Martinho Martins Bastos Tavares

Aprendizagem por Reforço aplicada a Jogos de Luta

Reinforcement Learning applied to Fighting Games

Universidade de Aveiro
2024

Martinho Martins Bastos Tavares

Aprendizagem por Reforço aplicada a Jogos de Luta

Reinforcement Learning applied to Fighting Games

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática , realizada sob a orientação científica do Doutor Armando José Formoso de Pinho, Professor catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor João Miguel Rafael de Carvalho, Professor adjunto convidado do Instituto Superior de Contabilidade e Administração de Aveiro da Universidade de Aveiro.

Quero agradecer ao Professor Armando Pinho e ao Professor João Carvalho, por aceitarem orientar-me num problema de iniciativa própria, mostrarem-se sempre disponíveis e ajudarem imenso a abordar um tópico ambicioso. A ajuda foi muito importante não só academicamente mas também pessoalmente, motivando-me durante um ano inteiro, principalmente em alturas em que eu estava a duvidar do meu esforço ou do que eu estava a fazer. Não podia pedir melhor orientação.
Também quero agradecer à minha família e amigos por me terem suportado, aturado e também dado feedback do meu trabalho. E, acima de tudo, por estarem presentes especialmente num ano que, fora isso, foi de trabalho solitário.

o júri / the jury

presidente / president

Professora Doutora Petia Georgieva Georgieva
professor associado c/ agregação da Universidade de Aveiro

vogais / examiners committee

Doutor Luís Filipe Barbosa de Almeida Alexandre
professor catedrático da Universidade da Beira Interior

Professor Doutor Armando José Formoso de Pinho
professor catedrático da Universidade de Aveiro

agradecimentos / acknowledgements

I want to thank Professor Armando Pinho and Professor João Carvalho, for accepting being the advisors for a problem of my own volition, being always available and helping immensely in addressing an ambitious topic. The help was very important not only academically but also personally, motivating me for a whole year, mainly in times I was doubting my efforts or what I was doing. I could not ask for better advising.

I also want to thank my family and friends for supporting me, putting up with me and also giving feedback of my work. And, above all, for being present especially in a year that, otherwise, was of solitary work.

Palavras Chave

aprendizagem por reforço, rede neuronal artificial, jogo de luta, comportamento tipo humano, tempo de reação, modelação do oponente.

Resumo

Os métodos de inteligência artificial podem ser usados para desenvolver agentes artificiais para videojogos. Estes agentes têm o potencial de serem ferramentas de aprendizagem para os jogadores, tal como dar a entender o quão equilibrado o jogo é. No género dos jogos de luta, os agentes são frequentemente desenvolvidos ou por imitação de jogadores humanos ou por algoritmos de tomada de decisão. O primeiro método não executa decisões explicitamente, enquanto que o segundo não incorpora características de como os humanos jogam, principalmente o tempo de reação. Este trabalho aborda ambos os problemas, explorando aprendizagem por reforço para a criação de um agente capaz de aprendizagem autónoma, sobre um tempo de reação emulado semelhante ao de humanos. Este tempo de reação depende do quão bem o agente prevê o comportamento do oponente. O tempo de reação impõe um atraso na perceção que o agente tem do jogo, que é corrigido extrapolando o presente a partir da observação atrasada. O método é avaliado no jogo de luta de código aberto *FOOTSIES*, que modificámos para acomodar o treino de um agente. Descobrimos que considerar as ações futuras do oponente explicitamente no algoritmo de aprendizagem por reforço é prejudicial ao seu desempenho, mas que outros ajustamentos pertinentes ao género ajudam na aprendizagem. Além disso, o tempo de reação é completamente corrigido com uma ligeira diminuição no desempenho da tarefa, mas sem comprometer o desempenho computacional. Concluímos que a seleção de oponentes utilizada para treino é de extrema importância na criação de agentes robustos, e que sinais de recompensa densos são importantes para guiar o agente em direção ao objetivo. Terminamos este estudo fornecendo diferentes caminhos por onde se pode continuar a explorar o problema, principalmente na simplificação da emulação do tempo de reação, inclusão de diferentes esquemas de recompensa e utilização de arquiteturas hierárquicas.

Keywords reinforcement learning, artificial neural network, fighting game, human-like behavior, reaction time, opponent modeling.

Abstract

Artificial intelligence methods can be used to develop artificial agents for video games. These agents have the potential to be learning tools for players, as well as providing automated insights into how balanced the game is. In the genre of fighting games, agents are often developed either by imitation of human players or by decision-making algorithms. The former method does not explicitly perform decisions, whereas the latter does not incorporate characteristics of human play, mainly reaction time. This work addresses both problems, by exploring reinforcement learning for creating an agent capable of autonomous learning, while under an emulated reaction time mimicking that of humans. This reaction time is dependent on how well the agent predicts the opponent's behavior. Reaction time imposes a delay on the agent's perception of the game, which is tackled by extrapolating the present from the delayed observation. The method is evaluated in the open-source fighting game *FOOTSIES*, which we modified to accommodate training of an agent. We find that explicitly considering the opponent's future actions in the reinforcement algorithm is detrimental to its performance, but that other adjustments pertinent to the genre aided in learning. Furthermore, reaction time was fully handled with a slight decrease in task performance, but without compromising computational performance. We conclude that the choice of opponents used for training is paramount in making robust agents, and that dense reward signals are important for guiding the agent toward the goal. We finish this study by providing different avenues through which the problem can be explored further, mainly in the simplification of the reaction time emulation, inclusion of different reward schemes and usage of hierarchical architectures.

acknowledgement of use of AI tools

Recognition of the use of generative Artificial Intelligence technologies and tools, software and other support tools.

I acknowledge the use of ChatGPT (OpenAI, https://chat.openai.com) to find answers to doubts for which I could not find satisfying explanations with a search engine. The prompts used include "Why is the Soft Actor-Critic algorithm off-policy?", "In the context of Reinforcement Learning, what does it mean for a policy to be 'convex'?" and "Why is the all-actions policy gradient method not used often in practice?". I acknowledge the use of the GitHub Copilot extension for Visual Studio Code (OpenAI, https://github.com/features/copilot) to speed up development, by having Copilot complete patterns in repetitive code segments. I acknowledge the use of the Grammarly extension for Visual Studio Code (Grammarly Inc., https://github.com/znck/grammarly) to fix typos and incorrect grammar in this document. I acknowledge the use of Google Translate (Google, https://translate.google.com) to translate some articles and websites in foreign languages to english. I acknowledge the use of Visual Studio Code (Microsoft, https://code.visualstudio.com) to develop the project's code and write this document.

Contents

List of Figures

List of Tables

Glossary

A2C	Advantage Actor Critic
A3C	Asynchronous Advantage Actor Critic
AI	artificial intelligence
ANN	artificial neural network
API	application programming interface
CNN	convolutional neural network
CPU	central processing unit
DDA	dynamic difficulty adjustment
DIAYN	Diversity Is All You Need
DL	deep learning
DNN	deep neural network
DP	dynamic programming
DQN	Deep Q-Network
DRL	deep reinforcement learning
FSP	fictitious self-play
HRL	hierarchical reinforcement learning
L2E	Learning to Exploit
MAML	Model-Agnostic Meta-Learning
MC	Monte Carlo
MCTS	Monte Carlo tree search
MDP	Markov decision process
ML	machine learning
NFC	near-field communication
OSG	opponent strategy generation
PFSP	prioritized fictitious self-play
PG	policy gradient
POMDP	partially observable Markov decision process
PPO	Proximal Policy Optimization
ReLU	Rectified Linear Unit
RL	reinforcement learning
RNN	recurrent neural network
SL	supervised learning
TD	temporal difference
TRPO	Trust Region Policy Optimization
UL	unsupervised learning

CHAPTER 1

Introduction

Presentation of the main topic, its relevance and our work.

Fighting games are competitive two-player zero-sum video games. These are real-time games where both players act simultaneously, as opposed to turn-based play. In these games, it is possible to play against human players, but artificial opponents are usually also implemented. Unfortunately, current approaches to creating these artificial opponents are not appropriate, in the sense that they do not play as human beings do [1]. One major factor in this discrepancy is human reaction time: many of the actions performed by each player cannot either be reacted to or it is difficult to do so. Because of this, strategies in fighting games are heavily based on prediction of the opponent's future actions [2]. We can think of these games as multiple consecutive trials of rock-paper-scissors to some extent [3], [4]. At their essence, fighting games require the player to be as unpredictable as possible while predicting the opponent's behavior patterns.

This dynamic however is either not captured by commercial games or developers are not transparent about whether or not it is present. Transparency in how artificial opponents work is important as players show bias when playing against them, and assume these opponents will not play intelligently or follow reasonable patterns [5]. Another factor we need to consider is the believability of the artificial opponents' behavior. Opponents exhibiting human-like behavior make for more engaging play [6], since they mimic the way people think and so players are more inclined to be competent against them as they would be against a human opponent. Only one commercial game successfully tackled these problems and was transparent about the solution [7], but the method involved copying past human behavior rather than reasoning.

1.1 Motivation

How would having human-like, fair artificial opponents for fighting games be beneficial? For one, we can have a range of opponents that better capture the experience of playing against a human opponent. With this, we can set up controlled learning environments in which new players can learn to play against opponents adjusted to their skill level, facilitating learnability which is difficult to attain in these games [8]. Another benefit is for playtesting: added to the use of human testers, we can have a population of artificial agents learn to play the game in order to assess its balance, assuming they are good representatives of human players [9]. This is especially important considering these games tend to be updated over time, with changes in mechanics that alter the game's balance. As such, we try tackling these problems by developing an artificial agent incorporating a reaction time constraint inspired by how reaction time works in human beings, which we believe should result in a more fair and human-like agent.

1.2 Approaches and challenges

There exist various approaches to the development of artificial agents, with the most basic being artificial intelligence (AI) methods involving the manual creation of rules that determine the agent's behavior. For instance, we can define a rule where, at some distance from the opponent, the agent performs one of three specific moves with equal probability. However, these methods not only require extensive expert knowledge and labor but also have limited performance [10]. Another approach is to use machine learning (ML) methods, which involve autonomous learning of behaviors or rules to accomplish a given task. These methods remove the burden of manually specifying these behaviors, but in turn are more unstable, unpredictable and less interpretable. Still, ML methods are shown to be successful in practice, achieving results that were hardly attainable in various tasks.

Of the different ML paradigms, we choose to focus on reinforcement learning (RL). RL is concerned with an agent learning to behave optimally in an environment. The agent perceives the environment and can perform actions on it according to what it perceives. The environment, in turn, changes and reacts to these actions, rewarding or penalizing the agent accordingly. The goal of the agent is to find behaviors capable of maximizing the reward received from the environment [11]. This general definition can be applied to an extensive number of situations, such as robotics [12]. The reward signal in RL is flexible, and is even able to characterize the behavior of human beings and animals, allowing the exhibition of seemingly intelligent behavior [13].

We study the application of RL methods to the creation of artificial opponents for fighting games. One reason is that, as previously mentioned, RL is capable of modeling reward-driven human behavior, which motivates its usage for attaining human-like behavior. The other is that we can create autonomous agents that, by themselves, learn to play the game appropriately. We can specify the goal easily by just rewarding or penalizing the agent at the appropriate times. In the case of fighting games, if the goal is to win, then we can reward the agent the

moment it wins a match and penalize it when it loses. This brings plenty of flexibility: if we want the agent to accomplish any other specific goal, all we need to do is provide appropriate rewards. For instance, if we want to create a limited agent emulating an easier difficulty, we can restrict the agent's action space (for instance, by removing special moves), resulting in an opponent that learns to act the best it can given the options laid out to it but using a less complex action set. Another is for practice. We may want an opponent whose goal is not to win explicitly, but to exploit specific bad habits of the human player so that they can learn to play appropriately. One example to achieve this is to reward the agent when they perform "counter hits", which happen when the player is hit while performing any attack.

In practice, however, specifying a reward scheme that is effective in allowing the agent to learn appropriate behaviors is difficult. For instance, the reward scheme that was just proposed is *sparse*, as we do not give any kind of feedback to the agent during the matches. This makes it difficult for the agent to learn what behaviors caused it to receive the reward, the so-called *credit assignment problem*. Added to this are other problems that make the application of RL in practice challenging, such as slow learning and instability.

Fighting games themselves bring their own set of challenges in developing an artificial agent. For one, time steps are processed at a fixed rate of 1/60th of a second, which imposes restrictions on performance. Additionally, because reaction time is incorporated, the agent will perceive the environment with some delay, making it more difficult in practice to learn the effect of current actions. Also, the agent should learn to perform specific, coherent sequences of actions characteristic of these games. Finally, learning the behavior patterns of the current opponent is key to success in these games, rather than finding a strategy that is only good against any opponent on average.

1.3 Objectives

Current applications of RL to fighting games only tackle some of the previously mentioned issues, with many not considering the reaction time constraint or not exploring quick adaptation to new opponents. In this work, we explore an RL-based solution to these problems, outline its performance and perform a study on the challenges in achieving an artificial agent appropriate for human play.

Still, we leave some aspects to be handled. We are interested in creating a purely competitive agent, leaving non-competitive behavior aside which might be important for attaining believability. Additionally, in our specified RL setting, the agent perceives the environment by receiving the specific values of the different game variables, such as the positions or health of the players. This contrasts with how human players mainly play these games, which is by visual perception. Because it is possible to use strategies based on visual obstruction or ambiguity, the agent will lose some fairness by becoming impervious to such tactics. But since these situations are rare in practice and we focus our attention on the learning problem rather than perception, we leave a visual-based agent for future work.

1.4 Document structure

The document is structured as follows:

- **Chapter 2, Background** introduces the main concepts essential for understanding the rest of the work, mainly RL and fighting games;
- **Chapter 3, State of the art** presents past and current approaches to developing artificial agents, mainly for fighting games, according to different aspects such as performance and adaptability to the opponent;
- **Chapter 4, Methodology** details the problem and the proposed solution to be evaluated;
- **Chapter 5, Evaluation** contains the evaluation of the solution and discussion of the results, as well as details about the game environment used;
- **Chapter 6, Conclusions and future work** summarises this work, presenting its limitations and future work.

CHAPTER 2

Background

Introduction to the area and concepts.

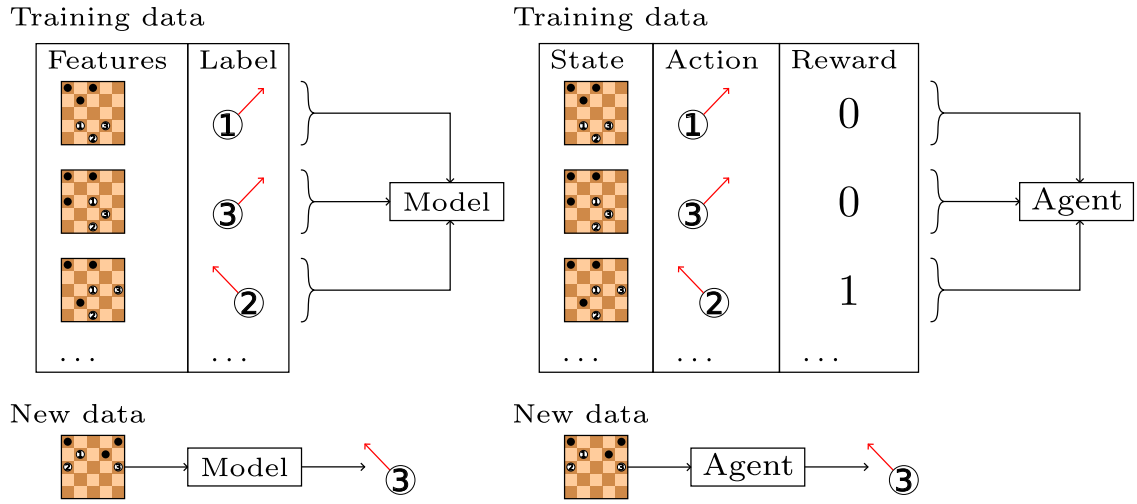
2.1 Machine learning

Machine learning (ML) is a sub-field of artificial intelligence (AI). AI encompasses problem-solving methods usable in computer programs to solve tasks usually requiring intelligence akin to that of humans. Examples of problem-solving methods are tree search algorithms such as Monte Carlo tree search (MCTS) for planning, which can be used to create programs capable of playing digital games. ML is more specific in that the problem-solving methods are not defined explicitly. Rather, what is defined is the problem and a learning algorithm that will autonomously form a solution to that problem. Within the field of ML, there are three main paradigms that can be differentiated by the class of tasks that they focus on. The following subsections introduce each of these paradigms, with Subsection 2.1.4 presenting an overview of artificial neural networks (ANNs) and deep learning (DL).

2.1.1 Supervised learning

Supervised learning (SL) is an ML paradigm where an algorithm learns to predict or classify phenomena from a set of labeled training examples. Each training example is characterized by a set of features and a label, and the objective is to predict the label associated with the features. In SL, we expect the predictor to learn not only to correctly predict the labels of the training examples, but also to correctly predict the labels of examples it has not seen during training, i.e. to generalize.

SL can prove to be useful for our objective of building an agent with reinforcement learning (RL), by using imitation learning for instance. Imitation learning is learning to imitate the behavior of the training examples, where the features are the current environment state and the chosen action is the label. Whereas in RL we learn behaviors by considering the reward signal from the environment, in SL we learn behaviors by imitating what an example



(a) Imitation learning (supervised learning) **(b)** Reinforcement learning

Figure 2.1: Illustration of the difference between imitation learning and reinforcement learning on a simple version of checkers.

agent would do in each situation, without regard for the reward signal. Figure 2.1 illustrates the difference between imitation learning using SL and RL on a simplified version of checkers. We can then merge both learning paradigms, with SL being able to guide RL agents toward effective behaviors during training, allowing learning to be more efficient. RL is detailed further in Subsection 2.1.3.

2.1.2 Unsupervised learning

Unsupervised learning (UL) is similar to SL. The main difference is that in UL the training data is not labeled. Instead, the objective of a model is to predict a label for the given features. In other words, UL methods find patterns or structure in the training data. One example of UL is clustering, where we attempt to group examples in the training data according to their similarity, such that examples in one group are similar to examples in the same group but dissimilar to those of other groups.

Although some similarities between UL and RL could be drawn, this paradigm still differs from RL. For instance, UL methods do not address issues arising from continuous interaction with an environment, such as the exploration-exploitation dilemma mentioned further [11, Section 1.1]. Unsupervised learning does not have obvious applications in the RL setting, and as such we do not focus on it in this work.

2.1.3 Reinforcement learning

Reinforcement learning (RL) is related to the development of autonomous agents that perceive and interact with an environment through sensors and actuators, respectively. The environment rewards and penalizes these agents, whose goal is to maximize the expected return, i.e. the cumulative reward they receive over time. Figure 2.2 illustrates the aforementioned interaction loop between an agent and its environment. The agent perceives state s and

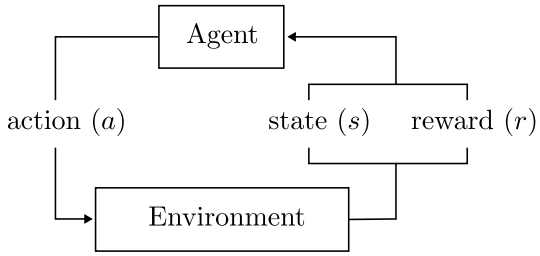


Figure 2.2: Basic interaction loop in a reinforcement learning setting.

receives the reward r for transitioning into s, and then performs action a which will in turn have an effect on the environment.

Formally, an RL environment can be defined as a finite Markov decision process (MDP). In our case, we assume the environment is discrete, i.e. interaction occurs in discrete time steps t, rather than continuous time steps as in real life. Figure 2.3 shows an example of an MDP formulated for the game of tic-tac-toe. This MDP is a tuple $(\mathcal{S}, \mathcal{A}, P, R)$ containing:

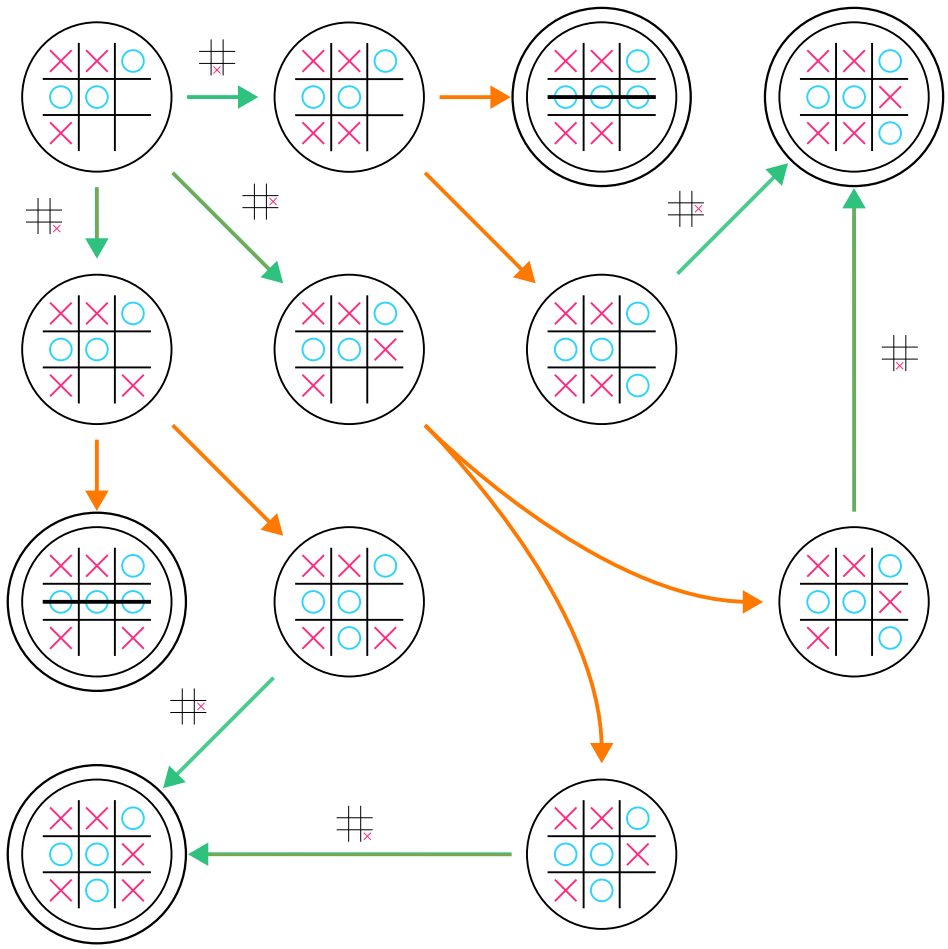
- $\mathcal{S}$, the set of all environment states;
- $\mathcal{A}$, the set of all performable actions. This is a simplification, as the set of actions that can be performed may be conditioned on the current state s;
- P, a function $P(s'|s, a)$ returning the probability of transitioning to state s' when performing action a in state s;
- R, a function $R(s'|s, a)$ returning the reward r for transitioning to state s' after performing action a in state s. The reward being deterministic is a simplification, and it is not the most general definition.

The goal of an RL algorithm is to find a policy π, which is a function $\pi(a|s)$ returning the probability of taking action a in state s, that maximizes the expected return. The return is the sum of future reward when acting with π on the environment, and the reward may be "discounted". The discounted return G_t from a reference time step t can be defined as

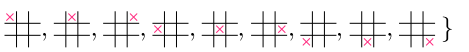
$$G_t \triangleq r_{t+1} + \gamma\, r_{t+2} + \gamma^2\, r_{t+3} + \cdots = \sum_{k=0}^{T} \gamma^k\, r_{t+1+k} \tag{2.1}$$

with r_t being the reward received at time step t and T being the time at which the environment terminates, which can be $T = +\infty$ for environments that do not terminate (continuous) or $T \in \mathbb{Z}^+$ for environments that terminate at some time in the future (episodic). The discount factor γ, a value between 0 and 1 inclusive, can be included so that we give more weight to rewards closer to the current state in time rather than rewards farther into the future. With $\gamma = 0.0$ we only consider the next immediate reward, whereas with $\gamma = 1.0$ we are infinitely far-sighted. Note that it does not make sense to have $\gamma = 1.0$ and $T = +\infty$, since the sum is not guaranteed to converge; for example, in an environment with only zero or positive reward any policy π would be a solution. The RL objective can thus be formulated as finding the policy π which maximizes the expected discounted return G_t as in

$$\arg\max_{\pi}\ \mathbb{E}_\pi\left[G_t\right], \tag{2.2}$$



State set: $\mathcal{S} = \{$  $\}$

Action set: $\mathcal{A} = \{$  $\}$

Reward: $\begin{cases} 1 & \text{on win} \\ -1 & \text{on loss} \\ 0 & \text{otherwise} \end{cases}$ Transitions: orange arrows

Figure 2.3: Example of an MDP for tic-tac-toe. To simplify, we illustrate it by starting with the state at the top left corner, which is near the end of the game. Green transitions are actions made by the agent, orange transitions are environment-intrinsic transitions. The probability of the orange transitions, of which the agent is not usually aware of, is dependent on the opponent in this case. In this scenario, the agent never perceives the states that are endpoints of green transitions, only states that are endpoints of orange/environment transitions. Double-circled states are terminal states, which means the environment terminates and the agent cannot act further. Note that the reward function is arbitrary, and the one we define is to serve as an example where we merely want the agent to learn how to win.

with the expectation $\mathbb{E}_\pi$ being performed over trajectories experienced while following π. RL algorithms essentially involve the agent exploring the environment through trial-and-error, keeping track of the rewards received as the environment reacts to its actions, and developing and following a policy indicating behavior that maximizes these rewards.

Figure 2.1b shows an example of RL in a simplified version of checkers, in which we reward the agent for capturing opponent pieces. Contrary to SL in Fig. 2.1a, in RL the agent itself is the one performing the actions, and so the action labels, and by extension the training data/experience, are provided by the agent. This brings issues in terms of efficient exploration of the environment, as the agent has to sufficiently *explore* the environment to get enough training data so that it learns as much as possible about it. But the agent also has to *exploit* and maximize reward, incurring in obtaining experience it is already familiar with, depriving itself of discovering new behavior that is potentially better than the current one. This exploration-exploitation dilemma is at the heart of RL, and is a difficult issue to tackle.

RL theory is predicated on the fact that environments, defined as MDPs, obey the Markov property, i.e. the probability of transitioning to state s' only depends on the previous state s and the action a executed on it, not on states or actions previous to s. This is important if we want to guarantee theoretical properties when designing an MDP for a particular problem. Having the Markov property ensures we are including all information at every time step needed for the agent to perform correct decision making, without needing knowledge of past states or previously performed actions. But even though it provides desirable properties that would, in theory, allow for greater effectiveness and efficiency, it is not strictly necessary in practice for an RL agent to succeed in a given environment. Consider imperfect information games like poker for instance, where players do not have access to the entirety of the environment state needed to fully describe it (the players' hands), but can nevertheless perform well.

Other concepts worth introducing are that of a value function and an action-value function. A value function $V_\pi(s_t)$ returns the discounted expected return when following the policy π after acting in state s_t at time t. An action-value function $Q_\pi(s_t, a_t)$ returns the discounted expected return after performing action a_t in state s_t at time t and then following the policy π. The value function $V_\pi(s_t)$ and action-value function $Q_\pi(s_t, a_t)$ can be written with respect to one another, according to

$$V_\pi(s_t) = \mathbb{E}_\pi \left[G_t | s = s_t\right] = \sum_{a \in \mathcal{A}} \pi(a|s_t) Q_\pi(s_t, a) \tag{2.3}$$

and

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi \left[G_t | s = s_t, a = a_t\right] = \sum_{s \in \mathcal{S}} P(s|a_t, s_t)(R(s|a_t, s_t) + \gamma V_\pi(s)), \tag{2.4}$$

with the expectation $\mathbb{E}_\pi \left[G_t | s = s_t\right]$ assuming that the start state is s_t and $\mathbb{E}_\pi \left[G_t | s = s_t, a = a_t\right]$ assuming that both the start state and action are s_t and a_t respectively when computing the return G_t. As such, we may also rewrite $V_\pi(s_t)$ and $Q_\pi(s_t, a_t)$ as a function of themselves on the immediate future state $s_{t+1} = s$, according to

$$V_\pi(s_t) = \sum_{a \in \mathcal{A}} \pi(a|s_t) Q_\pi(s_t, a)$$
$$= \sum_{a \in \mathcal{A}} \pi(a|s_t) \sum_{s \in \mathcal{S}} P(s|a, s_t)(R(s|a, s_t) + \gamma V_\pi(s)) \tag{2.5}$$

and

$$Q_\pi(s_t, a_t) = \sum_{s \in \mathcal{S}} P(s|a_t, s_t) \left(R(s|a_t, s_t) + \gamma V_\pi(s)\right)$$
$$= \sum_{s \in \mathcal{S}} P(s|a_t, s_t) \left(R(s|a_t, s_t) + \gamma \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a) \right). \tag{2.6}$$

We say that V_* and Q_* are the value and action-value functions when π is the optimal policy π^*, i.e. the best possible policy, which is the solution to the RL problem. Whenever we mention V or Q without specifying a policy, then it is because it is not relevant in that context. These concepts are important when discussing the application of RL to our problem.

Throughout this work, we use some more terminology to refer to concepts of the RL problem. Firstly, we distinguish between "states" and "observations": "state" refers to all variables that define the environment, whereas "observation" is a subset of those variables, which is what the agent has access to. It is perfectly possible to have observations match the environment state, which brings the benefit of the agent not having to rely on the past since the state has all the information needed to predict the future. This is not possible in many scenarios however, and this decoupling allows representing cases such as those of a robotic agent learning to interact in the real world; the agent perceives the world through sensors, which cannot capture everything that rules the world (if we assume the world to follow an MDP). Other examples are video games, in which the agent may just have access to the display but not to the internal variables that define the game, or imperfect information games in general. Nevertheless, the agent needs to learn how to act having only access to the observations. In these cases, we have a partially observable Markov decision process (POMDP) rather than an MDP. In POMDPs we define, in addition to the MDP components, the observation set Ω and the function $O(b|s', a)$ that returns the probability of observing b when the agent transitions into state s' after performing action a. In POMDPs, we try to create approximations of the true states s from the history of observations b. This distinction between states and observations is worth making when we apply RL to our problem, since we introduce a reaction time constraint that forces the agent to only have access to imperfect observations.

To the set of all possible states we call "state space", the set of all possible observations "observation space" and the set of all possible actions that the agent can perform "action space". The cartesian product of the observation and action spaces defines the domain of the policy π, although, as was previously mentioned, depending on the environment there are actions that cannot be performed on some environment states, and so the action space itself is dependent on the current state. This is the case for fighting games since there are states in which some actions are ineffectual. The way this is handled is detailed in Subsection 4.2.2.

