms. Our work is shown to improve such traditional algorithms in terms of the achieved win rate or game score in a setting that requires adaptivity. It would thus be promising to combine our techniques with AlphaStar or OpenAI Five with the goal of achieving super-human performance in fair competitions.

The three properties of adaptivity that were identified in the introduction, as well as the challenge of overcoming sparse rewards, are discussed next in the context of our contributions.

10.1 Intra-game and Inter-game Adaptivity

A static policy that can only execute one strategy regardless of the opponent’s strategy is usually exploitable by adaptive policies. When reinforcement learning is applied to a training setting against a single fixed opponent, there is a risk of learning such a static policy. For example, Sun et al. (2018) applied reinforcement learning to StarCraft against the built-in Blizzard bot, and the policy learned a rush strategy that did not adapt to the opponent. In Chapter 7, our baseline reinforcement learning algorithm learned

a non-adaptive policy using just one weapon, which was very effective in the training setting, but most likely exploitable by human players. Reinforcement learning thus seem to overfit to the training opponents, requiring a large set of diverse agents to train against. AlphaStar avoids overfitting through the AlphaStar League (Vinyals et al., 2019) where it plays against other reinforcement learning agents that are trained in parallel with different objectives. BRIL (Chapter 6) learns a diverse set of behaviors from human demonstrations which could serve as a simple way of seeding such a league with diverse and human-like policies. Imitation learning methods (explored in Chapter 5) has the property of imitating humans and should thus achieve intra-game adaptivity automatically, in the same way humans adapt in-game. This is a powerful feature of imitation learning, where reinforcement learning require a sophisticated training setting. Vanilla imitation learning is, however, by design incapable of achieving super-human performance alone. Applying imitation learning and reinforcement in parallel (Harmer et al., 2018) could provide the strengths of both approaches to achieve a strong policy with intra-game adaptivity.

There seems to be a trade-off between how well a policy expresses a certain type of strategy and how well it adapts. One could argue that a policy designed to execute a rush strategy in StarCraft should be able to adapt into a non-rush strategy. Rather, a high-level mechanism could be employed to switch (in-game) between static low-level policies. How to apply such mechanism for both intra-game and inter-game adaptivity is an interesting direction worth exploring.

An agent can perform inter-game adaptivity by either having access to an adjustable policy (Chapter 6) or a set diverse policies (Section 11.1 and 11.2). The first approach has clear advantages in terms of storage while it may also have the opportunity to be more expressive since it is possible interpolate almost infinitely between two behavioral descriptions. Additionally, it may be easier to learn multiple behaviors for the same game with a single set of model parameters (depending on the optimization procedure employed) allowing the algorithm to reuse general feature representations across multiple behaviors. It would be insightful to investigate these two potential advantages further.

While I have studied adaptation in a competitive game setting, it is also be a useful property in real-world cooperative settings, e.g. when robots interact with other agents with unknown behaviors/incentives. Recently, Canaan et al. (2019b) demonstrated how

a diverse set of policies allow for inter-game adaptation in the cooperative card game Hanabi.

10.2 Generality

In Chapter 8, I demonstrated how reinforcement learning agents trained on just one level were unable to generalize to unseen levels of the same game. We then introduced procedural content generation to present the agent with a completely new level at each training episode. This allowed the agents to achieve higher scores on unseen levels in several games. Using deep reinforcement learning from pixels it is hard to imagine how the agent can learn a general policy without training on a large set of levels. However, humans can easily transfer the knowledge learned in one level to another. Learning general policies without procedurally generating new training levels is also an interesting direction.

Learning policies that generalize to unseen levels in complex games such as StarCraft using reinforcement learning and procedural content generation would be an interesting experiment. To investigate this further, one could apply PPCG (Chapter 8) to progressively increase the size and difficulty of the training levels and in that way measure how far we can get with limited resources.

The challenge of achieving generality across several unseen levels in games has gained more interest along with several new game AI challenges such as the OpenAI Retro Contest (Nichol et al., 2018), the Unity Obstacle Tower Challenge (Juliani et al., 2019), and CoinRun (Cobbe et al., 2018). We believe this trend will continue toward robust methods that may allow policies trained in simulation to overcome the reality gap in robotic tasks.

10.3 Overcoming Sparse Rewards

Learning in environments with sparse rewards is difficult and thus most approaches, as well as the ones introduced in this dissertation, employs some mechanism that provides additional feedback to the agent. Procedural content generation can be used to generate smaller and simpler environments in the beginning of the learning phase (Chapter 8),

resulting in more frequent feedback. A different approach is to modify the reward function to provide an intrinsic reward signal. Our approach called Rarity of Events (RoE) (Chapter 7) is similar to other approaches that intrinsically reward the agent for experiencing rare situations. Our reward mechanism is based on the temporal rarity of pre-defined events while Bellemare et al. (2016); Ostrovski et al. (2017); Burda et al. (2018) are all based on the rarity of patterns in the raw observations. While it requires some domain knowledge to determine the events in RoE, it is usually simple to implement and allows the experimenter to specify the desired behavior on a high-level. Intrinsic rewards based on raw observations does not allow the experimenter to guide the performance toward a certain behavior while it is a more general approach. We believe RoE is specifically well suited for game developers that want an easy and yet flexible method that is easy to understand and implement.

We trained a build-order planner for StarCraft using imitation learning and applied it to a modular bot. This modular approach is suited for reinforcement learning as it greatly reduces the observation and action space as well as the number of interactions; each module only takes periodically actions based on selected features of the game state. Our approach could be extended such that all high-level modules are replaced by neural networks, sending high-level commands to low-level scripted modules that execute raw actions in the game. This idea seems feasible but requires a good amount lot of engineering and experimentation. Sun et al. (2018) implemented another kind of extension to our work wherein a single network was responsible for all high-level tasks, thus replacing several modules in the bot. We believe that such modular bots with both high-level neural networks trained with either imitation learning or reinforcement learning and low-level scripted modules deserves to be explored further. For example, the approach by Sun et al. (2018) could be extended with a training setting with procedural content generation to improve the generality on different levels and BRIL could be applied to ensure strategic robustness against a diverse set of training opponent. While AlphaStar was trained end-to-end (from raw observations to raw actions) and it was able to overcome sparse rewards (on one map), the optimization procedure requires computational resources that are not accessible by most researcher and game developers. The modular approach is thus an interesting alternative to end-to-end learning in these complex games. It would be interesting to investigate if the approach by Sun et al. (2018), with our proposed

improvements, could reach the same skill level as AlphaStar by using far less training resources.

Chapter 11

Future Directions

11.1 MAP-Elites for Noisy Domains

We have argued in this dissertation that a large set of diverse and high-quality policies enables inter-game adaptation. MAP-Elites (Mouret and Clune, 2015) is a popular Quality-Diversity algorithm that aims to fill (illuminate) an entire behavioral space with high-quality solutions and it is thus seem to be perfectly suited for our problem. The work presented here investigate the effect of noise on diversity, performance, and robustness of the solutions found by MAP-Elites, which has not been studied in depth despite most games and real-world problems are stochastic. To deal with the issues introduced by noise we propose an adaptive sampling technique that gradually increases the number of samples used in each solution evaluation.

We formulate the QD-optimization problem as the maximization of $M(x_1, \dots, x_N) = \sum_{i=1}^N \mathbb{E}(F(x_i))$, where x_i are the parameters of the solution in cell i within a user-defined feature space divided into N cells (or niches), $F(x) = f(x) + \delta_f(x)$, where $f(x)$ is the true performance/quality of x and δ_f is noise from an unknown distribution. The features determining which cell x belongs to is sampled from $B(x) = b(x) + \delta_b(x)$ and $M(\cdot)$ refers to the total expected quality and is maximal when the archive is filled with high-quality solutions.

A common approach to deal with noise in evolutionary algorithms is to evaluate each solution multiple times and take the average of all the performance measurements, also

known as explicit averaging (Paenke et al., 2006). For QD-algorithms, the same can be done to determine the behavioral features. Explicit averaging introduces several issues for an elitist algorithm like MAP-Elites: (1) The number of evaluation trials is usually found experimentally by balancing between inaccurate estimations, leading to unstable solutions, and longer running times for the algorithm, (2) domains with a high level of noise require many trials to reach accurate results, which require more computation, and (3) the algorithm will over time over-estimate the fitness and thus prioritize unstable solutions.

11.1.1 Adaptive Sampling & Drifting Elites

Intuitively, we need to re-evaluate more at the end of the evolutionary process (to get precise results) than at the beginning (when a rough approximate is acceptable). We thus propose an adaptive sampling approach along with an early-stopping rule. Previous adaptive sampling methods for EAs (Cantú-Paz, 2004) are not directly applicable to MAP-Elites. The approach for evaluating a solution is shown in Algorithm 9, where lines in gray are unnecessary when $B(x) = b(x)$.

Algorithm 8 MAP-Elites with **adaptive sampling** and **drifting elites** for noisy performance measures and feature descriptors.

```

1: procedure EVALUATE( $x$ )
2:    $e \leftarrow \emptyset$                                  $\triangleright$  The elite to challenge
3:    $V \leftarrow \emptyset$                              $\triangleright$  Visited cells
4:   while  $e = \emptyset$  ( $|x_P| < |e_P|$ ) do
5:      $b, p \leftarrow \text{SIMULATE}(x)$             $\triangleright$  Measure behavior and perf.
6:      $x_B \leftarrow x_B \cup \{b\}, x_P \leftarrow x_P \cup \{p\}$ 
7:      $e \leftarrow A(x_B)$                           $\triangleright$  Get current elite from archive A
8:      $c \leftarrow \text{CELLINDEX}(A, e_B)$             $\triangleright$  The cell occupied by  $e$ 
9:     if  $e = \emptyset$  then
10:      return
11:    else if  $x_P < e_P$  then                   $\triangleright$  Mean perf. of  $x$  and  $e$  is compared
12:       $V(c) \leftarrow V(c) + 1$                     $\triangleright$  Increment visit count
13:       $b, p \leftarrow \text{SIMULATE}(e)$             $\triangleright$  Re-evaluate  $e$  once
14:       $e_B \leftarrow e_B \cup \{b\}, e_P \leftarrow e_P \cup \{p\}$ 
15:       $c' \leftarrow \text{CELLINDEX}(A, e_B)$ 
16:      if  $c \neq c'$  then
17:        Remove  $e$  from cell  $c$                   $\triangleright$  Elite is drifting
18:        EVALUATE( $e$ )                           $\triangleright$  Resume evaluation of  $e$ 
19:         $e \leftarrow A(x_B)$ 
20:        if  $e \neq \emptyset$   $x_P < e_P$  then
21:          return
22:        if  $V(c) > \frac{|V|}{2}$   $|x_P| < |e_P|$  then
23:          return

```

In this scheme, solutions are evaluated until one of the following occurs: 1) The solution is estimated to be in an empty cell, 2) the solution has been evaluated the same number of times as the corresponding elite and the solution has a higher mean performance, or 3) the mean performance is lower than the corresponding elite's. If feature measures are noisy, case number 3 is only activated if the solution has visited the current cell more than 50% of the time; we believe it has *settled*. In case 1 and 2, the solution is added to the archive in the corresponding cell and in case 3 the solution is discarded. When discarding a solution, the elite is evaluated one additional time to improve our estimations.

As solutions in the archive are re-evaluated over time and thus their behavioral descriptors change, solutions will have to be moved to new cells; i.e. they are *drifting*. A naive implementation of this idea will leave behind empty cells whenever solution *drift*. Our solution to this is to store the k best solutions in each cell while only treating the fittest one as the elite. When an elite is moved to a new cell it will continue its evaluation procedure and the second most-fit solution in the cell becomes the new elite. We observed improvement by using $k = 10$ instead of $k = 1$ but saw no difference when using $k = 100$.

11.1.2 Experiments

We test three domains: (1) 6-D Rastrigin with noisy performance measures and deterministic feature measures: $f(\mathbf{x}) = 10n + \sum_{i=1}^n [x_i^2 - 10 \cos(2\pi x_i)]$ where x in $[-5, 10]$, $n = 6$, $F(x) = f(x) + \mathcal{N}(0, 625)$, and $B(x) = b(x)$; where $b(x)$ is equal to the two first values in x . 2) 6-D Rastrigin with noisy performance measures and noisy feature measures, such that $F(x) = f(x) + N(0, 625)$ and $B(x) = b(x) + \mathcal{N}(0, 0.01)$, and 3) the OpenAI Gym BipedalWalker environment (Klimov, 2016) which is stochastic and thus no artificial noise is added. The variances of the added noise were determined by randomly sampling the search space.

We use the CVT variant of MAP-Elites (Vassiliades et al., 2018)), a batch size of 100, random initialization of 1,000 solutions, and for the mutation operator, sigma iso was set to 0.01 and sigma line to 0.2, with 25,000 samples to generate the CVT archive. We use 5,000 niches in the two Rastrigin experiments; for the BipedalWalker the number of niches is set to 1,000 and the neural network has 24 inputs, two fully-connected layers of 256 tanh

units, and 4 outputs. Initial parameters are sampled uniformly random within $[-0.5, 0.5]$ ($[0, 1]$ for Rastrigin). We compare our approach to three baselines with explicit averaging of $n = 1$, $n = 10$, and $n = 100$. Rastrigin experiments were repeated three times each, while the BipedalWalker experiment was executed once. To analyze the solutions found by each algorithm we *correct* the archives by re-evaluating every elite (either by using the true value for Rastrigin or re-evaluating each elite 100 times for the BipedalWalker) and move them to their correct cells. This sometimes leaves cells empty (see Figure 11.1.1).

Figure 11.1.1 shows the *corrected* collection size, total normalized quality (the sum of all solution qualities normalized from the range $[-250, 0]$ for Rastrigin and $[-50, 300]$ for the BipedalWalker) and the number of elite evaluation trials. For Rastrigin with noisy performance measures our approach results in the best total normalized quality compared to the three baselines. For the baselines on Rastrigin with noisy performance and feature measures, the collection size and total quality degrade over time as unstable solutions populate the archive. While it seems that $n = 100$ is best here, if we were able to stop early, it requires many re-evaluations to monitor the degradation. For the BipedalWalker the same trend is not as apparent, only for $n = 1$. The best results here were obtained by $n = 10$ followed by our approach. These results suggest that the adaptive sampling approach needs to be controlled better as it may be domain-specific and depend on other hyper-parameters. How to control the growth of evaluation trials remain a challenge for future work while our results demonstrate the potential of adaptive sampling in elitist QD-algorithms.