Value function learning

Value functions are useful constructs to learn in an RL problem. From them, depending on the problem, we are able to determine an appropriate policy to follow. For instance, if we learned the optimal action-value function Q_*, then we are able to follow a policy that, at each state s, simply chooses the action a with the greatest return $Q(s, a)$, according to

$$\pi(a|s) = \begin{cases} 1 & \text{if } \ a = \arg\max_{a'} Q^*(s, a') \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

and, in this case, π happens to be the optimal policy π^*.

But how can we compute/learn these functions? There are different kinds of methods, of which three types are briefly introduced here. Firstly, let's consider only learning the value function, as learning the action-value function is analogous since the ideas translate directly from one case to the other. Then, let's consider the expression (2.3) for a value function. One possibility for computing $V_\pi(s_t)$, and perhaps the most straightforward, is to use a Monte Carlo (MC) approach. MC methods approximate an expected value by taking samples. In this case, we want to take samples of the return G_t we obtain when following policy π. As such, all we have to do is interact with the environment from s_t and keep track of the discounted sum of rewards $\hat{G}_t$. After the environment terminates, we can assume $V_\pi(s_t) = \hat{G}_t$. Then, we can interact with the environment again from s_t and repeat this process. In the end, we will get various returns G_t for a single $V_\pi(s_t)$; all we have to do then to approximate the expectation is to simply average all of them.

Two other possible methods can be derived if we consider the recursive form (2.5). From this formulation, we can write $V_\pi(s_t)$ in terms of the value of another state at the next time step $s_{t+1} = s$. If we are learning an estimate of $V_\pi(s_t)$, then we can use estimates on future states s_{t+1} to update the estimate for s_t. This is called *bootstrapping*. The only problem in (2.5) is that the environment's transition dynamics $P(s_{t+1}|s_t, a_t)$ are generally unknown. In a similar vein to MC methods, we can instead utilize samples from the environment to approximate $P(s_{t+1}|s_t, a_t)$. For the case of learning $V_\pi(s_t)$, we can even consider samples of $\pi(a_t|s_t)$. As such, from an environment transition of s_t to s_{t+1} sample we can repeatedly perform an incremental update

$$V_\pi(s_t) \leftarrow V_\pi(s_t) + \alpha \left(R(s_{t+1}|s_t, a_t) + \gamma V_\pi(s_{t+1}) - V_\pi(s_t) \right), \tag{2.8}$$

with s_{t+1} being the state the agent ended up in after performing a_t in s_t and α being the learning rate. This update is essentially a linear interpolation between the current prediction $V_\pi(s_t)$ and the target value $R(s_{t+1}|s_t, a_t) + \gamma V_\pi(s_{t+1})$ which, although also uses an estimate and is probably wrong, it should at least be more accurate than our current prediction since it incorporates an accurate value of $R(s_{t+1}|s_t, a_t)$. The difference between the target and current prediction $(R(s_{t+1}|s_t, a_t) + \gamma V_\pi(s_{t+1}) - V_\pi(s_t))$ is called the *temporal difference*, with the methods using this kind of updates being called temporal difference (TD) methods.

But what if we know the transition function $P(s_{t+1}|s_t, a_t)$ of the environment? Then we can just perform updates on many states s independently, utilizing dynamic programming (DP) algorithms. In this case we do not sample, and just perform updates on all states s independently until we converge.

In this work, we are interested in TD methods. DP methods are prohibitive since they not only require a perfect model of the environment, but they are also computationally expensive for large state spaces. Although in digital environments we always have access to a model of the environment, which is our case, the state space of the games we consider in this work is too large to be updated frequently. For the case of *FOOTSIES*, a relatively simple fighting game that is explained in Appendix A.1, we have over 1.08×10^9 states if we discretize one of the continuous variables into just 10 bins. For games with more state variables, there is a combinatorial explosion which makes these methods infeasible in practice. Also, they perform updates over *all* states, even though there are some states that the agent will not experience or will rarely do so. From this perspective, a method that utilizes agent experience and traverses the states that are actually relevant for the agent to learn should be more adequate. In MC methods we sample the environment, but do not bootstrap. Compared with TD methods, MC methods cannot perform updates online, i.e. cannot update at every time step and need to postpone learning to the end of an episode. Additionally, MC methods also have the disadvantage of only being applicable to environments with some defined form of termination, such as episodic environments, so that we can have a sample of the return for a value function update. On the other hand, TD methods are independent of this, and are thus more generally applicable. Finally, TD methods end up learning faster than MC methods in practice [11, Section 6.2].

Although in this work we do not explore methods solely based on value function learning for fighting games, we still utilize TD in our learning algorithm. We combine value function learning with policy gradient methods, another family of RL algorithms introduced further.

Policy gradient methods

The value-based methods previously introduced indirectly represent the policy, ideally by being one that attempts to choose actions that maximize the calculated value as in (2.7). Another class of methods is policy gradient (PG) methods, which directly learn the policy π. These methods seem more appropriate since we are actually learning the component we are looking for. PG methods learn a parameterized policy $\pi_\theta(a|s)$, which is some mathematical function of the state s that outputs the probability of performing a, and has parameters θ that can be tuned. These parameters are updated to maximize the objective $J(\theta)$ defined as

$$J(\theta) \triangleq V_\pi(s_0), \tag{2.9}$$

standing for the expected discounted return starting in state s_0, the initial state. This means that we want the policy π_θ to have behavior that maximizes the return from the beginning of interaction, and in our case the game. The parameters are then updated according to the gradient of this objective, named the *policy gradient*

$$\begin{aligned}\nabla J(\theta) &\propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} Q_\pi(s,a)\nabla\pi_\theta(a|s) \\ &= \mathbb{E}_\pi\left[\sum_{a\in\mathcal{A}} Q_\pi(s,a)\nabla\pi_\theta(a|s)\right] \end{aligned} \tag{2.10}$$

where $\mu(s)$ is the on-policy state distribution, i.e. the probability of ending up at s by following π. This result stems from the *policy gradient theorem* [11]. Intuitively, this means that we have to update the parameters θ so that, for action a performed in s, we increase its probability $\pi_\theta(a|s)$ if $Q_\pi(s,a)$ is positive, and decrease the probability if it is negative, with the update's magnitude being proportional to $Q_\pi(s,a)$. In other words, we increase the probability of good actions and decrease the probability of bad actions. Also, the update is further proportional to how often the agent ends up in state s, which makes sense since we want to focus updates on states that the agent actually experiences. Within PG methods, there are different ways we can approximate the gradient $\nabla J(\theta)$, since we cannot compute it in practice due to $\mu(s)$ which requires taking all possible states $\mathcal{S}$ into account. They vary mostly on how the sum inside the expectation in (2.10) is computed, with trade-offs in bias and variance. We focus on actor-critic methods, which learn the action-value function $Q_\pi(s,a)$ using the TD methods described in Subsection 2.1.3. Appendix A.2 gives a better explanation of learning parameterized functions according to some arbitrary objective $J(\theta)$.

It is worth noting that, if discounting is used, i.e. $\gamma < 1$, this gradient is slightly different in that $\mu(s)$ should be the discounted state distribution $\mu_\gamma(s)$ [14]. This has implications in PG algorithms, which are often ignored in practice. As such, many methods commonly utilized in practice are not theoretically correct. One of the reasons the effect of the discount factor on the gradient is ignored is that it causes updates that are farther from the initial state s_0 to contribute less, which makes sense since the objective is defined as the return of the initial state s_0, and so actions closer in time to $t = 0$ are more important. However, this makes it so that, after some timesteps, the agent stops learning altogether. In our case of fighting games, this is especially prominent since many timesteps have to occur before meaningful interactions with the environment, and so the gradients are discounted heavily very quickly. A different formulation that might be more apt is that of *average reward*, which considers the environment to be continuous. Since that formulation is not intuitive for our environment, which lends itself to be episodic, and is not often used in practice, we leave it for future work.

Hierarchical reinforcement learning

Hierarchical reinforcement learning (HRL) is a subset of methods in RL that represent the agent's policy hierarchically. The policy is subdivided into different layered policies, where policies at a higher level choose options or goals to which the lower-level policies are conditioned. The policies at the lowest level are the ones that choose the actions from the environment, which in this context are called *primitive* actions. An option or goal chosen by a high-level policy is persisted for some time.

In theory, these methods present many benefits. One is temporal abstraction, which allows reasoning over extended periods rather than just the current time instant. Another is allowing for hierarchical planning, which is similar to how humans rationalize many problems. For instance, when we want to approach a door, we do not think about the individual leg motor control required to move our legs, we merely think about walking in the direction of the door when forming a plan for this problem. Yet, walking is itself a skill necessary for higher-level tasks, which needs to be learned as well. This also allows abstraction of a problem, which should make learning analogous problems much easier (walking can be used for tasks other than approaching a door). The hierarchical structure of the policy additionally allows for better interpretability. Overall, HRL methods have the ability to mimic the way biological beings think [15].

These hierarchical organizations can be built manually by a human expert, but work has been done in letting agents learn these organizations autonomously. However, learning problem abstractions remains an open research question. Even then, HRL methods either tend to rely heavily on domain knowledge, limiting their applicability to various domains, do not scale well to complex environments or are sample-inefficient, requiring many environment interactions to learn [16].

One possibility of HRL for fighting games is representing long specific sequences of actions as options/goals. One example is representing "special moves", which require precise sequences of primitive actions, detailed further in Section 2.2. This is similar to how human players plan in terms of these moves rather than in terms of the individual primitive actions.

2.1.4 Deep learning

One final concept worth introducing is deep learning (DL). DL corresponds to the application of an ML method with deep neural networks (DNNs), which can provide results hardly achievable with previously known methods.

Let us consider the SL problem of Fig. 2.1a. If we want to predict the label according to the features, we could use the naive approach of keeping track of a table that dictates, for each encountered state in the features, which action to perform. But if for the same state multiple actions were performed, then we also need to keep track of a probability distribution for every encountered state. Considering this, we would have a table containing $C_6^{5^2} \times 2^3 \approx 1.42 \times 10^6$ elements for our simple checkers example, where C_k^n denotes the number of k-element combinations from a set of n elements. This is possible in practice, but for the actual game of checkers the table would contain $C_{24}^{8^2} \times 2^{12} \approx 1.03 \times 10^{21}$ entries, which is intractable. Of course, many of these states are either impossible or extremely rare, but it nevertheless illustrates the problem in dimensionality. Another problem is that this tabular method does not generalize to similar states unseen in the training data. This generalization would have to be defined manually in some form.

One structure for tackling both dimensionality and generalization is an ANN. ANNs are function approximators, i.e. they approximate a function $f(x) = y$ by being trained on the input x (features) and output y (labels). ANNs are useful when knowing the function f is

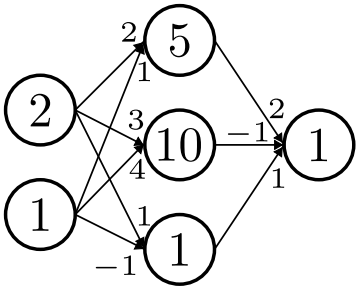


Figure 2.4: Example of a simple linear artificial neural network, with an input layer with one node, a hidden layer with three nodes and an output layer with one node. The value at each node corresponds to the summation over the multiplication of the connection's weights with their input.

either extremely difficult or impossible.

ANNs are chains of nodes representing mathematical operations over the input, inspired by biological information processing systems such as the brain. Figure 2.4 illustrates a simple linear ANN, containing weights on each connection which can be adjusted when the network is learning. In practice, the power of ANNs to represent various functions increases when we introduce non-linear operations, such as exponentiation. If, in addition to the nonlinearity, we increase the network's *depth*, i.e. the number of hidden layers including the output layer, we increase the network's ability to form abstract features over the input. However, introducing nonlinearity and depth comes at the cost of making the networks harder to train, and taking more time to learn. And so, to a deep non-linear ANN we call deep neural network (DNN).

The application of DL to RL in particular is called deep reinforcement learning (DRL), and it has become prevalent in the application of RL in practice. For instance, we can use DNNs to approximate the value and action-value functions V and Q, or even to create a policy π. DRL is also explored in this work.

2.2 Fighting games

Fighting games are competitive video games in which players fight each other in an arena. This work focuses on the subgenre of real-time one-on-one fighting games where movement is restricted to a two-dimensional plane, to which we refer as "traditional" fighting games hereafter. Famous examples of these games are those in the *Street Fighter* series. Players have a set amount of health, and the objective is to deplete the opponent's health within the time limit or remain with the health lead otherwise. Fighting games have a multitude of characters, and each player controls one of said characters. Each character has a specific set of offensive and defensive moves that the player has to make use of to succeed. Additionally, characters have access to "special moves" unique to them, usually performable using a combination or a precise sequence of actions.

Figure 2.5 illustrates the typical game elements of a traditional fighting game using an adapted screenshot from *FOOTSIES*[1]. Players 1 **(a)** and 2 **(b)** are facing each other on

[1] `https://hifight.github.io/footsies/`

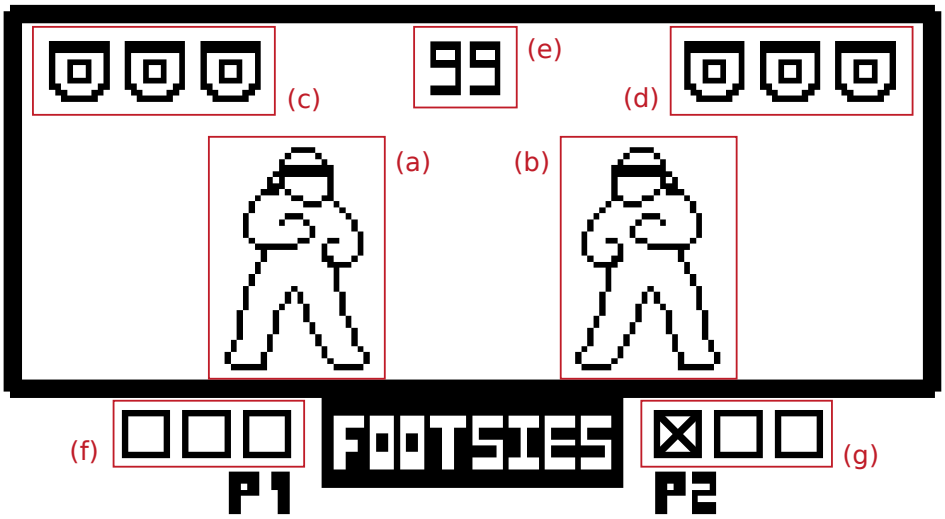


Figure 2.5: General structure of a game state in fighting games. Screenshot taken from *FOOTSIES* and adapted.

opposite sides of the arena. Each player's current health is indicated in **(c)** and **(d)**, and their number of rounds won in **(f)** and **(g)**. The time limit is shown in **(e)** as a countdown timer. The first player to get to the maximum number of won rounds wins the match.

Other important concepts worth introducing are those of "frames", or time steps, hitboxes and hurtboxes. Fighting games are discrete-time environments, with most of them processing frames with a fixed time interval between them of 1/60th of a second. As such, for every second occurring in real-time 60 frames are processed. This is important when discussing the duration of each performable move in the game. Figure 2.6 presents the duration of some moves in *FOOTSIES* in terms of frames. All presented moves, except walking forward and backward, take some time to fully perform, during which no other action can be made. For instance, both "attack" and "special move" take some time to perform, with the "special move" taking twice as long.

Figure 2.6 also shows the hitboxes and hurtboxes, represented as red and yellow rectangles respectively. These boxes determine whether damage is dealt to a player: if a player's hurtbox overlaps with the opponent's hitbox, then the player takes damage and loses health. With this, we can distinguish between neutral and damaging moves depending on whether they have a hitbox or not. The frames in which the hitbox appears are called "active frames", with the frames leading up to it being called "startup frames" and the frames until the player is actionable again "recovery frames". Notice how the presented damaging moves are in effect in less than 12 frames, corresponding to 200 milliseconds. This is less than the expected human visual reaction time [7], and has implications on how strategies in fighting games are formed, explained further in Subsection 2.2.1.

Moves also have various other properties that set them apart, presenting different risk-reward ratios and advantages or disadvantages depending on the situation. For example, the "dash backward" move in Fig. 2.6 has no hurtbox at the beginning of its duration, giving it the ability to avoid attacks. However, if the opponent anticipates this move, then they can

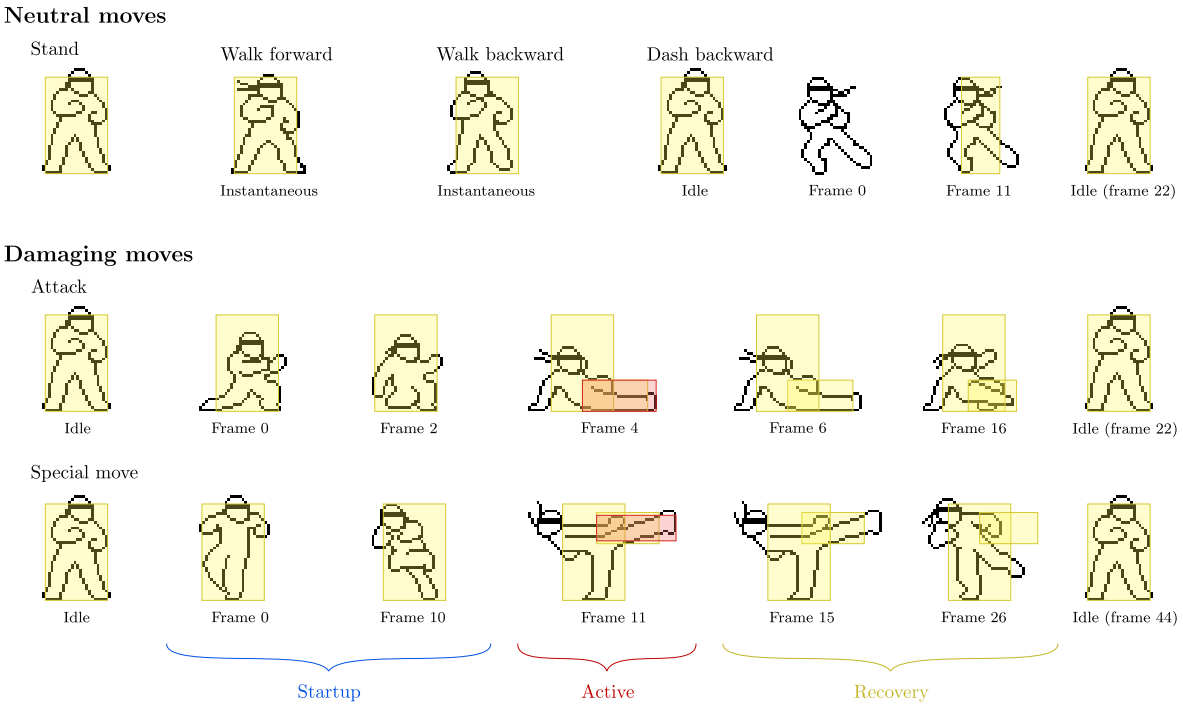


Figure 2.6: Illustration of the duration of actions in *FOOTSIES* in terms of frames, or time steps.

perform the illustrated "special move", advancing forward and hitting the dash when it already has the hurtbox. If the moves were by coincidence done at the same time, then the "special move" would win against the backward dash, at frame 11. Of course, the "special move" itself also loses in other situations. This dynamic makes up the non-transitivity of fighting games, similar to how rock-paper-scissors works, and will be detailed further in Subsection 2.2.1.

One other prominent feature of fighting games is the ability to perform "combos". A combo is a sequence of attacks performed with a short time interval between attacks, where if it hits the opponent it leaves no gaps for them to act back [17]. Since the release of *Street Fighter II*, combos have been a standard mechanic in fighting games, and are an important ability to learn in order to optimize play. Artificial agents should thus have the ability to perform them, which requires learning to perform sets of particular sequences of actions.

2.2.1 Strategy

Fighting games have imperfect information, contrary to most board games for instance. That is, each player does not have full knowledge of the current and previous environment states when making a decision. Considering chess as an example, each player has access to the full state of the board at all times, which is all that is required to make appropriate decisions. This contrasts with poker, another imperfect information game, in which all players' hands except their own are hidden to themselves, and so players have to make decisions at every instant without actually knowing the entire game state. Fighting games are real-time games with asynchronous actions from both players. Since most of these actions are too fast for a human being to react to, as illustrated in Fig. 2.6, players can never make instantaneous decisions based on those actions. That is, at each instant players have to make decisions

without being sure of what the opponent is doing at that moment, and so without knowing the whole game state. As such, it requires predicting what the opponent will do, and in fact, prediction-based play is the essence of these games. Because of this, a fair artificial agent should emulate this reaction time and thus rely on prediction as well as reaction, or else it would not be playing by the same game rules as a human player [7], [18].

Human players can reliably react to some moves from the opponent, but in many situations that is either not possible or extremely difficult to do. This is usually because reaction time is dependent on the number of options that the player expects the opponent to do. According to Hick's law, the relation between human reaction time and the number of options the human has to react to is logarithmic in the number of options [19]. More precisely, we say that this reaction time is the *choice* reaction time since choosing the appropriate response to one of many stimuli is required. Intuitively, if we expect the opponent to soon perform a quick action, we simply have to react to that one action by performing one appropriate move, which incurs in small reaction time. Otherwise, if the opponent is unpredictable and can perform one of many options, each of which requires different responses, then reaction time is slightly delayed due to greater uncertainty. However, there are exceptions in practice in which Hick's law does not seem to hold [20], [21]. Our case is one of sequential decision making, which is studied in particular by [20]. In this scenario, the relation between reaction time and uncertainty in the sequence is better explained by a more complex model. This difference will be explored in Chapter 4.

Because of the prediction-based play and non-transitivity of actions, fighting games can be for the most part thought of as multiple consecutive trials of rock-paper-scissors [3]. Decision making is done without knowledge of what the opponent's chosen action is, so they are mostly predictive rather than reactive. Unlike rock-paper-scissors however, the trials are not independent of each other, i.e. the outcome of one trial influences the trials performed in the future. Additionally, each option does not yield the same risk or reward, and has long-term consequences that influence the reward that that option is expected to bring in the future. This is one reason why RL methods could be helpful, by approximating the expected return that each option yields in every trial. Finally, since each player's actions take some time to perform, these rock-paper-scissors trials in fighting games are extended in time rather than being decided in an instant.

Figure 2.7 illustrates one instance of this rock-paper-scissors dynamic in the *FOOTSIES* game. At the beginning of each round, no player has the advantage, and they engage in a careful back-and-forth play. There are three main strategies that they can employ. One of them is to simply approach the opponent and attack them. To counteract this, the opponent can simply attack preemptively to prevent our approach. If the opponent is attacking preemptively, we can in turn wait for them to miss an attack and hit them while they are recovering, also known as "whiff punishing". However, the opponent can take advantage of the fact that we are being watchful of a possible attack and approach, making sure that their next attack will hit rather than miss. With this, a cycle is formed, making these strategies non-transitive. This

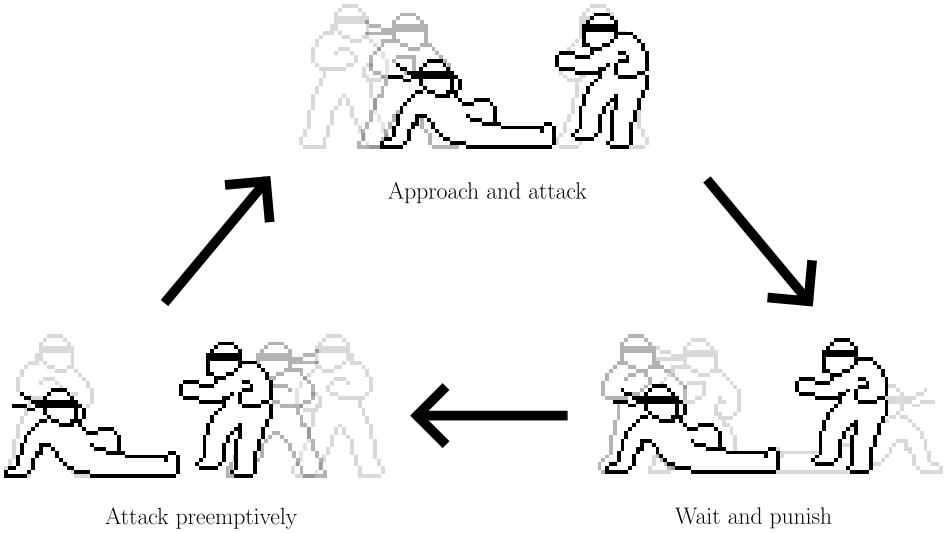


Figure 2.7: Illustration of the rock-paper-scissors dynamic in fighting games, with *FOOTSIES* as an example.

particular state in play is colloquially known as "footsies", commonplace in fighting games [2].

In a competitive game featuring multiple players, the Nash equilibrium is the strategy which if followed by a player, there is no other strategy that could be employed by the other players that is better than the equilibrium. The Nash equilibrium exists for rock-paper-scissors, which involves randomly choosing any option uniformly [22]. However, fighting games are more complex than rock-paper-scissors, and uniform random play may not constitute the Nash equilibrium. An analysis of the Nash equilibrium for fighting games has been done [4], suggesting the Nash equilibrium is indeed a mixed strategy involving stochastic play, but it is incomplete as ample player positioning and the duration of actions were not considered. Still, due to their non-transitivity, we can expect the Nash equilibrium for fighting games in practice to always involve randomness to some extent, so that the strategy cannot be exploited by being predictable.

However, as is in the rock-paper-scissors game, if we are facing an opponent with a specific strategy, we can gain an advantage by attempting to use a counter-strategy, yielding greater return than when the equilibrium is followed. In fact, human players tend to deviate from random play in order to exploit patterns in the opponent's behavior [23]–[25]. Even when attempting random play, human players tend to have biases that drive them to choose actions not randomly but according to some subconscious pattern [26], [27]. This is because humans are not a good entropy source, i.e. cannot generate randomness [28]. This means that, against a specific human player, the Nash equilibrium is not the optimal strategy and, as such, a method that adapts to the opponent will be more appropriate in this scenario. Additionally, in terms of enjoyment, it is more fun for the human player to find and exploit patterns in the opponent than to face an optimally random opponent following a Nash equilibrium. Essentially, in ML terms, we want to *overfit* to the current opponent to a certain extent, rather

[2]Glossary: `https://glossary.infil.net/?t=Footsies`. Video illustrating the "footsies" dynamic, by the developers of *FOOTSIES*: `https://youtu.be/96MKkqOlpKs?t=23`.

than *generalize* to multiple opponents. In RL, we can consider the fighting game environment described here to be *non-stationary*, i.e. it may change over time while the agent is learning, which in this case means the opponent may differ, although we can also consider adjustments to the game itself as a source of non-stationarity, which is also worth exploring.

2.2.2 Challenges

This genre poses challenges in implementing an efficiently adaptive artificial agent:

- Decision time is standardized to 1/60th of a second, since these games currently process game state at a fixed rate of 60 frames per second;
- The games are prediction- rather than reaction-based. Human players cannot react quickly enough to many of the actions that the opponent can make, and these games are built with that in mind. As such, artificial agents should take this into account;
- Combinations of primitive actions (button presses) can be performed;
- Specific sequences of primitive actions, commonly known as "motion inputs", yield special moves that cannot be performed otherwise. These moves tend to be what distinctly defines each character, and are important to use them properly;
- An ability to adapt to the opponent's behavior is paramount. As explained previously, these games share strategic similarities to rock-paper-scissors, and playing optimally against a human opponent requires exploiting biases in decision making rather than being wholly random.

As such, an appropriate agent for fighting games should exhibit consistently efficient decision-making, be prediction-based, be able to execute combinations of actions, perform long-term planning and be quickly adaptive. The long-time planning and action combinations requirements can be slightly alleviated by considering combinations and special moves to be actions themselves, although it makes the action space dependent on the character. Additionally, the input systems of current games are complex in the sense that considering these moves to be single actions robs the agent of some flexibility. Even considering this slight simplification, it is an ambitious goal. Nevertheless, a solution to some of these problems is explored. In Chapter 3, current work that tackles some of these facets is presented.

CHAPTER 3

State of the art

Literature review of past and current approaches for the development of artificial agents.

3.1 Commercial games

Almost all commercial fighting games either do not employ sophisticated artificial intelligence (AI) systems or are not transparent about them. Of note, the 2013 release of *Killer Instinct* contains a "Shadow System", which creates artificial opponents based on play data of human players [7]. An artificial opponent from this system has a replay bank of matches played by a specific human player, and during decision making the agent chooses the moment in the replay bank that most closely resembles the current situation according to a similarity metric. This similarity metric includes heuristics that allow incorporating historical data into the current state, requiring some expert knowledge. After the moment in the bank is chosen, the agent imitates the human player's actions from that moment forward, until a significant state change is detected. This is a form of temporal abstraction since replays are not chosen at every instant, but only when necessary.

This method is an instance of case-based reasoning, in which an AI system utilizes solutions to past problems to solve new similar problems. Although decision making is reduced to copying past human experience, the method ended up working well in practice with a positive reception from the players. The agents are also easy to train, requiring few samples of human play data to exhibit seemingly meaningful and complex behaviors. Because human play is copied, the agents also inherently retain human limitations such as reaction time and execution imperfection. The method also has the added benefit of adapting to the players' strategies as they evolve, by simply prioritizing more recent play data, without needing any further tuning.

However, compared to the approach explored in this work, the AI from the Shadow System does not explicitly perform decisions, but simply imitates other players. This means that the AI is not explicitly playing to win. If the human player it is imitating is losing on purpose, then it will lose on purpose as well. The AI does not have any notion of how good or bad

each decision is, i.e. the replay moment it chooses, in terms of future reward. This allows the AI to exhibit all kinds of behaviors including non-competitive ones such as taunting, but not to autonomously provide insight into how balanced the game is for instance.

There are other noteworthy mentions that, although revealing their general approach, were not fully transparent on the solution. *Samurai Shodown* [29], released in 2019, uses deep learning (DL) for its AI. This system, similar to Killer Instinct's, imitates human players. However, details on the system's architecture or reception by the players have not been disclosed. *Tekken 8*, released in 2024, features a "Super Ghost Battle" mode that allows each player to build an AI that learns to imitate them, akin to the above mentions. The AI utilizes reinforcement learning (RL) to learn, more specifically the Q-learning algorithm [30], and is relatively quick at learning, being able to learn behaviors within a round. However, no more details have been disclosed, and it is not clear whether the reward for the Q-learning algorithm is a reward for closely imitating the human player or for playing the game successfully, only utilizing the human play data to know what experience to use for the updates. The *Super Smash Bros.* series of fighting games feature "Figure Players", which are trainable AI that learn to play similarly to the human player they fight against, scanned and saved through near-field communication (NFC) on amiibo toys [31]. Like the previous mentions, it is not known exactly how these "Figure Players" learn. Overall, we can note that commercial games tend to apply imitation learning for building human-like AI, with *Killer Instinct* being the only one whose learning process is known.

3.2 Learning environments

There are not many fighting game environments built for the development of AI, especially since fighting games are mostly commercial games. Still, two such platforms can be noted, which are detailed in the following subsections.

3.2.1 FightingICE

FightingICE is a fighting game made for research of AI algorithms using any methodology, such as heuristic-based and machine learning (ML) methods, released in 2013. This game works like usual fighting games, with processing occurring 60 times per second and specific action sequences activating special moves [32]. Figure 3.1 shows an example screenshot with players using different characters.

The Intelligent Computer Entertainment Lab[1] at Ritsumeikan University hosts the yearly DareFightingICE competition, in which participants develop AI agents for FightingICE to fight each other in a tournament [33]. The tournaments are divided into two leagues, each having a different objective:

- Standard: the objective is the same as in any fighting game, completely deplete the opponent's health or remain as the one with the largest amount of health when the time limit is reached. This is the goal we focus on in this work;

[1] `https://www.ice.ci.ritsumei.ac.jp/links.html`

Figure 3.1: Screenshot of a round in FightingICE, version 6.1, between the characters ZEN and LUD.

- Speedrunning: defeat the baseline Monte Carlo tree search (MCTS)-based AI (MctsAi) in the shortest amount of time possible.

The main goal of the competition is to advance research in the development of general AI, specifically for fighting games. A general fighting game AI should be able to adapt to and surpass unseen opponents, using any kind of character that the game offers. This work shares a similar goal: strong adaptability to different opponent strategies. The ability to effectively learn any playable character is left for future work.

Artificial agents can perceive the environment using three distinct representations: audio, visual or structured. Audio-based features are a recent addition that has been experimented with in the 2022 and 2023 editions of the competition. It has been proven that, with appropriate sound design, fighting games can be played based on sound cues alone [2]. Utilizing audio-based features for creating artificial agents is also an interesting problem that has not been as extensively explored in the application of RL to video games as visual-based ones. Still, we focus on structured environment representations alone as previously mentioned.

Three playable characters are offered. Until 2021, inclusive, only motion data regarding some of the three characters was publicly provided. Motion data details the numerical properties of each move, such as the data in Fig. 2.6, allowing accurate perception of their effects on the game environment. The intention behind hiding this data was to evaluate robustness toward character change in the competitions, by not allowing participants to train on that data beforehand. The offering of three playable characters, each behaving differently, allows the experimentation of transfer learning.

The game attempts to emulate the reaction time of human players by delaying the state information sent to the artificial agents by 15 frames, which corresponds to 250 milliseconds. However, audio and visual data are not delayed. Even then, the reaction time employed in FightingICE is static, and we explore a dynamic reaction time better mimicking that of

[2]Examples at: `https://www.ice.ci.ritsumei.ac.jp/~ftgaic/Downloadfiles/2022.pdf`

human beings.

One important feature of the game is the ability for the agent to perform simulations of the game on demand. This heavily simplifies the application of algorithms dependent on knowing the environment dynamics, i.e. the environment model, since they are already provided. This makes the implementation of planning algorithms such as MCTS more straightforward, relieving the burden of learning the environment model from the agent. Nevertheless, simulation may not be available in all environments, so we study environment model learning in our work. Additionally, even for FightingICE, simulation is computationally expensive enough that the time between time steps (which, reminding, is 1/60th of a second) is too small to allow extensive simulation [34], [35]. The ability to learn environment models could allow for the use of more abstract representations of the dynamics that encode information that is not only useful for the agent but is also faster to compute. Consider, for instance, temporally abstracted simulations that allow leaps of frames to be performed at once. Still, learning such a model is an open research question [16].

Figure 3.2 shows the evolution of methods used among the competitors for the DareFightingICE competition over the years. We can broadly classify approaches as being in one of three main categories: rule-based, pure planning-based and RL-based [36]. Note that each competitor may combine multiple methods, e.g. planning may be combined with RL which is often done. Methods that are classified into "other" are those that neither used RL, planning nor manual rules, such as evolutionary algorithms. In earlier versions of the competition, mainly rule- and heuristic-based agents were developed. Since 2016, planning using MCTS has been widely adopted among competitors, matching the year AlphaGo achieved a milestone in RL by combining RL with neural networks and MCTS-based planning [37]. In recent years, there has been more interest in using supervised learning (SL) and RL for agent development, as the performance of rule-based and pure planning agents tends to fall behind that of automated learning approaches [10]. Since 2022, the competition has switched to audio-based input features.

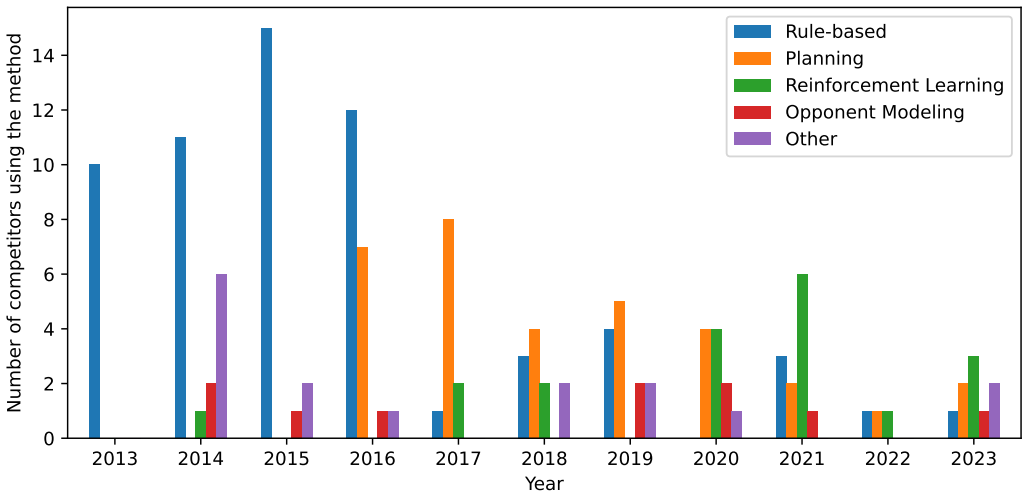


Figure 3.2: Method adoption of DareFightingICE competitors over the years.

Various works use FightingICE as a testbed for experimentation. The platform also provides a baseline MctsAi that purely plans with MCTS and two other baselines based on RL: one using a Deep Q-Network (DQN), introduced in 2018, and another using Proximal Policy Optimization (PPO), introduced in 2022. These AIs, mainly the one based on pure MCTS, are frequently used for comparatively evaluating performance both in the competitions and the literature.

3.2.2 DIAMBRA Arena

DIAMBRA is an ecosystem mainly comprising two platforms for development of and competition among RL-based AI respectively. DIAMBRA Arena is the development platform, allowing development of agents in competitive video games for research, competition and experimentation [38]. It provides environments for various commercial games with the current selection featuring fighting games only, most of them traditional. The platform also already offers training features such as self-play, in which the agent trains against past versions of itself, imitation learning and human-in-the-loop training.

3.3 Performance

Most works in the development of artificial agents for fighting games tend to focus purely on performance rather than creating agents for play against human players. For two-player zero-sum games, performance is hard to quantify in absolute terms. The evaluation metrics that are used instead are comparative, one example being Elo [39]. These metrics are calculated by having developed agents match against other players. For fighting games, the metric most used is win rate, and the artificial opponents used for testing tend to be the ones already provided in those games, which for commercial games are closed-source. For FightingICE the opponents used for comparison tend to be the submissions to past FightingICE competitions. For the case of FightingICE however using a large set of opponents was shown to be required, and ranking within the competitions is not directly indicative of the competitors' performance [34], [40], [41]. This suggests that there is still much non-transitivity in the strategy space covered by current agents, and as such using many test opponents is required to get a better grasp of agent strength. There is no standard benchmark for evaluating fighting game agents. Ideally, a tournament-based benchmark would be used, with a diverse set of opponents ensuring transitive strength of the agent, such as the one in [42] for the game of rock-paper-scissors which includes forty-three opponents.

Sections 3.3.1–3.3.3 will present the recent work in this area for fighting games in general, FightingICE and DIAMBRA, respectively.

3.3.1 Fighting games in general

ML methods have been applied to a diverse range of commercial fighting games, which makes it difficult to compare each method in terms of performance. This is exacerbated by the fact that agents tend to be trained by playing against the in-game AI, which has no

guarantees of having been implemented the same way across games. As such, each method needs to be evaluated in isolation.

The work in [43] implemented a convolutional neural network (CNN) for the fighting game "King of Fighters '97", with the goal of creating an agent capable of performing combos. This was achieved by representing the agent's action history as a binary image, and using that as input to the CNN. CNNs are frequently used for finding patterns in image-based input, and so the idea is to make use of this architecture to detect patterns in the agent's input sequence and leverage that information to know which action to perform next in a combo. The agent was able to beat in-game AI at the highest difficulty setting, but was only able to perform small combos.

Another work applied a CNN as well for the fighting game *Street Fighter V*, utilizing a matrix representation for the output to tackle the action complexity of these games and allow combinations of actions to be performed [44]. However, the matrix representation incurs some redundancy, as some row-column pairs are repeated in their role. Even though good performance was achieved against the in-game AI, only the AI of average skill level was used for evaluations, and more should have been used to better outline the agent's performance.

In [2], an ML algorithm was trained on *Super Smash Bros. Melee*, a fighting game inserted into the subgenre of platform fighters. Platform fighters share similarities with traditional fighting games, but contain largely different goals and mechanics [45]. The authors also focused their attention on the learning problem rather than perception, and so used structured representations of the environment as observations. Two model-free RL algorithms were tested with function approximation: Q-learning and a variation of actor-critic. The algorithms were successful, with positive average reward within one day of training on a cluster of training agents. Although the two algorithms achieved similar average reward values, they employed vastly different strategies, with the actor-critic algorithm behaving most similarly to a human player and the Q-learning algorithm merely attempting to exploit a single flaw of the training opponent. Evaluation with human players indicated that these agents did not generalize well, so self-play was employed for the actor-critic one. With this, performance rivaling that of experienced players was achieved, indicating that self-play is a crucial technique in achieving generalization. However, it was still possible for the agents to be exploited and perform odd behavior under certain conditions.

At present, DL-based approaches are not the most appropriate for a general fighting game agent [46]. The diversity of opponent strategies and respective counter-strategies means that, for a fighting game agent to succeed, quick adaptation is required for generality. However, RL and deep reinforcement learning (DRL) in particular are sample inefficient, i.e. they take plenty of environment interactions to learn [47], [48]. As such, these algorithms fall short of accomplishing this goal. One possible way to alleviate this sample inefficiency is to adopt off-policy RL algorithms, which allow for learning on experience other than the agent's, and so we can recycle old data for learning.

3.3.2 FightingICE

Various approaches based on structured representations of the environment have been studied. One work used Q-learning with deep neural networks (DNNs) to develop a pool of agents with diverse behavior [49]. Based on the discriminatory procedure of Diversity Is All You Need (DIAYN) [50], the generated behaviors were not only more diverse than baselines developed through reward shaping but also exhibited better performance. However, these behaviors were not developed to be evolved, but set in an initial training phase and then slowly modified in a posterior fine-tuning stage to avoid radically changing the set behavior, which may hinder adaptiveness. Additionally, the 15-frame delay of the state was removed, circumventing the reaction time constraint. Still, this work highlights the need for developing a pool of diverse agents for implementation of AI in fighting games, which will be further discussed in Subsection 3.4.2.

Other works utilized SL to develop an agent or a component of one. The work in [51] used imitation learning to have an agent learn to imitate recorded play data of human players. The method unfortunately was highly sensitive to training data, since for each observation the agent can only learn to perform a single action. This limits the stochasticity of the agent and will make them wholly predictable, which is not desirable in our case. Some other works used SL to create action predictors based on structured representations of the environment [52], [53]. These works suggest the applicability of SL techniques for opponent modeling.

Training agents with visual and audio-based environment representations has also been attempted [46]. In particular, RL has been used with audio-based features, specifically for testing a new sound design for the FightingICE platform [54]. This work used, besides win rate, the average difference in health between players at the end of each round as a performance metric. These metrics were evaluated in matches against a weakened version of the MctsAi, which performed planning for a shorter amount of time [33]. In this work, performance was used as a comparative measure to evaluate the new sound design against the old one, which seemed to be slightly more effective if considering one of the audio encoders in particular.

Some works experimented with Deep Q-learning and Sarsa(λ) algorithms, but were not able to outperform the baseline MctsAi [55], [56].

In [41], a new evaluation metric for fighting game agents is proposed, that takes win rate, health difference and elapsed round time as factors. The authors developed an RL agent using PPO with a self-play phase incorporating both the baseline MctsAi and versions of the agent throughout training. This work provides relevant insight in performance evaluation. The developed agent was able to surpass the top three agents of the 2017 competition, but had difficulty facing the 4th ranked one. This is evidence of the non-transitivity of strategies, and so evaluation by matching against other artificial agents should be done with a varied pool of test opponents, and not only those that seemed to perform the best in competition.

Hierarchical reinforcement learning (HRL) has also been mixed with MCTS such that MCTS plans for each high-level option of the hierarchy [40]. Each option specifies a particular set of actions that can be performed, and when an option is chosen by the high-level policy MCTS plans using only the actions specified by that option. This approach brought mixed

results against previous FightingICE competitors when the policy was frozen, since training was only performed against one of them which hampered generalization. Results improved substantially when the policy was allowed to be updated during evaluation. This again highlights the need to consider a large breadth of strategies when training, in order to have a general agent. Additionally, adaptation to the specific opponent is important for good performance, and while the authors achieved better results after adaptation it took 1000 rounds to do so for each opponent. The hierarchy was also built by a human expert, rather than learned.

An integration of self-play with the DQN algorithm was also explored [57], with the goal of creating agents that can generalize to unseen opponents and avoid specializing only to opponents seen during training. The intuition is that using various versions of the agent throughout training as opponents will force it to keep a general behavior that is effective against all past versions of itself. The proposed method achieved slightly better performance against past competitors when the versions to be used as opponents were randomly picked from the past rather than just choosing the latest version. This makes sense since it is expected that the latest version will be similar to the current agent version, which does not provide the agent with diverse behaviors needed for generalization. This corresponds to the difference between fictitious self-play (FSP), in which we sample uniformly from the past, and original self-play, in which we prioritize the latest agent versions [58], respectively. However, self-play still does not guarantee that past versions of the agent are a good source of diverse opponents. Evolutionary algorithms might be a better source of opponents since it is easier for each agent to have large differences in behavior due to independent training, which can lead them to converge to different local minima. Subsection 3.4.2 delves into the issue of creating diverse opponents, which can aid in self-play.

A pure MCTS-based agent was also created, similar to MctsAi, but using an action table for opponent modeling [34]. Since fighting games, including FightingICE, only allow 1/60th of a second in decision-making time, MCTS simulations must be performed within this time frame. In practice, MCTS-based approaches are only able to perform a small number of simulations per expansion, and may require additional mechanisms to make the search space smaller [59]–[62]. The authors performed opponent modeling to optimize the simulation step of MCTS, by determining which actions the opponent and the agent itself are likely to perform in simplified versions of the environment state. This directs the MCTS simulation to game states that are more likely to happen. Results show improvement over bare MCTS, but it still proved insufficient for one particular test opponent. This again stresses the importance of evaluating agents against a large and varied pool of opponents. Nevertheless, this work highlights the usefulness of incorporating opponent modeling into planning methods based on simulation.

3.3.3 DIAMBRA Arena

Due to the platform's recency, there is only one work of note that used this platform [63]. Its main objective was producing a performant agent for the *Street Fighter III* environment,

evaluating different RL algorithms in the process and outlining their advantages and drawbacks. Despite the results being presented non-homogeneously and the lack of conclusions drawn from comparisons between algorithms, we can note that PPO seemed more efficient and performant than Advantage Actor Critic (A2C), but Asynchronous Advantage Actor Critic (A3C) was still able to have greater reward on the long-term. This work was hindered by the heavy computational requirements for training an agent. To the best of our ability, we try to create a simple agent that should not require extensive computer resources. Additionally, the agent lacked proper adaptation to different opponents, which was left for future work.

3.4 Agent qualities

Some works focus on creating autonomous agents with one or more particular qualities rather than pure performance. For competitive and cooperative video games, we can have artificial agents fill the role of human players. This is useful when either human players are not available or, if they are, they are not compatible for fair or engaging play, due to being of a disparate skill level for instance. Research has been made on endowing these agents with aspects that enhance the human players' game experience, which are laid out in the following subsections.

3.4.1 Dynamic Difficulty Adjustment

In video games, providing a fair and engaging challenge to players can be done by automatically adjusting game parameters so that adequate difficulty is presented. This automatic process is called dynamic difficulty adjustment (DDA), and it is meant to enhance the end users' experience [64]. Usually, estimations of the player's performance are made and parameters are adjusted according to perceived skill. As such, DDA is also concerned with downgrading the performance of the artificial opponent so that they better match the skill of the human player. For our purposes, even though providing opponents adjusted to the current human player is a concern taken into consideration, this performance downgrading aspect is not explored for simplification. Instead, we can train a set of opponents at varying skill levels, where skill is proportional to the training time used for the agent. Still, state of the art in DDA can provide insight into the perception of difficulty in human-agent interaction, which could even be employed in our simplified case by detecting whether agents used at a certain skill level are not appropriate for the current human player.

The authors in [55] used RL to develop an artificial opponent with DDA in FightingICE. DDA was performed by saving versions of the agent throughout training at certain time intervals, emulating different skill levels. Another possible approach is to use a performance metric rather than fixed time intervals, which was left for future work. With these different opponent versions, DDA is performed by evaluating the difference between both player's health points throughout the games and switching to different versions accordingly. However, there were some limitations to the approach, such as the inability to surpass FightingICE's baseline MctsAi and the fact that the easiest skill level features an agent with no training, which is a random agent. In the context of fighting games, this is hardly engaging to play

against since human players cannot predict random play, making these games lose their appeal. This needs to be a major consideration in future work, since the main objective of DDA is to provide engaging experiences to the end user. One possible approach for creating easier difficulties in these games is to limit the number of options available to the artificial agent which limits the complexity of their behavior.

Instead of considering the player's performance for DDA, the work in [65] uses the player's "affective states" to provide an appropriate experience. This is relevant since some players are looking for a strict challenge while others are seeking a more relaxing experience, and the proposed method aims to take this into account. SL was applied to FightingICE to determine a player's perceived emotional state and integrate it in an MCTS-based opponent, but survey results indicated differing opinions between participants. Although interesting in terms of providing satisfying experiences to players, in our work we merely focus on providing a competent agent, regardless of the player's emotional state.

3.4.2 Diverse playstyles

Playstyle can be defined as the way someone or something plays a game, characterized by the strategies that they employ. For games, especially complex ones, a particular skill level is not defined by a single strategy. Figure 3.3 illustrates the geometry of the strategy space in real world games, proposed in [27]. The strategy space of real world games tends to resemble a spinning top, in the sense that the vertical axis encodes the strategy's transitive strength and the width at a particular point in the vertical axis represents the non-transitive dimension, i.e. the breadth of distinct strategies of the respective strength. Transitivity stands for the property in which, for three distinct strategies A, B and C, such that A beats B and B beats C, we have that A beats C. Without transitivity, we might form a cycle, in which C beats A. At the topmost tip of the spinning top we have the Nash equilibria, and the bottom tip contains strategies that purposefully attempt to lose. As such, it is natural that each player has their own playstyle, defining the way they approach the game, with this being especially true on lower levels of skill. We use this geometry further to illustrate our points.

It is worth noting that the geometry of Fig. 3.3 was proposed considering *symmetric* games, i.e. games in which any two players have access to the same conditions. For example, turn-based games are asymmetric because one player has to start first, and the other one can only act afterward. Fighting games are also asymmetric unless both players are utilizing the same character. The authors include asymmetric games in their analysis by considering each player to play once in the other player's conditions, which in the case of fighting games means two matches are performed where each player plays once with their character and then again with their opponent's character.

This is an important formalism especially when considering population-based training methods that are often used to achieve high performance in competitive environments by artificial agents. In these competitive environments, in which the artificial agent is playing against other adversaries, we need to supply a good opponent pool such that we improve the agent by increasing their transitive strength. However, we may find that when using an

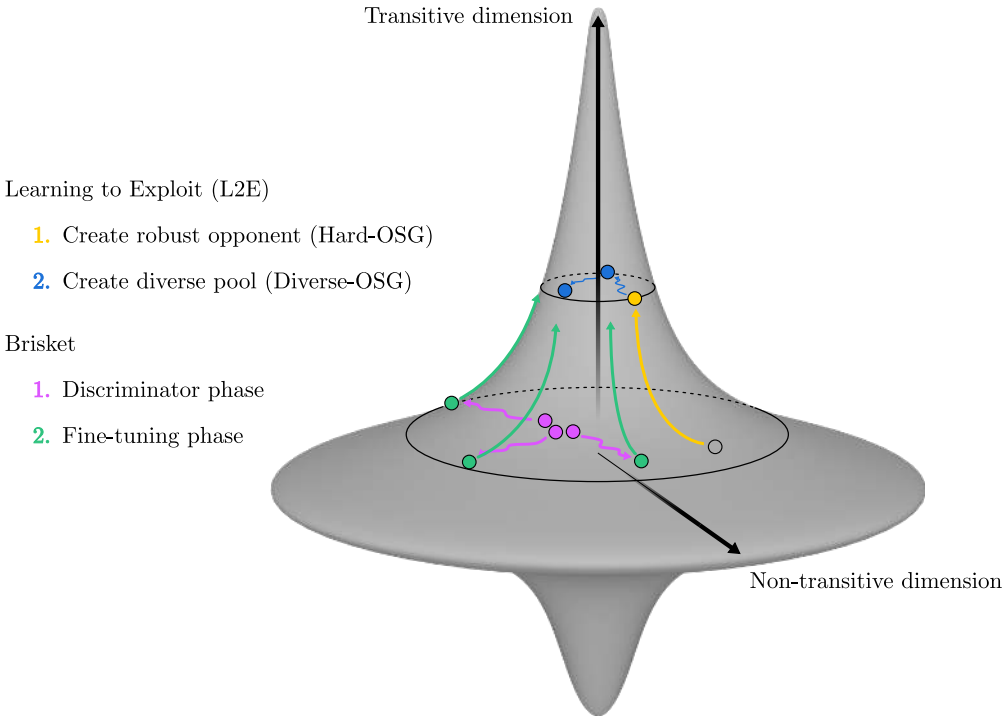


Figure 3.3: Strategy space geometry of real world games proposed in [27]. The agent generation strategies in Brisket [49] and L2E [66] are illustrated using this geometry.

opponent pool populated by naive self-play, for instance, the agent is just navigating along the non-transitive dimension, learning policies that merely form strategy cycles during training and not improving overall. Different self-play strategies have been adopted to attempt to avoid this, such as populating the opponent pool with "exploiter" agents that are focused on exploiting the main agent's weaknesses rather than trying to be robust against many opponents [58]. Essentially, we want to avoid traversing the non-transitive dimension and traverse the transitive one as much as possible. One theoretical result is that, if the population size is representative enough of the number of strategies of a given transitive strength, then we can guarantee improvement in terms of transitive strength. Moreover, we can note that non-transitive complexity lowers as the agent gets stronger, assuming the agent is beyond the largest disc of the spinning top. As such, there is interest in properly initializing the opponent pool so that the agent gets to a higher transitive strength as quickly as possible, e.g. by initializing the population with highly skilled opponents. In the DareFightingICE competition, for instance, since 2020 it has been frequent for competitors to use submissions from past years as training opponents in self-play [41].

We can additionally induce that, to create engaging play in fighting games, it is not enough to just create a single opponent. There is interest in having a population of agents, each exhibiting diverse behaviors that better represent the game's strategy space. This way, human players have to figure out and better adapt to various playstyles of the artificial opponents, at any skill level. Additionally, given the spinning top structure, some skill levels exhibit a greater set of strategies, so the population size can vary among skill levels.

Not many works in the literature explore this area. Of note, [49] developed Brisket, a

method for creating a population of agents with diverse behaviors for FightingICE. Training is separated into two phases, where in the first phase agents merely learn to exhibit diverse behaviors between each other, whether they be good or bad for performance at the game. Then, with behavioral bias set from the previous phase, the second phase focuses on learning how to play the game, using Deep Q-learning with reduced learning rate to avoid radical changes in behavior. These two phases can be thought of as the first one focusing on traversing the game's spinning top space in the non-transitive dimension, while the second phase focuses on traversing the transitive one. This behavior diversity is achieved using an adaptation of the DIAYN method [50], which is a method for allowing an agent to learn a fixed set of skills that allow it to traverse distinct portions of the state space. When applied to robotics, for instance, we can have an agent autonomously learn to walk or to perform frontflips, without being instructed to do so. In this work, the same principles are used, but instead of having diversity between skills in a single agent, we have diversity between different agents in an agent pool. As such, whereas DIAYN is focused on within-agent diversity, this work is focused on between-agent diversity. The developed agents were diverse, contrasting with the baselines which converged to the same local maxima. At the same time, all agents exhibited the same performance, which was also superior to the baselines' performance.

Another work worth mentioning, although not applied to fighting games, is the Learning to Exploit (L2E) framework [66]. In this work, the authors propose an opponent strategy generation (OSG) algorithm for creating a diverse pool of training opponents, with the aim of increasing the agent's robustness when trained against it, i.e. make them less exploitable. Similarly to [49], the algorithm is separated into two phases but with inverted goals: in the first phase (Hard-OSG), a single randomly-initialized opponent is optimized to be robust against the current agent; in the next phase (Diverse-OSG), a diverse opponent pool is iteratively created by creating opponents similarly to the first phase, but with the additional constraint of being dissimilar to all opponents in the pool so far. Diversity is enforced by ensuring opponents have behavior that produces distinct experiences/trajectories during gameplay. Through ablation tests, the authors found that utilizing both optimization phases (optimizing for robustness and diversity) helps increase the agent's expected return, with diversity being a large contributing factor. This work highlights the need to have not only a robust but also a diverse set of opponents during training, especially important when employing self-play.

3.4.3 Adaptability

A major problem in creating efficient artificial opponents for human players in fighting games is ensuring a fast enough adaptation to the player's behavior. That is, the artificial agent should change its strategy in response to the opponent's perceived strategy, such that it would be an appropriate response in the sense that its expected return would be maximized. As explained in Chapter 2, optimizing play against human players requires adapting to their playstyle, and not only playing randomly. Adaptation to the player has to be done quickly enough, and not require an extensive number of matches.

The work in [48], not applied to a fighting game, incorporated opponent modeling in DRL

in an attempt to improve the adaptation efficiency of an agent to a non-stationary environment, on which DRL methods are known to be slow due to their sample inefficiency. They proposed using a Bayesian neural network to model the behavior of k different opponents, and evaluating the uncertainty of the network using Monte Carlo dropout. When uncertainty is deemed too high, the network in its entirety, as well as the policy, are switched to more appropriate models considering the current opponent. Results show that, while the baseline DRL algorithm gets to a good performance faster at the beginning of training, in the long-term it can only produce behavior that is the best for all opponents on average, whereas the proposed method can adapt better to each opponent individually after various switches. This work highlights how radical changes to the learned model are needed for adaptation, but such is done by training completely independent models that are switched by an external component, that tracks the uncertainty of the currently performing model. Ideally, aspects learned for one opponent should be transferrable to other opponents.

In [67], Q-learning was used with artificial neural networks (ANNs) on a non-traditional fighting game made by the authors. The goal of the work was to create an adaptive agent, and although adaptation was successful to fixed artificial opponents, the adaptation efficiency was not evaluated.

The authors in [68] used hierarchical planning against previous FightingICE competitors to create an adaptive agent, contrasting with reactive agents that were prevalent in the FightingICE competition. Although good performance was achieved against the top three competitors of 2016, evaluation should have been done with more competitors to properly cover the set of possible strategies. Additionally, the hierarchy needs to be built by a human expert, limiting automation and easy application to different games.

Heuristic search was also merged with MCTS for FightingICE [35]. According to the authors, MCTS and evolutionary algorithms are useful in cases where the environment changes slightly since they are less biased to the training data [35]. Also, MCTS applied to FightingICE is not efficient enough due to the demanding performance requirements, not allowing extensive simulation. As such, a heuristic search method that takes priority over MCTS is tested. This method differs from our desired approach since there is heavy human intervention in determining an appropriate set of heuristics for the search. Additionally, no tests over efficient adaptability were made.

An approach for adaptation is to use an ensemble of static agents rather than just one static agent [10], [69]. During play, only one of the agents in the ensemble is used, and a performance metric is tracked. After a threshold, the current agent is switched to another one in the ensemble. The idea is that each agent encompasses a specific strategy or playstyle, and switching according to the current performance allows the overall agent to adapt to the opponent. However, the performance of these methods falls behind those that employ more sophisticated methods such as evolutionary algorithms or ML [10]. Additionally, tests with human players indicated that care should be taken in the switching mechanism since the radically different playstyle switch can cause disorientation [69].

Despite not having been applied to fighting games, the L2E framework [66] introduces

a gradient-based approach to producing agents that adapt quickly to new opponents. The authors define an objective function $J(\theta)$ for the agent policy that is equal to the sum of the discounted expected return against each opponent of an opponent pool *after* some small number n of gradient ascent updates against it, and the policy is updated to maximize $J(\theta)$. This encourages the agent's policy to have some set of parameters θ such that, after learning to play against a new opponent for just a small number of gradient updates, it nevertheless has increased return. For the two games of poker and a grid soccer game, often used to test opponent modeling, the proposed framework proved to be much more apt to adapt in very few iterations than other baselines including the RL algorithm Trust Region Policy Optimization (TRPO) [70] and the meta-learning algorithm Model-Agnostic Meta-Learning (MAML) [71].

3.4.4 Human likeness

Artificial agents developed for these games tend to behave in a way that human players do not, giving away their artificiality. This is due to the complex decision making and control that human players have, which may even be influenced by emotions, and therefore is hard for agents to emulate. As a result, the players' experience is often diminished due to the lack of believability in the opponent's behavior [72]. This is an imperative concern for fighting games since human players assume that the opponent has exploitable biases in their playstyle that can be used to their advantage. If it is clear that the opponent is artificial, human players will not think to play in this pattern-seeking way [5], diminishing engagement.

We can consider two approaches to this problem: either the artificial agents are constrained in ways similar to human players which should result in exhibiting similar behavior, or the agents directly attempt to imitate human behavior. For constraint-based approaches, we can denote the FightingICE platform, which includes a static delay to the state provided to the agents. Other than that, current work, including commercial games, tends to focus on the second approach of imitating human behavior.