11.1.3 Conclusions

Our results show that the traditional MAP-Elites algorithm does not handle noisy fitness and behavior evaluations well, as the true collection size and quality degrades due to over-estimations. One solution is to simply increase the number of trials used in the evaluations. However, finding the right balance between accuracy and running speed can be tricky. We have presented a variant of MAP-Elites that used adaptive sampling to automatically increase the number of trials as the algorithm runs. This approach achieves satisfying results on our artificial benchmark function Rastrigin. To also deal with noisy behavior evaluations, we introduce the idea of storing multiple elites in each cell

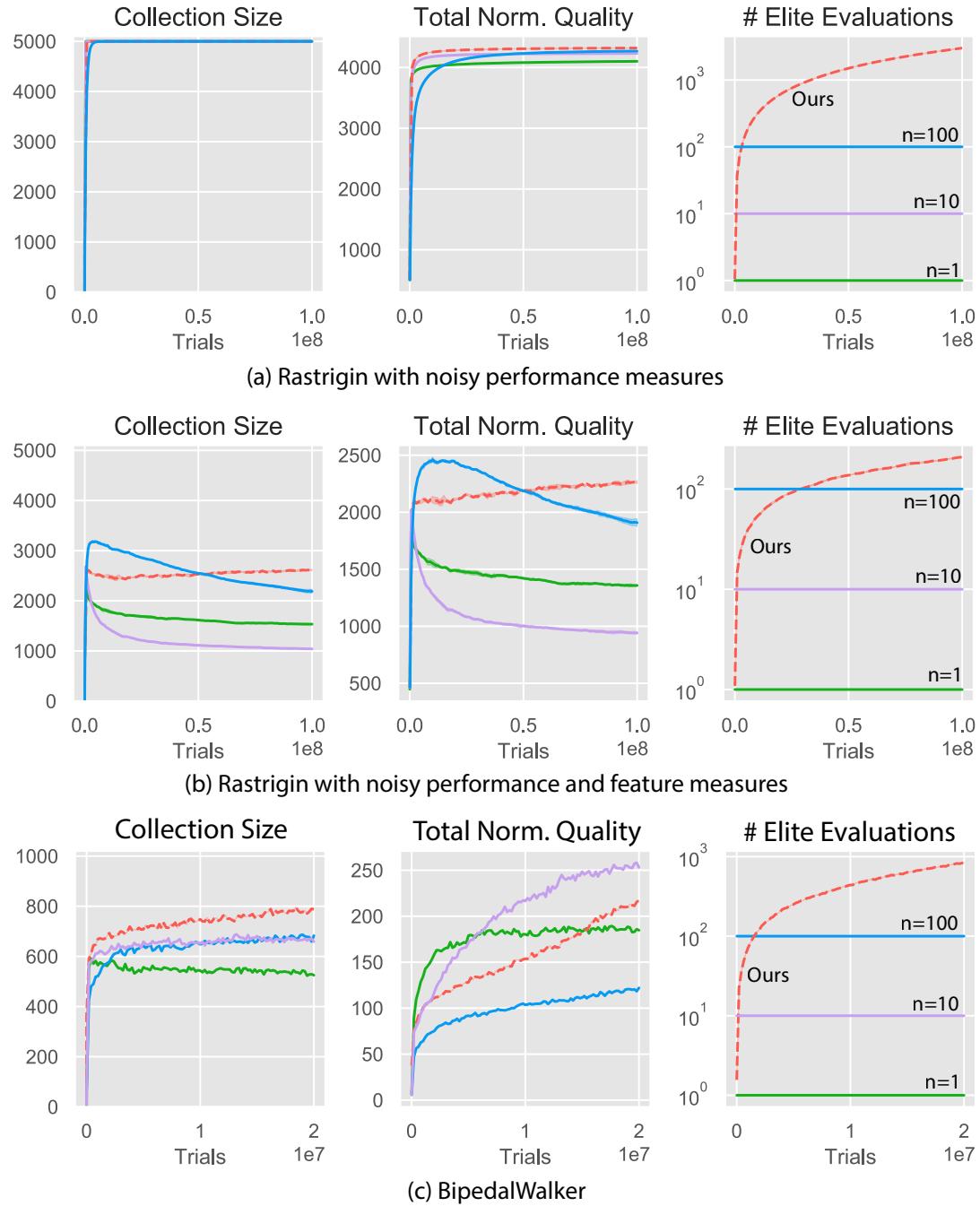


Figure 11.1.1: The *corrected* collection size, total normalized quality, and the mean number of elite evaluations for (a) Rastrigin with noisy performance measures and (b) feature measures, and (c) for the BipedalWalker.

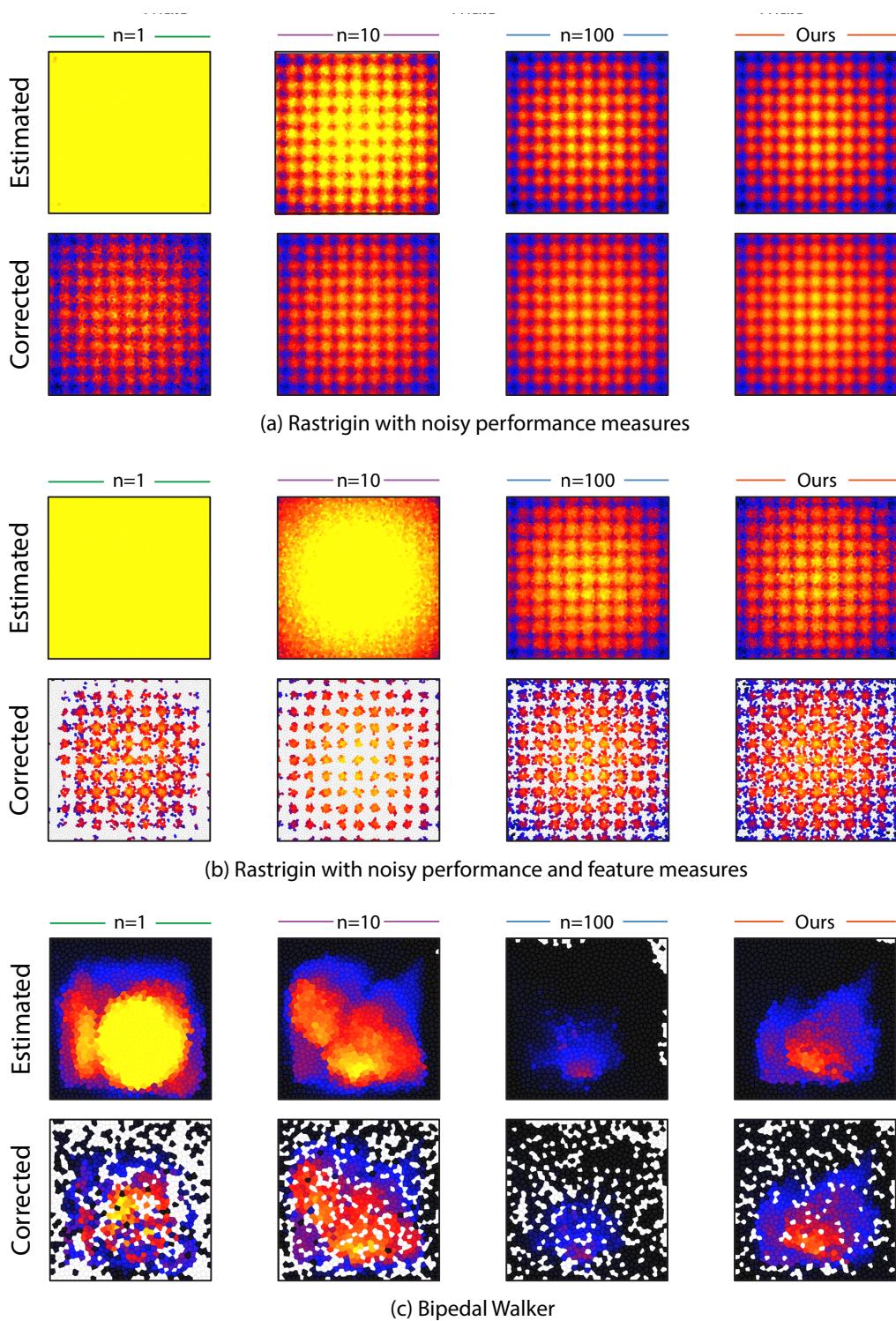


Figure 11.1.2: Examples of *estimated* and *corrected* performance-behavior maps for (a) Rastrigin with noisy performance measures and (b) feature measures, and (c) for the BipedalWalker. When behavioral measures are noisy, we can see how solutions drifts to areas of higher performance over time. This effect is smaller when using more evaluations.

as we can then re-evaluate them and move them to other cells if needed, without leaving behind an empty cell. This approach looks promising while it can be outperformed by our baseline with a certain number of trials. We believe that adaptive sampling is promising for MAP-Elites in noisy domains and future work should look into better mechanisms for adapting the number of trials.

11.2 Diverging Policies using Rarity of Events

The goal of traditional reinforcement learning algorithms is to find the optimal policy for a given training environment with a given reward function. The policy found by reinforcement learning might be optimal, or near-optimal, in the training environment, but usually performs far worse when employed in other environments (as we saw in Chapter 7 and 8). Quality-Diversity algorithms have been successful for many problems but are usually implemented using black-box optimization techniques that are inefficient when evaluation episodes are stochastic, as we saw in the previous section, or long, which is the case in most games. Here, we present a novel QD algorithm wherein a number of parallel instances of a reinforcement learning algorithm train on a fixed environment while they are rewarded to behave differently. The parallel instances submit copies of its intermediate policy to a shared archive during training when it shows novel and high-quality performance. To ensure diversity, each instance is rewarded by exploring behaviors that are novel compared to policies submitted to the archive by other instances, with the aim that it leads to divergence in the behavioral space. Our preliminary results in the game Doom show that this approach leads to divergence, resulting in a diverse set of high-quality policies. The implemented approach is based on a new map-based reinforcement learning approach and a multi-policy variant of Rarity of Events (from Chapter 7).

11.2.1 Map-based Reinforcement Learning

Previous map-based reinforcement learning algorithms (Kume et al., 2017) imitate the process of the original MAP-Elites algorithm, neglecting that parameter updates in

reinforcement learning are guided by a fixed reward function which is very different from the randomized perturbations and crossover operators employed in evolutionary search. With reinforcement learning, we expect that learning will follow a guided path that will be similar in several repeated trials from the same some point in parameter space. We thus speculate that re-initializing to previously encountered parameters in reinforcement learning (Kume et al., 2017) has minimal effect on the search for diversity. We will attempt to overcome this issue while relying on the strengths of gradient-based reinforcement learning rather than settling on random and inefficient exploration of the parameter space.

We first propose a simple modification to the previous map-based reinforcement learning algorithms, that in itself will not serve as a good QD-algorithm, but can serve as a foundation for several interesting variations. This map-based algorithm implements a weaker form of MAP-Elites; it is only used for storing solutions, and not for retrieval and re-initialization. The algorithm is thus unaffected by the map, while it is still populated by diverse policies throughout training. The behavioral features and performance of the current policy are periodically evaluated, averaged across several episodes, and stores policies to the archive following the original procedure. This algorithm can run sequentially, maintaining the map between runs, or run asynchronously with multiple instances of reinforcement learning algorithms sharing the same map. The asynchronous variant is applied in our experiments.

11.2.2 Expanding Archives

MAP-Elites stores high-performing solutions in a multi-dimensional archive consisting of rigid cells, i.e. cells have a fixed size and location throughout the optimization process. A drawback of having rigid cells, is that the experimenter has to specify the lower and upper limit of each dimension manually. If the span is too small, solutions lying outside of the bounds will share niches on the edge, despite being different. If the span is too large, fewer niches will be considered due to a low resolution in the area of interest.

To deal with these limitations, we propose an expandable archive that does not explicitly use cells. Instead, a *virtual* cell is imposed on the behavioral space when a new solution is encountered, wherein a local competition between the solutions within the cell takes place.

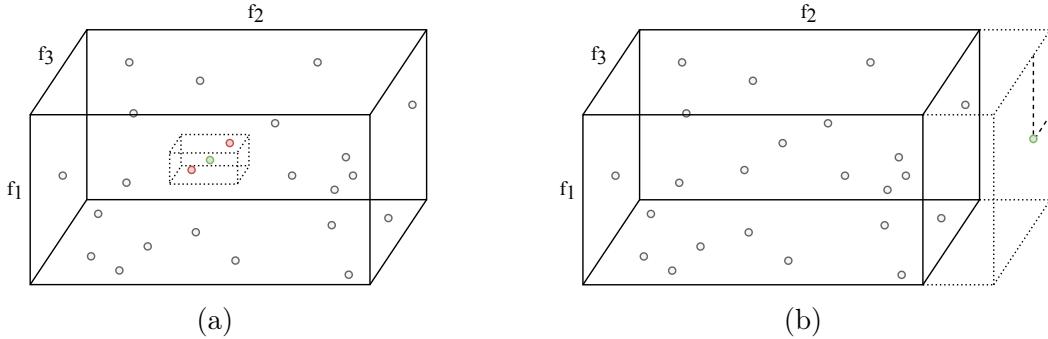


Figure 11.2.1: Visualizations of an expanding archive with behavioral feature dimensions f_1 , f_2 , and f_3 . (a) A new solution (the green dot) must compete with neighboring elites (red dots) to enter the archive. The size of the cell (dotted lines) surrounding the new solution is determined by a constant factor of each dimension’s length. (b) The length of f_2 is increased as a new solution (green dot) exceeds the previous bound. Because the size of the feature space is altered, the size of the future cells surrounding new solutions is also changed.

The length of the virtual cell along any dimension is determined by dividing the maximum encountered value along this dimension with a constant representing the number of cell divisions along dimensions. The virtual cell is then centered on the newly encountered solution, forming the perimeter of the neighborhood. If the new solution outperforms all existing solutions within the virtual cell, it will replace all of them in the archive. The expandable archive approach with virtual cells is visualized on Figure 11.2.1. Because expandable archives do not feature fixed cells, it can be seen as a combination MAP-Elites and novelty search with local competition.

11.2.3 Diverging Policies using Rarity of Events

Our approach is a multi-policy variant of Rarity of Events (RoE) for Quality-Diversity optimization. RoE assumes a set of n predefined events \mathcal{E} that can occur in a state transition. Each state transition gives the usual state-action-reward triplet as well as a vector \mathbf{x} containing n scalars representing the number of occurrences of each event (typically 0 or 1 for each event type). In RoE, all the accumulated numbers of events during an episode are recorded in a fixed-sized FIFO buffer from which the episodic mean occurrence $\boldsymbol{\epsilon}$ at iteration is computed by averaging all event vectors in the buffer. The mean occurrence is thus a temporal measure because the buffer only holds recent event vectors. The reward function R_k at iteration k in RoE is conditioned on $\boldsymbol{\epsilon} = [\epsilon_0, \epsilon_1, \dots, \epsilon_n]$,

and given a vector of event occurrences in one step $\mathbf{x} = [x_0, x_1, \dots, x_n]$, is defined by:

$$R(\mathbf{x}|\boldsymbol{\epsilon}) = \sum_{i=1}^n x_i \frac{1}{\max(\epsilon_i, \tau)}, \quad (11.2.1)$$

where τ is a small threshold (e.g. 0.01) to handle events that have not occurred. Due to this reward scheme, agents are incentivized toward new behaviors, involving events that are rare relative to itself. The agent effectively attempts to diverge from its own behavior.

So far, the description follows the traditional RoE with one policy (with a slightly different notation). However, in the variant presented here, multiple instances of a reinforcement learning algorithm run in parallel, sharing the same buffer while the episodic events are labelled with the ID of the instance it came from. The key idea in our multi-policy variant, is that policies do not have access to their own records when computing $\boldsymbol{\epsilon}$. This should allow each instance to diverge, not away from its own behavior, but away from the behaviors of the other instances, ultimately to converge to their own niche.

We combine this new multi-policy variant or RoE with Map-based reinforcement learning, such that all instances submit solutions to a shared archive. Additionally, we let the archive act as the buffer containing episodic events to reduce redundancy. The *episodic occurrences* is a vector of scalars corresponding to the number of occurrences of each of the pre-defined events in an episode, and a *behavior* is the number of occurrences per step. Each feature in the behavioral space is thus defined by the per-step ratio of each event. We still maintain the temporally by only considering a fixed number of recently added elites. There is a key difference between using a buffer and an elitist archive for RoE. With a buffer, each instance would diverge from the others' current behaviors, and with an archive, each instance would diverge from the others' last high-performing behaviors.