Once we have an approach, how do we evaluate "human likeness"? One test seen in some works is the Turing test, alternatively called the "imitation game", in which human evaluators attempt to determine whether a given entity is human or artificial merely by its actions on a particular task [73]. For instance, in one-on-one competitive games, video footage of the artificial agent playing against a human or another artificial opponent is shown to human evaluators. The evaluators have to specify whether each, or one of, the players is human or not. The usefulness of this test is currently a point of controversy [74], but for our purposes the Turing test seems valid. Since the main objective is to develop an artificial agent behaving similarly to a human, the ability of an agent to pass the Turing test directly represents this goal. This is in contrast to the broad original objective of Turing tests, which is to evaluate whether an artificial agent can think intelligently.

The authors in [75] built upon [2], in which an agent was implemented to play *Super Smash Bros. Melee* using RL, by incorporating a static reaction time constraint for fairness. The authors suggest that adding human constraints may allow the agent to feature human-

like behavior. However, human likeness was not evaluated, but task and computational performance instead. The authors delay the agent's actions by a fixed number of frames, which should be at least 15 frames, matching the average reaction time of human beings. Usually, to deal with this delay we augment the state that the agent perceives by including which actions the agent performed 15 frames earlier; this way, we maintain the Markov property. However, this method tends to not work well in practice due to the large increase in the state space size, which is augmented by a factor of $|\mathcal{A}|^d$ with $\mathcal{A}$ being the agent's action space and d being the delay. The authors found that task performance is inversely proportional to the amount of delay, both in *Atari* games and *Super Smash Bros. Melee*. Therefore, the authors opted for correcting the action delay by directly predicting the state in which that action will end up being executed. That is, the agent should predict the state 15 frames into the future and act according to it. The prediction is done through an environment model built using residual-like recurrent neural networks (RNNs). After experiments with predicting the future by different amounts, i.e. predicting only a fraction of the delay, the authors found that predicting the full delay is necessary to achieve good performance. Still, there were performance issues, and the agent was not able to play in real-time with a reaction time delay of over eight frames.

Based on findings from psychology, the work in [76] incorporates the emotional appraisal engine Gamygdala into the built-in Fuzzy AI in the Universal Fighting Engine. The authors propose integrating emotions into parameter adjustments of the Fuzzy AI, but their work has not been tested yet. Also using the Fuzzy AI in the Universal Fighting Engine, the authors in [77] propose a method for evaluating the playstyles of human and artificial players. Their method is based on which triplets of actions are likely to be performed by each player. The probabilities of each triplet being performed are laid out in a playstyle vector, which can be compared between players using cosine similarity to get a value representing how similar the players' playstyles are. However, this notion of playstyle may not be in line with how humans perceive a playstyle. Additionally, the number of triplets is cubic in terms of the number of actions available. As such, the playstyle vectors will be high-dimensional, which should require a large amount of sampling to correctly approximate the triplet probabilities. Therefore, for playstyle evaluation Turing tests were also performed, in the manner specified earlier. The authors obtained mixed results, with evaluators overall providing incorrect answers or being unsure of what to answer. Not only do people have different perceptions of what constitutes a "playstyle", but the lack of interactivity in the tests may also hinder the evaluators' ability to correctly determine the behavior patterns of any player. In the future, tests in which evaluators are active players in the game rather than a spectator for two other players might be attempted.

A set of heuristics has also been incorporated into FightingICE to create an agent that learns to exploit opportunities and perform combos [78]. After Turing tests, the proposed method presented some general hesitation in human evaluators to classify the agent as human, but was still considered the most human-like of other artificial baselines. However, the method is rule-based, and no automated learning is being performed, which limits applicability to

different games that have slightly different mechanics.

In [79], the authors improve upon a previous work of theirs, which utilizes MCTS to create an agent for DDA, by removing seemingly artificial behaviors that would diminish the players' experience, also in FightingICE. Their previous agent, for the purposes of adjusting difficulty, was often performing uncompetitive behavior, rather than actively engaging in the game. To mitigate these behaviors, the authors define believability as being active and aggressive as opposed to being passive, and aim to achieve higher believability in those terms. Of course, this excludes behaviors that might be technically passive but which human players still perform, such as taunting. Nevertheless, the authors aim for an objective aligned with ours, of merely making a competent agent. To achieve this, the authors incorporate a different evaluation function for MCTS, which is a linear interpolation between their previous DDA evaluation function and an evaluation function for believability. This evaluation function is essentially the difference in the opponent's health, encouraging the agent to damage the opponent. The coefficient is dynamically adjusted during the game to ensure high believability/aggressiveness when the agent has lower health than the opponent, and better difficulty adjustment otherwise. After evaluating the level of challenge, enjoyment and human likeness with human players, the authors note that although the improvement was able to increase challenge and human likeness slightly, the level of enjoyment was lower than expected against expert human players. Utilizing human play data for imitation of human behaviors with MCTS was left for future work.

The work in [80] evaluates both RL and SL for the creation of human-like agents in a custom fighting game. For the RL algorithm, tabular Q-learning, i.e. without function approximation, was used with three different reward functions: reward by victory, by whether the agent hits the opponent or gets hit by them, and by following a specific strategy imposed by the authors. SL, on the other hand, was used with function approximation through ANNs, trained with imitation learning. After training the agents, a questionnaire was used to evaluate which of them was the most challenging, enjoyable and human-like, similarly to [79]. The authors found that imitation learning was more capable of producing not only human-like behavior but also engaging experiences when compared with RL. Among the different reward schemes, imposing a behavior that abides by the authors' defined strategy seemed the most human-like. However, RL was applied with Q-learning, which is a greedy algorithm from which the policy has to be indirectly developed, e.g. for manually introducing stochasticity. Additionally, other than the strategy-based reward scheme, no constraints such as reaction time were imposed on the agent, having less incentive to act like a human. Yet, despite the dependence on training data with SL, it still proves to be a useful paradigm for incorporating human-like behavior.

A mixture of SL and RL has also been tested. The idea behind incorporating both learning paradigms is that SL can be effectively used to imitate behaviors according to training data, while RL can be used to learn how to behave optimally in an environment. The authors in [81] experimented with three different approaches to mixing SL and RL, using DNNs. The authors found that one of the the proposed architectures, which uses separate networks for

each learning paradigm, did create more believable agents than the baseline DQN agent, but achieved poorer performance. Additionally, the calculated human likeness is far below that of a human player. Although the application of SL brought improvements in terms of human likeness, the results are still far from desired. Also, the most human-like architecture seems to not be apt for an adaptive agent. SL was applied in a first stage before RL was used in the final second stage. This second stage had to be truncated in practice, since the RL algorithm would gradually transition the agent into optimal but artificial behavior, losing the human-like bias induced in the first stage, a problem similarly faced in [49].

HRL was also used to create opponents exhibiting human-like, believable behavior for fighting games, more specifically *Street Fighter IV* [82], using a learnable model of the environment. Turing tests indicated that the developed agents exhibited behavior similar to human players. However, human reaction time was not taken into consideration for fairness, and the hierarchical decomposition of the tasks has to be done by a human expert rather than learned. The method also requires some human bias to be introduced at the beginning of the training process, so that the behavior of the agent during training does not deviate from expected. Additionally, even though the method learns an environment model that could give the agent adaptive capabilities, since we can consider the opponent to be part of the environment, this was not evaluated.

The work in [1] highlights some of the problems possibly faced when implementing human-like AI for fighting games. One is that the mere requirement of adaptability for agents does not guarantee that the exhibited behaviors are human-like. Additionally, there is the problem of unpredictability. Being unpredictable and exploring the environment is important for these games, but the agent should do so without being unreasonable. That is, there are specific states at specific times in which we expect a human player to attempt experimentation or to vary their approach, and an artificial agent should not be blind when exploring but be guided in some sense. With this in mind, a planning-based approach was proposed that learns a model of the environment and a strategy from demonstrations of human play, akin to [82]. After implementation in a purposefully made game, the agent replicated human play better than other artificial baselines, but its believability was still low. Also, the environment model architecture is not scalable, and environment transitions should be made more compact for more complex environments.

3.5 Transfer learning

Transfer learning denotes the process of a machine learning model, trained for a given task, to be trained for a different task such that the bias brought by the first task improves learning and performance on the second task. This is desirable when the two tasks are only slightly different and makes training much more efficient. In fighting games, we can think of transfer learning as the ability for an agent, trained on playing as a specific character, to be able to play as another character, but without requiring the onerous training as was needed for the first character since some skills should be transferable. Or, in a similar manner, an

agent trained to play *against* a character or opponent being able to efficiently learn how to play against another one.

In [83], PPO was applied to FightingICE, based on an ANN with representation transfer. The first part of this network is focused on building a latent representation of the input vector, which is the environment state. With this representation, the network branches out in two predictors: the value of the input state and the action to execute on that state, which are approximators of the value function and the policy respectively. A faster adaptation to a slightly altered environment was perceived, but it ended up exhibiting slightly worse performance than a pre-trained model trained directly in the altered environment. This might be due to the pre-trained model biasing the agent into a local optimum that ended up being worse than the optimum found without any preset bias.

In [43], an agent was developed to play against a single character in "King of Fighters '97" and then, to test transfer learning capabilities, trained to play against three other characters that the game offered. Training time was reduced to about a half, indicating the transferability of skills.

The authors in [2] developed various agents for *Super Smash Bros. Melee*, each learning one of six distinct characters. After training, each agent learned to play as the five other characters. The agents successfully learned to play the new character in much less time than would be required to learn from scratch. Another interesting result was that the transfer learning efficiency of each character largely corroborated the human players' perception of learning difficulty for those characters.

A survey on the current state of HRL and the different systems built in past research that mimic components of biological learning agents has also been conducted [15]. One of the promises of HRL methods is transfer learning, since specific problems are abstracted in a hierarchical fashion, generalizing the solution to similar problems. The authors report that although different HRL approaches encompass different aspects of biological beings, there is no unified architecture that merges these ideas stably and efficiently. A guide for future research is laid out, so that transfer learning and decision-time planning can be leveraged for solving problems without requiring extensive interaction with the environment, which is a known detriment to RL methods and DRL in specific.

3.6 Conclusions

In terms of the performance of artificial agents in fighting games, it is hard to make meaningful comparisons between approaches given the lack of standard benchmarks and the calculation of performance metrics on an insufficiently representative number of opponents. Exhibition of human-like behavior is even harder to compare due to the subjective nature of the evaluation, but the success of hierarchical reinforcement learning (HRL) and policy gradient methods is of note nevertheless. Still, the proposed approaches for human likeness do not take into account the fairness of the agent, such as emulation of reaction time. The influence of the reaction time constraint on human-like behavior is yet to be explored.

Deep reinforcement learning (DRL) methods for fighting games are shown to be successful, even able to beat expert human opponents, but can still show odd behavior sometimes. The policies learned by these methods are also highly uninterpretable, leading to unpredictability in how the agent will behave. Current applications of HRL methods to fighting games are successful but utilize expert knowledge to build the hierarchy. Future work can experiment with automated approaches to building these hierarchies, which already exist [15]. Unlike DRL, HRL methods provide greater interpretability of their policies.

There are also some key insights for the development of engaging artificial agents. For one, implementing various difficulty levels through versions of the agent at different stages of training should be handled with care, so that the under-trained agents are not close to unreasonable behavior. Additionally, exploration of the environment by the agent should be guided rather than completely random. One example is to use an intrinsic "curiosity" reward for exploring environment states on which the agent is unsure of what to expect according to a learned environment model [15], [84], [85]. Another important takeaway is to generally avoid using ensembles of agents as a single agent since they can incur unnatural behavior shifts which can disorient the human player. Ideally, these ensembles exist at a higher level, where the agent pool is used as an opponent pool offering agents with varied strategies for each match, rather than being a single opponent that can be changed on a whim in the middle of a match.

All works that attempted to perform transfer learning corroborate its success in the context of fighting games, both when considering transferability to playing as and against different characters.

In terms of issues, we note that there still has not been extensive work on the efficiency of adaptation in non-stationary environments, more specifically to varying opponents in fighting games. Works evaluate whether adaptation occurs at all and if it leads to good performance, but do not focus on its efficiency. Additionally, some of the challenges brought by fighting game environments still need to be explored further, with only a few works exploring action combinations and extended input sequences for performing combos and special moves.

CHAPTER 4

Methodology

Problem definition and solution architecture.

4.1 Problem

Before applying reinforcement learning (RL) in fighting games, we need to properly define the problem, i.e. frame them as a Markov decision process (MDP) $(\mathcal{S}, \mathcal{A}, P, R, O)$ as detailed in Subsection 2.1.3. Fighting games are episodic environments, i.e. they start and terminate at some states. We call an *episode* the agent's interaction with the environment from beginning to end. In fighting games, matches are usually played in rounds in a best-of-three format. For simplification, we assume each episode is a round rather than a match, and as such termination occurs after either a player loses all their health or the time limit has been reached. However, with this simplification we are depriving ourselves of between-round decision making that may be useful in some games, as one round can affect the next. This can still be somewhat circumvented by attributing additional reward to the agent depending on how well-off it terminated the current round. Therefore, within this MDP there are some states of S which are considered *terminal* and represent episode termination. These states have to be treated specially, as the agent cannot perform actions on them and so constructs such as the value or action-value functions V and Q are not evaluated for these states.

The following subsections 4.1.1–4.1.4 describe the state space $\mathcal{S}$ and observation function O, transition function P, action space $\mathcal{A}$ and reward function R of the MDP $(\mathcal{S}, \mathcal{A}, P, R, O)$, respectively.

4.1.1 States and observations

It is desirable to guarantee the Markov property, that is, to make sure that the next state of the MDP can be determined merely from its previous state, without requiring extra information such as other past states. Having the Markov property, we can ensure we provide the agent with all information necessary to reason about the future. In fighting games, most

variables required to determine the next state from the current one are perceivable by the players, but not all. Perhaps the most important of these unperceivable variables are each player's input buffers, which store the history of actions from each player. These buffers are used to determine whether a player has performed a special move. As explained in Subsection 4.1.3, we simplify the problem and consider special moves as part of the action space, making input buffers less important for determining future states. In fact, many fighting games allow performing special moves through action *combinations*, i.e. multiple actions at the same time, rather than through action sequences, so this is not a universal problem. As such, we can consider all variables that both players perceive visually to be the environment state, such as position, health points, etc. of each player. Hereafter, we assume the Markov property.

We introduce a varying reaction time constraint on the agent, so the observations will not match the environment state exactly. The observation space Ω will match the state space S, as no pertinent information is hidden to the players. However, the observations are delayed versions of the state, that are delayed by a number of time steps equal to the computed reaction time. The formal definition introduced in Subsection 2.1.3 of the function $O(b|s', o, a)$ for determining observations given the state is not convenient in our case, since b is equal to a past state s_k but the formulation does not allow that to be represented. This is because the history of past states will depend on factors such as the agent's behavior π which is not captured by O. Adhering to the definition requires appending the history of past states to the state as well as considering the agent policy π and opponent model ω, which should not be needed for the rest of the MDP. As such, we consider O to be a function $O(b|s_t, \dots, s_{t-k}, \pi, \omega)$, where k is the maximum reaction time. The way reaction time is computed and tackled to get approximations of the true states s is detailed in Subsection 4.2.3.

Because reaction time is a function of the agent's uncertainty of the opponent's behavior, the agent will internally have some opponent model ω. We decided to integrate this opponent model into other components of the agent, by explicitly considering the opponent action o along with the state s. Therefore, the transition function is defined as $P(s', o'|s, o, a)$, the reward function as $R(s', o'|s, o, a)$ the value function as $V(s, o)$, the action-value function as $Q(s, o, a)$ and the policy as $\pi(a|s, o)$, with $o \in \mathcal{A}_o$ and $\mathcal{A}_o$ being the opponent's action space. Explicitly considering the opponent's action like this is useful for further modeling and implementation in practice, especially since the opponent's action space is finite. This also has the potential to allow faster adaptation to new opponents, since the agent has the ability to act completely differently depending on what it predicts the opponent will do, and so different opponent models allow different behaviors.

4.1.2 Transition function

Save for rare exceptions, fighting games are completely deterministic, since from the current state we can exactly know the next state given only the agent's and opponent's next action. However, from the agent's perspective, the environment is not only composed of the game itself but also the opponent. As such, the transition function $P(s', o'|s, o, a)$ requires

knowing the opponent's policy $\omega(o'|s')$ which dictates the probability with which action o' will be performed in state s'. We can divide the transition function into two separate components: the opponent's policy $\omega(o|s)$, which is stochastic, and a function $G(s'|s, o, a)$ which outputs 1 if s' follows from (s, o, a) and 0 otherwise. G represents the deterministic environment dynamics since only one state s' can follow from (s, o, a). This facilitates learning an approximation of P since the output of the function is only a probability distribution over $|\mathcal{A}_o|$ states rather than a probability distribution over all possible next states. In fact, the transition function can be written as

$$P(s', o'|s, o, a) = \omega(o'|s')G(s'|s, o, a). \tag{4.1}$$

When we are training in an environment, we are always perceiving transitions (s, a, o, s') in which $G(s'|s, o, a) = 1$. Because of this, we can disregard G in practice, which is what we do hereafter, unless we are performing prediction of future states.

4.1.3 Actions

We simplify the players' action space by including special moves. Considering special moves to be actions alleviates the need to consider extended past sequences of primitive actions. However, as pointed out in Section 2.2, considering special moves a part of the action space limits transferability to learn other characters since the action spaces will be different. The advantage is that the problem is heavily simplified, at the cost of having a less general agent.

If we do not consider special moves to be actions and have the action space consist only of primitive actions, similar to how a human plays, then the agent needs to somehow consider the past sequence of actions performed by themselves in order to be able to perform special moves. We could include a queue of past actions into the state, which is inclusively available in DIAMBRA [38], but that greatly increases the state's dimensionality. If we apply hierarchical reinforcement learning (HRL) and consider a not to be primitive actions but to be options or goals, then we can perform special moves as high-level actions that persist for a small amount of time and internally execute the actions. We can consider the inclusion of special moves into the action space described previously to be a specific case of this, where special moves are options that when chosen execute a particular sequence of primitive actions until termination. Considering these trade-offs, we evaluate only the action space which treats special moves as actions to simplify the problem.

4.1.4 Reward

We experimented with two reward schemes: one based only on the game's outcome (win, loss or draw), and another that considers not only the game's outcome but also the difference in health between players in a state transition. Providing reward only at the end of the game based on the outcome results in a very sparse reward scheme, since during the game the agent gets no reward signal. This can make it harder for the agent to learn, as the credit assignment problem is exacerbated. To alleviate this, the second reward scheme is denser, providing

reward during the game depending on health difference. Since the goal in fighting games is to deplete the opponent's health before ours is, we are guiding the agent toward the goal.

These reward functions can be computed by only considering the state s_t and next state s_{t+1}, without the agent and opponent's actions a_t and o_t. As such, the reward function $R(s_{t+1}, o_{t+1}|s_t, o_t, a_t)$ can be simply evaluated as $R(s_{t+1}|s_t)$, and we will adopt this simplification throughout the rest of the document.

We denote the two previously detailed reward functions as R_s and R_d respectively. The sparse reward $R_s(s_t|s_{t-1})$ of an environment state s_t as time t is

$$R_s(s_t|s_{t-1}) = \begin{cases} 1, & \text{if agent won} \\ -1, & \text{if agent lost} \\ 0, & \text{otherwise} \end{cases}. \tag{4.2}$$

To clarify, in the case of a draw at time T the reward is $R_s(s_T|s_{T-1}) = 0$. The dense reward function R_d provides the same reward sum over each episode as R_s, but the reward signals are more distributed throughout the episodes according to the agent's progress toward the goal. Since the goal is to deplete the opponent's health before ours, we can consider depletion of health as progress toward or away from our goal depending on whose health is being lost. As such, the dense reward function $R_d(s_t|s_{t-1})$ of an environment state s_t at time t is

$$R_d(s_t|s_{t-1}) = \begin{cases} 1 - C(s_{t-1}), & \text{if agent won} \\ -1 - C(s_{t-1}), & \text{if agent lost} \\ -C(s_{t-1}), & \text{if a draw occurs} \\ R'_d(s_t|s_{t-1}), & \text{otherwise} \end{cases}, \tag{4.3}$$

where $C(s_t)$ is the cumulative sum of R'_d until state s_t at time t,

$$C(s_t) = \sum_{i=1}^{t} R'_d(s_i|s_{i-1}), \tag{4.4}$$

and $R'_d(s_t|s_{t-1})$ is the amount of reward obtained in state s_t according to health depletion of both players from s_{t-1} to s_t,

$$R'_d(s_t|s_{t-1}) = \frac{h_1(s_t) - h_1(s_{t-1})}{\hat{h}_1} - \frac{h_2(s_t) - h_2(s_{t-1})}{\hat{h}_2}, \tag{4.5}$$

where $h_j(s_t)$ is the health of player j at state s_t and $\hat{h}_j$ is the total health of player j, considering the agent to be player $j = 1$ and the opponent $j = 2$. Below is a proof that the cumulative reward $\sum_{t=1}^{T} R_d(s_t|s_{t-1})$ over an entire episode is equal to $\delta = 1 \vee \delta = -1 \vee \delta = 0$ on win, loss or draw respectively:

$$\begin{aligned}\sum_{t=0}^{T} R_d(s_t|s_{t-1}) &= R_d(s_T|s_{T-1}) + \sum_{t=1}^{T-1} R'_d(s_t|s_{t-1}) \\ &= \delta - C(s_{T-1}) + C(s_{T-1}) \\ &= \delta.\end{aligned}$$

Ensuring the sum is still 1, -1 or 0 on episode termination cements the idea that, regardless of what happens during the episodes, the outcome of the match is what matters. Still, the dense reward signal should be better in guiding the agent toward the goal during training than the sparse one, which should result in more efficient training.

Regarding the discount factor, which represents how much weight we give to future rewards, we initially set it to 1 since it is an episodic environment and does not require many environment steps to complete (6000 steps, or 100 seconds, at maximum). This enforces the idea that the result, winning or losing a match, is what matters most.

4.2 Solution

The objective is to develop an RL agent capable of online adaptation to the current opponent under a dynamic reaction time constraint. To accomplish both of these objectives, we employ explicit opponent modeling, learning a model of the opponent's behavior for training and inference. This can help in tackling the environment's non-stationarity, i.e. different opponents [86]. Additionally, if we have an explicit model of the opponent's behavior we can relate the prediction of the opponent's future actions with the agent's uncertainty of which response is appropriate, allowing us to modulate the reaction time according to this uncertainty. Finally, the opponent model can be used to make predictions of the near future, which can prove useful when correcting the reaction time delay.

In the following subsections we detail the different components of the developed solution. In Subsection 4.2.1, we present how environment modeling is performed, which includes opponent modeling. In Subsection 4.2.2, we detail the custom actor-critic RL algorithm used. The reaction time emulator is explained in Subsection 4.2.3, along with its relation with the environment model. Finally, some extra considerations are detailed in Subsection 4.2.4, and the overall architecture and algorithms are presented in Subsections 4.2.5 and 4.2.6 respectively.

4.2.1 Environment model

We can employ one of two kinds of RL methods: model-based and model-free. Model-based RL utilizes a model of the environment's dynamics, specifically for planning [11, Section 1.3]. Essentially, the agent has access to the transition function P or an approximation of it. With this, it can perform simulations of environment interactions without needing to actually experience them.

Environment models are useful for planning. At each environment state, we can look ahead into future states to anticipate the consequences of the actions we can take at this

moment. As such, we can make more informed decisions and increase the quality of the policy. A popular choice of a planning algorithm is Monte Carlo tree search (MCTS).

Another use case for environment models is the creation of training data on demand. The agent can train on environment interactions it has not experienced or has experienced infrequently, which is useful when we want to manipulate the experience to which the agent is exposed. For instance, we can train on simulated states that the agent does not experience often in an attempt to improve generalization. Training an agent in this manner, without requiring interaction in the actual environment, is called *offline* RL, as opposed to *online* RL [87].

The disadvantage of model-based methods is the need for a sufficiently accurate environment model to avoid misguiding the agent, and in scenarios where such a model is not provided it needs to be learned, which comprises most practical applications. Additionally, planning imposes a performance overhead at decision time.

Model-free algorithms do not require a model of the environment. Although each individual decision only considers its value at that specific moment, independence on the dynamics of the environment makes these methods more applicable to complex environments. These algorithms are appropriate in cases where having an accurate model of the environment is extremely difficult, such as in robotics where the real world presents noise and uncertainty.

Disregarding some exceptions, fighting games are deterministic, and therefore it is much simpler to learn a model of the environment if that need be. Since the environment is inherently digital, it is also possible for the game to provide a queryable model, as in FightingICE, which saves the need to learn one.

The only stochastic component of the environment is the opponent, which is handled by a separate opponent model. To avoid confusion, we denote the deterministic component the *game* model g and the stochastic one the *opponent* model ω. These two components in conjunction form the environment model.

As such, model-based RL should be applicable in this scenario. Planning can be especially useful for these games since we can leverage it for performing combos [68]. However, time is fine-grained in these games, and a naive planning implementation that considers each time step individually will waste the simulation of various states that are very similar between each other. Player actions take a fixed number of time steps to complete, and so the simulation of a single action from either player may need prediction over many timesteps. If we want to efficiently perform planning over the players' future courses of action, we need to employ temporal abstraction. One possibility is to only consider one time step every n time steps, although it would be a fixed value. It is not immediately obvious how planning can be efficiently performed over long periods of time, and so we opt to use a model-free method, using a model of the environment only for correcting the reaction time delay detailed in Subsection 4.2.3, but not for planning.

Game model

The game model g is learned using supervised learning (SL). It is a function $g(s_t, o_t, a_t)$ that outputs a prediction of the immediate next state s_{t+1}. Since the state is structured, the game model will predict a set of variables, with each of them being discrete or continuous. For instance, the particular state a player is in, such as moving forward, is a discrete variable, whereas the absolute position is continuous. Continuous variables do not require special treatment; for discrete variables, we consider them to be probability distributions over all possible values they can take.

The final prediction s_{t+1} may be built in different ways. The straightforward way is to directly predict s_{t+1} with the same structure as s_t. Another one is to predict a *residual*, i.e. the game model predicts the difference between states $s_{t+1} - s_t$. When we want to build a prediction from the game model's output, we simply compute $s_t + g(s_t, o_t, a_t)$. This may simplify the problem for the game model, by having it predict the rate of change rather than the change itself. Consider for example the task of predicting the player's next position when they are moving forward, assuming no acceleration. In the former method, the model has to predict that the player's next position is dependent on both its previous position and the fact that they are moving forward. Then, the model needs to add some constant to the player's position to obtain the next one. If we use the latter method, the model simply needs to predict a single additive constant when it detects that the player is moving forward, removing the dependency on the player's previous position. Essentially, we explicitly model velocity rather than position in this case. Note that this strategy only makes sense for continuous variables.

The work in [75], which tackled the same problem of reaction time correction, utilized a recurrent environment model with a residual-like architecture. The authors developed three distinct models D, F and N that together compute a prediction as

$$g(s_t, o_t, a_t) = F(s_t, o_t, a_t) \times (s_t + D(s_t, o_t, a_t)) + (1 - F(s_t, o_t, a_t)) \times N(s_t, o_t, a_t). \quad (4.6)$$

Here, the model D has exactly the same function as the second prediction strategy mentioned previously, of predicting the state *difference*. The model F determines which information to *forget* from the previous state s_t, and model N includes *new* information. We evaluate this more complex architecture in addition to the two previously mentioned methods, although we do not use recurrency since, in theory, the states have the Markov property, i.e. we always have enough information to determine the next state without checking the past. Additionally, the opponent model detailed in Subsection 4.2.1 is the one that addresses time dependencies in the environment, and for which recurrency makes sense in our case.

The authors in [75] however found a computational performance problem when predicting multiple time steps at once. Their model predicted the state at the next immediate time step $t + 1$, so n chained predictions have to be performed in order to have a prediction for the state at $t + n$. To trade computational performance for correctness, we can skip some time steps and have the game model g directly predict n steps ahead. We evaluate this trade-off.

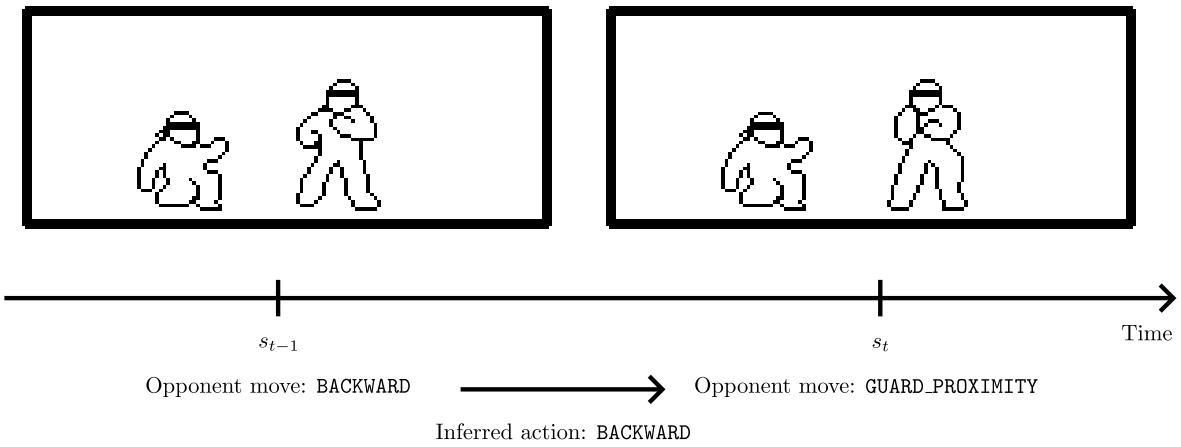


Figure 4.1: Example of opponent action inference. Here, we infer that the opponent kept performing the action of going backward at state s_{t-1}.

Opponent modeling

Similarly to the game model, the opponent model ω is learned using SL. We could consider the opponent model ω to be a function of the current environment state s_t, outputting the probability distribution over actions. If using HRL however, we can take advantage of temporal abstraction and consider the model's output to be a probability distribution over *options*, as in [88]. This goes in line with the opponent modeling that human players do in fighting games: players do not think of what the opponent will do in terms of individual button presses, but rather of the overall plan that they will follow or the special move that they will perform. However, we don't explore HRL in this work, and consider a simplified version of the opponent's action space which includes special moves, similar to the agent's. The opponent's underlying action is manually inferred from state transitions (s_{t-1}, s_t) by checking which moves they are performing. Figure 4.1 illustrates this inference process on *FOOTSIES*. The opponent's move, which is what can be visually perceived, transitioned from walking backward into a guarding state. From this, we can infer that the opponent performed the action of going backward, coinciding with what they were previously doing.