11.2.4 Experiments

The synchronous actor-critic (A2C) reinforcement learning algorithm is applied to the Deathmatch scenario in Doom with the same parameters as specified in Chapter 7. We test our new map-based reinforcement learning approach with n diverging policies, using $n = 2$, $n = 3$, and $n = 4$. Here, the fitness, which determines when to replace solutions in

Algorithm 9 Map-based Reinforcement Learning with Diverging Policies

```

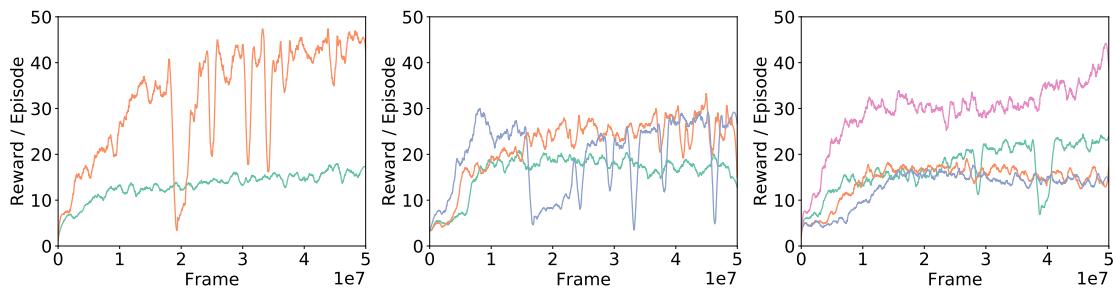
( $\mathcal{X} \leftarrow \emptyset, \mathcal{P} \leftarrow \emptyset$ )     $\triangleright$  Shared multi-dimensional map of elites: solutions  $\mathcal{X}$ , and their performances  $\mathcal{P}$ 
for  $i = 0, k$  do                                 $\triangleright k$  instances running in parallel
    Initialize parameters  $\theta$ 
     $\mathcal{R} \leftarrow \emptyset$                            $\triangleright$  History of accumulated rewards
     $\mathcal{B} \leftarrow \emptyset$                            $\triangleright$  History of episodic behaviors
    for  $k = 0, 1, 2, \dots$  do
         $\epsilon \leftarrow$  mean event occurrences in  $\mathcal{X}$  not labelled  $i$            $\triangleright$  Explained in Section 11.2.3
        Collect trajectories  $\mathcal{D}_k$  on policy  $\pi(\theta_k)$        $\triangleright$  Using Rarity of Events with rewards based on  $\epsilon$ 
        terminal episode  $e$  in  $\mathcal{D}_k$ 
            Append accumulated reward of  $e$  to  $\mathcal{R}$ 
            Append episodic behavior of  $e$  to  $\mathcal{B}$            $\triangleright$  Apply behavioral feature descriptor
            if  $k \geq N$  then                                 $\triangleright$  Do not store solutions before  $N$  episodes.
                 $p \leftarrow \text{AVG}(\mathcal{R})$                    $\triangleright$  Estimated fitness of  $\pi(\theta_k)$ 
                 $b \leftarrow \text{AVG}(\mathcal{B})$                    $\triangleright$  Estimated behavioral features of  $\pi(\theta_k)$ 
                if  $\mathcal{P}(b) = \emptyset$   $p > \mathcal{P}(b)$  then
                     $\mathcal{P}(b) \leftarrow p$ ,  $\mathcal{X}(b) \leftarrow \theta_k$            $\triangleright$  Labelled  $i$ 
                Pop the oldest element in  $\mathcal{R}$  and  $\mathcal{B}$            $\triangleright$  To maintain a fixed size
                Perform gradient update to obtain  $\theta_{k+1}$            $\triangleright$  Algorithm-specific gradient update

```

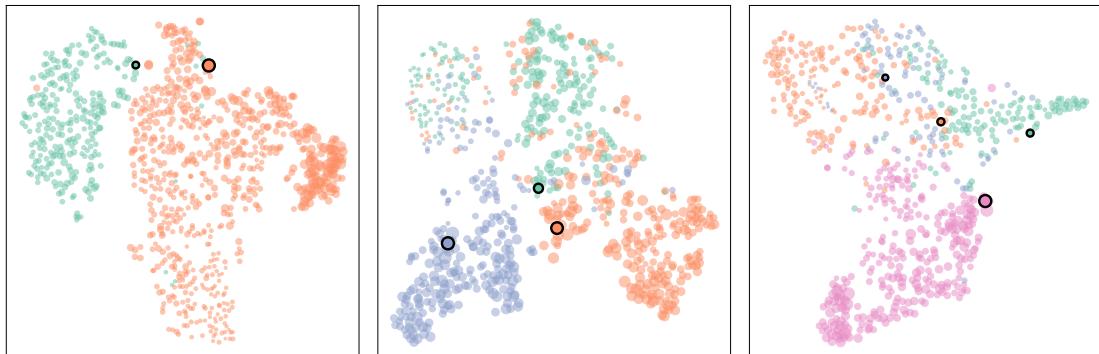
the archive, corresponds to the mean reward across the workers in A2C. Figure 11.2.2 shows the reward per episode for each instance and dimensionality-reduced behavioral space (using PCA) of the policies found by each instance. It is clear to see that the instances have diverged in the behavioral space such that each instance occupy a niche.

11.2.5 Conclusions

Our results confirm that our new approach can result in diverging policies which we believe are useful for learning behavioral repertoires with reinforcement learning. Additionally, this approach could be useful to multi-player games such as StarCraft, and possibly directly applicable to AlphaStar’s league system, to ensure that different policies behave differently while they compete with each other. While Rarity of Events is well suited for this new approach, other intrinsic motivation methods could possibly be applied as well, such as Random Network Distillation (Burda et al., 2018). Future work will explore the robustness of this approach and its ability to produce strong and diverse solutions through reinforcement learning.



(a) Reward per episode of the parallel instances of A2C in Doom which uses our Map-based Reinforcement Learning with Diverging Policies approach. It is noticeable that some niches in the behavioral space (see the plots below) occupy policies which are stronger than policies in other niches.



(b) caption of the four small images

Figure 11.2.2: A dimensionality-reduced behavioral space using PCA of the policies in the archive. The colors correspond to the different instances of A2C which are the same on the plots above. The size of the dots represent the fitness of each solution.

11.3 Blood Bowl: A New Board Game Challenge and Competition for AI

The advancements made in AI for game-playing has led to a common misconception that computers can now play all interesting board games. We thus propose the popular board game Blood Bowl (Games Workshop, 1986) as the next grand board game challenge for AI. The *turn-wise* branching factor of Blood Bowl is several orders of magnitude larger than those of classic board games, and our experiments have shown that a random agent was unable to score any points in 350,000 Blood Bowl matches, making it infeasible to apply vanilla reinforcement learning. In retrospect, recent AI game challenges such as go and most Atari games are in fact particularly suitable for deep reinforcement algorithms as they have image, or image-like, observations as well as a fixed action space. Blood Bowl does not have these properties, as observations consist of both spatial and non-spatial information and the available actions depend on the game state, similarly to the StarCraft II Learning Environment (SC2LE) (Vinyals et al., 2017).

Besides the high complexity of Blood Bowl, the game also requires agents to be highly adaptive. With 24 customizable teams, and millions of possible starting formations, the space of initial game states is much larger than for most complex games. This feature allows Blood Bowl to be a fruitful testbed for adaptive game-playing methods.

To facilitate this challenge, we present the Fantasy Football AI (FFAI) framework, a Python implementation of Blood Bowl with an API for scripted bots and several OpenAI Gym environments, including scaled down versions of Blood Bowl. We also describe a scripted Blood Bowl bot called GrodBot and present preliminary results of training a deep reinforcement learning agent in three smaller variants of Blood Bowl. Additionally, we detail the plans for several future AI competitions using FFAI.

11.3.1 Game Overview

Blood Bowl is a board game designed by Jervis Johnson in 1986 and published by Games Workshop. It is a so-called *fantasy football* game (not to be confused with the American

football manager games) that is played on a board of 26×15 squares mimicking a football / rugby-like pitch (Figure 11.3.1). Two players each control a team of miniatures and the goal is to score the most touchdowns. We will refer to players as *coaches* (or sometimes *teams* in a traditional sports-like manner) and the miniatures on the board as *players*. Each coach can field 11 players on the board where after coaches take turns to move all their players. Players can either move, pass, hand-off, block (attempt to knock down opposing players), blitz (move and block) or foul (stomp on down players) during their *player turn*. When the ball carrier reaches the opponent's end zone their team/coach scores a point.

11.3.1.1 Game Rules

The rules of Blood Bowl have gone through major alterations since the first release in 1986, especially in the 2nd edition (1988), and 3rd edition (1994), where after the rules were periodically updated by the Blood Bowl Rules Committee (2002-2009) resulting in the Living Rulebook 6 (LRB6) (Johnson, 2016). The 2016 Edition of Blood Bowl came with a new ruleset very similar to LRB6. We will at all times refer to the rules in the LRB6 as they have been distributed online for free by Games Workshop and are thus easy to obtain.

An important concept in Blood Bowl that plays a role in most aspects of the game is the *tackle zone*. A player's tackle zone consists of the eight surrounding squares in which it is risky for opponent players to perform actions. Examples of the effects of tackle zones are shown in Figure 11.3.2. Many coaches make use of the famous *Cage* formation, wherein the ball carrier is surrounded by team-mates, usually positioned on the diagonally adjacent squares, such that any opponent trying to block the ball carrier has to make a hard Dodge roll. Players that are prone, hypnotized, etc. do not have a tackle zone. Another important concept is that players can get *knocked down* either as a result of a block or a failed move near opponent players. When a player is knocked down, an armor roll is typically required: two 6-sided dice (D6) are rolled, and if the result is higher than the player's *Armor Value* (AV) attribute its armor is broken. If this occurs, an injury roll is made to determine if the player is *stunned* (becomes prone and inactive for one turn), *knocked out* of the game (may return at next kickoff), or injured/dead (will not return).

11.3.1.2 Turns

A game of Blood Bowl is separated into two halves, each with eight turns for each coach alternating back and forth. The game starts with a kick-off, where each coach sets up a maximum of eleven players on their respective halves of the pitch and the kicking team places the ball on the receiving team's half. Hereafter, the receiving team's turn starts. Within a turn, each player on the team, that is not *stunned*, can take one of six player actions: Move, Pass, Hand-off, Block (attempt to knock down opposing players), Blitz (move and block), or Foul (stomp on prone players). If a player at any point stands in the opponent's endzone with the ball, it is a touchdown; the scoring team scores one point and both teams must set up for kick-off again. When there are no more turns in the first half, each team similarly sets up again for kick-off. When the second half is over, the team with the most touchdowns wins, or if it is a tie, the game ends in a draw. A critical rule in Blood Bowl is the turnover rule. If any player fails a dice roll, such that the acting player falls over, fails to pick up, or catch the ball, the coach's turn ends immediately.

11.3.1.3 Setup

Before each kickoff, the kicking/defending team first sets up a maximum of eleven players on their half of the pitch. It often happens that a coach cannot field eleven players because of injuries. Besides the maximum number of players on the pitch, there must be a maximum of two players on each wing and a minimum of three players on the line of scrimmage. An example of a defensive zone formation (the blue team) and an offensive wing formation (the red team) is shown in Figure 11.3.1. Notice that the offensive team has one player positioned to get the kicked ball in the backfield. Coaches usually have their own repertoire of formations that are tailored to their strategy and playing style.

11.3.1.4 Movement and Dodging

Moving players into positions that give tactical advantage is perhaps the most important part of Blood Bowl. Unless a player is stunned or takes a block action, it is allowed to move a number of squares equal to its *Movement Allowance* (MA) attribute. For example,

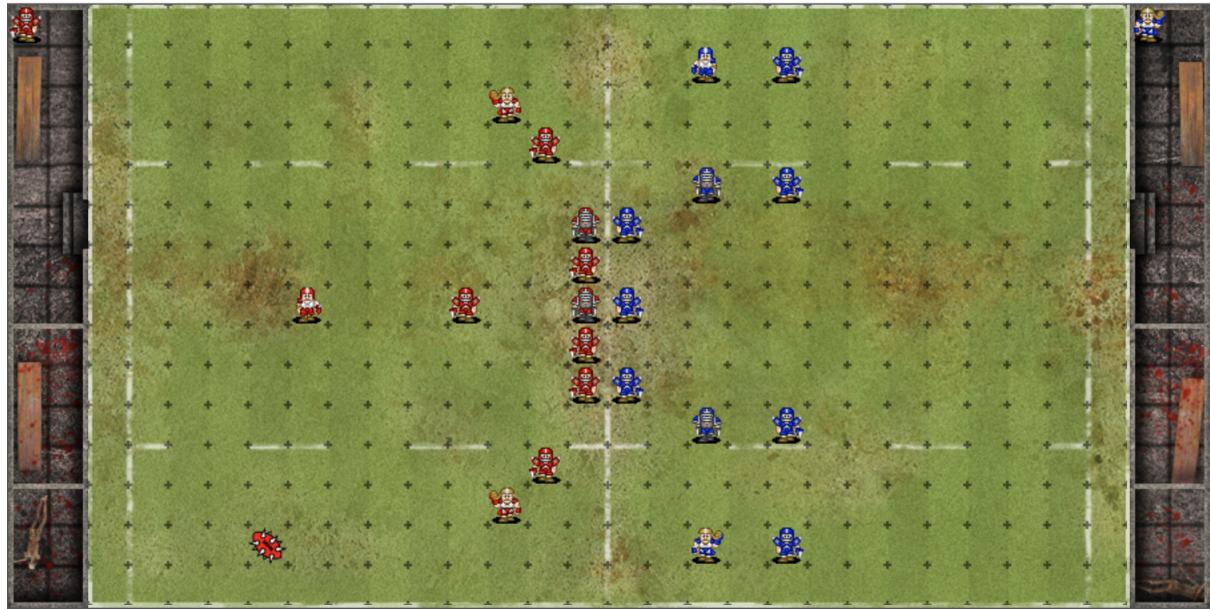


Figure 11.3.1: The game board in FFAI after both teams have set up. The blue team just kicked the ball to the red team and assumed a defensive cover formation, while the red team is in an offensive wedge formation, protecting the wings against blitzing opponents.

a Halfling can only move five squares while a Wood Elf Catcher can move eight squares. To move from a square that is within an opponent tackle zone, a player has to pass a Dodge roll that depends on its *Agility* (AG) attribute (thus also called an Agility roll). The higher the agility, the higher the chance is for the roll to succeed. Moving from an opponent tackle zone into other opponent tackle zones add further modifiers to the roll. An example of a tough dodging situation for a red Catcher is shown in Figure 11.3.2b. Also, notice in Figure 11.3.1 how the defending blue team has positioned itself such that the red team cannot easily run through its cover defense. When a player has moved the number of squares equal to its MA, it has the option to make up to two risky *Going For It* (GFI) moves, allowing the player to move an additional square on a roll of 2 or more. Players that fail a dodge or GFI roll are knocked down. If a player moves to a square with a ball an Agility roll must be made to attempt to pick it up.