Care should be taken in how the opponent model is built. A naive SL algorithm may not constitute the opponent modeling that human players perform. One important form of reasoning in opponent modeling is recursive reasoning [89]. In recursive reasoning, we perform actions according to what the opponent thinks, but at diverse levels of thought [24]. At level 0, we do not consider the opponent's actions at all when deciding what to do. At level 1, we consider what the opponent will probably perform, which is the most simple form of opponent modeling. This means that we believe the opponent is performing reasoning at level 0. At level 2, we act according to what we believe the opponent thinks *we* will perform, meaning we think the opponent themselves is employing simple opponent modeling, i.e. level 1 reasoning. The levels go on, but human players do not tend to employ strategies greater than level 2 [90], [91]. We employ a simple learning algorithm for the opponent model, which should correspond to level 1 reasoning since we assume the opponent does not take into account any information other than the current environment state s_t for making decisions. We are not interested in

creating an agent with level 0 reasoning since it does not require an opponent model, and thus would not theoretically need a dynamic reaction time.

As such, opponent modeling is performed by simply predicting that the opponent will keep performing the most recent actions. If the opponent often performs an action x, then the model will predict that x will keep being likely. This may contrast with the opponent modeling human players perform, since they can, for instance, predict the opponent will break their patterns to avoid repetition. The opponent modeling we perform is similar in essence to rollback networking, which is how modern games usually handle online play [92]. Fighting games utilize rollback networking to make the player's inputs feel more responsive, by running the game normally as if running locally while predicting the opponent's actions before they have been received over the network. If by the time the opponent's inputs are received the game notices it has been running the game wrong, then it performs a *rollback* of the current game state to the point of the prediction error and corrects it with the actual input, and then updates the state back to the current time step. The opponent prediction algorithm tends to be simple, merely predicting the opponent's action to be the same as their last. Despite the simplicity, it is very effective in practice. Therefore, we essentially perform opponent modeling in the same vein, although it is not clear whether it is an appropriate methodology for general opponent prediction. This prediction strategy proves effective for the case of rollback in which we want gameplay inconsistencies to be nearly imperceptible, but we cannot be sure whether it is adequate for our case.

One problem of traditional artificial neural networks (ANNs) is that they are stateless, i.e. they do not incorporate any kind of memory. This is not a problem when approximating the policy, action-value function and game model, since we made sure that past information should not be necessary to determine the desired output from input. However, this is not the case for an opponent model. Opponents can range from simple algorithms to human players, which can incorporate state into their decision making. For this scenario, recurrent ANN architectures are more appropriate than ones that do not consider state. However, recurrent networks have shown to be harder to effectively use in practice [2], [93], [94]. In fact, for the task of prediction for rollback networking, recurrent neural networks (RNNs) have shown to be too computationally expensive for real-time play [93]. Nevertheless, we evaluate whether a recurrent architecture is able to improve opponent prediction performance.

RNNs work by having nodes that receive some hidden state along with the network input, and compute another hidden state from the pair. The hidden state that is computed for a given input-hidden state pair is kept as the hidden state for the next inference step. There are two concerns we need to address regarding the hidden state: which hidden state to consider on reset, i.e. the very first input, and at which points in time should the hidden state reset be performed, since we do not need to keep context all the time. In terms of which hidden state to consider on reset, we can just use zeros, which is inclusively the default behavior in PyTorch [95]. Regarding at which points to reset the hidden state, we can consider episode termination as a context ender. With this, we are effectively learning the sequential behavior of the opponent from the beginning of the episode up until the end. However, we can increase

the context granularity. It is likely that the opponent is not going to factor in long-term information such as the beginning of the episode when determining decisions in the far future. Instead, we also propose two other strategies for considering different contexts: one in which after any player is hit, the hidden state is reset, and another that uses the same strategy but also resets the hidden state when the players are positioned too close to one another. With this separation strategy, we better differentiate between different "neutral" stages during an episode. In fighting games, neutral is the stage in play where players are spaced apart enough to allow free movement, and without any of them being hit. In these situations the opponent's behavior is more ambiguous, and so it is very important to understand how an opponent behaves during neutral. Additionally, the opponent is likely not going to take into account what happened at the previous neutral stages to decide what to perform at the current one. We evaluate the effect of these three context reset strategies.

We also attempt to include an adjustment to tackle class imbalance. There may be some actions that the opponent barely uses, or that they use infrequently compared to other actions. Consider, for example, the actions of walking forward or backward, which are instantaneous, and therefore occur much more frequently than attack actions. This means that these actions will be updated much more frequently, biasing the opponent model. To alleviate this imbalance, we apply a multiplier that will give an importance to each action inversely proportional to the frequency, in percentage, with which the opponent performs it. This way, the updates will be larger for infrequently used actions, and much smaller for frequently used ones. Whether this adjustment improves prediction performance is evaluated, and will be denoted as the *dynamic adjustment of loss weights* hereafter.

To avoid overfitting to a specific opponent, which would make the model harder to adjust to different opponents, we introduce an entropy coefficient β that encourages the model to retain some randomness and not degenerate into a deterministic model.

4.2.2 Reinforcement learning

For the RL algorithm, we employ a policy gradient method with function approximation. Policy gradient methods, mainly Proximal Policy Optimization (PPO), produced more human-like behavior than value and action-value algorithms which have indirect policy representation such as Q-learning [2], [38]. This is due in part to the fact that policy gradient methods learn parameterized policies which can be stochastic, whereas value and action-value methods need to introduce stochasticity manually, such as through an ϵ-greedy policy or a softmax distribution, which are not learned. Additionally, policy gradient methods lend themselves better to performing imitation learning since the policy itself can be directly trained to copy other actions rather than being driven by reward, which can allow for starting training with a more sensible policy than a randomly initialized one. With this in mind, we study an actor-critic method, a subclass of policy gradient methods. Actor-critic methods jointly learn a parameterized policy, which is the actor, as well as a value or action-value function, which is the critic used to evaluate said policy. Estimating the value or action-value function has the advantage of providing diagnostics on the game, allowing us to get insights on the balance

of the game at different states for each player.

As such, we opt to experiment with an adapted version of the Advantage Actor Critic (A2C) algorithm [96]. This algorithm is relatively simple in terms of implementation, especially when compared with PPO which is predominant in the state of the art [97], [98]. A2C is also appropriate for discrete action spaces, which is our case. ANNs are used for function approximation of the agent policy π and action-value function Q_π. In actor-critic methods, the value function V_π is often used instead of the action-value function Q_π since it is simpler, but we learn an action-value function since it considers the agent's action as well, allowing the creation of payoff matrices for diagnostics. Because we are explicitly considering the opponent's action in π and Q_π, for a single state s_t we will have a matrix as output, with a row for each opponent action and a column for each agent action, hence why we can naturally have payoff matrices.

We could perform representation transfer as in [83], having both the policy and action-value networks sharing parameters initially before a branching point. The initial part of these networks is the representation function, creating an abstract representation of the input state, for use by the policy and action-value networks. This allows both networks to share an abstract representation of the input state, which should contain useful information for both performing an action at that state and evaluating how good that state is. This has the advantage of allowing better adaptation to slightly different environments, in theory requiring only a change in the representation function. Since fighting games can be updated over time, changing the internal game mechanics, this can prove to be a useful construct in improving adaptation efficiency [83]. Despite this, we experiment with not sharing parameters between these networks, as representation transfer may disrupt the training of the actor and the critic.

Actor learning

The actor is learned using the policy gradient theorem introduced in Subsection 2.1.3. The challenge in policy gradient methods is approximating the gradient $\nabla J(\theta)$, and different methods exist bringing different trade-offs. We opt for using the advantage formulation, which is used both for A2C and PPO. The advantage A of action a_t at state s_t is defined as the difference between the action-value of the state-action (s_t, a_t) pair and the value of s_t, according to

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t). \tag{4.7}$$

With the reformulations of Section 4.1, we have

$$A(s_t, o_t, a_t) = Q(s_t, o_t, a_t) - V(s_t, o_t). \tag{4.8}$$

The advantage dictates how good the given action a_t is in state s_t when compared with other actions, being positive if it is better than other actions in expectation and negative if it is not.

The actor is then updated after performing an action a_t at state s_t using the update rule

$$\theta \leftarrow \theta + \alpha A(s_t, o_t, a_t) \nabla \log \pi(a_t | s_t, o_t), \tag{4.9}$$

with θ being the parameters of policy π, α being the learning rate, and o_t being the opponent's action inferred from the transition from s_t to s_{t+1}. This update rule has an intuitive meaning: if the advantage is positive/negative, then we will increase/decrease the probability of doing a_t on state s_t when the opponent does o_t, with the update magnitude being proportional to the absolute value of that advantage. The logarithm accounts for the fact that we sampled a_t: there is an inherent bias when we perform an update using a_t because a_t was sampled from the policy π, so actions that were more likely to be sampled will have updates more often, which is "unfair". Recall that in the policy gradient formula (2.10), we never sample a *single* action, we just consider *all* of them. As such, to overcome this bias we divide the gradient by the probability of a_t being sampled, which ends up being equal to the gradient of the log-probability according to

$$\nabla \log \pi(a_t|s_t, o_t) = \frac{\nabla \pi(a_t|s_t, o_t)}{\pi(a_t|s_t, o_t)}.$$

This has the effect of increasing the update when the probability of sampling a_t is small, since we would sample a_t infrequently, and decrease the update when the probability is high.

We need to address how the advantage A is computed in practice. Since we learn a critic, which is an approximation of the action-value function, we need to compute the value $V(s_t, o_t)$ using action-values. Therefore, we compute it according to

$$V(s_t, o_t) = \sum_{a \in \mathcal{A}} \pi(a|s_t, o_t) Q(s_t, o_t, a), \tag{4.10}$$

and so the advantage is calculated according to

$$\begin{aligned} A(s_t, o_t, a_t) &= Q(s_t, o_t, a_t) - V(s_t, o_t) \\ &= Q(s_t, o_t, a_t) - \sum_{a \in \mathcal{A}} \pi(a|s_t, o_t) Q(s_t, o_t, a). \end{aligned} \tag{4.11}$$

There is an alternative formulation of the advantage that may allow faster learning, especially during the beginning of training. It basically involves writing $Q(s_t, o_t, a_t)$ in (4.11) as a function of the action-value at the next state s_{t+1} according to

$$\begin{aligned} A(s_t, o_t, a_t) &= Q(s_t, o_t, a_t) - \sum_{a \in \mathcal{A}} \pi(a|s_t, o_t) Q(s_t, o_t, a) \\ &= \sum_{s \in \mathcal{S}} \sum_{o \in \mathcal{A}_o} \omega(o|s) G(s|s_t, o_t, a_t) \left[R(s, o|s_t, o_t, a_t) + \gamma V(s, o) \right] - V(s_t, o_t) \\ &= \sum_{o \in \mathcal{A}_o} \omega(o|s_{t+1}) \left[R(s_{t+1}, o|s_t, o_t, a_t) + \gamma V(s_{t+1}, o) \right] - V(s_t, o_t) \\ &= R(s_{t+1}|s_t) + \gamma \sum_{o \in \mathcal{A}_o} \omega(o|s_{t+1}) V(s_{t+1}, o) - V(s_t, o_t) \\ &= R(s_{t+1}|s_t) + \gamma \mathbb{E}_{o \sim \omega} \left[V(s_{t+1}, o) \right] - V(s_t, o_t), \end{aligned} \tag{4.12}$$

with s_{t+1} being the only state $s \in \mathcal{S}$ for which $G(s|s_t, o_t, a_t) = 1$. This formulation has the advantage of using the reward signal from the environment R directly, without the agent having to wait for it to be integrated into the estimation of the action-value function. For instance, if the agent receives a reward of 1 for performing action a_t in s_t, the agent will immediately use that reward of 1 to drive its update if using (4.12), but in (4.11) we need to trust $Q(s_t, o_t, a_t)$ already has that reward of 1 integrated into its value, which it will not have at the beginning of training. However, this formulation has a dependence on the opponent model ω, to determine which future opponent action to consider. In this scenario, we can have updates similar to the ones explained in the next section on critic learning. For example, we can use Sarsa-like updates for the opponent, which means we simply wait for another state s_{t+2} and observe the next opponent action o_{t+1}, and use that rather than the opponent model. Since each formulation has its trade-offs, we evaluate both of them. Hereafter, we denote (4.11) the *original* advantage formula and (4.12) the *alternative* advantage formula.

Similarly to the opponent model, we also introduce an entropy coefficient β into (4.9) to incentivize the policy to retain some randomness and not degenerate to an almost deterministic one, with which it would be difficult to explore, according to

$$\theta \leftarrow \theta + \alpha \left[A(s_t, o_t, a_t) \, \nabla \log \pi(a_t|s_t, o_t) + \beta \, \nabla H(\pi(\cdot|s_t, o_t)) \right]. \tag{4.13}$$

During experimentation, we found the need to include gradient clipping. Even with the entropy coefficient, the actor is still susceptible to performance collapse, where the agent becomes deterministic due to occasional large updates, and does not recover. We found gradient norm clipping of the actor's gradients to be enough to avoid these performance collapses.

Finally, fighting games have a dynamic action space, i.e. the player's available actions change depending on the current game state. This may be tackled by using action masking [99], [100], which sets the probability of performing invalid actions to zero. Since we use the softmax function to create the probability distribution π, we can multiply the values before the softmax, called "logits", by a mask consisting of ones and $-\infty$, with ones for valid actions and $-\infty$ for invalid actions. This has the added benefit that, during gradient-based optimization, the gradient concerning these actions will be zero, and so invalid actions will not interfere with learning. We evaluate whether this trick improves performance.

Critic learning

We learn the action-value function online through temporal difference (TD) learning, i.e. bootstrapping. We know that the action-value of a given state-action pair (s_t, a_t) can be written as a function of the action-values of the next state according to (2.6). Considering the reformulations in Section 4.1, we have that

$$
\begin{aligned}
Q_\pi(s_t, o_t, a_t) &= \sum_{s\in\mathcal{S}} \sum_{o\in\mathcal{A}_o} P(s, o|s_t, o_t, a_t) \left(R(s, o|s_t, o_t, a_t) + \gamma \sum_{a\in\mathcal{A}} \pi(a|s, o) Q_\pi(s, o, a) \right) \\
&= \sum_{s\in\mathcal{S}} \sum_{o\in\mathcal{A}_o} \omega(o|s) G(s|s_t, o_t, a_t) \left(R(s|s_t) + \gamma \sum_{a\in\mathcal{A}} \pi(a|s, o) Q_\pi(s, o, a) \right) \\
&= \sum_{o\in\mathcal{A}_o} \omega(o|s_{t+1}) \left(R(s_{t+1}|s_t) + \gamma \sum_{a\in\mathcal{A}} \pi(a|s_{t+1}, o) Q_\pi(s_{t+1}, o, a) \right) \\
&= \sum_{o\in\mathcal{A}_o} \omega(o|s_{t+1}) R(s_{t+1}|s_t) + \sum_{o\in\mathcal{A}_o} \gamma\omega(o|s_{t+1}) \sum_{a\in\mathcal{A}} \pi(a|s_{t+1}, o) Q_\pi(s_{t+1}, o, a) \\
&= R(s_{t+1}|s_t) + \gamma \sum_{o\in\mathcal{A}_o} \omega(o|s_{t+1}) \sum_{a\in\mathcal{A}} \pi(a|s_{t+1}, o) Q_\pi(s_{t+1}, o, a) \qquad (4.14)
\end{aligned}
$$

with s_{t+1} being the only state $s \in \mathcal{S}$ for which $G(s|s_t, o_t, a_t) = 1$.

There are different ways we can update our estimate of $Q_\pi(s_t, o_t, a_t)$ based on (4.14), depending on which kind of agent π or opponent ω we assume for the returns. Fixing the opponent choice (i.e. the environment's transition dynamics), we may approximate the following expectation

$$
\sum_{a\in\mathcal{A}} \pi(a|s_{t+1}, o) Q_\pi(s_{t+1}, o, a) = \mathbb{E}_{a_{t+1}\sim\pi} Q_\pi(s_{t+1}, o, a_{t+1}) \qquad (4.15)
$$

by sampling an action a_{t+1} from $\pi(\cdot|s_{t+1}, o)$, and then interacting with the environment using a_{t+1} on s_{t+1}. So, in place of the summation, we merely use the action-value of the next state with our sampled action. These correspond to Sarsa updates, from the fact that for a single action-value update we need a quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ with $r_{t+1} = R(s_{t+1}|s_t)$. This update method is useful in case the policy distribution π is not represented explicitly or is difficult to derive, as in value-based methods, but since we are using a policy gradient method we have an explicit representation of π. As such, we can save ourselves from sampling a_{t+1} and use π directly to obtain the exact value of the expectation. These are called Expected Sarsa updates.

We can consider a special case in which we consider π to be a policy different from the one that is being used to generate samples (i.e. interact with the environment). If we consider π to be a policy that always chooses the action a with the greatest action-value for the current state, then the policy is called "greedy" and is also an optimal policy if $Q_\pi = Q_*$. Therefore, the summation above is computed as

$$
\sum_{a\in\mathcal{A}} \pi(a|s_{t+1}, o) Q_\pi(s_{t+1}, o, a) = \max_a Q_\pi(s_{t+1}, o, a). \qquad (4.16)
$$

These correspond to Q-learning-like updates. In fact, Q-learning is a specialization of Expected Sarsa, in which π is greedy. As such, we also call these "greedy" updates.

These updates can similarly be considered for the opponent model ω, and so we can have Sarsa, Expected Sarsa and greedy updates. If we utilize Sarsa or Expected Sarsa updates, then whenever we perform an update we are always computing values assuming the current

opponent. On the other hand, if we utilize greedy updates, then we are being pessimistic and assume the opponent will always choose their best possible action. We obtain these values regardless of the opponent used, as long as the opponent continuously permits us to experience all environment states, i.e. does not restrict our experience to a small subset. It is important to note that the greedy opponent policy ω chooses an action that *minimizes* our action-value, and so the summation in (4.14) becomes

$$\sum_{o \in \mathcal{A}_o} \omega(o|s_{t+1}) \sum_{a \in \mathcal{A}} \pi(a|s_{t+1}, o) Q_\pi(s_{t+1}, o, a) = \min_o \sum_{a \in \mathcal{A}} \pi(a|s_{t+1}, o) Q_\pi(s_{t+1}, o, a). \tag{4.17}$$

It is not obvious whether this is true, but considering we are in a zero-sum environment, the minimization of the other player's returns should correspond to the maximization of ours, assuming the returns are correct.

In the actor-critic setting, using Q-learning updates for the policy π, i.e. assuming it to be greedy, does not make sense, since the policy gradient is formulated using the value and action-value functions assuming policy π. However, we can assume the opponent ω to be greedy, which should allow us to obtain a policy π that is robust against worst-case scenarios and avoids exploitation, but as a downside does not learn how to exploit the current opponent. If we had considered a different method, however, we could assume both players to be greedy, and solve a minimax problem to find a policy in the Nash equilibrium [101]. We are not interested in finding an unexploitable agent approximating a Nash equilibrium, rather, we want an agent that learns to exploit the current specific opponent.

In practice, learning value and action-value functions can be challenging when function approximators are being used. One of the instabilities that can be found is when performing bootstrapping, in which we update the action-value at the current state s_t using an estimate of the action-value at the next state s_{t+1}. When using function approximators such as ANNs we indirectly update the output for many inputs even when training on just one example, so if we increase the action-value at state s_t then it is likely that the action-values at states similar to s_t are increased as well. Since s_{t+1} is very likely to be similar to s_t since it is just one time step away (depending on how the state is structured), then when we update the action-value at the current state s_t we are also updating the estimate of the action-value at the next state. For example, if we increase the action-value of the (s_t, o_t, a_t) triple because the target value computed using (4.14) is larger, then we will also increase the action-value of s_{t+1}, and so for the next update at (s_t, o_t, a_t) we can expect the target to be larger again since the action-value of s_{t+1} increased. This has the undesirable effect of diverging action-values, in which action-values increase/decrease indefinitely. To alleviate these dependencies, one technique that can be employed is the usage of target networks. Essentially, the action-value estimates of the next state are provided by a slowly-changing version of our action-value function network. This means that the next state estimates do not update jointly with the action-value of the current state when it is updated, breaking these conjoined updates.

4.2.3 Reaction time

The developed agent has a reaction time constraint emulating the human beings' choice reaction time. As such, the imposed artificial reaction time should not be static, but rather be dependent on the uncertainty of the agent [19]. The more options the agent needs to react to *differently*, the longer it will take to react, but not proportionally. The "options" we are concerned with are those of the opponent, since they are the only factor of the environment that the agent cannot predict with certainty. Hick's law states that choice reaction time increases logarithmically with the number of options, and an approximation k can be calculated as

$$k = m \log(n + 1) + c, \tag{4.18}$$

where n is the number of options and m and c are constants requiring tuning. Note that the base of the logarithm does not matter since it can be compensated by the constant m. n is incremented by 1 in the logarithm to model the inexistence of an option and the inherent uncertainty it brings. That is, during the time until an option occurs, the agent does the special response of doing nothing. In the original work, this produced slightly better fits to human data than when 1 was not added [19]. In both the agent's and opponent's action space, we explicitly consider the existence of the action of doing nothing, so we don't need to perform this compensation.

However, (4.18) only works if the n options are equiprobable. In the context of fighting games, it means the opponent is uniformly random, which is unrealistic. Rather, the probability with which they perform each option is different and can vary over time. If those probabilities follow the discrete probability distribution ρ, we can calculate k as

$$k = mH(\rho, s) + c, \tag{4.19}$$

where $H(\rho, s)$ is the entropy associated with the distribution ρ measured over the agent's action space $\mathcal{A}$ at the environment state s, given by

$$H(\rho, s) = -\sum_{a \in \mathcal{A}} \rho(a|s) \log(\rho(a|s)). \tag{4.20}$$

It is worth noting that decision making is performed sequentially, as mentioned in Chapter 2, and so a model based on the joint probability of consecutive events would be more appropriate [20]. But due to the added complexity of this model which is harder to translate to our scenario, we opt for using the simpler equation (4.19), adding some error.

Since we are using opponent modeling, we can make use of the opponent model to gauge how uncertain the agent is of the opponent's behavior. The opponent model outputs a probability distribution $\omega(\cdot|s)$ over their actions to which the agent has to form an appropriate response, and so we could calculate the entropy H_ω to measure this uncertainty. But we not only have to consider the probabilities with which the opponent is going to perform each action, but we also need to consider which *responses* are required of the agent against those actions, since in these games there may be more than one suitable response for each action

and the sets of responses can overlap. For example, if the opponent is expected to perform one of three actions, but all of those actions require the same response, then we would not see an increase in reaction time. After all, the three actions do not impose more than a single decision on the player, so there are no choices to be made when reacting.

Therefore, knowing our predictions $\omega(o|s)$ for what the opponent will perform, we then need to check which responses we would perform. The agent policy π outputs the probability $\pi(a|s,o)$ of the agent performing action a at state s coupled with the fact that the opponent will perform o. So, for each action o of the opponent we have a probability distribution $\pi(\cdot|s,o)$. We then build a *decision* distribution ρ, which dictates the probability with which the agent will perform an action a in state s, implicitly taking into account what the opponent would perform. To obtain the decision distribution ρ, we compute a weighted sum of ω and π, with each distribution $\pi(\cdot|s,o)$ being weighted by $\omega(o|s)$, according to

$$\rho(a|s) = \sum_{o \in \mathcal{A}_o} \omega(o|s)\pi(a|s,o), \tag{4.21}$$

with $\mathcal{A}_o$ being the opponent's action space and $a \in \mathcal{A}$ being an action from the agent's action space $\mathcal{A}$. Essentially, $\rho(a|s)$ is the sum of joint probabilities of the opponent performing o in s and the agent performing a in s knowing the opponent would indeed perform o, along all $o \in \mathcal{A}_o$. In other words, $\rho(a|s)$ is the marginal distribution of A on the joint probabilities $p(A \cap B)$, where A and B are random variables corresponding to which action the agent and the opponent performed respectively. These joint probabilities are calculated using the definition of conditional probability: $p(A \cap B) = p(A|B)P(B)$, with $p(A|B)$ being given by the policy π and $p(B)$ being given by the opponent model ω. In this context, $p(X)$ stands for the probability of the random event X. Having ρ, we can compute k for the current environment state s using (4.19).

The architecture of this system is presented in Fig. 4.2. The system works on a history of states from the environment. For the current environment step t, we estimate the reaction time k and provide the agent the state s_{t-k}, with s_t being the state associated with the current instant. The agent's currently perceived observation $b_t = s_{t-k}$ is a function of the decision entropy calculated on the previous observation $b_{t-1} = s_{t-k'-1}$. The reaction time k at the current state is

$$k = \min\left\{k' + 1, \lceil mH(\rho, b_{t-1}) + c \rceil\right\}, \tag{4.22}$$

where $H(\rho, b_{t-1})$ is the entropy of the decision at the previous state, k' is the reaction time computed at the previous instant $t-1$, and m and c are parameters requiring tuning. Taking the minimum avoids the agent erroneously perceiving an event further in the past than $s_{t-k'-1}$; the most that can happen is $s_{t-k'-1} = s_{t-k}$ when uncertainty is even higher than in the previous time step. With this, s_{t-k} will always be a more recent observation than $s_{t-k'-1}$ or equal to it. The rate at which reaction time increases is no more than one time step per time step, but the rate at which it decreases is not limited.

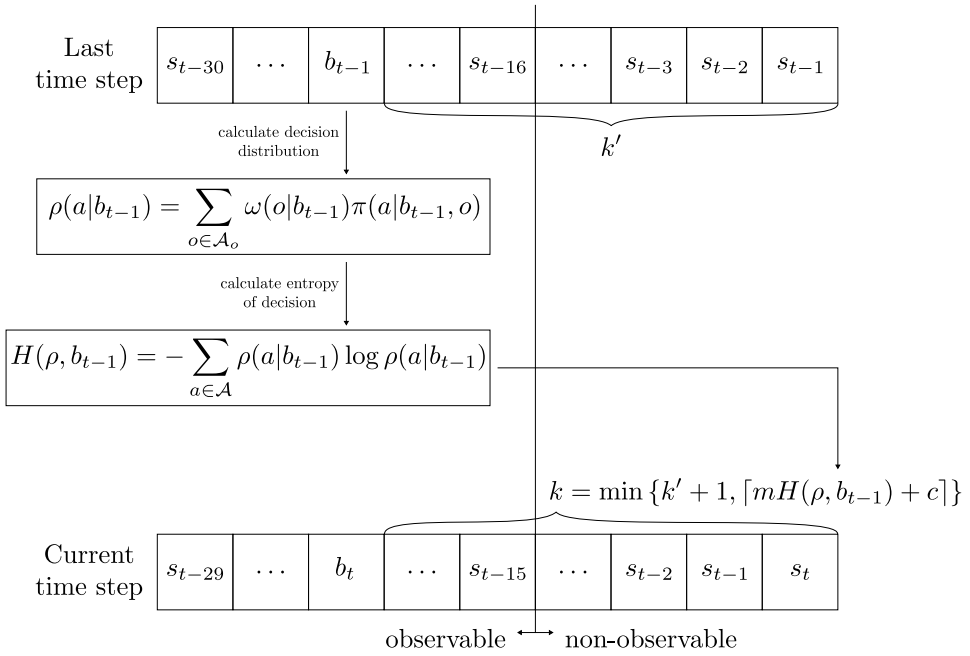


Figure 4.2: Architecture of the human reaction time emulation component.

All environment states that are no more than c time steps old are not observable, since 0 is the minimum entropy value. As such, we can consider c as the minimum human reaction time without any uncertainty, which is the simple reaction time. The developers of *Killer Instinct* performed experiments evaluating single reaction time to visual stimuli and noted that most players had between 12 and 19 frames (time steps) of reaction time [7]. Considering this, we set the minimum reaction time to be $k_{min} = c = 15$ time steps, a middle point between 12 and 19, which is equivalent to 250 milliseconds. We use visual reaction time since it is the predominant form of reaction time in these games, although audio is also possible. We also consider a maximum reaction time of $k_{max} = 29$ time steps. We believe this is enough as players rarely need more than half a second to react, reflected in the fact that actions in fighting games rarely take half a second to become active. Since $\log(|\mathcal{A}_a|)$ is the maximum entropy value, with $\mathcal{A}_a$ being the agent's action space and $|\mathcal{A}|$ its cardinality, we set m and c to

$$m = \frac{k_{max} - k_{min}}{\log(|\mathcal{A}|)} \quad \text{and} \quad c = k_{min}.$$

With the use of delayed observations, we need to address one potential issue. Since we are using observations that are in the past but acting in an unobserved state in the future, there is a discrepancy between what the agent thinks the current state is and what state it is actually acting in, which may introduce instability during training. This is further exacerbated by the fact that observations are delayed by different amounts depending on the decision entropy above. One possible way to remedy this issue is to make use of the environment model to predict the current state $\tilde{s_t}$, and consider that predicted state instead for training and inference. We can even consider the predicted state for inference only, and train with delay since what matters most is that reaction time is dealt with at interaction time. We do

not need to consider the policy π when predicting since we can make use of the history of the agent's executed actions when making this prediction. For instance, in Fig. 4.2, the agent executed an action a_{t-1} at state s_{t-1}, which was the current instant in the last time step; when predicting the current state s_t, we consider the transition from s_{t-1} to s_t using action a_{t-1}. This correction is motivated by the compensation human beings subconsciously perform on their reaction time. There are many hypotheses explaining the phenomenon, with one of them dictating that human beings internally predict the future ahead of time [102], similarly to what was previously explained. For instance, to catch an airborne ball, we instinctively predict its trajectory since we cannot possibly react to its actual position. The flash-lag effect is an optical illusion illustrating this phenomenon [1]. Although we correct the perceived observation to always be the state at the current instant, we are introducing some error by relying on the environment model, as it is not completely accurate. In fact, we found the need to manually fix every prediction in order for the state variables to have values that make sense.

The authors in [75] found that introducing an environment model to fully compensate for the delay yielded greater returns than when the environment model was either not used completely or only partially, at the expense of more computational time required. Even though this work considered action delay rather than observation delay, they end up being equivalent [103] since the only difference is whether the agent's perceived observation at time step t is the actual state s_t, as the action the agent chooses is always performed some time steps after the observation. The authors additionally found that predicting over more than 8 time steps was too computationally expensive for the algorithm to be run in real-time. Contrary to this work, we decouple the environment model into two components, not using a recurrent architecture for the game model and leaving recurrency for the opponent model only, so there is an opportunity for improved performance. Nevertheless, we also evaluate whether this correction improves task performance in our case and its impact on computational performance.