11.3.1.5 Blocking, Blitzing and Fouling

Another important part of Blood Bowl is blocking. In fact, some coaches focus more on blocking, attempting to knock out opponent players, than worrying about the ball. A player taking the Block action can perform a block on an adjacent standing opponent. If the two players have the same value in their *Strength* (ST) attribute, one block die

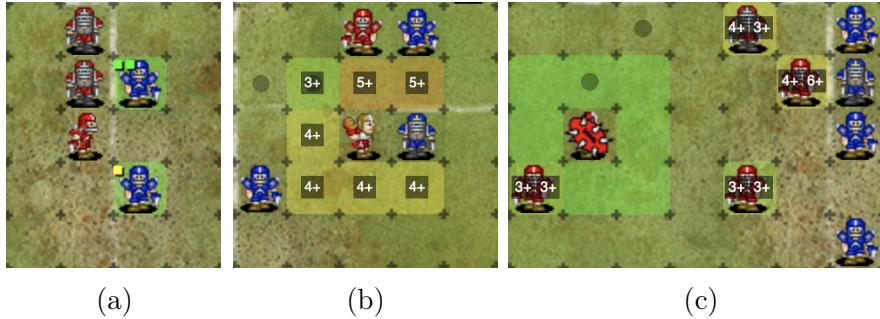


Figure 11.3.2: Effects of tackle zones on different dice rolls visualized in FFAI. (a) The red Lineman can block one of two blue Linemen. When blocking the top-most blue Lineman, two block dice are used due to the two assisting red Blitzers in the top, while it only gets one block die when blocking the bottom-most Lineman. (b) The red Catcher can move to seven adjacent free squares. Since it is already in a blue player’s tackle zone, a Dodge roll is required. The player has $AG = 3$, which makes the dodge successful on a roll of $3+$. However, this number is increased by one for each opponent tackle zone covering the target square. (c) A red human Thrower can attempt to pass the ball to four nearby team-mates. Two of these team-mates are in the *quick pass* range where a pass will be accurate on a roll of $3+$ while the other two are in the *short pass* range requiring a roll of $4+$. If the pass is accurate, the ball can be caught on $3+$ with an additional modifier for each opponent tackle zone covering the catching player. Note that (b) and (c) show the required dice rolls after modifiers have been added.

is rolled. If one player is stronger than the other, two (and sometimes three) block dice are rolled, where after the stronger player’s coach must choose one of the rolled dice to take effect. A block die has six sides with the following outcomes: 1) Attacker Down: the attacking player is knocked down, 2) Both Down: both players are knocked down unless they have the Block skill, 3-4) Push: a player is pushed one square back, 5) Defender Stumbles: the defender is pushed and knocked down unless it has the Dodge skill. 6) Defender Down: the defender is pushed and knocked down. A player on the blocking team can assist the block if it is inside the blocked player’s tackle zone but not in any other tackle zones of the blocked player’s team. Likewise, an opponent player can assist the blocked player if it is within the blocking player’s tackle zone but not in any other tackle zone of the blocking player’s team. Each assist on each side adds one to the attacker or defenders ST. Figure 11.3.2a shows an example where a red lineman can get two assists by blocking the top-most blue lineman while the opponent gets a single assist, resulting in two block dice. One Blitz action can be taken each turn, which corresponds to a Move action wherein the player can perform one Block action. Similarly, one Foul action can be made each turn which consists of a Move action followed by a foul. A foul (kicking a player that is down) can only be done to prone players and do not require a block

roll. Instead, an armor roll and eventually an injury roll is made directly where assists are applied to modify the armor roll. However, kicking players that are down is strictly against the rules, and thus if either the armor roll or the injury roll is doubles the referee spots the foul and sends the fouling player out of the game.

11.3.1.6 Passing and Hand-offs

A player taking the Pass action can first move as if it was a Move action and thereafter pass the ball. A pass consists of three parts. First, the passer declares a target square. Then the opponent coach declares a player that stands between the passer and the target (if any) who will attempt to intercept the ball. Interceptions are hard and require an Agility roll with additional modifiers. If the interception is successful, the intercepting player catches the ball and it is a turnover. If not, the passer must make an Agility roll that depends on the range to the target and the number of opponent tackle zones the passer is in. If that fails, the ball is either fumbled to a nearby square or becomes inaccurate and is scattered three random squares from the target. Otherwise, the ball lands on the target square. If there is a player on the square the ball lands on, that player must finally attempt to catch it by passing an Agility roll. If the pass does not result in a catch by a team-mate it is a turnover. An example of a pass situation is shown in Figure 11.3.2c where both the Agility roll for passing and catching is shown for each friendly player. A Hand-off action is similar to a Pass action with a few differences. The ball can only be “passed” to an adjacent square with a team-mate, the “pass” is always accurate, and it cannot be intercepted. A maximum of one pass and one hand-off actions can be performed each turn.

11.3.1.7 Teams and Races

The board game Blood Bowl comes with two teams: Humans and Orcs. The Human team does not have any particular strengths nor weaknesses while the Orcs are stronger, have more armor, are slower, and less agile. Each team can be built from a restricted set of positional players (different types of players) with different attributes and skills; any Human team can have up to 16 Linemen, 4 Catchers, 2 Throwers, 4 Blitzers, and 1 Ogre.

Qty	Title	MA	ST	AG	AV	Skills
0-16	Linemen	6	3	3	8	
0-4	Catchers	8	2	3	7	Dodge, Catch
0-2	Throwers	6	3	3	8	Sure Hands, Pass
0-4	Blitzers	7	3	3	8	Block
0-1	Ogre	5	5	2	9	Loner, Bone-head, Mighty Blow, Thick Skull Throw Team-Mate

Table 11.3.1: The positional players allowed on a Human team. MA=Movement Allowance, ST=Strength, AG=Agility, AV=Armour value. The Ogre is a special type of player called *Big Guys* and its skills are not explained here.

Table 11.3.1 shows each positional player's attributes and skills. There are 24 teams in LRB6, all with their own strengths and weaknesses. In competitive play, teams are built starting with a *treasury* of 1 million or 1.1 million gold coins, where after the roster is created by buying from the available positional players.

11.3.2 Competitive Play

Blood Bowl is very popular among competitive tabletop gamers with 161,080 recorded tabletop tournament matches registered by NAF¹, the international association of Blood Bowl coaches and organizers of the Blood Bowl world cup. NAF also maintains the ranking of competitive Blood Bowl coaches based on their participation in tournaments². Not all teams are balanced to be equally good. For example, the Undead team has the record low loss rate of 31.9% and the Ogres team has the highest loss rate of 58.6%.

Blood Bowl is also played competitively in leagues where players (the miniatures) are rewarded experience points for completing passes, scoring touchdowns, and causing casualties. Players can then level up when enough experience points are reached, resulting in new skills or attribute increases. Players can also gain permanent injuries that reduce their value. The coach can in-between matches decide to fire old players and hire new players. A team thus progresses throughout a league, making each league match a unique experience.

A video game adaptation with 3D graphics was released by Cyanide Studios in 2009 which features online play. The video game includes an AI which is far from human-level;

¹http://naf.talkfantasyfootball.org/total_for_all_competitions.html

²<https://www.thenaf.net/rankings/glicko-rankings/>

it presumably follows a set of scripted rules combined with a pathfinding algorithm. FUMBBL (the acronym combines the football term Fumble with BBL; Blood Bowl League) is a community-driven online league with more than 2,500,000 recorded matches³. Matches in FUMBBL are played using an unofficial game client with simple 2D graphics⁴. The source code is kept secret by the developers to prevent cheating.

11.3.3 Game Variants

Blood Bowl has several variants using extensions of the core rules⁵. Dungeon Bowl (Games Workshop, 1988) is, as the name suggests, Blood Bowl in a dungeon. There are no fixed rules on the structure of this dungeon, only that it is grid-based, has setup zones and endzones for each team. The rules also include trapped chests, teleporters, lava pits, monsters, etc. that makes every game unique. Other variants exist, such as Death Bowl that allows four teams to play on a cross-shaped pitch with two balls. Street Bowl, Blood Bowl Sevens, Blitz Bowl are smaller variants of Blood Bowl, with a smaller game board and fewer players.

11.3.4 Characteristics

This section contains a short analysis of Blood Bowl using the game characteristic dimensions defined by Yannakakis and Togelius (2018a). These dimensions include observability (perfect or non-perfect information), stochasticity (deterministic or stochastic), and time granularity (turn-based or real-time). Additionally, we consider at the state representation.

Blood Bowl has **perfect information** as the board state is fully observable and coaches have no hidden information. The game has an optional rule that allows coaches to have secret *special play* cards but these are rarely used in competitions.

Blood Bowl is **stochastic** as most of the interesting actions require dice rolls to succeed. Coaches have a limited number of re-roll tokens they can use to re-roll a failed roll.

³<https://fumbbl.com/p/stats>

⁴<https://fumbbl.com/>

⁵<https://www.thenaf.net/blood-bowl/variants/>

Experienced Blood Bowl coaches usually start their turn with safe actions that require easy or no dice rolls and postpone risky actions to the end of their turn.

Blood Bowl is a **turn-based** and a **multi-action** game as coaches take turns to move *multiple* players on the board. Other multi-action games that have been the basis for research on AI methods are Arimaa (Syed and Syed, 2003) and Hero Academy (Robot Entertainment, 2012) (Justesen et al., 2017). What makes Blood Bowl even more complicated than these is that players can be moved several steps each turn, making it a *nested* multi-action game. A turn consists of multiple player actions which consists of multiple individual actions. Blood Bowl can be played with a maximum time limit for each turn, usually of two to four minutes per turn, and is sometimes enforced in competitive play.

The **state representation** is especially relevant for deep learning methods. Go and most Atari games are particularly suitable for deep learning methods as they have an image, or image-like, state representation as well as a fixed action space, which is not the case for Blood Bowl. Here, the state is represented by players on the board, each with multiple dimensions (player attributes, whether it is standing, knocked down etc.), a dugout for both coaches with reserve, knocked out, and injured players, weather conditions, and occasionally information on a dice roll, etc. The state representation thus consists of both multi-layered spatial features and non-spatial features very similar to SC2LE (Vinyals et al., 2017). Another similarity shared with SC2LE is that the **action space** varies between steps.

11.3.5 Complexity

To get a grasp of the complexity of Blood Bowl we will first attempt to estimate the branching factor analytically and then empirically measure it relative to the game-play of two baseline agents in FFAI.

The size of the action space in Blood Bowl depends on the state and varies between 1 and 395. Sometimes the coach has to select between a few dice results and other times one of the 395 squares on the board to kick or pass the ball to. In most situations, the coach has to select one of up to eight adjacent squares to move a player to or select

one of six different action types for a player. A reasonable estimation of the average *step-wise* branching factor is 10 and the average number of steps in a complete player action could be around 5. The number of unique action sequences for just one player action is thus 10^5 . With 10 players able to take actions in a fixed order, the average *turn-wise* branching factor is $10^{5 \times 10} = 10^{50}$. As players can take actions any order, this is a lower-bound estimate. In comparison, the *turn-wise* branching factor is around 30 in chess and 300 in go. Long action sequences with sparse scores make both search algorithms and reinforcement learning harder to apply. A game of Blood Bowl consists of approximately $10 \times 5 \times 32 = 1600$ steps, using the previous estimations multiplied by 32 turns.

We performed an experiment in FFAI, simulating 100 games with two agents that samples actions uniformly. 169.8 actions were taken on average by each agent (10.6 actions per turn) with a *step-wise* branching factor of 29.8. This gives us a *turn-wise* branching factor of around 10^{30} . Randomly sampled actions are not representative for human play and we thus also performed an experiment with the scripted bot GrodBot (which is described in Section 11.3.7.2). However, this experiment only includes 10 games because GrodBot runs much slower than the random bot. Here, the measured average *step-wise* branching factor was 17.2 with 41.6 actions per turn and thus the *turn-wise* branching factor is around $17^{42} = 4.8 \times 10^{51}$.

The branching factor was estimated for one setup of Blood Bowl with two simple Human teams while the game has many possible setups; a coach can choose among 24 races to play and has the option to customize the roster. Especially in leagues, it is typical to play against a team with a unique combination of players that the coach has never encountered before, which contrasts sharply with classic board games that have just one or a few initial game states. We describe these different setups in more details in Section 11.3.9.2.

11.3.6 Motivation

There are several motivations for proposing Blood Bowl as a new AI challenge. First of all, Blood Bowl is, due to its high complexity, significantly harder for AIs to play than classic board games. Having a hard task that is focused on tactical planning, long-term planning, and careful risk management, without dealing with the array of challenges in

real-time video games, is of high value. We can effectively do research on an environment that is fast, easy modifiable and lies somewhere in the large gap between go and StarCraft in terms of complexity. Exploring video games, which are more similar to real-world problems, are obviously very useful to use as AI test beds. However, by staying in the realm of board games, it is easier to compare the cognitive level of AI systems to humans, which has been argued to be impossible in video games (Canaan et al., 2019a). Another main motivation for proposing Blood Bowl is its expressiveness. Every game of Blood Bowl is never the same. Not only because it quickly branches out to new situations, but because the number of possible initial board states is astronomical, taking into account the number of permutations of players on each of the 24 teams and the relatively non-restricted freedom to set up the players on the board. Additionally, Dungeon bowl introduces the possibility of playing in new and unseen dungeons. These challenges are not in classic board games.

Game Engine Existing Blood Bowl implementations are closed source and do not have an AI interface. Thus, we have developed our own game engine named the *Fantasy Football AI* (FFAI) client⁶. Figure 11.3.1 shows a screenshot of a part of the user interface in FFAI with our own 2D graphics⁷. FFAI is implemented in Python allowing a simple way to interface with popular machine learning libraries. We considered implementing the engine in C++ with a Python interface on top, but the state updates in Blood Bowl are fairly simple, and thus fast, even in Python. FFAI implements the Open AI Gym interface for reinforcement learning algorithms (Brockman et al., 2016b). The observation object includes several spatial feature layers as well as several non-spatial features. Aside from the reinforcement learning interface, the engine itself can be used as a forward model. One game step with a randomly sampled action takes on average 0.9ms on a regular laptop and a complete game takes on average 0.16s. While the forward model is rather fast itself, the game state object is object-oriented and thus slow to clone. Tree search is possible, but not very feasible using the current data structure. A simpler array-based data structure could be implemented while it would complicate the game logic. FFAI also comes with a simple web application that implements a user interface for humans to play against other humans, online or local, or against bots.

⁶<https://github.com/njustesen/ffai>

⁷Nicholas Kelsch has the copyright to the player icons.

11.3.6.1 Competition Functionalities

To support AI competitions, FFAI comes with built-in functionalities to handle a sequence of games between two bots, restricting and penalizing hanging or crashing bots. FFAI can be configured with time limits for a complete turn, a single action taking place in the opponent’s turn (such as block die selection or skill usage - we refer to this as an *opponent procedure*), initialization, and termination procedures. The game waits for the acting agent to return an action where after it can be penalized in several ways:

- **Delay of Game:** the acting agent fails to end its turn, or returns an action during an opponent procedure, within the time limit. If a Delay of Game penalty occurs, actions are randomly taken by the system until the turn or opponent procedure ends.
- **No Response:** if there is a Delay of Game and the agent responded later than a specified disqualification threshold, it will be disqualified directly.
- **Crash:** if the acting agent crashed, it will be disqualified directly. In the unexpected case that FFAI crashed during an internal procedure, the game will end in draw.

FFAI also saves a report of the competition with aggregated results and a list of individual game results.

11.3.6.2 Replays

We are currently working on a module for FFAI to replay matches from FUMBBL. This would enable us to extract state and action pairs from the 2,500,000 available FUMBBL matches which can be used for imitation learning. Here, the goal is to learn a policy function that maps states to actions using traditional supervised learning techniques, thus imitating the playing styles expressed in the dataset. This technique has been applied to several games, including StarCraft (Justesen and Risi, 2017b), and Candy Crush Saga (Gudmundsson et al., 2018). Currently, individual replays can be fetched from the FUMBBL API. However, this process is slow and we are thus planning to release publicly available data sets in the future.

11.3.7 Baseline Agents

11.3.7.1 Random Agent

FFAI comes with an agent that samples actions from a uniform distribution. To be more precise, it first samples a legal action type uniformly and if that action type requires a position as well, such as a movement action, a legal position is sampled uniformly as well. When setting up, it randomly selects between two predefined formations on the offense and two on the defense. In 350,000 games against itself, no touchdowns were scored and thus all games ended in a draw. Games in which random agents practically never score points or wins are extremely challenging for many algorithms such as Monte Carlo Tree Search and Q-learning as they rely on random exploration. Thus, we do not expect vanilla implementations of such algorithms to score any points either.

11.3.7.2 GrodBot

GrodBot⁸ is a scripted bot with an estimated skill level slightly higher than a rookie player. GrodBot repeatedly evaluates all possible moves for all players. The move with the highest score is then executed at each step in the game. The *end turn* action is always assigned a score of zero so that the turn will be passed when no positive actions are left. The scoring of moves first applies pathfinding to identify the set of possible squares a player can reach along with a probability of success for moving there. The probability of successfully moving to each reachable square is thus the maximum probability of all possible paths leading to it; we always follow the optimal path. GrodBot maintains a list of possible purposes, their values, and some rules that apply modifiers to the values (e.g. it is preferred to pass the ball to an unused team-mate close to the end-zone rather than a used team-mate in the backfield). The final score for a move is then computed by multiplying the probability of success with the modified value of the move. Each purpose was initially valued using an experienced Blood Bowl player's intuition where after it was gradually improved by playing GrodBot against itself and tune the values in an ad-hoc manner. GrodBot's move purposes are:

⁸<https://github.com/njustesen/ffai/blob/master/examples/grodbot.py>

- **Move** to a receiving position
- **Move** to the ball
- **Move** to a player holding the ball somewhere safe or towards the end zone
- **Move** to a player to form a defensive sweep position
- **Move** to a player to form a defensive screening position
- **Move** to a player to form an offensive screening position
- **Move** to a player to form a cage around the ball
- **Move** to a player to exert an opponent tackle zone
- **Foul** an opponent on the ground
- **Blitz** an opponent (preferably the ball carrier)
- **Hand-off** the ball to a team-mate
- **Pass** the ball to a team-mate
- **Block** an adjacent opponent using two block dice

The probability of a successful move in Blood Bowl depends on a number of different factors, such as the number of tackle zones it moves through, or whether or not the player needs to attempt a GFI. We do not believe there is a useful admissible heuristic for path finding in Blood Bowl and we therefore apply Dijkstra's algorithm. The cost function is probability-based in the interval $[0,1]$ and is not additive. For sequences of moves, their costs are simply multiplied. In general, if $C(\{s_1, s_2\})$ is the cost of the path going from square s_1 to square s_2 and the following cost of going from s_2 to s_3 is $C(\{s_2, s_3\})$, then the total cost of the path from s_1 to s_3 is $C(\{s_1, s_2, s_3\}) = 1 - (1 - C(\{s_1, s_2\})) \times (1 - C(\{s_2, s_3\}))$. The probabilities of success are multiplied and subtract from 1 to convert them back to the probability of failure, i.e. the cos). GrodBot's path-finding functionality has become a part of FFAI and can easily be used by other bots.

We tested GrodBot in ten games against the random baseline (five as the away team and five as the home team) where both agents controlled the basic human team. Grodbot won all ten games with an average of 4.2 touchdowns and 1 inflicted casualty a game. The

random agent scores 0 touchdowns and had 0.6 inflicted casualties per game. The number of inflicted causalities also include casualties inflicted by the crowded, failed dodges, etc.

11.3.8 FFAI Gym

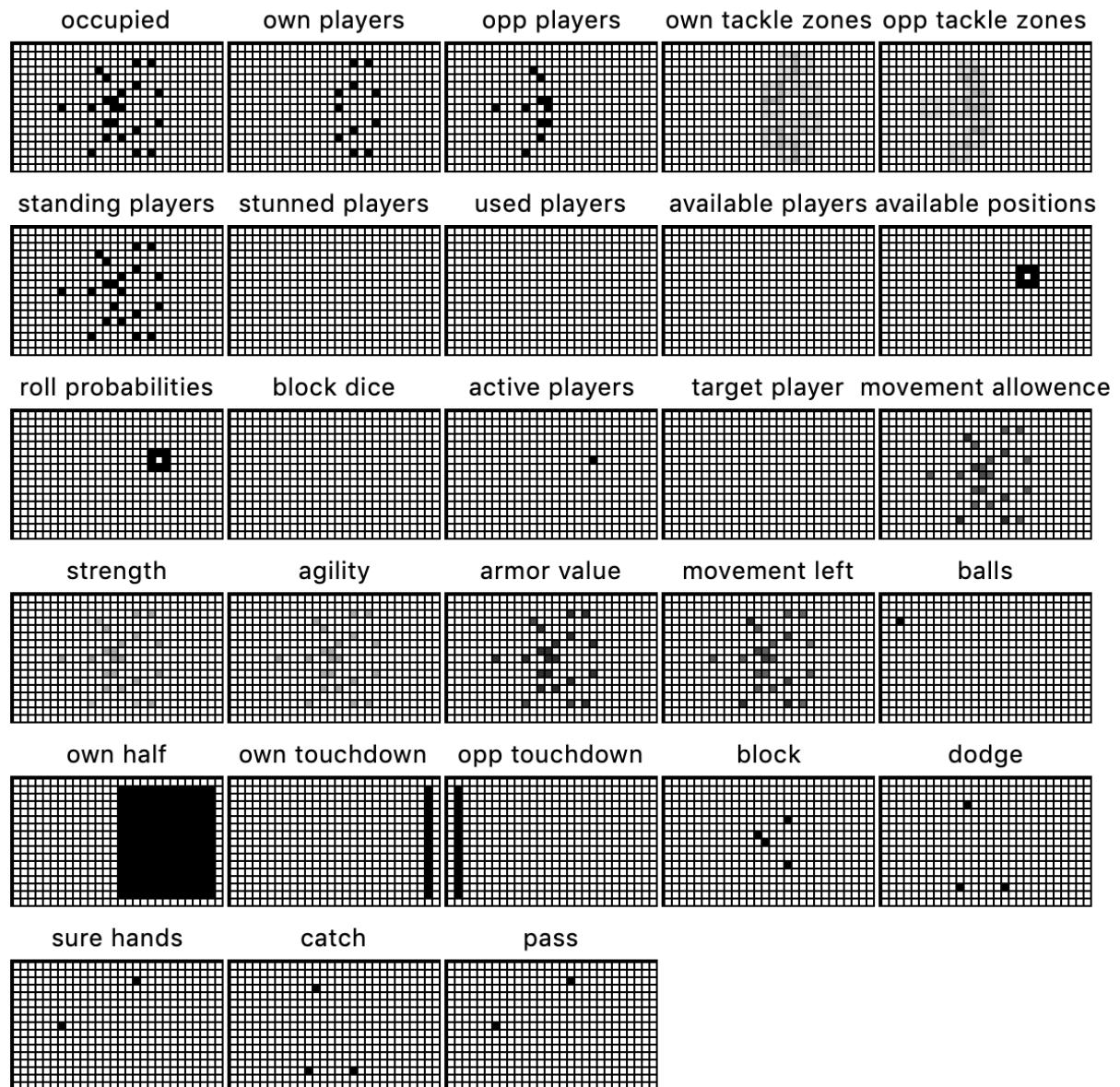


Figure 11.3.3: The 28 spatial feature layers in the FFAI Gym observation. Each layer has a name, which is shown above the visualization. Here, black squares represent a value of 1 and white squares represent a value of 0.

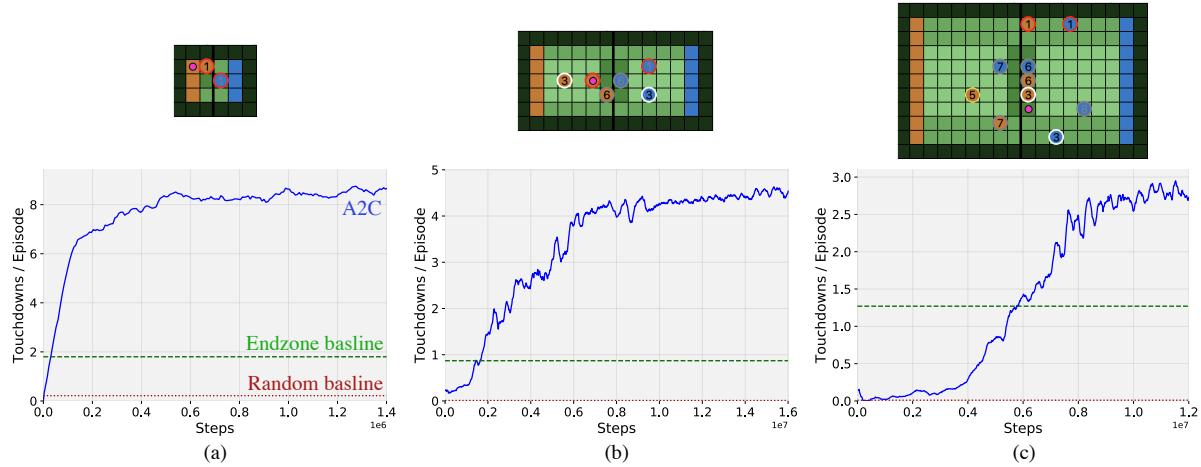


Figure 11.3.4: Touchdowns per episode of A2C during training in the three smallest FFAI Gym environments: (a) FFAI-1-v1, (b) FFAI-1-v3, and (c) FFAI-1-v5, which features 1, 3, and 5 players on the pitch for each team. Simple renderings of each environment is shown above the plots. The agent plays against an agent that takes random actions. The touchdowns are smoothed over 200,000 steps. The red and green lines show the touchdowns per episode the Random and Endzone baselines. We see that A2C learns a policy that is better than the baselines in all three environments.

11.3.8.1 The Gym Interface

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms by implementing a simple interface to handle communication between the agent and the environment (Brockman et al., 2016b). FFAI comes with several Gym environments, one with the original game board and 11 players on each side and several smaller variants (see Figure 11.3.4). In these environments, the agent faces the random agent (see Section 11.3.7.1), which probably will lead to sub-optimal behaviors against stronger agents. It is, however, easy to modify the FFAI Gym implementation to e.g. support self-play, which is an effective learning method in two-player board games (Silver et al., 2018). Both agents control a basic Human team, but this can also be modified easily. For simplicity, the agent always plays as the home team, playing on the right side of the field. This limitation can easily be resolved after training by simply flipping the board and swapping a few values in the observations, if it has to play as the away team.

The observation space is split into three parts; (1) a vector containing 50 non-spatial normalized values representing the score, turn number, half, re-rolls left, etc. (2) a one-hot encoded vector representing the *procedure* (phase) the game is in, e.g. *Setup*, *Turn*, *PlayerAction*, and *Push*, and (3) a set of 2D feature layers, each with one value per square

on the board such as whether a square is occupied by the ball or by a player controlled by the agent or the opponent. All 28 features layers can be seen in Figure 11.3.3 for the same game state. The design of the observation space is very similar to that of SC2LE (Vinyals et al., 2017) as it also has spatial and non-spatial components.

The action space has two parts: (1) the type of the action to perform among 31 choices such as *Block*, *Select Both Down* (block dice result), *Use Reroll*, *Heads* (for the coin toss), etc., and (2) a position, which is only relevant for some action types, e.g. a *Block* action requires a position to determine which opponent player to block.

The built-in reward function only gives a reward of 1 when winning the game, -1 when losing, and 0 otherwise. The complete game state is, however, accessible in the environment to allow for reward shaping.

11.3.8.2 Preliminary Results with A2C

We applied the deep reinforcement learning algorithm synchronous advantage actor-critic (Mnih et al., 2016) (A2C) on the three smallest/easiest Gym environments in FFAI, which has a board of 4×3 , 12×5 , and 16×9 squares with 1, 3, and 5 fielded players on each team, respectively (see Figure 11.3.4). On the two smaller environments, we used only touchdowns as rewards. However, on the larger one, touchdowns are rarely obtained, and thus the following reward shaping was used (rewards are shown in brackets): winning (5), touchdown (4), knock out opponent (3), push opponent into the crowd (3), completion (3), cause opponent fumble (2), knock down opponent (2), handoff (2), opponent failing a dodge (1), moving ball carrier closer to the opponent endzone (1), and gaining the ball (1).

We use a convolutional neural network with an additional fully-connected input stream to handle the non-spatial features. We use all the 28 spatial feature layers from FFAI for the convolutional input stream and all the 50 non-spatial features (in the current version of FFAI there are now 50 non-spatial features) for the fully-connected input stream. The convolutional stream has two layers, one with 16 filters of size 3×3 (mimicking the tackle zone area around a player), and a second layer with 32 filters of size 2×2 . Both layers use stride 1 and padding, preserving the spatial structure of the observation. The non-spatial

input stream consists of a single fully connected layer of size 25 which is concatenated with the flattened convolutional stream followed by a single fully-connected layer and two output streams for the critic and the policy. When actions are sampled from the policy distribution, we use a masking technique to filter out illegal actions before applying softmax.

A2C was configured to use eight parallel workers, a learning rate of 0.001, a discount factor $\gamma = 0.99$, entropy coefficient of 0.01, value loss coefficient of 0.5, max. gradient normalization of 0.5, and steps per update $t_{max} = 10$. We used the RMSprop optimizer tieleman2012lecture with $\epsilon = 1e-8$ and $\alpha = 0.99$. Figure 11.3.4 shows the touchdowns per episode during training, where it reached an average around 8/5/3 in the three environments. A random baseline agent scores around 0.2/0.0/0.0 touchdowns per episode, and a simple *endzone* baseline that always goes toward the endzone with the ball scores around 1.8/0.9/1.3 touchdowns/episode. The code is made available online⁹. When observing the trained agents play¹⁰ we can see that it plays quite well but not optimal.

11.3.9 AI Competition

11.3.9.1 Bot Bowl I

Based on our analysis of Blood Bowl we believe it can offer a new and exciting testbed for AI due to the high complexity of the game while still resembling many classic board games. To encourage researchers and hobbyists to explore algorithms that can play Blood Bowl, we are organizing an annual AI competition with progressively more difficult challenges using FFAI. The first competition, Bot Bowl I¹¹, was held at the IEEE Conference on Games in August 2019. The competition allowed all types of methods, including controllers that are scripted, search-based, neural network-based, as well as hybrids combining several of these approaches. Bot Bowl only featured a pre-fixed Human team, to keep the format of the first competition simple. The submitted agents played in a round-robin tournament with ten matches against each other and the two best agents (with the most wins) played

⁹https://github.com/lasseuth1/blood_bowl2/

¹⁰<https://youtu.be/xk4AMutuyaA> & <https://youtu.be/7xmwB8hn3qM>

¹¹<https://bot-bowl.com/bot-bowl/>

in a final series. The competition had four entries and we added our random baseline agent into the mix. Two entries were extensions of our preliminary results using A2C (Section 11.3.8.2) where one of the agents achieved an impressive five wins and five ties against the random baseline. The two finalists were GrodBot (11.3.7.2) and EvoGrod (an extension of GrodBot with evolved constants), and here GrodBot won. There was a clear difference in performance between the scripted bots and the reinforcement learning bots. We hope to see even more entries in Bot Bowl II.