4.2.4 Other considerations

In fighting games, there are many time steps in which a player cannot act. This is the case when the player is performing a move, for instance. As such, there are no decisions to be made in these cases, and so we can skip decision making. Although this technique has been referred to as "frame skipping" in past work [56], we refer to it as "decision skipping" hereafter to avoid confusion with the technique of only perceiving the environment once every N steps which has the same name. Decision skipping simplifies training and avoids unnecessary computation, and is a form of temporal abstraction since the agent's actions lose their temporal component. It is worth noting however that the degree to which decision skipping can actually be done depends on the game it is being applied to. Ideally, an HRL technique offering temporal abstraction should alleviate the need for decision skipping.

[1] Flash-lag effect example at: `https://michaelbach.de/ot/mot-flashLag/`

Even though decision skipping allows us to avoid performing decision making when it is not needed, it might still be useful to use the step-by-step interactions for training the agent. One example is improving our estimates of state/action values, or improving the environment model if one is being learned. Additionally, we could even make use of these non-actionable periods to perform planning. This means, for instance, we do not care about $Q(s, o, a)$ if at state s the agent could not act. Note that decision skipping only concerns the actor-critic components, which are those mainly responsible for making decisions, and not the environment model components. We evaluate the effect of decision skipping when applied to training.

The opponent also has periods in which they cannot act. Since we explicitly consider which action the opponent will perform, we need to be careful of what action we assume the opponent to be doing in these periods. For example, if at state s the agent could act but not the opponent, and so their action o is irrelevant, then what action o do we consider for $Q(s, o, a)$? For this work, we evaluate three different strategies: consider any action, i.e. $Q(s, o, a)$ should be the same for any $o \in \mathcal{A}_o$, the last valid action, or the special action of doing nothing, which we already include in the agent's and opponent's action spaces. In terms of theoretical correctness, the first option sounds the most appropriate. However, since we are approximating these components using ANNs, we can have problems similar to the one with correlated states detailed in Subsection 4.2.2. ANNs evaluate their inputs continuously, so unless strong abstractions are formed, an update of $Q(s, o, a)$ can affect the action-values $Q(s', o, a)$ for states s' similar to s. Basically, the updates performed during the opponent's unactionable periods can "leak" into similar states s', causing a change in the action-values $Q(s', o, a)$ even though we technically did not see the opponent perform o in s'. In fact, we technically did not see the opponent perform o at all. Considering o to be the action of doing nothing seems the next most reasonable option, but we may be introducing biases into that action by constantly performing updates considering that one action.

Another aspect worth taking into account is the "hitstop" and "blockstop" that are present in fighting games. Hitstop refers to the period in which, after any player is hit, the game completely freezes in time for some frames. Blockstop is analogous for situations in which a player blocks an attack. This is meant to enhance the user experience by providing clear feedback on key moments of the game. This however has implications on how the agent perceives the environment. If we consider hitstop as being part of the environment, then the agent will perceive these freezing periods as environment transitions where $s_t = s_{t+1}$. This means that, to ensure the Markov property, the agent needs to also know at which point in the hitstop/blockstop period it is so that it can predict at which point it will end. Because of this, unless we add this very specific state information, hitstop/blockstop will affect how the agent learns. We perform an adjustment where, for agent updates, we ignore hitstop/blockstop freeze periods completely. The agent performs an action as soon as it perceives hitstop/blockstop freeze, and does not perform any other action until it is finished. We evaluate the effect of this adjustment on agent performance. Note that hitstop and blockstop should still be considered for the state history in the reaction time emulator. This means that hitstop and blockstop give the agent a better chance to react, similar to its effect on human players.

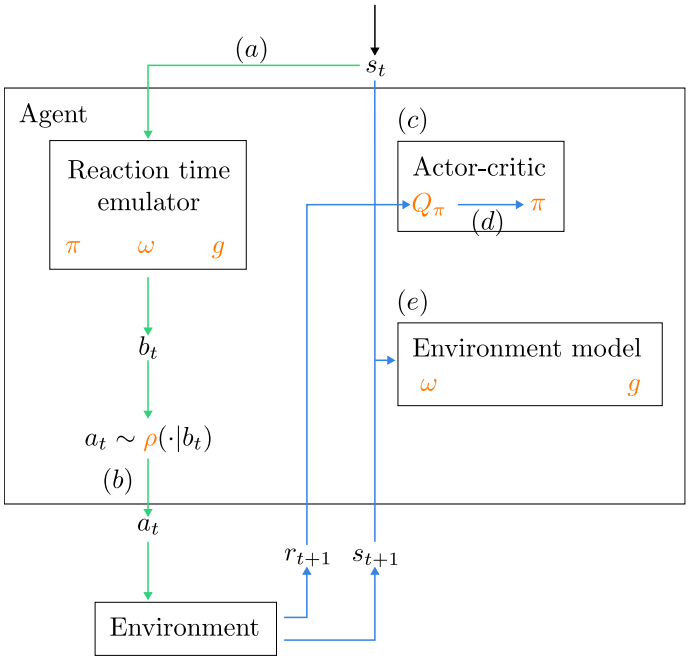


Figure 4.3: Solution architecture, illustrating training and inference when interacting with the environment. Orange components are learned, green paths indicate inference and blue paths indicate training.

4.2.5 Architecture

Figure 4.3 illustrates the generic training loop of the agent. At every interaction step in the environment, we obtain the environment state s_t of the current time instant. **(a)** Firstly, we pass the state s_t through the reaction time emulator to obtain an observation b_t, which is delayed in time. The emulator may also correct b_t to point to the current instant. If no reaction time emulation is to be performed, then $b_t = s_t$. **(b)** Then, to interact with the environment an action a_t is sampled from the decision distribution ρ. The decision distribution ρ is created using the opponent model ω and policy π according to (4.21). After acting in the environment, the agent's components are updated using the reward r_{t+1} and next state s_{t+1}. **(c)** The actor-critic components, the action-value function Q_π and policy π, are updated using the update methods described in Subection 4.2.2. **(d)** One peculiarity worth noting is that the critic is updated before the actor, so that the actor can benefit from the updated critic values since it depends on the critic. **(e)** Finally, the environment model components, the opponent model ω and game model g, are updated, ignoring the reward signal. Note that the game model g is only used for correcting the observation b_t into an estimation $\tilde{s}_t$. We can train the environment model offline, separated from all other components.

4.2.6 Algorithm

Algorithm 1 shows how to perform interaction with the environment for one episode, without considering reaction time. We first sample the initial episode state s_0 before we get in an interaction loop until episode termination. At each iteration we first sample the action o that we think the opponent will perform, and then sample action a conditioned on o. Note

that sampling first from ω and then from π as shown is equivalent to sampling once from ρ. Then we execute a and get the next state s, repeating this process.

Algorithm 1: Interaction over an episode, without reaction time.

Observe initial environment state: $s \leftarrow s_0$
repeat
  Sample opponent action o from $\omega(\cdot|s)$
  Sample action a from $\pi(\cdot|s, o)$
  Interact with the environment with a
  Observe s from the environment
until *s is terminal*

Introducing reaction time adds some more complexity. Algorithm 2 presents the additional steps required when incorporating the reaction time system of Subsection 4.2.3. In essence, the interaction loop of Algorithm 1 needs to occur on delayed states. For predicting the future, we could sample π to get the actions the agent would do, but we can increase correctness and performance by instead considering the actions that it already performed, as was explained in Subsection 4.2.3.

Algorithm 2: Interaction over an episode, with reaction time.

Observe initial environment state: s_0
Consider first perceived observation: $b \leftarrow s_0$
$k \leftarrow +\infty$
repeat
  `/* Perceive delayed state */`
  Compute decision distribution: $\rho \leftarrow \pi(\cdot|b, \cdot)\omega(\cdot|b)$
  Calculate reaction time: $k \leftarrow \min\{k+1, \lceil mH(\rho, b) + c \rceil\}$
  $b \leftarrow \begin{cases} s_{t-k}, & t \geq k \\ s_0, & t < k \end{cases}$
  `/* Correct perceived observation */`
  $k' \leftarrow k$
  while $k' > 0$ **do**
    Sample opponent action o from $\omega(\cdot|b)$
    $b \leftarrow g(b, o, a_{t-k'})$
    $k' \leftarrow k' - 1$
  end
  `/* Interact with environment */`
  Sample opponent action o from $\omega(\cdot|b)$
  Sample action a_t from $\pi(\cdot|b, o)$
  Interact with the environment with a_t
  Observe s_t from the environment
until *s_t is terminal*

Algorithm 3 shows the learning loop. In practice, the learning and interaction loops are interleaved, where at each iteration the agent acts first according to either Algorithm 1 or 2

and then the agent learns from that interaction. It is merely shown separately for clarity. There are four different parameterized functions to learn, each approximated using ANNs in implementation. The policy's parameters θ_a are updated using an approximation of the policy gradient with advantage. The action-value function's parameters θ_q are updated using mean-squared error between the target action-value and the current action-value. One noteworthy detail is that the gradient with respect to θ_q considers $V_\pi(s_t, o_t)$ to be parameterized, but not q_{target}, i.e. $\nabla_{\theta_q}\, q_{target} = 0$. The opponent model's parameters θ_o are updated using the cross-entropy loss, which is appropriate for classification problems in which we output a probability distribution over the classes (in this case, opponent actions). Finally, the game model's parameters θ_g are updated to minimize the cross-entropy loss in the case of discrete state variables such as the move a player is performing, and the mean-squared error for continuous variables such as a player's position. It is worth reminding that $V(s_T, \cdot) = 0$ when at step T the environment has terminated. It is also important that the environment transitions are sampled using the constantly-updating π since the actor-critic method employed is on-policy. The value $V_\pi(s_{t+1}, o_{t+1})$ is calculated according to (4.10).

Reaction time is not considered at all when performing updates. We want to avoid introducing instability during training when considering corrected delayed observations. Even then, it would be fairer to queue updates and only perform them some time later, e.g. update after a number of time steps equal to the maximum reaction time. This would not allow the agent to learn from transitions it could not even react to.

Algorithm 3: Learning over an episode.

Input: policy π with parameters θ_a, learning rate α_a and entropy coefficient β_a
Input: action-value function q with parameters θ_q and learning rate α_q
Input: opponent model ω with parameters θ_o, learning rate α_o and entropy coef. β_o
Input: game model g with parameters θ_g and learning rate α_g

$v_i(s) \triangleq$ partition of state s corresponding to variable i
repeat
  Sample transition $(s_t, a_t, r_{t+1}, s_{t+1})$ from the environment using π and ω
  Infer opponent action o_t from the transition (s_t, s_{t+1})
  $q_{target} \leftarrow r_{t+1} + \gamma \mathbb{E}_{o\sim\omega} V_\pi(s_{t+1}, o)$
  Update actor:
    $\theta_a \leftarrow \theta_a + \alpha_a \left[(q_{target} - V_\pi(s_t, o_t))\ \nabla_{\theta_a} \log \pi(a_t|s_t, o_t) + \beta\ \nabla_{\theta_a} H(\pi(\cdot|s_t, o_t))\right]$
  Update critic: $\theta_q \leftarrow \theta_q - \alpha_q \nabla_{\theta_q} (q_{target} - V_\pi(s_t, o_t))^2$
  Update opponent model: $\theta_o \leftarrow \theta_o + \alpha_o \left[\nabla_{\theta_o} \log \omega(o_t|s_t) + \beta\ \nabla_{\theta_o} H(\omega(\cdot|s_t))\right]$
  Update game model:
    $$\theta_g \leftarrow \theta_g - \alpha_g \nabla_{\theta_g} \sum_i \begin{cases} (v_i(s_{t+1}) - v_i(g(s_t, o_t, a_t)))^2, & i \text{ is continuous} \\ H(v_i(s_{t+1}), v_i(g(s_t, o_t, a_t))), & i \text{ is discrete} \end{cases}$$
until s_{t+1} *is terminal*

These algorithms were implemented in Python, using the PyTorch library [95] for creating and training ANNs. The project's code is hosted on GitHub[2].

[2] `https://github.com/martinhoT/footsies-agents/tree/checkpoint`

CHAPTER 5

Evaluation

Evaluation of the work through various metrics.

To evaluate the degree to which each component of the system contributes to the agent's performance, we performed ablation tests. Additionally, we varied some components of the system, and evaluated its performance according to our expectations. The reaction time component is evaluated in isolation, in order to assess the rest of the system assuming the reaction time correction is perfect. Table 5.1 summarises all the evaluations that were performed. The hyperparameter values of the algorithms used for evaluation as well as the environment parameters are detailed in Appendix A.3.

Component	Variable	Changes	Page
Actor	Advantage formula	Original versus alternative	67
	Entropy coeffcient	$\beta \in \{0.02, 0.04, 0.08, 0.16\}$	68
	Action masking	Active or inactive	69
Actor-critic	Agent decision skip	Whether to ignore action-values when the agent cannot act	70
	Opponent decision skip	Which opponent action to assume when they cannot act	71
	Discount factor	$\gamma \in \{0.9, 0.99, 1.0\}$, and correct discounted policy gradient	72
	Reward function	Sparse versus dense formulation	73
	Opponent update style	Sarsa, Expected Sarsa, greedy and uniform updates	74
	Special moves	Active or inactive	75
Opponent model	Dynamic loss weights	Active or inactive	76
	Entropy coefficient	$\beta \in \{0.04, 0.08, 0.16, 0.32\}$	77
	Recurrency	Recurrent architecture, different context reset strategies	77
Reaction time	Observation correction	Skip a varying number of time steps per prediction	79
	Game model method	Which prediction method to use	81
General	Comparison with baselines	Compare with PPO, A2C and DQN	85
	Hitstop/blockstop freeze	Whether to ignore or keep	85
	Adaptability	Pre-train agents with opponents, then play with new ones	88
	Consider opponent actions	Whether to consider the opponent's actions at all	90
	Performance with self-play	Performance doing self-play for a long period of time	90
	Transfer learning	Pre-train an action-value function or not, then play from scratch	91

Table 5.1: Evaluation of the proposed solution's components.

In terms of performance metrics, we use win rate, as is frequently used in the literature, and episode length. Winning is the main goal of the agent, but episode length is also important to evaluate especially in the case of *FOOTSIES*, in which the duration of the episodes is unbounded. For each metric, its mean and standard deviation were computed among 10 random seeds (all integers from zero to nine). For all line plots, the line represents the mean and the shaded area represents one standard deviation. Some metrics, such as the policy's entropy, were calculated every 1000 time steps as an average over 5000 test states which were collected before training using a random policy.

5.1 Environment

We evaluated the system on one fighting game: *FOOTSIES*, an open-source game with a separate paid version. *FOOTSIES* is chosen for its great simplicity in capturing the essence of the fighting game genre, having simple state and action spaces. *FOOTSIES*, contrary to most fighting games, does not have multiple playable characters, but does feature special moves that require a much larger sequence of actions. In particular, special moves in *FOOTSIES* are performed by holding the attack action for at least 60 consecutive time steps and then releasing it. During these 60 time steps, the agent can still act by moving, which makes learning these special moves complex. Additionally, it involves multi-step planning: choosing to hold the attack action is dependent on the agent's previous choices of whether to hold it or not. *FOOTSIES* already includes a hand-made stochastic artificial intelligence (AI), against which we perform evaluations.

FightingICE is also an appropriate choice for a testbed, as it is a fighting game purposefully made for the research and development of artificial agents. The game offers multiple playable characters as well as an extensive selection of agents from past competitions against which we can have agents training. We initially planned to evaluate our solution on FightingICE as well, but due to time constraints we decided to focus on one game only.

Interfacing with the environment follows the Gymnasium API [104], utilizing the Python programming language. We created the API from scratch, utilizing sockets for communication between the game and Python. The game modification, as well as the API, are hosted publicly on GitHub[1].

We use structured representations of the games' state as observations for the agent. We consider the standard game variables such as the players' positions, health values and moves that they are performing. More information regarding *FOOTSIES* such as game mechanics is present in Appendix A.1.

FOOTSIES differs from usual fighting games in that it does not feature a time limit for matches. This is important since if we are employing self-play, the agent can get stuck in a loop in which it does not hit or get hit by the opponent, never advancing the match. This occurs because there is no reward or penalization for stalling the match. To counteract this, we impose a time limit to force the environment to truncate after some time. We could then

[1] `https://github.com/martinhoT/Footsies-Gym`

incentivize the agent to not stall, for instance by penalizing it at every time step and thus encouraging it to finish the match as soon as possible, but that would invalidate strategies involving waiting [49]. Additionally, depending on how it is implemented, it could have the undesirable effect of the agent trying to finish the match earlier by purposefully losing. Note that, from the perspective of the agent, truncation is not part of the game; however, we treat truncation as termination of the game when computing win rate, with wins being attributed as they are in fighting games when the time limit is reached, so as to not merely consider truncated games invalid.

To aid in evaluation on *FOOTSIES*, a curriculum of hand-made, deterministic opponents was developed to assess whether the agent is learning correctly. These opponents are used to perform *curriculum learning*, where the agent learns to play against each opponent in the curriculum in sequence and in increasing difficulty. The curriculum is detailed in Appendix A.5. Additionally, a dataset of 1000 episodes between the in-game AI and a human (the author) was created, in which the game model and opponent model can be trained and evaluated. The actor-critic component cannot be learned on the dataset since the reinforcement learning (RL) algorithm employed is not off-policy. For evaluation with the dataset, 90% of it is used for training and the other 10% is for validation, with the split being performed in terms of episodes. The dataset is also shuffled in terms of episodes, and state transitions are not shuffled in order to keep state sequentiality, which would otherwise compromise training of recurrent opponent models. Additionally, to help troubleshoot and analyze the agents, a tool was developed using the DearPyGui toolkit [105]. Details regarding this tool are in Appendix A.6, and its code is hosted on GitHub[2].

5.2 Results

5.2.1 Actor

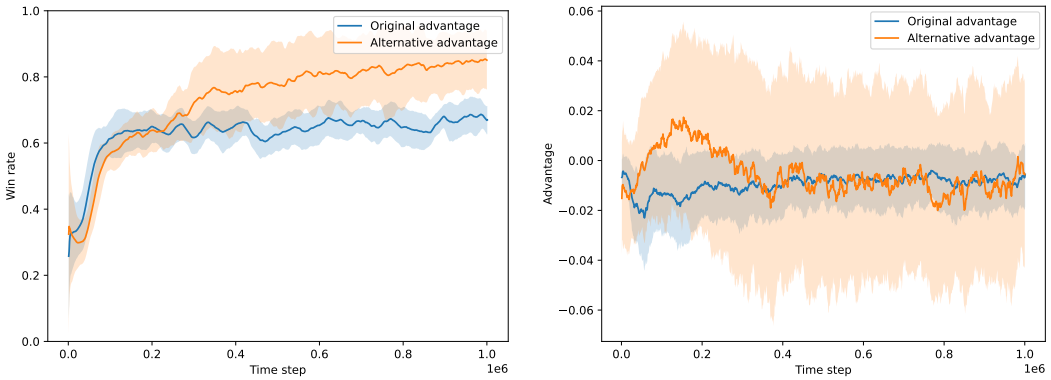
Advantage formula

We derived two different ways for computing the advantage, detailed in Subsection 4.2.2. Figure 5.1a presents the win rate of the agent against the in-game AI of *FOOTSIES*, using the two different methods. From Fig. 5.1a, we can note that using the alternative formula allows the agent to learn much better. This makes sense, as the alternative formula is less dependent on the critic learning the appropriate action values, despite incorporating the opponent model. However, the agent's performance with the alternative formula has higher variance.

In terms of the advantage values themselves, Fig. 5.1b shows that the alternative advantage formula allows for a larger breadth of values, and so more informative updates. This is in contrast with the original formulation, which has less variance. With the original advantage formula, the actor might require a larger learning rate in order to compensate the small advantage values.

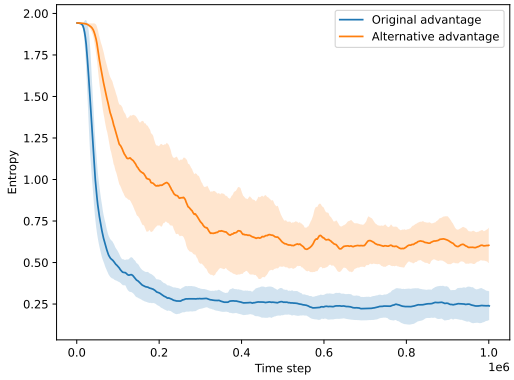
Despite the learning signal being smaller, the policy entropy as shown in Fig. 5.1c ends up being much smaller in the original advantage formulation. This is contrary to our expectations,

[2] `https://github.com/martinhoT/footsies-agents/tree/checkpoint`



(a) Win rate, over the last 100 episodes

(b) Advantage



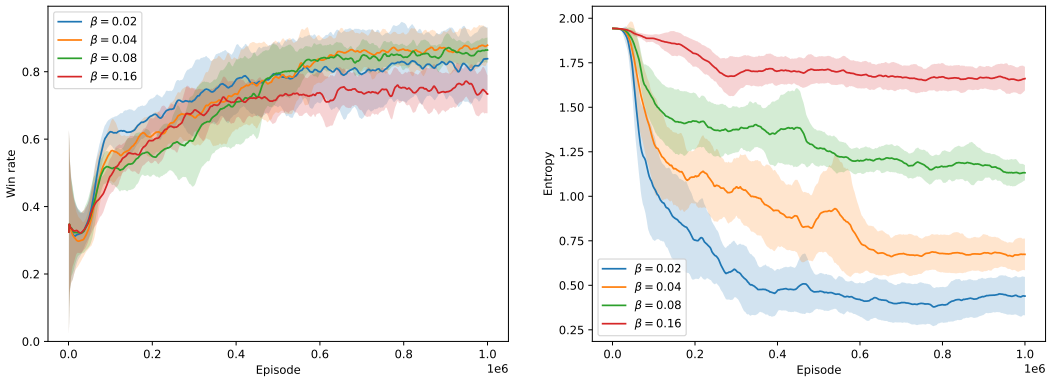
(c) Policy entropy, over 5000 test states

Figure 5.1: Effect of the advantage formula against *FOOTSIES*'s in-game AI.

since we expected that the original formula would have slower learning rather than learn a specific behavior strongly. We believe this is because the agent only has access to the critic's values with the original formula; if these are incorrect, which is to be expected at the beginning of training, then the agent will be trained with consistently wrong and biased signals. On the other hand, the alternative advantage incorporates the reward signal directly from the environment, so the agent has a more stable learning source.

Entropy coefficient

The entropy coefficient indicates how much to care about maximizing the policy's entropy. Figures 5.2a and 5.2b show the win rate and policy entropy respectively when against the in-game AI of *FOOTSIES*. We can note that the policy's entropy does have an effect on the agent's performance, with some degree of entropy maximization being beneficial. Naturally, the policy's entropy is lower the lower the entropy coefficient is. This suggests encouraging the policy to not stay deterministic is important in achieving good performance, as the agent is able to explore the environment better. Too much entropy maximization can be still detrimental as the agent is not able to exploit good actions, so a balance needs to be achieved.

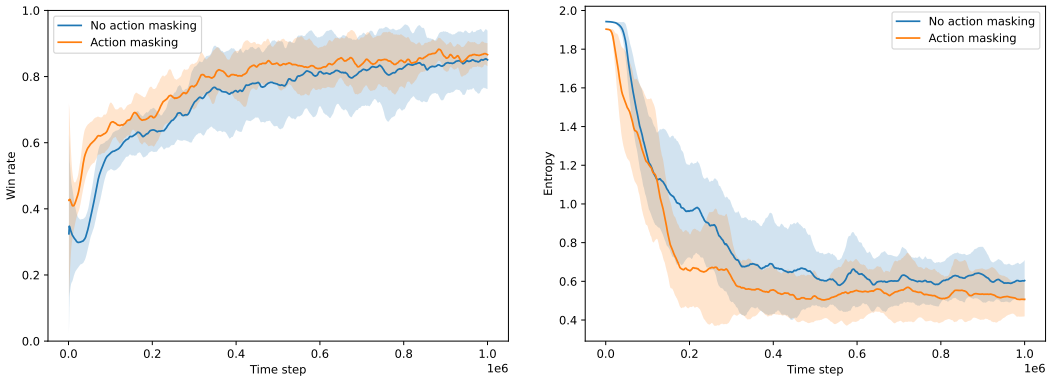


(a) Win rate, over the last 100 episodes **(b)** Policy entropy, over 5000 test states

Figure 5.2: Effect of the entropy coefficient against *FOOTSIES*'s in-game AI.

Action masking

Action masking is the process of explicitly invalidating some actions in the action space depending on the situation, introduced in Subsection 4.2.2. This allows both the agent to not execute those actions, as well as not consider them for learning. In the case of *FOOTSIES*, action masking is only applied at hitstop/blockstop, in which case only two actions are effectively available: not doing anything or attacking. Not doing anything is performed by sending `STAND` to the environment, and attacking by sending `N_ATTACK`. We expected action masking would help the agent learn better, since we reduce the number of actions it needs to explore. Figure 5.3a presents the agent's win rate depending on action masking against the in-game AI of *FOOTSIES*. Action masking brought a slight performance improvement, in addition to reducing the agent's win rate variance. As such, we can have more confidence that the agent achieves greater win rate values. In addition to the win rate, we also evaluate the policy's entropy, shown in Fig. 5.3b. There is not a large difference in entropy, although action masking expectedly causes lower entropy values since during hitstop/blockstop only two actions are considered, and so the distribution's entropy will be smaller.



(a) Win rate, over the last 100 episodes **(b)** Policy entropy, over 5000 test states

Figure 5.3: Effect of action masking against *FOOTSIES*'s in-game AI.

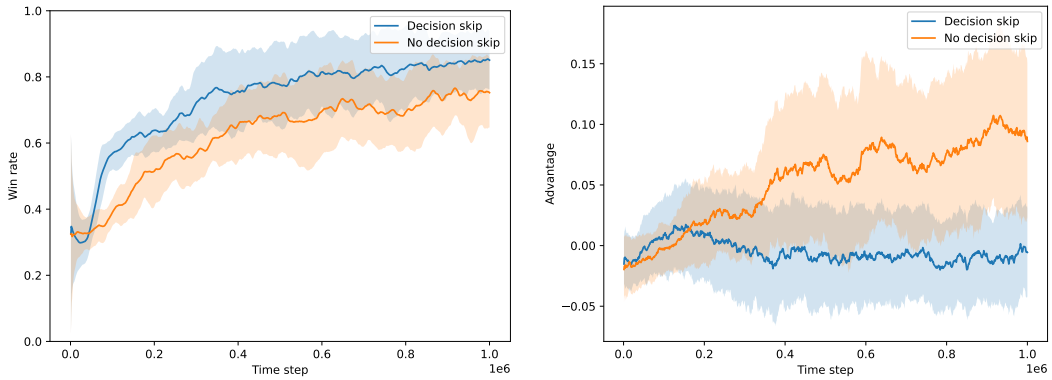
5.2.2 Actor-critic

Agent decision skip

As explained in Subsection 4.2.4, there are moments in gameplay where the agent cannot act, whose action-values we can choose to ignore during training. We denote the skipping of these time steps as "decision skipping". Figure 5.4a shows the win rate against the in-game AI when skipping and not skipping inconsequential decisions. Decision skipping helps attain good performance in this scenario. Credit assignment is harder without decision skipping, i.e. it is harder to attribute reward to the relevant actions.

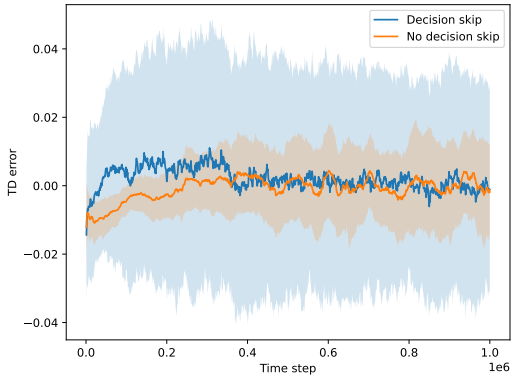
Assume, for example, that the agent performs an attack at state s_t which ends at state s_{t+22}. That attack hits the opponent, which under the dense reward scheme means the agent got positive reward, but it was at a time step $t + 6$. This means that, to properly credit the action that was performed at state s_t, the critic needs to propagate action-values from the time $t + 6$ back to s_t. Since we are performing one-step temporal difference (TD) updates, i.e. we only consider the immediate next state s_{t+1} during the updates, it will take plenty of updates to properly update the action-value at state s_t. This contrasts with the case in which we have decision skipping. With decision skipping, the time at which reward is given between t and $t + 22$ does not matter; that reward is attributed to the transition (s_t, s_{t+22}), and so the action performed at s_t is credited immediately. Conceptually, this makes sense, since having action-values for states where actions cannot be performed is not useful, unless we want diagnostics at every state.

Figures 5.4b and 5.4c show the advantage values and TD error when learning the actor and critic respectively. Despite achieving higher performance, decision skipping had lower advantage values, whose mean stays around zero, whereas no decision skipping had increasing values. Additionally, the TD error had lower variance when not decision skipping, but the same mean. When not decision skipping, the action-value function is updated much more frequently, e.g. in the previous example it would be updated 21 times since there were 21 transitions, whereas with decision skipping only one transition effectively occurred. As such, since the states are very similar, each update has similar targets and updates are frequently performed, it is natural that the error will be smaller. When manually analyzing the action-value function output, we noticed that when decision skipping the action-values for all actions are high, being generally at or above 0.3, whereas when not decision skipping the values are mostly near zero. However, when not decision skipping, the action-values of rewarding actions, such as attacks, contrast heavily with the action-values of other actions, which results in a large advantage. This difference in the magnitude of action-values might be because action-value propagation *between* actions is easier when decision skipping, whereas when not decision skipping many more time steps need to be propagated through. For instance, in the action string `DASH_FORWARD` and `N_ATTACK`, there are effectively only three states: the neutral state, the state when `DASH_FORWARD` finishes, and the state when `N_ATTACK` finishes. As such, there are only two transitions that need propagating and therefore two action-value updates, contrasting with the usual case which would require more than 30 updates.



(a) Win rate, over the last 100 episodes

(b) Advantage



(c) TD error

Figure 5.4: Effect of decision skipping against *FOOTSIES*'s in-game AI.

Opponent decision skip

There are situations in which the opponent cannot act. Since we are explicitly considering the opponent's actions in many of the agent's components, we need to define which action to assume in these periods. As mentioned in Subsection 4.2.4, we defined three strategies: consider them to be doing no action in particular, and so updates consider all actions simultaneously, to be doing the last valid action and to be doing the action of doing nothing. Figure 5.5 shows the difference between the three strategies in terms of win rate against the in-game AI. The strategies are rather different performance-wise, with the strategy of considering no action in particular being the most performant and stable. This is because, in practice, this strategy comes close to not considering the opponent's actions at all, evaluated in Subsection 5.2.5. Through manual analysis, we find the action-values and the policy end up being very similar independently of the opponent's actions. This is in contrast with the other two strategies: for the one that considers the last valid action, we have updates distributed among all opponent actions, whereas when considering only the no-op action we notice that that single action gets disproportionately more updates compared to the others. These results indicate that not considering an opponent action explicitly is better for learning.