11.3.9.2 Future Competition Formats

Future competitions can be extended in several exciting directions that will add further challenges for the competitors. Four concrete ideas are:

- **All teams:** The most obvious extension of the current competition format is to allow all teams. This addition will require a higher flexibility with less hard-coded strategies and formations.
- **Custom rosters:** To align our competition format with real Blood Bowl tournaments, competitors should be allowed to buy players for their roster from a starting treasury. As agents need to generalize to possibly unseen teams, it will further require agents to rely less on hard-coded strategies.
- **League format:** In Blood Bowl leagues, players get star player points and can gain new skills when leveling up between games. Additionally, players can get permanent injuries. This format thus adds a meta-game, as agents must manage their team in-between games, selecting new skills when players level up, hire new players, etc.
- **Dungeon Bowl:** Instead of playing on the same board, competitions could be played on procedurally generated dungeons using the official Dungeon Bowl rules while building on the numerous algorithms for dungeon generation in the procedural content generation literature (Van Der Linden et al., 2014; Liapis et al., 2015; Shaker et al., 2016). By playing on procedurally generated *unseen* dungeons in the competition, agents need to either use extensive forward search or learn a policy on a vast training set of dungeons in a similar fashion to our approaches in Chapter 8.

These four suggestions for future competitions add new challenges that all require agents

to generalize to new game variations (such as different levels), something that current reinforcement learning algorithms struggle with.

Chapter 12

Conclusions

This dissertation explored a new perspective on the challenges of creating artificial game-playing agents in complex games. This perspective is based on three properties of adaptivity that I believe any player must have to master the most complex games that involve multiple players. These properties are (see the introduction for longer descriptions): (1) intra-game adaptivity: the ability to adapt to opponent strategies within a game, (2) inter-game adaptivity: the ability to intelligently switch strategy in-between games, and (3) generality: the ability to generalize to many different, and most likely unseen, variations of the game (e.g. different levels).

I show that intra-game adaptivity can be achieved in the real-time strategy game StarCraft through a continual variant of evolutionary planning (Chapter 4) or imitation learning (Chapter 5). This was achieved using a modular approach, in which the two approaches only deal with high-level build-order decisions and low-level tasks are performed by scripted modules. This modular approach was later taken by Tang et al. (2018) and Sun et al. (2018), in which reinforcement learning was applied, and one the resulting bot by Sun et al. (2018) was able to outperform the best (cheating) bots developed by Blizzard Entertainment (the developers of StarCraft).

I also presented an extension to the modular approach based on imitation learning called Behavioral Repertoire Imitation Learning (BRIL) in which a large repertoire of diverse policies are learned from human demonstrations (Chapter 6). Our experiments demonstrate how BRIL can be used for inter-game adaptation by switching between

different policies in-between games. I also present preliminary work on an extension to MAP-Elites for noisy domains, such as games, which could be useful for learning a large set of diverse and strong policies without human demonstrations (Section 11.1). MAP-Elites is, however, very sample inefficient and thus I have explored how several parallel instances of a reinforcement learning algorithm can efficiently learn diverse behaviors by forcing them to diverge in a behavioral space (Section 11.2).

Reinforcement learning is a promising approach to learning strong policies in games, and they seem to scale well to large state and action spaces as more computational resources are applied (Vinyals et al., 2019; OpenAI, 2018a). Still, these methods are very data inefficient, and small research or game development teams may not have access to such resources. To overcome sparse rewards with limited domain knowledge and tweaking by the experimenter, I have presented an automatic reward shaping approach called Rarity of Events (RoE), in which the rewards of a set of predefined events, dynamically adjusts to match the rarity of their occurrence, such that rare events are more rewarding (Chapter 7). Several scenarios with sparse rewards were solved faster with RoE, and the learned policies expressed better generality in a number of test scenarios as they tend to express a balanced behavior. RoE can thus be seen as a form of regularization technique for reinforcement learning.

I have also demonstrated that reinforcement learning overfits when trained on just one or a few game levels (Chapter 8). This is perhaps not surprising but it highlights how a reinforcement learning trained on a single level, such as AlphaStar (Vinyals et al., 2019) and OpenAI Five (OpenAI, 2018a), may not learn general concepts about the game but rather overfitted behaviors that are tied to the particular configuration used in training. I argue that to fully master a game, one must be able to compete similarly to how humans compete in the game, which typically involves series of matches against multiple players and with different variations of the game (e.g. different levels/maps) for each match (Chapter 9). To deal with these challenges I presented a new framework for reinforcement learning in which a new level is generated for each episode (Chapter 8). Using a constructive level generator, we were able to achieve general policies with reasonable test performance on unseen levels. In games with sparse rewards, we further had to dynamically adjust the difficulty of the generated levels to match the performance

of the learning agent. Using this approach, we were able to achieve strong policies in games that were otherwise infeasible to learn due to their sparse rewards. Our results also highlight the challenges of generating levels that allows the agents to generalize to a different distribution of levels.

Finally, I presented Blood Bowl as a new board game challenge and competition for AI (Chapter 11.3). Blood Bowl is one of the most complex board games that exist and it demonstrates that board games are not yet passé for AI research. I have implemented the game rules in python and developed an interface for scripted bots as well as an OpenAI Gym implementation for reinforcement learning algorithms. I aim to run and extend this competition annually in the coming years and I hope it will become useful to future research on adaptive game-playing agents in complex games.

This dissertation has surveyed state-of-the-art within research in game-playing algorithms and presented promising new approaches to the problem of achieving artificial game-playing agents in complex games. Several approaches have been studied for each property of adaptivity in the context of games with sparse rewards (see and overview in Table 1.0.1 in the introduction). It remains a challenge for the future, how to best combine these approaches to achieve strong game-playing agents with all four properties that can outperform the best human players in fair competitions, in the most difficult games.

Bibliography

- Aarseth, E. (2014a). Ludology. In Wolf, M. J. P., editor, *The Routledge Companion to Video Game Studies*. Routledge.
- Aarseth, E. (2014b). Ontology. In Wolf, M. J. P., editor, *The Routledge companion to video game studies*, pages 510–518. Routledge.
- Aarseth, E. and Calleja, G. (2015). The word game: The ontology of an undefinable object. In *Foundations of Digital Games Conference*.
- Aarseth, E., Smedstad, S. M., and Sunnanå, L. (2003). A multidimensional typology of games. In *DiGRA Conference*.
- Andrychowicz, M., Crow, D., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, O. P., and Zaremba, W. (2017). Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5055–5065.
- Arjoranta, J. (2014). Game definitions: A wittgensteinian approach. *Game Studies: the international journal of computer game research*, 14.
- Asada, M., Veloso, M. M., Tambe, M., Noda, I., Kitano, H., and Kraetzschmar, G. K. (2000). Overview of robocup-98. *AI magazine*, 21(1):9.
- Avedon, E. M. and Sutton-Smith, B. (1971). *The study of games*. John Wiley & Sons.
- Bakker, P. and Kuniyoshi, Y. (1996). Robot see, robot do: An overview of robot imitation. In *AISB96 Workshop on Learning in Robots and Animals*, pages 3–11.
- Barriga, N. A., Stanesco, M., and Buro, M. (2017). Combining strategic learning and tactical search in real-time strategy games.
- Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq,

- A., Green, S., Valdés, V., Sadik, A., et al. (2016). Deepmind lab. *arXiv preprint arXiv:1612.03801*.
- Bellemare, M., Naddaf, Y., Veness, J., and Bowling, M. (2015). The arcade learning environment: An evaluation platform for general agents. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.
- Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479.
- Bellemare, M. G., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*.
- Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.
- Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM.
- Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.
- Bhatti, S., Desmaison, A., Miksik, O., Nardelli, N., Siddharth, N., and Torr, P. H. (2016). Playing doom with SLAM-augmented deep reinforcement learning. *arXiv preprint arXiv:1612.00380*.
- Bhonker, N., Rozenberg, S., and Hubara, I. (2017). Playing SNES in the retro learning environment.
- Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.
- Bjork, S. and Holopainen, J. (2004). *Patterns in game design*. Charles River Media.
- Blackford, J. and Lamont, G. B. (2014). The real-time strategy game multi-objective build order problem. In *AIIDE*.

- Bontrager, P., Khalifa, A., Mendes, A., and Togelius, J. (2016). Matching games and algorithms for general video game playing. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 122–128.
- Bostrom, N. (2014). *Superintelligence: paths, dangers, strategies*. Oxford University Press.
- Bouzy, B. (2005). Move-pruning techniques for monte-carlo go. In *Advances in Computer Games*, pages 104–119. Springer.
- Branavan, S., Silver, D., and Barzilay, R. (2011). Non-linear monte-carlo search in Civilization II. AAAI Press/International Joint Conferences on Artificial Intelligence.
- Brant, J. C. and Stanley, K. O. (2017). Minimal criterion coevolution: a new approach to open-ended search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 67–74. ACM.
- Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016a). OpenAI gym. *arXiv preprint arXiv:1606.01540*.
- Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016b). OpenAI gym.
- Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43.
- Buck, J. (2015). *Mazes for Programmers: Code Your Own Twisty Little Passages*. Pragmatic Bookshelf.
- Burda, Y., Edwards, H., Storkey, A., and Klimov, O. (2018). Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*.
- Buro, M. (2003). Real-time strategy games: A new AI research challenge. pages 1534–1535.
- Caillois, R. (2001). *Man, play, and games*. University of Illinois Press.
- Campbell, M., Hoane Jr, A. J., and Hsu, F.-h. (2002). Deep Blue. *Artificial intelligence*, 134(1-2):57–83.

- Canaan, R., Salge, C., Togelius, J., and Nealen, A. (2019a). Leveling the playing field-fairness in ai versus human game benchmarks. *arXiv preprint arXiv:1903.07008*.
- Canaan, R., Togelius, J., Nealen, A., and Menzel, S. (2019b). Diverse agents for ad-hoc cooperation in Hanabi. *arXiv preprint arXiv:1907.03840*.
- Cantú-Paz, E. (2004). Adaptive sampling for noisy problems. In *Proc. of GECCO*.
- Carter, M., Gibbs, M., and Harrop, M. (2012). Metagames, paragames and orthogames: A new vocabulary. In *Proceedings of the international conference on the foundations of digital games*, pages 11–17. ACM.
- Caruana, R. (1997). Multitask learning. *Machine learning*, 28(1):41–75.
- Čertický, M. and Churchill, D. (2017). The current state of StarCraft AI competitions and bots. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.
- Chang, Y.-H., Ho, T., and Kaelbling, L. P. (2003). All learning is local: Multi-agent learning in global reward games. In *NIPS*, pages 807–814.
- Chaplot, D. S., Lample, G., Sathyendra, K. M., and Salakhutdinov, R. (2016). Transfer deep reinforcement learning in 3D environments: An empirical study. *Deep Reinforcement Learning Workshop, NIPS 2016*.
- Chaslot, G., Bakkes, S., Szita, I., and Spronck, P. (2008a). Monte-carlo tree search: A new framework for game AI. In *AIIDE*.
- Chaslot, G. M. J., Winands, M. H., van den Herik, H. J., Uiterwijk, J. W., and Bouzy, B. (2008b). Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357.
- Chebotar, Y., Hausman, K., Zhang, M., Sukhatme, G., Schaal, S., and Levine, S. (2017). Combining model-based and model-free updates for trajectory-centric reinforcement learning. *arXiv preprint arXiv:1703.03078*.
- Chen, C., Seff, A., Kornhauser, A., and Xiao, J. (2015). Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730.

- Chen, X., Duan, Y., Houthooft, R., Schulman, J., Sutskever, I., and Abbeel, P. (2016). Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Advances in neural information processing systems*, pages 2172–2180.
- Cho, H.-C., Kim, K.-J., and Cho, S.-B. (2013). Replay-based strategy prediction and build order adaptation for StarCraft AI bots. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–7. IEEE.
- Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.
- Churchill, D. and Buro, M. (2011). Build order optimization in StarCraft. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*.
- Churchill, D. and Buro, M. (2013). Portfolio greedy search and simulation for large-scale combat in StarCraft. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE.
- Churchill, D. and Buro, M. (2015). Hierarchical portfolio search: Prismata’s robust AI architecture for games with large search spaces. In *Proceedings of the Artificial Intelligence in Interactive Digital Entertainment Conference*, pages 16–22.
- Churchill, D., Buro, M., and Kelly, R. (2019). Robust continuous build-order optimization in starcraft. In *Proceedings of the IEEE Conference on Games (COG)*.
- Churchill, D., Preuss, M., Richoux, F., Synnaeve, G., Uriarte, A., Ontanón, S., and Certický, M. (2016). Starcraft bots and competitions.
- Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. (2018). Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*.
- Conti, E., Madhavan, V., Such, F. P., Lehman, J., Stanley, K., and Clune, J. (2018). Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. In *Advances in Neural Information Processing Systems*, pages 5032–5043.
- Côté, M.-A., Kádár, A., Yuan, X., Kybartas, B., Barnes, T., Fine, E., Moore, J.,

- Hausknecht, M., Asri, L. E., Adada, M., Tay, W., and Trischler, A. (2018). Textworld: A learning environment for text-based games. *arXiv preprint arXiv:1806.11532*.
- Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. In *International Conference on Computers and Games*, pages 72–83. Springer.
- Cowling, P. I., Ward, C. D., and Powley, E. J. (2012). Ensemble determinization in monte carlo tree search for the imperfect information card game Magic: The gathering. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(4):241–257.
- Cuccu, G., Togelius, J., and Cudre-Mauroux, P. (2018). Playing atari with six neurons. *arXiv preprint arXiv:1806.01363*.
- Cully, A., Clune, J., Tarapore, D., and Mouret, J.-B. (2015). Robots that can adapt like animals. *Nature*, 521(7553):503.
- Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197.
- Debus, M. S. (2017). Metagames: on the ontology of games outside of games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, page 18. ACM.
- Degriz, T., Pilarski, P. M., and Sutton, R. S. (2012). Model-free reinforcement learning with continuous action in practice. In *American Control Conference (ACC), 2012*, pages 2177–2182. IEEE.
- Dosovitskiy, A. and Koltun, V. (2017). Learning to act by predicting the future. In *Proceedings of the International Conference on Learning Representations*.
- Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*.
- Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

- Elverdam, C. and Aarseth, E. (2007). Game classification and game design: Construction through critical analysis. *Games and Culture*, 2(1):3–22.
- Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. (2018). IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1407–1416, Stockholm, Sweden. PMLR.
- Eysenbach, B., Gupta, A., Ibarz, J., and Levine, S. (2018). Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*.
- Florensa, C., Held, D., Wulfmeier, M., Zhang, M., and Abbeel, P. (2017). Reverse curriculum generation for reinforcement learning. *arXiv preprint arXiv:1707.05300*.
- Foerster, J., Nardelli, N., Farquhar, G., Torr, P., Kohli, P., Whiteson, S., et al. (2017). Stabilising experience replay for deep multi-agent reinforcement learning. In *Proceedings of the International Conference on Machine Learning*.
- Foerster, J. N., Assael, Y. M., de Freitas, N., and Whiteson, S. (2016). Learning to communicate to solve riddles with deep distributed recurrent q-networks. *arXiv preprint arXiv:1602.02672*.
- Foerster, J. N., Farquhar, G., Afouras, T., Nardelli, N., and Whiteson, S. (2018). Counterfactual multi-agent policy gradients. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 2974–2982.
- Fortunato, M., Azar, M. G., Piot, B., Menick, J., Hessel, M., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. (2018). Noisy networks for exploration. In *International Conference on Learning Representations*.
- Fragkiadaki, K., Agrawal, P., Levine, S., and Malik, J. (2016). Learning visual predictive

- models of physics for playing billiards. In *International Conference on Learning Representations (ICLR)*.
- Frasca, G. (2003). Ludologists love stories, too: notes from a debate that never took place. In *DiGRA conference*.
- Fulda, N., Ricks, D., Murdoch, B., and Wingate, D. (2017). What can you do with a rock? affordance extraction via word embeddings. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 1039–1045. AAAI Press.
- Gaina, R. D., Liu, J., Lucas, S. M., and Pérez-Liébana, D. (2017a). Analysis of vanilla rolling horizon evolution parameters in general video game playing. In *European Conference on the Applications of Evolutionary Computation*, pages 418–434. Springer.
- Gaina, R. D., Lucas, S. M., and Perez-Liebana, D. (2017b). Rolling horizon evolution enhancements in general video game playing. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 88–95. IEEE.
- Galway, L., Charles, D., and Black, M. (2008). Machine learning in digital games: a survey. *Artificial Intelligence Review*, 29(2):123–161.
- García-Sánchez, P., Tonda, A., Mora, A. M., Squillero, G., and Merelo, J. (2015). Towards automatic starcraft strategy generation using genetic programming. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 284–291. IEEE.
- Gelly, S. and Wang, Y. (2006). Exploration exploitation in go: UCT for monte-carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*.
- Gers, F. A. and Schmidhuber, J. (2000). Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE.
- Goldberg, D. E. (1987). Simple genetic algorithms and the minimal. *Deceptive Problem, Genetic Algorithms and Simulated Annealing*, pages 74–88.
- Gomez, F. and Miikkulainen, R. (1997). Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342.

- Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.
- Graves, A., Bellemare, M. G., Menick, J., Munos, R., and Kavukcuoglu, K. (2017). Automated curriculum learning for neural networks. *arXiv preprint arXiv:1704.03003*.
- Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.
- Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476.
- Groshev, E., Goldstein, M., Tamar, A., Srivastava, S., and Abbeel, P. (2017). Learning generalized reactive policies using deep neural networks. *arXiv preprint arXiv:1708.07280*.
- Gu, S., Lillicrap, T., Sutskever, I., and Levine, S. (2016). Continuous deep Q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838.
- Gudmundsson, S. F., Eisen, P., Poromaa, E., Nodet, A., Purmonen, S., Kozakowski, B., Meurling, R., and Cao, L. (2018). Human-like playtesting with deep learning. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE.
- Guo, X., Singh, S., Lee, H., Lewis, R. L., and Wang, X. (2014). 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.
- Ha, D. and Schmidhuber, J. (2018). World models.
- Harmer, J., Gisslén, L., del Val, J., Holst, H., Bergdahl, J., Olsson, T., Sjöö, K., and Nordin, M. (2018). Imitation learning with concurrent actions in 3d games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE.
- Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

- Hasselt, H. V. (2010). Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621.
- Hausknecht, M., Lehman, J., Miikkulainen, R., and Stone, P. (2014). A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366.
- Hausknecht, M., Mupparaju, P., Subramanian, S., Kalyanakrishnan, S., and Stone, P. (2016). Half field offense: An environment for multiagent learning and ad hoc teamwork. In *AAMAS Adaptive Learning Agents (ALA) Workshop*.
- Hausknecht, M. and Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. In *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents (AAAI-SDMIA15)*.
- Hausknecht, M. and Stone, P. (2016). On-policy vs. off-policy updates for deep reinforcement learning. In *Deep Reinforcement Learning: Frontiers and Challenges, IJCAI 2016 Workshop*.
- Hawkins, D. M. (2004). The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12.
- He, J., Chen, J., He, X., Gao, J., Li, L., Deng, L., and Ostendorf, M. (2016a). Deep reinforcement learning with a natural language action space. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1621–1630.
- He, K., Zhang, X., Ren, S., and Sun, J. (2016b). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.
- Helmbold, D. P. and Parker-Wood, A. (2009). All-moves-as-first heuristics in monte-carlo go. In *IC-AI*, pages 605–610.
- Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In *AAAI*.
- Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan,

- J., Sendonaris, A., Dulac-Arnold, G., et al. (2018). Deep q-learning from demonstrations. In *Proceedings of AAAI*.
- Hingston, P. (2012). *Believable Bots: Can Computers Play Like People?* Springer.
- Hinton, G., Srivastava, N., and Swersky, K. (2012). Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14.
- Ho, J. and Ermon, S. (2016). Generative adversarial imitation learning. In *Advances in Neural Information Processing Systems*, pages 4565–4573.
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.
- Holland, J. H. (1975). Adaptation in natural and artificial systems ann arbor. *The University of Michigan Press*, 1:975.
- Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., and Silver, D. (2018). Distributed prioritized experience replay. In *International Conference on Learning Representations (ICLR)*.
- Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.
- Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. (2017). Reinforcement learning with unsupervised auxiliary tasks. In *International Conference on Learning Representations (ICLR)*.
- Jaworska, N. and Chupetlovska-Anastasova, A. (2009). A review of multidimensional scaling (mds) and its utility in various psychological domains. *Tutorials in quantitative methods for psychology*, 5(1):1–10.
- Johnson, J. (2016). Blood bowl handbook: Blood bowl living rulebook 6.0.
- Johnson, L., Yannakakis, G. N., and Togelius, J. (2010). Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 10. ACM.
- Johnson, M., Hofmann, K., Hutton, T., and Bignell, D. (2016). The malmo platform for

- artificial intelligence experimentation. In *International joint conference on artificial intelligence (IJCAI)*, page 4246.
- Juliani, A., Khalifa, A., Berges, V.-P., Harper, J., Henry, H., Crespi, A., Togelius, J., and Lange, D. (2019). Obstacle tower: A generalization challenge in vision, control, and planning. *arXiv preprint arXiv:1902.01378*.
- Justesen, N. (2015). Artificial intelligence for hero academy. *Master's thesis*.
- Justesen, N., Mahlmann, T., Risi, S., and Togelius, J. (2017). Playing multi-action adversarial games: Online evolutionary planning versus tree search. *IEEE Transactions on Computational Intelligence and AI in Games*.
- Justesen, N., Mahlmann, T., and Togelius, J. (2016). Online evolution for multi-action adversarial games. In *European Conference on the Applications of Evolutionary Computation*, pages 590–603. Springer.
- Justesen, N. and Risi, S. (2017a). Continual online evolution for in-game build order adaptation in StarCraft. In *The Genetic and Evolutionary Computation Conference (GECCO)*.
- Justesen, N. and Risi, S. (2017b). Learning macromanagement in StarCraft from replays using deep learning. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 162–169. IEEE.
- Justesen, N., Tillman, B., Togelius, J., and Risi, S. (2014). Script-and cluster-based uct for StarCraft. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE.
- Juul, J. (2011). *Half-real: Video games between real rules and fictional worlds*. MIT press.
- Kansky, K., Silver, T., Mély, D. A., Eldawy, M., Lázaro-Gredilla, M., Lou, X., Dorfman, N., Sidor, S., Phoenix, S., and George, D. (2017). Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. In *Proceedings International Conference on Machine Learning*.
- Kaplan, R., Sauer, C., and Sosa, A. (2017). Beating atari with natural language guided reinforcement learning. *arXiv preprint arXiv:1704.05539*.

- Karnin, Z., Koren, T., and Somekh, O. (2013). Almost optimal exploration in multi-armed bandits. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1238–1246.
- Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. (2016). Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8. IEEE.
- Keogh, B. (2014). Across worlds and bodies: Criticism in the age of video games. *Journal of Games Criticism*, 1(1):1–26.
- Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
- Klimov, O. (2016). Bipedalwalkerhardcore-v2. <http://gym.openai.com/>.
- Komincz, G. M. DeepMind Starcraft 2 demonstration - MaNa’s personal experience.
- Kostka, B., Kwiecieli, J., Kowalski, J., and Rychlikowski, P. (2017). Text-based adventures of the golovin ai agent. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, pages 181–188. IEEE.
- Köstler, H. and Gmeiner, B. (2013). A multi-objective genetic algorithm for build order optimization in starcraft ii. *KI-Künstliche Intelligenz*, 27(3):221–233.
- Koutník, J., Cuccu, G., Schmidhuber, J., and Gomez, F. (2013). Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068. ACM.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.
- Kuchem, M., Preuss, M., and Rudolph, G. (2013). Multi-objective assessment of pre-optimized build orders exemplified for starcraft 2. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE.
- Kulkarni, T. D., Narasimhan, K., Saeedi, A., and Tenenbaum, J. (2016). Hierarchical

- deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 3675–3683.
- Kume, A., Matsumoto, E., Takahashi, K., Ko, W., and Tan, J. (2017). Map-based multi-policy reinforcement learning: enhancing adaptability of robots by deep reinforcement learning. *arXiv preprint arXiv:1710.06117*.
- Lample, G. and Chaplot, D. S. (2017a). Playing FPS games with deep reinforcement learning. In *AAAI Conference on Artificial Intelligence*.
- Lample, G. and Chaplot, D. S. (2017b). Playing FPS games with deep reinforcement learning. In *AAAI*, pages 2140–2146.
- Laud, A. D. (2004). Theory and application of reward shaping in reinforcement learning. Technical report.
- LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.
- LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.
- Lehman, J. and Stanley, K. O. (2008). Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pages 329–336.
- Lehman, J. and Stanley, K. O. (2011a). Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223.
- Lehman, J. and Stanley, K. O. (2011b). Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 211–218. ACM.
- Lehman, J. and Stanley, K. O. (2011c). Novelty search and the problem with objectives. In *Genetic programming theory and practice IX*, pages 37–56. Springer.
- Lerer, A., Gross, S., and Fergus, R. (2016). Learning physical intuition of block towers by example. In *International Conference on Machine Learning*, pages 430–438.
- Levine, J., Congdon, C. B., Ebner, M., Kendall, G., Lucas, S. M., Miikkulainen, R., Schaul, T., Thompson, T., Lucas, S. M., Mateas, M., et al. (2013). General video game playing. *Artificial and Computational Intelligence in Games*, 6:77–83.

- Li, Y. (2018). Deep reinforcement learning. *arXiv preprint arXiv:1701.07274*.
- Li, Y., Song, J., and Ermon, S. (2017). InfoGAIL: Interpretable imitation learning from visual demonstrations. In *Advances in Neural Information Processing Systems*, pages 3812–3822.
- Liapis, A., Holmgård, C., Yannakakis, G. N., and Togelius, J. (2015). Procedural personas as critics for dungeon generation. In *European Conference on the Applications of Evolutionary Computation*, pages 331–343. Springer.
- Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016). Continuous control with deep reinforcement learning. *International Conference on Learning Representations (ICLR)*, *arXiv:1509.02971*.
- Lin, L.-J. (1993). *Reinforcement learning for robots using neural networks*. PhD thesis, Fujitsu Laboratories Ltd.
- Louis, S. J. and McDonnell, J. (2004). Learning with case-injected genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 8(4):316–328.
- Louis, S. J. and Miles, C. (2005). Playing to learn: Case-injected genetic algorithms for learning to play computer games. *IEEE Transactions on Evolutionary Computation*, 9(6):669–681.
- Lu, Z., Pu, H., Wang, F., Hu, Z., and Wang, L. (2017). The expressive power of neural networks: A view from the width. In *Advances in neural information processing systems*, pages 6231–6239.
- Maaten, L. v. d. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605.
- Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M., and Bowling, M. (2018). Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562.
- Marcus, G. (2018). Innateness, alphazero, and artificial intelligence. *arXiv preprint arXiv:1801.05667*.

- Matiisen, T., Oliver, A., Cohen, T., and Schulman, J. (2017). Teacher-student curriculum learning. *arXiv preprint arXiv:1707.00183*.
- McInnes, L., Healy, J., and Melville, J. (2018). Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*.
- Merel, J., Tassa, Y., Srinivasan, S., Lemmon, J., Wang, Z., Wayne, G., and Heess, N. (2017). Learning human behaviors from motion capture by adversarial imitation. *arXiv preprint arXiv:1707.02201*.
- Miikkulainen, R., Bryant, B. D., Cornelius, R., Karpov, I. V., Stanley, K. O., and Yong, C. H. (2006). Computational intelligence in games. *Computational Intelligence: Principles and Practice*, pages 155–191.
- Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.
- Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A. J., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., et al. (2016). Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*.
- Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*.
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*.
- Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.
- Montufar, G. F., Pascanu, R., Cho, K., and Bengio, Y. (2014). On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pages 2924–2932.
- Mouret, J.-B. and Clune, J. (2015). Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*.

- Muñoz-Avila, H., Bauckhage, C., Bida, M., Congdon, C. B., and Kendall, G. (2013). Learning and game AI. In *Dagstuhl Follow-Ups*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- Murray, J. H. (2005). The last word on ludology v narratology in game studies. In *International DiGRA Conference*.
- Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., De Maria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., et al. (2015). Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*.
- Narasimhan, K., Kulkarni, T. D., and Barzilay, R. (2015). Language understanding for textbased games using deep reinforcement learning. In *In Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Citeseer.
- Neufeld, X., Mostaghim, S., and Perez-Liebana, D. (2015). Procedural level generation with answer set programming for general video game playing. In *2015 7th Computer Science and Electronic Engineering Conference (CEEC)*, pages 207–212. IEEE.
- Ng, A. Y. (2003). *Shaping and policy search in reinforcement learning*. PhD thesis, University of California, Berkeley.
- Ng, A. Y., Harada, D., and Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287.
- Nichol, A., Pfau, V., Hesse, C., Klimov, O., and Schulman, J. (2018). Gotta learn fast: A new benchmark for generalization in rl. *arXiv preprint arXiv:1804.03720*.
- Nowlan, S. J. and Hinton, G. E. (1992). Simplifying neural networks by soft weight-sharing. *Neural computation*, 4(4):473–493.
- Oh, J., Chockalingam, V., Satinder, and Lee, H. (2016). Control of memory, active perception, and action in minecraft. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48, pages 2790–2799.
- Oh, J., Guo, X., Lee, H., Lewis, R. L., and Singh, S. (2015). Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems*, pages 2863–2871.

- Ontanón, S. (2013). The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 58–64. AAAI Press.
- Ontanón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and Preuss, M. (2013). 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.
- OpenAI (2017). More on dota 2. <https://openai.com/blog/more-on-dota-2/>.
- OpenAI (2018a). Openai five. <https://blog.openai.com/openai-five/>.
- OpenAI (2018b). Openai five. <https://openai.com/five/>.
- OpenAI (2018c). Openai five benchmark. <https://openai.com/blog/openai-five-benchmark/>.
- OpenAI (2019). How to train your openai five. <https://openai.com/blog/how-to-train-your-openai-five/>.
- Osband, I., Blundell, C., Pritzel, A., and Van Roy, B. (2016). Deep exploration via bootstrapped DQN. In *Advances In Neural Information Processing Systems*, pages 4026–4034.
- Ostrovski, G., Bellemare, M. G., van den Oord, A., and Munos, R. (2017). Count-based exploration with neural density models. pages 2721–2730.
- Paenke, I., Branke, J., and Jin, Y. (2006). Efficient search for robust solutions by means of evolutionary algorithms and fitness approximation. *IEEE Trans. on Evolutionary Computation*, 10(4):405–420.
- Pan, S. J. and Yang, Q. (2009). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359.
- Parisotto, E., Ba, J. L., and Salakhutdinov, R. (2016). Actor-mimic: Deep multitask and transfer reinforcement learning.
- Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017a). Curiosity-driven exploration by self-supervised prediction. In *ICML*.
- Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017b). Curiosity-driven exploration

- by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 16–17.
- Peng, P., Yuan, Q., Wen, Y., Yang, Y., Tang, Z., Long, H., and Wang, J. (2017). Multiagent bidirectionally-coordinated nets for learning to play StarCraft combat games. *arXiv preprint arXiv:1703.10069*.
- Perez, D., Rohlfsen, P., and Lucas, S. M. (2012). Monte-carlo tree search for the physical travelling salesman problem. In *Applications of Evolutionary Computation*, pages 255–264. Springer.
- Perez Liebana, D., Dieskau, J., Hunermund, M., Mostaghim, S., and Lucas, S. (2015). Open loop search for general video game playing. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 337–344. ACM.
- Perez-Liebana, D., Liu, J., Khalifa, A., Gaina, R. D., Togelius, J., and Lucas, S. M. (2018). General video game AI: a multi-track framework for evaluating agents, games and content generation algorithms. *arXiv preprint arXiv:1802.10363*.
- Perez-Liebana, D., Samothrakis, S., Togelius, J., Lucas, S. M., and Schaul, T. (2016). General Video Game AI: Competition, Challenges and Opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*, pages 4335–4337.
- Pohlen, T., Piot, B., Hester, T., Azar, M. G., Horgan, D., Budden, D., Barth-Maron, G., van Hasselt, H., Quan, J., Večerík, M., et al. (2018). Observe and look further: Achieving consistent performance on atari. *arXiv preprint arXiv:1805.11593*.
- Powley, E. J., Whitehouse, D., and Cowling, P. I. (2012). Monte carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 234–241. IEEE.
- Pugh, J. K., Soros, L. B., and Stanley, K. O. (2016). Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:40.
- Rechenberg, I. (1978). Evolutionsstrategien. In *Simulationsmethoden in der Medizin und Biologie*, pages 83–114. Springer.