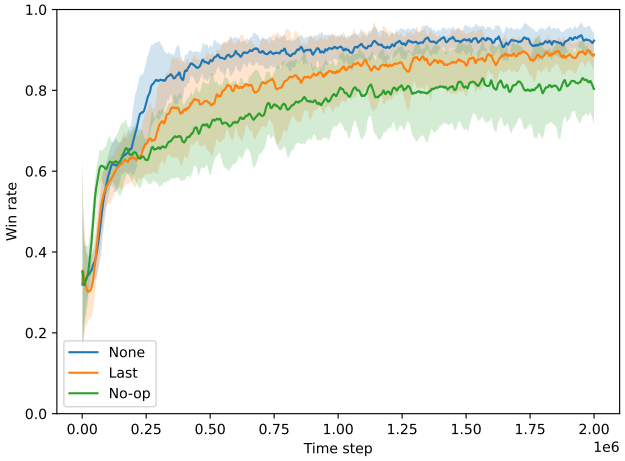


Figure 5.5: Effect of the assumed opponent action when they cannot act on the agent's win rate in the last 100 episodes against *FOOTSIES*'s in-game AI.

Discount factor

The discount factor dictates how far-sighted the agent is when computing the returns for its actions. For a discount factor $\gamma = 1.0$, the agent considers an infinitely large horizon, whereas for $\gamma = 0.0$ the agent only considers the next immediate reward. We considered $\gamma = 1.0$ to be an appropriate value since in our case the agent should always care to maximize the reward until the end end of the episode, which encodes whether the agent wins or loses. We compare using maximum gamma with two other values that are slightly lower $\gamma \in \{0.99, 0.9\}$.

Additionally, as briefly mentioned in Subsection 2.1.3, the policy gradient implementations commonly seen in practice are not theoretically correct, since they do not consider discounting fully. We determine the effect of considering a theoretically correct implementation of the policy gradient with discounting.

Figure 5.6 presents the effect of the discount factor on the agent's performance against the in-game AI in *FOOTSIES*. The run where $\gamma = 1.0$ and the correct discounted policy gradient is used is exactly equal to the one with the incorrect discounted policy gradient since discounting is not performed in either of them, and so it is omitted. Different values of γ impact performance heavily, and we can note that some amount of discounting $\gamma = 0.99$ is more performant. If we consider the correct discounted policy gradient formula, the agent's performance diminishes drastically, not being able to learn at all once $\gamma = 0.9$. This goes in line with what was discussed in Subsection 2.1.3, where in the original policy gradient objective we are interested in increasing the returns of the starting state s_0, and so future states are increasingly less important and thus its updates are also much smaller. Still, the agent is able to achieve good performance for $\gamma = 0.99$. This is because episodes, on average, took about 100 time steps to complete (around 2 seconds), which means that the discount would still be significant at state s_{100} since $\gamma^{100} \approx 0.37$. On the other hand, with $\gamma = 0.9$ we have that the state discount at $t = 100$ becomes $\gamma^{100} \approx 2.66 \times 10^{-5}$, which essentially nullifies the updates. The agent is therefore only able to succeed with $\gamma = 0.99$ because episodes can end within a reasonable number of time steps.

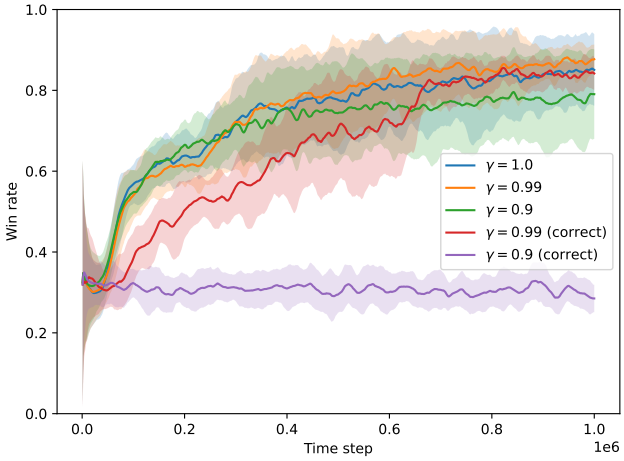
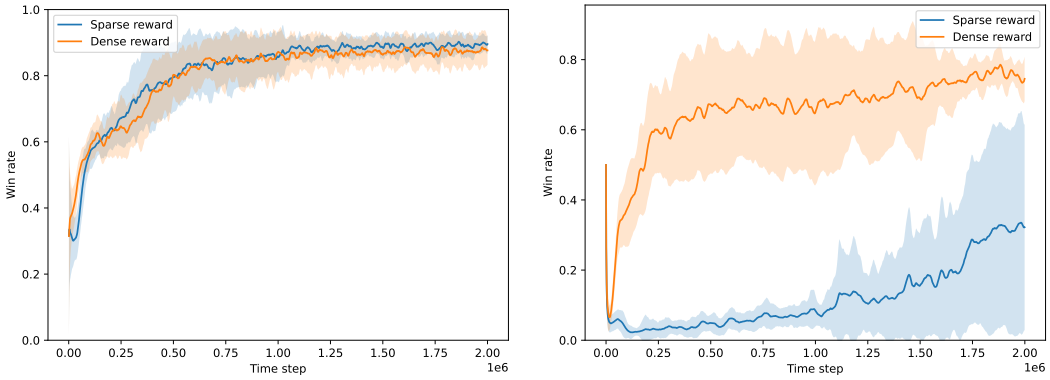


Figure 5.6: Effect of the discount factor and the correct discounted formulation on the agent's win rate over the last 100 episodes against *FOOTSIES*'s in-game AI.

Reward function

In Subsection 4.1.4, we defined two reward schemes: a sparse and a dense scheme. The sparse scheme rewards the agent on win or loss, whereas the dense scheme also rewards on hits. We expected that, since the dense reward scheme provides the agent with more reward signal, it should be able to learn the task faster. Figure 5.7a shows the win rate against the in-game AI with the two schemes. The dense reward scheme does not have a discernible impact on performance against the in-game AI. However, dense reward still proves to be useful in some scenarios. As shown in Fig. 5.7b, the agent is able to learn against `WhiffPunisher` much better with dense reward. This is because against `WhiffPunisher` the agent always needs to carefully diminish the opponent's health to zero before it can win [3]. As such, dense reward is able to guide the agent toward the goal, since it rewards the agent for damaging the opponent.



(a) *FOOTSIES*'s in-game AI **(b)** `WhiffPunisher`

Figure 5.7: Effect of the different reward schemes on the agent's win rate in the last 100 episodes against different opponents.

[3] It is not strictly necessary to diminish the opponent's health to zero in order to win in *FOOTSIES*. More details in Appendix A.1.

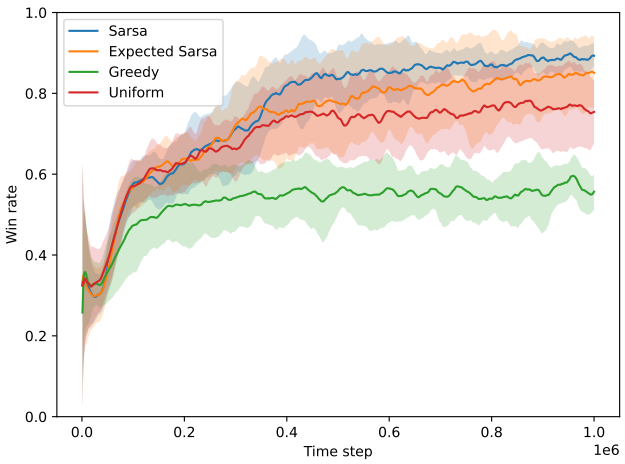
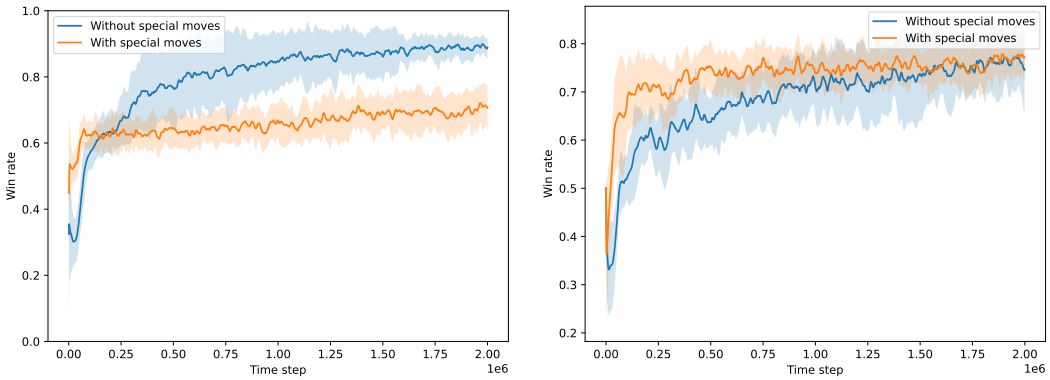


Figure 5.8: Effect of the opponent update style on the agent's win rate in the last 100 episodes against *FOOTSIES*'s in-game AI.

Opponent update style

When calculating future action-values, we can have different assumptions for how the opponent will behave. The most straightforward assumption is to assume the opponent to be the current one. As such, we can perform action-value updates considering samples of what the opponent actually performs in the future (Sarsa-like update) or use the opponent action distribution, such as an opponent model, directly (Expected Sarsa update). But we can also consider other behaviors. We experiment with two other behaviors: a greedy one, where the opponent chooses the action that minimizes our action-values, and a uniform one to serve as control.

Figure 5.8 presents the win rate against the in-game AI when using these four update styles. Surprisingly, assuming the opponent to perform any action yields performance comparable to Expected Sarsa updates, even though a uniform distribution is not at all representative of the current opponent. This might be because using a uniform policy removes the agent's fixation on which specific action the opponent is going to perform next, which in Subsection 5.2.5 we found to be more performant. Also, assuming the opponent to be greedy yields lower performance, without it increasing noticeably during training. This makes sense, since assuming the opponent to be greedy would not necessarily make the agent perform well against the current opponent, but would rather make the agent robust against exploits. Overall, the Sarsa-like updates are the most performant. As such, it is better to simply not use the opponent model to predict what the opponent is going to perform next, and just wait for their next action. Therefore, we can note that introducing an opponent model for action-value calculation was not beneficial to the agent, and that simple Sarsa-like updates should suffice. The only case in which an opponent model would be useful is in offline learning, where an opponent model is learned independently and then used to generate experience, but for the case of online learning it only increases complexity and decreases task performance.



(a) No pre-training **(b)** Pre-trained against `WhiffPunisher`

Figure 5.9: Effect of special moves on the agent's win rate in the last 100 episodes against *FOOTSIES*'s in-game AI.

Special moves

During development, we noticed that the agent tends to develop a large bias in utilizing the special moves instead of other actions, getting into a loop of always utilizing those moves and not learning more useful behavior. This led us to remove these same special moves in order to see if the agent is still capable of learning appropriately. Figure 5.9a shows the win rate against the in-game AI with and without the inclusion of special moves in the agent's action space. Noticeably, the agent's performance using special moves plateaus quickly. On the other hand, if not using special moves, the agent is able to achieve greater success over time, without converging soon.

This is perplexing, as we expected that given the agent's action space is smaller, it should either have the same performance as the full action space or worse, since it has fewer tools to achieve its task. However, we notice the opposite. We believe this is due to the worse exploration that the agent performs when it has special moves available. With special moves, the agent quickly finds a strategy that, at least in the *FOOTSIES* game, is able to win the game and thus get immediate reward: to constantly use special moves. However, this is largely because the in-game AI can be exploited by these moves, which will fail if performed against an opponent expecting them. Because the agent finds an easy strategy and is not incentivized to explore other strategies, which are relatively harder to find, it gets stuck having the same behavior, exploiting what it knows best.

In order to determine if the agent, even with access to special moves, is able to learn how to play appropriately, we pre-train the agent against the `WhiffPunisher` curriculum opponent, which consistently exploits the agent's tendency to do those moves frequently. Figure 5.9b shows the win rate against the in-game AI after pre-training for one million time steps against `WhiffPunisher`. Surprisingly, not only was considering special moves successful, but it also allowed the agent to converge in its performance very quickly. This suggests that `WhiffPunisher` is a suitable learning opponent since it makes the agent's behavior more robust. After some more training, the agent is able to achieve roughly the same performance when

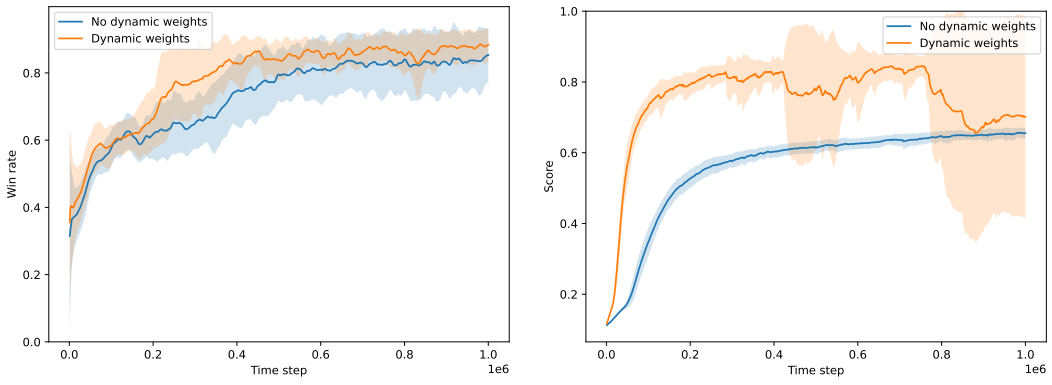
not using special moves. This indicates that the problems previously found with including special moves are mainly exploration issues.

We may interpret the agent's initial behavior of constantly attacking as the "attacking preemptively" strategy of Fig. 2.7. When the agent trains against `WhiffPunisher`, it is forced to stop adopting this behavior, which is exploitable by "wait and punish". Ideally, the agent then learns the "approach and attack" strategy[4], but we found that the agent is prone to stalling against this opponent. This is because the agent is not incentivized to engage.

5.2.3 Opponent model

Dynamic loss weights

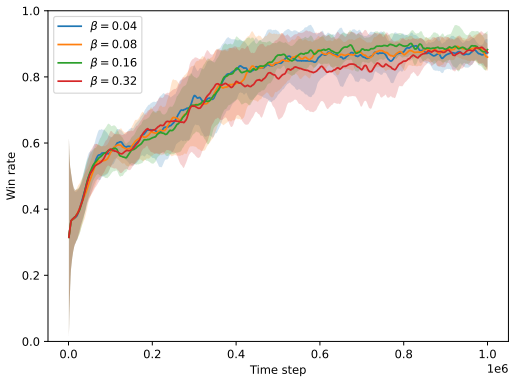
Some actions are performed much more frequently than others; yet, it does not mean that they are more important to predict. As explained in Subsection 4.2.1, we introduced an adjustment that tries to compensate for the fact that some actions, despite being important, are executed less frequently. In Fig. 5.10a, we can see that this adjustment seems to aid the agent in early stages of learning, and ends up slightly improving the agent's performance in the long term. The main question, however, is whether the model is actually predicting the opponent's actions better. Figure 5.10b presents the model's prediction score, which is equal to the mean of the model's assigned probability to the opponent action actually performed in 5000 test states. We can notice that the adjustment does improve prediction performance in early stages of training, but it becomes extremely unstable in later stages. The adjustment heavily increases the updates of infrequent actions, therefore increasing their gradient, potentially causing unstable updates. Still, the parameters's values were not extremely large, and were in a range not much different than when dynamic weights were not used. Nevertheless, this adjustment has hyperparameters of its own which need to be adjusted, but tuning was left out since the adjustment proved to not increase performance in early rounds of experimentation.



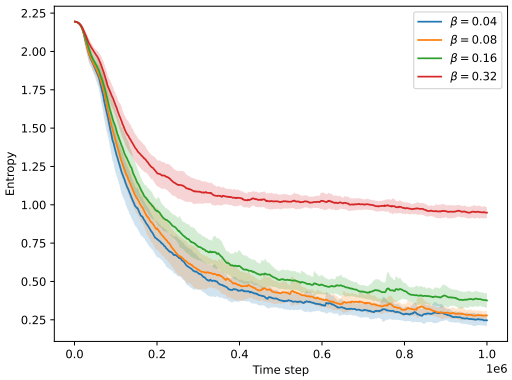
(a) Win rate, over the last 100 episodes **(b)** Opponent model score, over 5000 test states

Figure 5.10: Effect of dynamic loss weights against *FOOTSIES*'s in-game AI, with dense reward.

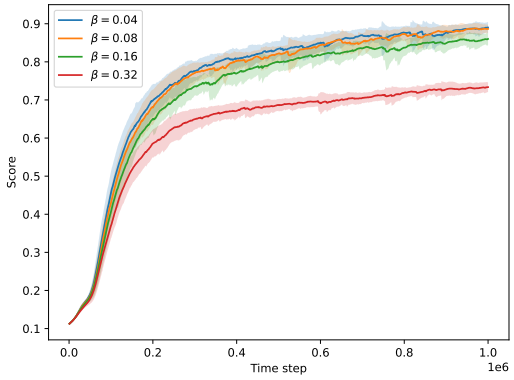
[4]This strategy is only valid if the opponent needs to react, but here the opponent reacts instantaneously. Nevertheless, the strategy of approaching and attacking is still the correct behavior against `WhiffPunisher`.



(a) Win rate, over the last 100 episodes



(b) Opponent model entropy, over 5000 test states



(c) Opponent model score, over 5000 test states

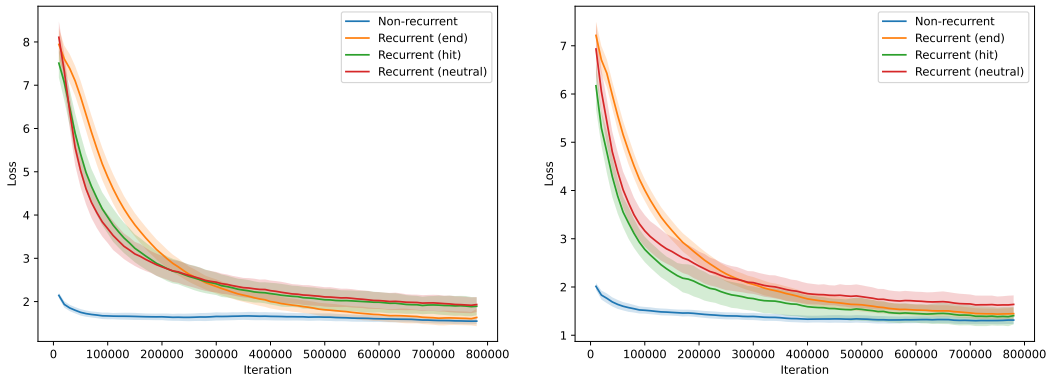
Figure 5.11: Effect of the opponent model entropy coefficient against *FOOTSIES*'s in-game AI, with dense reward.

Entropy coefficient

Similarly to the agent's policy, we also introduce an entropy coefficient to the opponent model, to avoid it becoming deterministic and allow some plasticity. In terms of task performance, Fig. 5.11a shows the coefficient does not have a considerable effect against the in-game AI. Still, a large value for the coefficient has the potential to be detrimental. Figure 5.11b shows the effect of the coefficient on the model's entropy, with the entropy being higher for higher coefficients, as expected. The prediction score as shown in Fig. 5.11c suggests that the entropy coefficient is inversely proportional to the score. This makes sense, as the entropy coefficient purposefully attempts to make the model less confident of its predictions. We can also note that ensuring some amount of entropy does not greatly hamper the model's prediction capability.

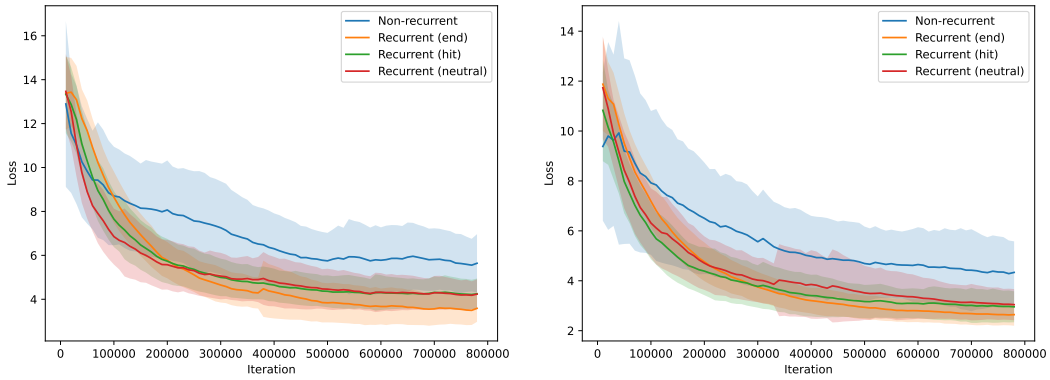
Recurrency

Since the opponent model is meant to learn the behavior of an arbitrary opponent, it ideally needs to consider the past when predicting the opponent's actions, i.e. have some kind of memory. This is possible by using recurrent neural networks (RNNs), and we evaluate their contribution in predicting the opponent's next action. Additionally, RNNs have a hidden state, which may need to be reset depending on which context we want to consider. In



(a) Loss for player 1 **(b)** Loss for player 2

Figure 5.12: Effect of opponent model recurrency and context reset strategies on the model's training loss on the dataset, for each player.

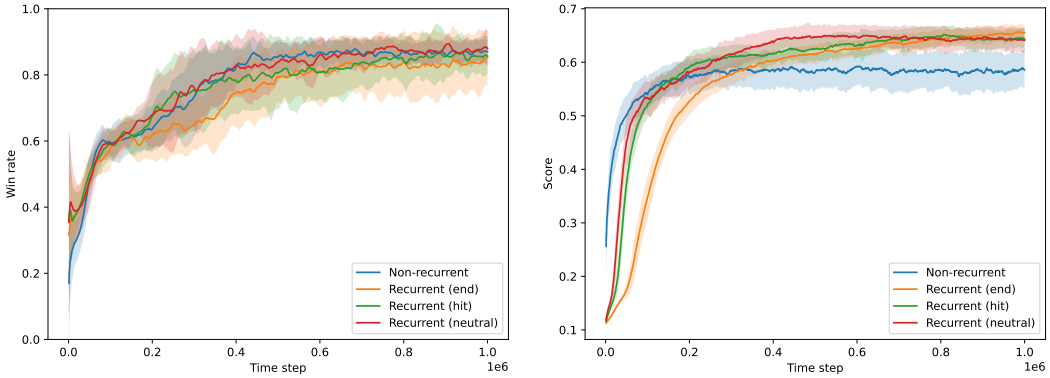


(a) Loss for player 1 **(b)** Loss for player 2

Figure 5.13: Effect of opponent model recurrency and context reset strategies on the model's validation loss on the dataset, for each player.

Subsection 4.2.1 we detail three context reset strategies: reset only at the end of the game ("end"), whenever a player is hit ("hit"), and whenever a player is hit or both players are too close to one another ("neutral").

We first evaluate the different strategies on the dataset. Figures 5.12 and 5.13 present the training and validation loss, respectively, of the different strategies for an opponent model learned for player 1, the in-game AI, and another for player 2, the author. Analyzing training loss, the recurrent models seem to learn slower than the non-recurrent one, due to their increased complexity and the fact that they are updated less frequently, only after the context is reset. Additionally, the recurrent models could not achieve a loss as low as the non-recurrent model's. On the other hand, through the validation loss we can actually note that the recurrent opponent models are more accurate than the non-recurrent model, generalizing better to unseen experience. The task of predicting player 2 seemed easier than that of predicting player 1, as the loss is lower overall when predicting the former. However, loss is not indicative of task performance when it is integrated into the agent, since some predictions may be more useful than others.



(a) Win rate, over the last 100 episodes **(b)** Opponent model score, over 5000 test states

Figure 5.14: Effect of opponent model recurrency and context reset strategies against *FOOTSIES*'s in-game AI, with dense reward.

As shown in Fig. 5.14a, the opponent model architecture did not end up having much of an impact on performance against the in-game AI. Of note, the "end" and "hit" context reset strategies learned only slightly worse than the other configurations. There are noticeable differences in terms of prediction score however. Figure 5.14b presents the model's prediction score. The recurrent models learn slower, but are able to make better predictions after some time. Additionally, the different context reset strategies do not seem to have much difference among them in terms of final prediction performance. The recurrent model with the "end" context reset strategy learns slower because its updates are less frequent. Still, the training complexity and imperceptible task performance increase put into question whether a recurrent opponent model is worth using.

5.2.4 Reaction time

Observation correction

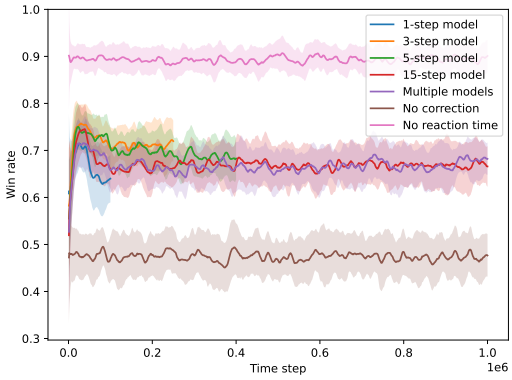
For the observation correction, we evaluated two different methods. One is using a single model for predicting the future: if we want to predict n time steps ahead and the model predicts $k \leq n$ time steps ahead then we chain $\lfloor \frac{n}{k} \rfloor$ consecutive predictions, i.e. one prediction is the input to the next. The other is utilizing multiple models, each responsible for predicting a fixed number k of time steps into the future: if we want to predict n steps ahead then we choose the model that predicts closest to n steps and perform a single prediction. The former method sounds theoretically more correct especially if we use a one-step model, but when performing approximation in practice any error in the model's predictions compounds quickly. Additionally, there is a large performance impact in using small step models, since it requires more predictions to fulfill the correction. The latter method is much more performant at inference time, since we always perform a single prediction only, but can be seen as wasteful since we are learning multiple separate models whose tasks are very similar. Additionally, although we save in inference, we sacrifice learning speed since we are learning multiple models. Nevertheless, in practice we notice that the decrease in learning speed is small and so the trade-off is worth it.

To evaluate the observation correction method, we use an agent pre-trained against the in-game AI, but without the game model. Then, we freeze all learned components (the actor, critic and opponent model) and learn blank game models using the methodologies previously described, while playing against the in-game AI. Additionally, the agent can act at any time; without reaction time, we could infer whether the agent could skip decision making by checking whether their action would have an effect on the current state, but here we allow the agent to act anyway since corrected observations are not perfect and so we should not manually make that inference. Reaction time was tuned by changing the temperature of the policy and opponent model distributions to 0.2, which decreases their entropy and essentially makes the agent more sure of its decisions. We found this produced appropriate reaction time values in practice. The configurations with single model prediction had to be truncated since they took considerably longer to run when compared with the other configurations. The multiple models agent utilized three models that predicted $t \in \{15, 20, 25\}$ time steps ahead. Figure 5.15 presents different metrics collected while learning the game model under reaction time. In terms of performance, shown in Fig. 5.15a, we found that any method of observation correction was able to maintain a fraction of the pre-trained agent's win rate. Still, we can note that the short-step models were able to achieve slightly greater win rate at the beginning of training, perhaps because they are more accurate.

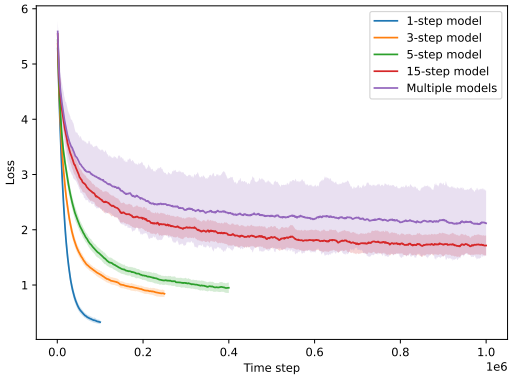
Figure 5.15b presents the loss of the game model(s) while training. As would be expected, the loss is larger as the step size increases, which means the predictions are less accurate. This is because with larger step sizes the game model also has to implicitly form a model of the agent's and opponent's behavior during those skipped steps, making the task harder.

Figure 5.15c shows the reaction time in number of frames. Reaction time is only dependent on the policy and opponent model, which are frozen, and so it tends to be constant. Still, despite appearing to be the case, reaction time is not constant during play, as was noted in manual analysis. In fact, reaction time can spontaneously go above 20 frames. This may be because reaction time is computed on the previous *corrected* observation, and so the game model influences the observations the reaction time is computed for. Reaction time is also dependent on the *states* the agent encounters, as it will increase in states where the agent has greater uncertainty, either over the policy, the opponent model or both.

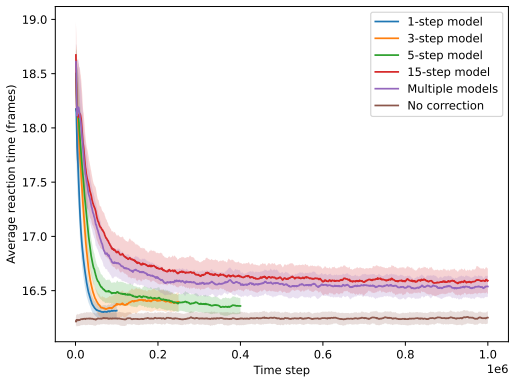
Finally, we also evaluate the computational performance during inference, i.e. when choosing an action given a delayed state. Figure 5.15d shows the distribution of time it took to make a decision, in CPU time. Inference time appears inversely proportional to the step size of single models, with the constant of proportionality depending on the reaction time itself, disregarding overhead not pertaining to observation correction. The hard limit for inference time is 16 milliseconds, which is the time used to process a frame in fighting games. One is inclined to think that one-step models are enough for running in real-time, but this evaluation does not take into account the time budget required for online learning of all four components as well as the inter-process communication time between the agent and the game. For that reason, the multiple models strategy seems to be the most appropriate, since its predictions are accurate enough and it allows constant-time inference, not depending on reaction time.