- Risi, S., Hughes, C. E., and Stanley, K. O. (2010). Evolving plastic neural networks with novelty search. *Adaptive Behavior*, 18(6):470–491.
- Risi, S. and Togelius, J. (2015). Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*.
- Robertson, G. and Watson, I. D. (2014). An improved dataset and extraction process for Starcraft AI. In *FLAIRS Conference*.
- Rodriguez Torrado, R., Bontrager, P., Togelius, J., Liu, J., and Perez-Liebana, D. (2018). Deep reinforcement learning for general video game AI. In *Computational Intelligence and Games (CIG), 2018 IEEE Conference on*. IEEE.
- Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.
- Rusu, A. A., Colmenarejo, S. G., Gulcehre, C., Desjardins, G., Kirkpatrick, J., Pascanu, R., Mnih, V., Kavukcuoglu, K., and Hadsell, R. (2016a). Policy distillation. *International Conference on Learning Representations (ICLR)*.
- Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. (2016b). Progressive neural networks. *arXiv preprint arXiv:1606.04671*.
- Sadeghi, F. and Levine, S. (2016). Cad2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*.
- Salimans, T., Ho, J., Chen, X., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.
- Schaul, T. (2013). A video game description language for model-based or interactive learning. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE.
- Schaul, T., Horgan, D., Gregor, K., and Silver, D. (2015). Universal value function approximators. In *International conference on machine learning*, pages 1312–1320.
- Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay. In *International Conference on Learning Representations*, Puerto Rico.

- Schmidhuber, J. (2013). Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in psychology*, 4:313.
- Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61:85–117.
- Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897.
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.
- Shaker, N., Togelius, J., and Nelson, M. J. (2016). *Procedural Content Generation in Games*. Springer.
- Shannon, C. E. (1988). Programming a computer for playing chess. In *Computer chess compendium*, pages 2–13. Springer.
- Shelhamer, E., Mahmoudieh, P., Argus, M., and Darrell, T. (2016). Loss is its own reward: Self-supervision for reinforcement learning. In *International Conference on Learning Representations*.
- Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.
- Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.
- Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395.
- Simon, H. and Chase, W. (1988). Skill in chess. In *Computer chess compendium*, pages 175–188. Springer.

- Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.
- Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.
- Stanescu, M., Barriga, N. A., Hess, A., and Buro, M. (2016). Evaluating real-time strategy game states using convolutional neural networks. In *IEEE Conference on Computational Intelligence and Games (CIG 2016)*.
- Stenros, J. (2017). The game definition game: A review. *Games and Culture*, 12(6):499–520.
- Stone, P. and Sutton, R. S. (2001). Keepaway soccer: A machine learning test bed. In *Robot Soccer World Cup*, pages 214–223. Springer.
- Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. (2017). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*.
- Suits, B. (2014). *The Grasshopper-: Games, Life and Utopia*. Broadview Press.
- Sukhbaatar, S., Lin, Z., Kostrikov, I., Synnaeve, G., Szlam, A., and Fergus, R. (2017). Intrinsic motivation and automatic curricula via asymmetric self-play. *arXiv preprint arXiv:1703.05407*.
- Sun, P., Sun, X., Han, L., Xiong, J., Wang, Q., Li, B., Zheng, Y., Liu, J., Liu, Y., Liu, H., et al. (2018). TStarBots: Defeating the cheating level builtin AI in StarCraft II in the full game. *arXiv preprint arXiv:1809.07193*.
- Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.
- Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063.
- Sweetser, P. (2008). *Emergence in games*. Cengage Learning.

- Syed, O. and Syed, A. (2003). Arimaa-a new game designed to be difficult for computers. *ICGA Journal*, 26(2):138–139.
- Synnaeve, G. and Bessiere, P. (2011). A bayesian model for plan recognition in rts games applied to StarCraft. *arXiv preprint arXiv:1111.3735*.
- Synnaeve, G. and Bessiere, P. (2012). A dataset for StarCraft AI\& an example of armies clustering. *arXiv preprint arXiv:1211.4552*.
- Synnaeve, G., Nardelli, N., Auvolat, A., Chintala, S., Lacroix, T., Lin, Z., Richoux, F., and Usunier, N. (2016). Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*.
- Szita, I., Chaslot, G., and Spronck, P. (2009). Monte-carlo tree search in Settlers of Catan. In *Advances in Computer Games*, pages 21–32. Springer.
- Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, J., Aru, J., and Vicente, R. (2017). Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4):e0172395.
- Tan, C., Sun, F., Kong, T., Zhang, W., Yang, C., and Liu, C. (2018). A survey on deep transfer learning. In *International Conference on Artificial Neural Networks*, pages 270–279. Springer.
- Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337.
- Tang, Z., Zhao, D., Zhu, Y., and Guo, P. (2018). Reinforcement learning for build-order production in StarCraft II. In *2018 Eighth International Conference on Information Science and Technology (ICIST)*, pages 153–158. IEEE.
- Teetzel, S. (2006). On transgendered athletes, fairness and doping: An international challenge. *Sport in Society*, 9(2):227–251.
- Teh, Y., Bapst, V., Pascanu, R., Heess, N., Quan, J., Kirkpatrick, J., Czarnecki, W. M., and Hadsell, R. (2017). Distral: Robust multitask reinforcement learning. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 4497–4507. Curran Associates, Inc.

- Tessler, C., Givony, S., Zahavy, T., Mankowitz, D. J., and Mannor, S. (2017). A deep hierarchical approach to lifelong learning in minecraft. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 1553–1561.
- Thurau, C., Bauckhage, C., and Sagerer, G. (2004). Imitation learning at all levels of game-ai. In *Proceedings of the international conference on computer games, artificial intelligence, design and education*, volume 5.
- Tian, Y., Gong, Q., Shang, W., Wu, Y., and Zitnick, C. L. (2017). Elf: An extensive, lightweight and flexible research platform for real-time strategy games. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 2656–2666. Curran Associates, Inc.
- Tobin, J., Zaremba, W., and Abbeel, P. (2017). Domain randomization and generative models for robotic grasping. *arXiv preprint arXiv:1710.06425*.
- Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE.
- Togelius, J., Karakovskiy, S., and Baumgarten, R. (2010). The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE.
- Togelius, J. and Lucas, S. M. (2006). Evolving robust and specialized car racing skills. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 1187–1194. IEEE.
- Tong, X., Liu, W., and Bin, L. (2019). Enhancing rolling horizon evolution with policy and value networks. In *2019 IEEE Conference on Games (COG)*. IEEE.
- Usunier, N., Synnaeve, G., Lin, Z., and Chintala, S. (2016). Episodic exploration for deep deterministic policies: An application to starcraft micromanagement tasks. *arXiv preprint arXiv:1609.02993*.
- Van Der Linden, R., Lopes, R., and Bidarra, R. (2014). Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89.

- Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100.
- Van Seijen, H., Laroche, R., Fatemi, M., and Romoff, J. (2017). Hybrid reward architecture for reinforcement learning. In *Advances in Neural Information Processing Systems 30*, pages 5396–5406. Curran Associates, Inc.
- Vassiliades, V., Chatzilygeroudis, K., and Mouret, J.-B. (2018). Using centroidal voronoi tessellations to scale up the multidimensional archive of phenotypic elites algorithm. *IEEE Trans. on Evolutionary Computation*, 22(4):623–630.
- Vinyals, O., Babuschkin, I., Chung, J., Mathieu, M., Jaderberg, M., Czarnecki, W. M., Dudzik, A., Huang, A., Georgiev, P., Powell, R., Ewalds, T., Horgan, D., Kroiss, M., Danihelka, I., Agapiou, J., Oh, J., Dalibard, V., Choi, D., Sifre, L., Sulsky, Y., Vezhnevets, S., Molloy, J., Cai, T., Budden, D., Paine, T., Gulcehre, C., Wang, Z., Pfaff, T., Pohlen, T., Wu, Y., Yogatama, D., Cohen, J., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Apps, C., Kavukcuoglu, K., Hassabis, D., and Silver, D. (2019). AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. <https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/>.
- Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., et al. (2017). StarCraft II: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.
- Vinyals, O. e. a. (2019). AlphaStar: Mastering the real-time strategy game StarCraft II.
- Volz, V., Schrum, J., Liu, J., Lucas, S. M., Smith, A., and Risi, S. (2018). Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 221–228. ACM.
- Wang, C., Chen, P., Li, Y., Holmgård, C., and Togelius, J. (2016a). Portfolio online evolution in StarCraft. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 114,120.
- Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2017a). Sample efficient actor-critic with experience replay.

- Wang, Z., Merel, J. S., Reed, S. E., de Freitas, N., Wayne, G., and Heess, N. (2017b). Robust imitation of diverse behaviors. In *Advances in Neural Information Processing Systems*, pages 5320–5329.
- Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. (2016b). Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*.
- Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.
- Whitley, L. D. (1991). Fundamental principles of deception in genetic search. In *Foundations of genetic algorithms*, volume 1, pages 221–241. Elsevier.
- Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J., and Schmidhuber, J. (2014). Natural evolution strategies. *The Journal of Machine Learning Research*, 15(1):949–980.
- Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.
- Wolf, M. J. (1997). Inventing space: Toward a taxonomy of on-and off-screen space in video games. *Film Quarterly (ARCHIVE)*, 51(1):11.
- Wu, H., Zhang, J., and Huang, K. (2017a). Msc: A dataset for macro-management in StarCraft II. *arXiv preprint arXiv:1710.03131*.
- Wu, Y., Mansimov, E., Grosse, R. B., Liao, S., and Ba, J. (2017b). Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *Advances in neural information processing systems*, pages 5285–5294.
- Wu, Y. and Tian, Y. (2017). Training agent for first-person shooter game with actor-critic curriculum learning. In *ICLR 2017*.
- Wymann, B., Espié, E., Guionneau, C., Dimitrakakis, C., Coulom, R., and Sumner, A. (2000). TORCS, the open racing car simulator. *Software available at <http://torcs.sourceforge.net>*.
- Yannakakis, G. N. (2012). Game ai revisited. In *Proceedings of the 9th conference on Computing Frontiers*, pages 285–292. ACM.

- Yannakakis, G. N., Spronck, P., Loiacono, D., and André, E. (2013). Player modeling. In *Dagstuhl Follow-Ups*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- Yannakakis, G. N. and Togelius, J. (2015). A panorama of artificial and computational intelligence in games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):317–335.
- Yannakakis, G. N. and Togelius, J. (2018a). *Artificial Intelligence and Games*. Springer.
<http://gameaibook.org>.
- Yannakakis, G. N. and Togelius, J. (2018b). *Artificial Intelligence and Games*. Springer.
- Yu, H., Zhang, H., and Xu, W. (2017). A deep compositional framework for human-like language acquisition in virtual environment. *arXiv preprint arXiv:1703.09831*.
- Zahavy, T., Haroush, M., Merlis, N., Mankowitz, D. J., and Mannor, S. (2018). Learn what not to learn: Action elimination with deep reinforcement learning. *arXiv preprint arXiv:1809.02121*.
- Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.
- Zhang, C., Vinyals, O., Munos, R., and Bengio, S. (2018). A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*.

Appendix

A1 StarCraft Build-order Forward Model

Algorithm 10 StarCraft Build-Order Forward Model

$\text{MINERALWORKERS}(s)$ and $\text{GASWORKERS}(s)$ return the number of workers gathering minerals and gas, respectively, in s .

```

1: PREDICT( $s$ ,  $buildOrder$ ,  $endFrame$ ) returns the resulting game state of producing
   the builds in  $buildOrder$  from state  $s$  until frame  $endFrame$  is reached.
2: procedure PREDICT(GameState  $s$ , BuildType[]  $buildOrder$ , int  $endFrame$ )
3:   for each BuildType  $type$  in  $buildOrder$  do
4:      $nextFrame = \text{PRODUCEFRAME}(s, type)$ 
5:     if  $nextFrame \leq endFrame$  then
6:        $\text{PROGRESS}(s, nextFrame)$ 
7:        $\text{BUILD}(s, type)$ 
8:     else
9:        $\text{PROGRESS}(s, endFrame)$ 
10:    return  $s$                                  $\triangleright$  The altered game state.
11: procedure PRODUCEFRAME(GameState  $s$ , BuildType  $type$ )
12:   Returns the latest frame in which all requirements, resources, and production
      buildings/units are available in  $s$  in order to produce  $type$ .
13: procedure PROGRESS(GameState  $s$ , int  $toFrame$ )
14:    $t = toFrame - s.frame$ 
15:    $s.minerals += t \times \text{MINE\_SPEED} \times \text{MINERALWORKERS}(s)$ 
16:    $s.gas += t \times \text{GAS\_SPEED} \times \text{GASWORKERS}(s)$ 
17:   for each Build  $b$  in  $s.\text{underProduction}$  do
18:     if not  $b.done$  and  $toFrame \geq b.doneAt$  then
19:       Add one build of type  $b.type$  to  $s$ 
20:        $b.done = \text{true}$ 
21: procedure BUILD(GameState  $s$ , BuildType  $type$ )
22:    $b = \text{Build}()$ 
23:    $b.type = type$ 
24:    $b.doneAt = s.frame + type.buildTime$ 
25:    $s.\text{underProduction}.Push(b)$ 
26:    $s.minerals -= type.mineralCost$ 
27:    $s.gas -= type.gasCost$ 
28:    $s.supply += type.supplyCost$ 

```

A2 Configuration of COEP

- Population size: 64
- Mutation operators and rates:
 - Clone: 0.5

- Swap: 0.5
- Add: 0.5
- Remove: 0.5
- Crossover operator: Two-point crossover
- Survival rate: 0.25
- Fitness function:
 - Step size: 2 minutes (one fastest) = 9810 frames
 - Horizon: 8 minutes (one fastest) = 11429 frames
 - Discount: 0.9

A3 Configuration of A2C+RoE

A2C	
Learning rate	7e-4
γ (discount factor)	0.99
Entropy coefficient	0.01
Value loss coefficient	0.5
Learning rate	0.0007
Max. gradient-norm	0.5
Worker threads	4 (16 in DM)
t_{max} (Steps per. update)	20
Batch size	64
Frame skip	4
RMSprop Optimizer	
ϵ	1e-5
α	0.99
RoE	
N (event buffer size)	100
τ (mean threshold)	0.01

Table A3.1: Experimental configurations for A2C and A2C+RoE. 16 worker threads were used in *Deathmatch*.