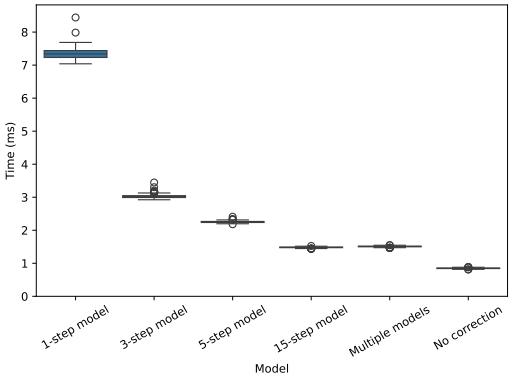
(a) Win rate, over the last 100 episodes



(b) Game model loss. The multiple models loss was divided by the number of models



(c) Reaction time



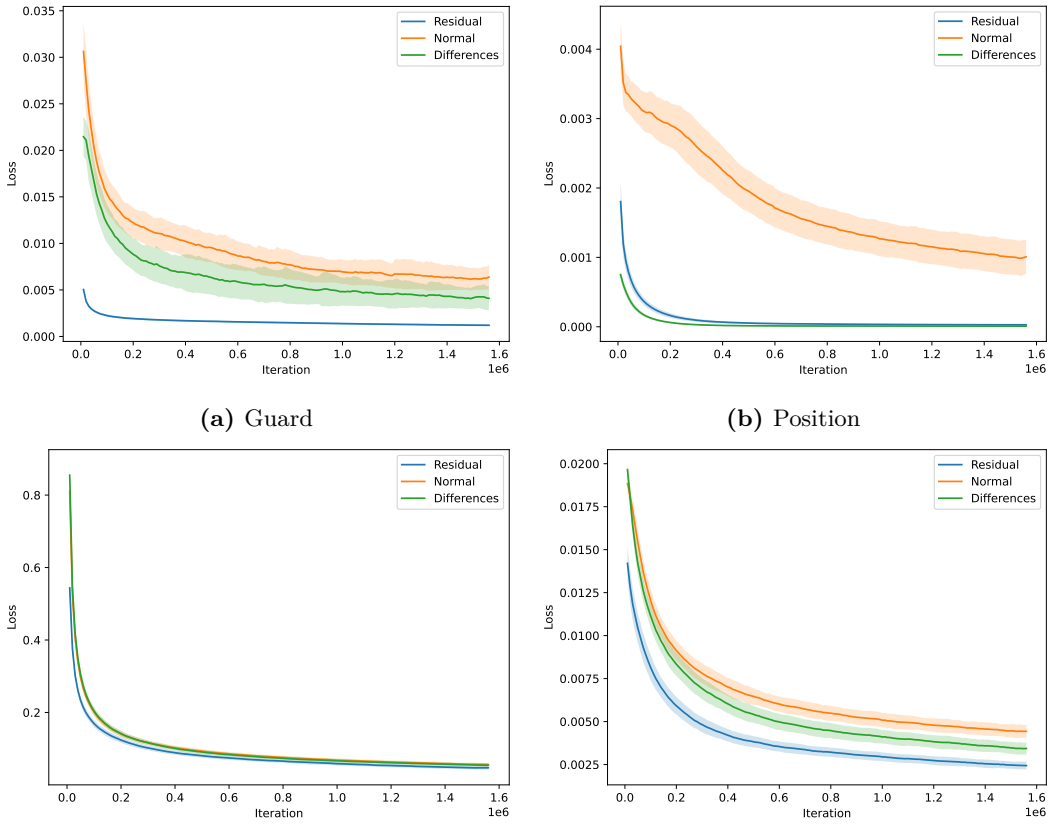
(d) Inference time distribution, in CPU time in milliseconds

Figure 5.15: Agent metrics while playing against *FOOTSIES*'s in-game AI with reaction time. The agent was pre-trained against the in-game AI for 1M time steps, and then only the game model was learned.

Game model method

The game model is responsible for approximating the function $g(s_t, a_t, o_t)$, which outputs a prediction for the next state s_{t+1}. We evaluated three different ways to make this prediction, detailed in Subsection 4.2.1. The most simple one is to predict the next state in the same structure as the input state. The other one is to predict, for the continuous state variables, not the absolute value but the difference from one state to the next. Finally, we also experiment with the architecture proposed in [75], which is more elaborate by including three separate models in a residual-like architecture, one of which focuses on the difference between state variables like the second method.

Figures 5.16 and 5.17 present the game models' training loss and validation loss, respectively, for each state variable when they trained on the dataset. We can note that predicting state differences aids in prediction of all continuous state variables. This makes sense since these variables change linearly in time, i.e. they change in constant increments. As such, if the player is walking forward or backward, then the model merely needs to predict a constant. This is the same for the move progress variable, but it is more dependent on the current move that



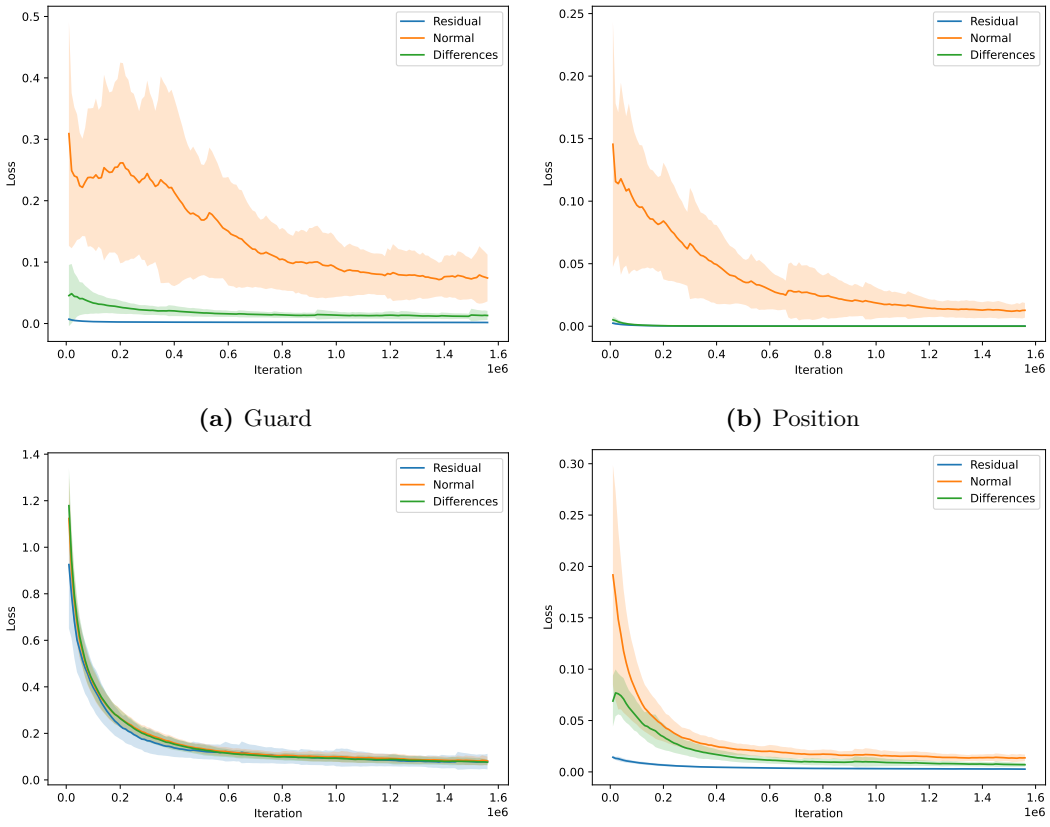
(a) Guard **(b)** Position

(c) Move **(d)** Move progress

Figure 5.16: One-step game model training loss in the dataset for each state variable.

is being performed. The guard variable is more complicated, as the model needs to identify situations with specific moves and move progress values. Overall, however, the residual-like architecture is the most performant. This is because it not only predicts differences, but it also permits adding new information completely unrelated to the previous state, which can help predicting the player's move. It is also considerably more performant in predicting the guard variable, perhaps due to the added flexibility in the architecture. It is worth noting however that it involves conjoining three separate models, so it is more computationally complex and thus takes longer to train. In fact, we found this strategy was more unstable for this exact reason, and required a smaller learning rate.

We need to keep in mind however that game model loss is not necessarily indicative of performance when the model is used by the agent. Figure 5.18 presents the agent's win rate against the in-game AI, when subject to the reaction time constraint with observation correction using the different strategies. The same setup was used as the one for observation correction explained in pag. 79. We can note that, in terms of performance, the methods are not very distinct from one another, despite there being slight differences in prediction quality. Figure 5.19 presents the game model's loss for each state variable. The residual-like architecture has the least loss overall, but the difference is only perceptible for the move variable. Still, we can notice that the residual-like architecture of [75] is more successful in



(a) Guard **(b)** Position

(c) Move **(d)** Move progress

Figure 5.17: One-step game model validation loss in the dataset for each state variable.

FOOTSIES than the simpler approaches.

An important factor worth noting is the scale of the variables' loss. Each variable uses a different loss function according to its type. The mean squared error is smaller than the cross-entropy loss by around two orders of magnitude. This has an impact on the gradients, being much larger for the move variable, which is discrete, than for the continuous ones. At the beginning of training, gradients for the discrete variable are around five orders of magnitude larger than the gradients for the others. Incidentally, the move variable is very important to model correctly when compared to the others, but care should still be taken when mixing different loss gradients. Still, it is not immediately obvious how scaling should be performed, as the cross-entropy loss is unbounded. Perhaps normalization of the gradients using running estimates of their mean and standard deviation can be performed, or prediction of different state variables can be performed by separate models.

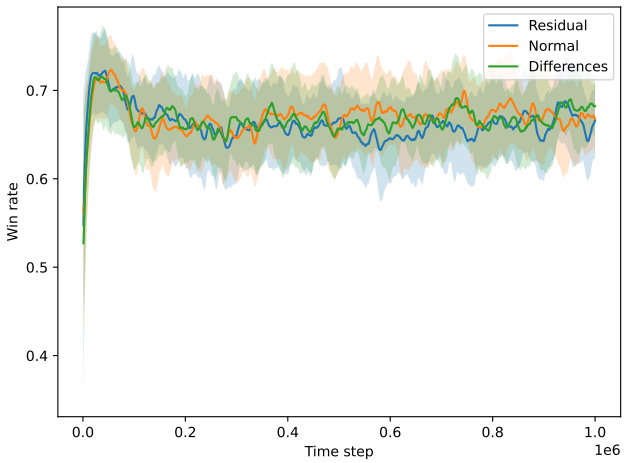
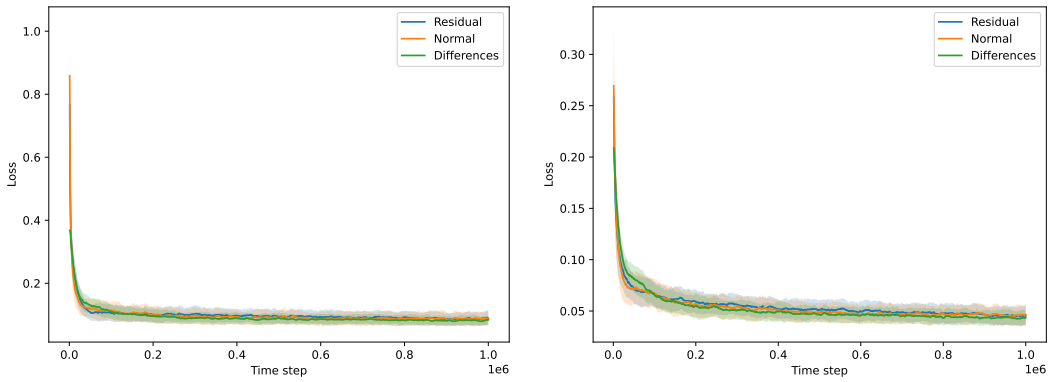


Figure 5.18: Effect of the game model method on the agent's win rate in the last 100 episodes against *FOOTSIES*'s in-game AI, with reaction time. The agent was pre-trained against the in-game AI for 1M time steps, and then only the game model was learned.



(a) Guard

(b) Position

(c) Move

(d) Move progress

Figure 5.19: Game model loss while playing against *FOOTSIES*'s in-game AI for each state variable, with reaction time. The agent was pre-trained against the in-game AI for 1M time steps, and then only the game model was learned. Note that the loss is larger than on the dataset since three separate models are used for correcting reaction time, and the loss presented corresponds to the sum of each model's loss.

5.2.5 General

Comparison with baselines

To validate whether the algorithm is learning appropriately, we compare its performance with baseline implementations of common RL algorithms for discrete action spaces. We compare our method with Advantage Actor Critic (A2C), Proximal Policy Optimization (PPO) and Deep Q-Network (DQN) implementations in the Stable Baselines3 module [106], of which PPO is state of the art for discrete action spaces. Figure 5.20 compares the tuned parameterization of our algorithm with the three tuned baselines, in terms of win rate over the last 100 episodes against the in-game AI, with dense reward. The A2C algorithm proved difficult to tune well, ending up losing more often than winning, and DQN did not have improved performance over time. Our algorithm got slightly better performance in the end than PPO against the in-game AI, although PPO was able to learn slightly more quickly. This learning speed can be attributed to PPO's greater sample efficiency, since it learns using the same batches of experience multiple times, whereas our algorithm learns purely online, using each example only once.

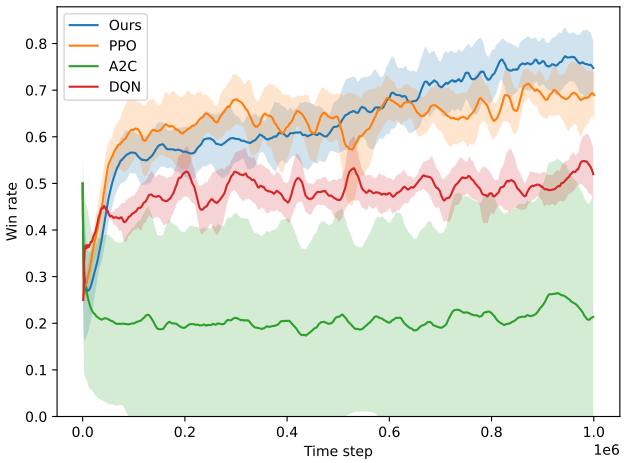


Figure 5.20: Performance between the agent and baseline RL algorithms, in terms of win rate of the last 100 episodes against *FOOTSIES*'s in-game AI.

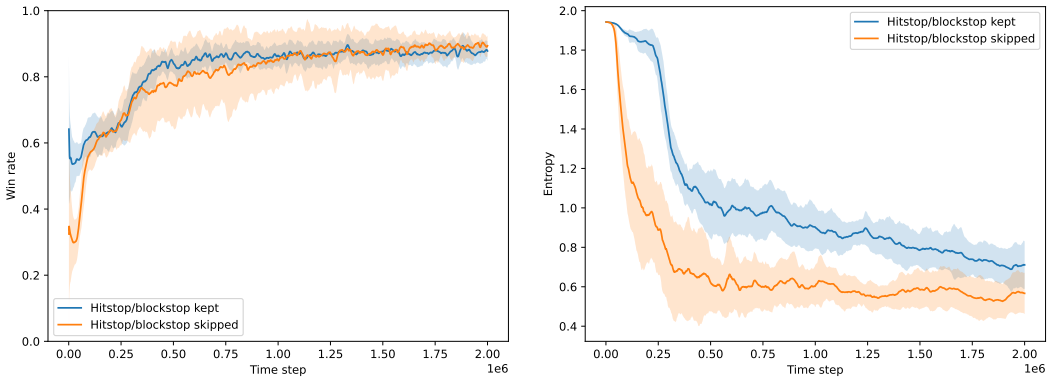
Hitstop/blockstop freeze

As discussed in Subsection 4.2.4, hitstop and blockstop are periods where the game freezes momentarily after an attack hits or is blocked, respectively. If we treat these freezing periods as transitions, there is no possibility for the agent to know when the hitstop/blockstop will end since there is no information in the state about its duration. Including hitstop/blockstop information in the state increases its dimensionality, while only being useful information in specific moments of gameplay. As such, we skip the hitstop/blockstop freeze altogether, and evaluate whether it is worthwhile to do so. In Fig. 5.21a, we can note that keeping hitstop/blockstop freeze allows the agent to achieve better performance more quickly and stably against the in-game AI. Ignoring the freeze had greater performance variance initially because three of the seeds had much lower performance than expected in the first half of

training. Nevertheless, whether the freeze is ignored or not did not have a large impact in the end. From Fig. 5.21b, we can note that the policy entropy is different between the two modes, likely attributed to the fact that credit assignment during hitstop/blockstop is incorrectly handled when the freeze is not skipped. Any attack action performed during hitstop/blockstop will schedule a special move, as mentioned in A.1. If these periods are kept, then the very last action performed during those periods will be the one that is credited if that special move is rewarded, and not the attack action that actually scheduled that move. As such, the updates will be more distributed among the different actions, keeping the entropy high.

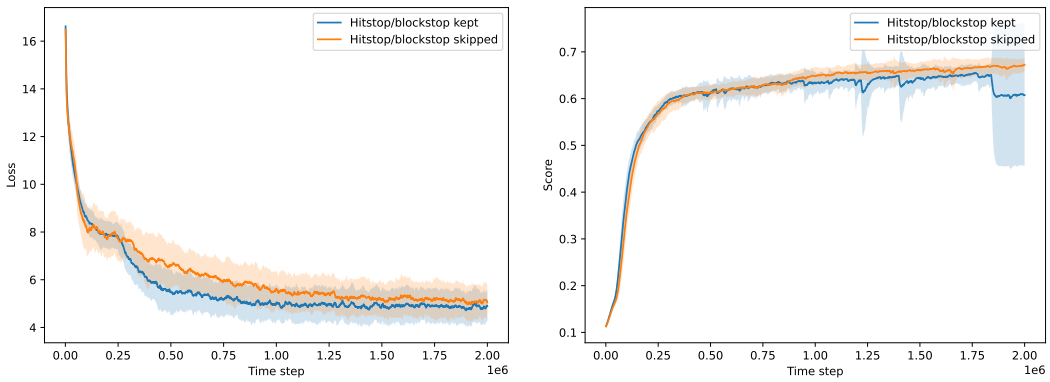
Ignoring hitstop/blockstop also affects the opponent model and game model. In Fig. 5.21c we can note that the game model is able to achieve slightly lower loss when these periods are kept. This goes contrary to our expectations, as it is impossible for the game model to determine when these periods end since no information about hitstop/blockstop is given in the state, and so it should be harder to make predictions. This is likely because the loss is still being evaluated during these periods, and since the model that keeps the freeze is still learning, it is able to improve its predictions repeatedly. Also, during these periods the task is easier, as the model merely needs to approximate the identity function. The opponent model on the other hand has only slightly better performance and lower variance when ignoring the freeze, as shown in its prediction score in Fig. 5.21d. The opponent model's loss is omitted for brevity since it is inversely correlated with the score, with a Pearson correlation coefficient lower than -0.99. Keeping the freeze will cause the opponent model to perform multiple updates during hitstop/blockstop, with these updates being the same since the game is frozen. These updates might be disruptive to the model, as it is trained on the same examples multiple times. In fact, the score and loss have large variance at the end of training because in one of the seeds the model got unstable and regressed, suddenly having score lower than 0.2. For the opponent model, we can note that ignoring the freeze mainly aids in stability.

Ignoring hitstop/blockstop is still highly beneficial in scenarios where decision making during these periods is important. One such case is when playing against `WhiffPunisher`, where the agent needs to actively avoid scheduling special moves during hitstop/blockstop unless the opponent has no health. If hitstop/blockstop is kept, then the agent is at a greater risk of accidentally performing an attack during those periods since actions are being constantly sampled. As shown in Fig. 5.22, ignoring these periods allows the agent to be successful. The variance for when hitstop/blockstop is skipped is very high since only some runs were able to achieve high win rate, with many still being unsuccessful. These results were obtained using the sparse reward scheme, which as evaluated in Subsection 5.2.2 proved to be insufficient against this particular opponent, as the agent needs plenty of luck in order to receive positive reward.



(a) Win rate, over the last 100 episodes

(b) Policy entropy, over 5000 test states



(c) Game model loss

(d) Opponent model score, over 5000 test states

Figure 5.21: Effect of ignoring hitstop/blockstop freeze against *FOOTSIES*'s in-game AI.

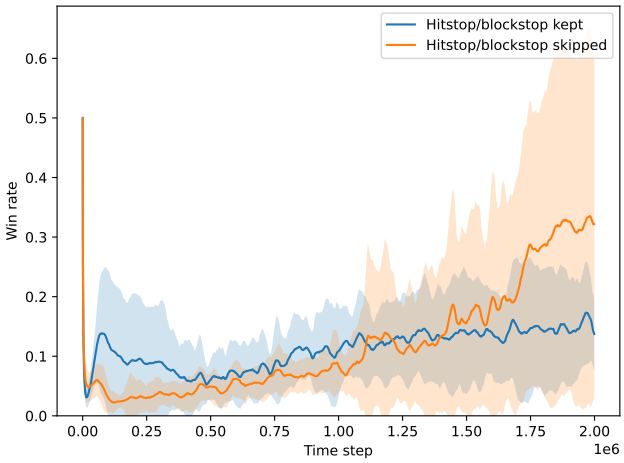


Figure 5.22: Effect of ignoring hitstop/blockstop freeze on the agent's win rate in the last 100 episodes against `WhiffPunisher`.

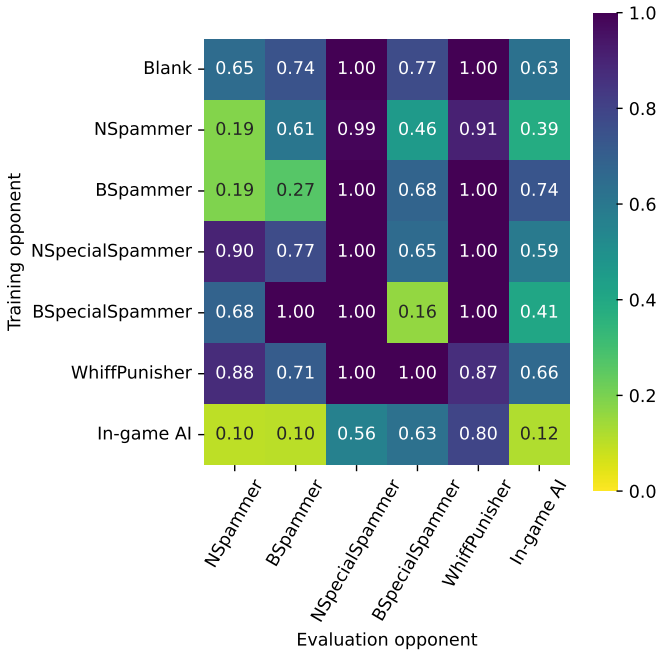
Adaptability

One of the requirements is to have quick adaptability to new opponents. The agent should be able to, within a reasonable number of episodes, achieve satisfactory performance against an opponent that acts differently than the one it was trained against. To evaluate adaptability, we have the agent train against one specific opponent for one million time steps, and then train against another opponent for another million time steps, on which the agent's performance is evaluated. We utilize the in-game AI as well as the curriculum opponents.

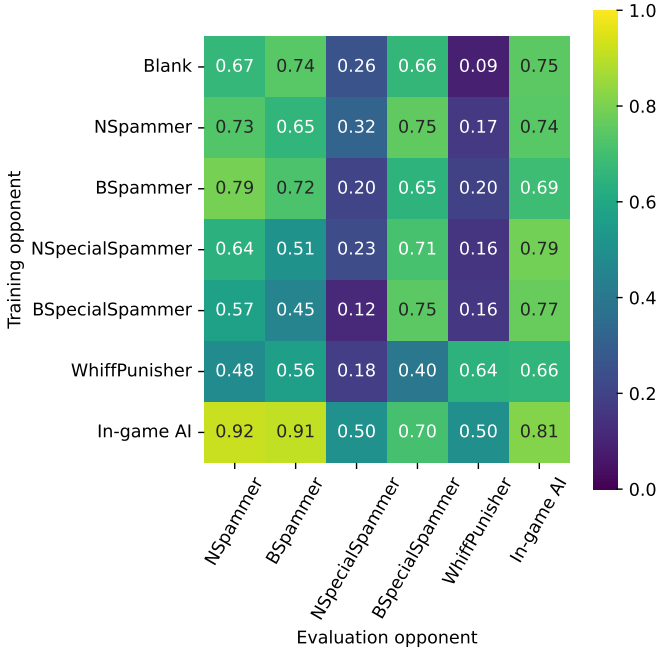
Figure 5.23a presents the time it took for the agent to achieve 80% win rate over the last 100 episodes on the second opponent, in terms of the fraction of training required[5]. We also include an agent without any pre-training for comparison. If the fraction is 1.0, then the agent was not able to achieve that threshold. We define 80% win rate as the threshold since it requires the agent to adapt well to the opponent, and is not likely to be achieved due to luck. The threshold can only be surpassed after the 10000th time step. From the matrix, we can note that the agent had a large difficulty adapting to certain opponents, mainly `NSpecialSpammer` and `WhiffPunisher`, even when pre-training against them. The in-game AI also proves to be useful as a training opponent, allowing the best adaptation to the other opponents. This is because the in-game AI incorporates plenty of stochasticity, allowing the agent to experience diverse game situations, whereas the curriculum opponents are deterministic and meant to have the agent learn specific game situations. The inverse can be seen in `WhiffPunisher`, which worsens adaptation to opponents other than itself, when compared with the case in which the agent did not pre-train at all. This is because `WhiffPunisher` punishes overly aggressive behavior, which otherwise works against the other opponents, and so forces a very deliberate playstyle on the agent. This demonstrates the need to expose the agent to a wide range of opponents, but also that training opponents can heavily bias the agent into behaviors that make it harder to adapt to future opponents.

The agent could not adapt to many of the configurations, so we also evaluate the agent's final win rate in Fig. 5.23b, from which we can draw more conclusions. Firstly, the training opponents `NSpammer`, `BSpammer`, `NSpecialSpammer` and `BSpecialSpammer` were not proper for adaptation to `WhiffPunisher`. This is because an aggressive playstyle works sufficiently well for these four opponents, which is the behavior the agent tends to gravitate toward by default. On the other hand, `WhiffPunisher` requires the agent to be more passive. The in-game AI proves to be a suitable pre-training opponent because it can occasionally punish overly aggressive behavior. This will avoid learning an almost deterministic policy that chooses actions that are bad against `WhiffPunisher`, from which it would be hard to escape. We can also notice some performance regression if we consider cases in which the agent surpassed the 80% win rate threshold, mainly with `BSpecialSpammer` as an evaluation opponent. Some seeds, despite achieving the threshold, eventually degenerated into an underperforming strategy close to or below 50% win rate. Performance regression was a problem we frequently experienced, and was often associated with the agent suddenly collapsing to a deterministic policy.

[5]The adaptability results were run with the first six seeds, as they took too long to compute for all 10 seeds.



(a) Fraction of 1M time steps needed to achieve 80% win rate



(b) Win rate over the last 100 episodes after 1M time steps

Figure 5.23: Adaptation speed against the in-game AI and the curriculum opponents.

In summary, the agent takes very long to adapt to other opponents. Still, we can note the capability of the in-game AI as a good pre-training opponent due to the fact the agent is able to experience varied game situations, only failing to expose the agent to those where overly aggressive behavior is consistently exploited.

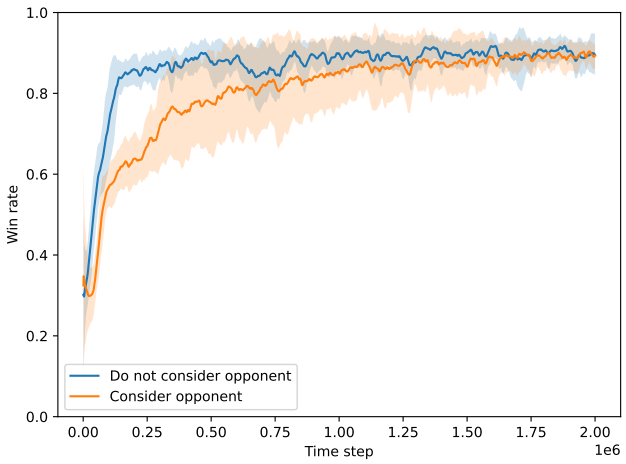


Figure 5.24: Effect of considering the opponent's immediate next action, in terms of win rate of the last 100 episodes against *FOOTSIES*'s in-game AI.

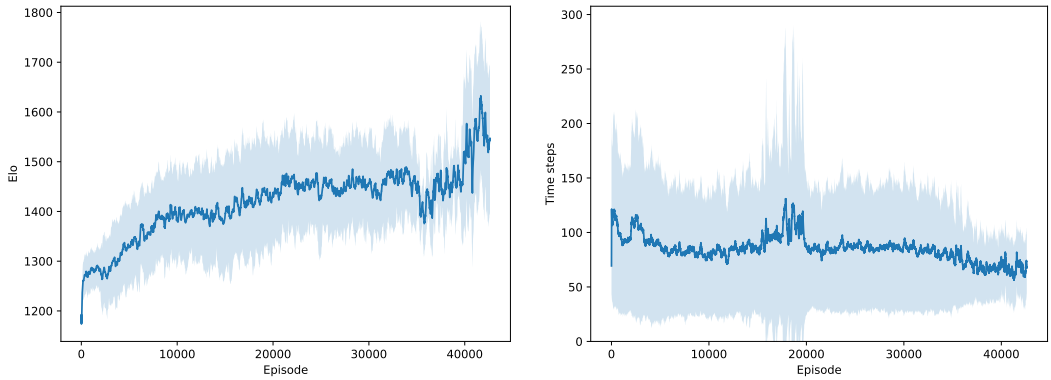
Consider opponent actions

There is a major question that needs to be asked: should we really consider the opponent's future actions explicitly? This amounts to essentially ingraining the environment dynamics into both the policy and action-value functions by considering the opponent's next action along with the state. In Subsection 4.1.1 we noted that it not only integrates well with the reaction time emulator but it also has the potential to allow better adaptation to different opponent behaviors. It should also make decision making more transparent, since we can see what the agent would do against each future action of the opponent. However, we risk making the problem more complex. In our algorithm, we can stop considering the opponent's actions simply by assuming their action space to only consist of one action, which is always performed. Figure 5.24 presents the agent's win rate considering and not considering the opponent's next action. We can notice that the agent learns much quicker when not considering the opponent's actions, even though the same peformance is achieved in the long term. Contrary to our expectations, considering the opponent's next action explicitly brings much more complexity to the problem, hampering the agent's learning.

The way we consider the opponent's next action is by having the agent's policy and action-value functions output a matrix, with a row for each opponent action and a column for each agent action. This resulted approximately in a 52% increase in the number of parameters for the policy, from 6983 parameters to 10623, and a 33% increase for the action-value function, from 22151 parameters to 29375. Added to the increase in the number of parameters, the updates are also dispersed between the opponent's different actions, instead of having updates focused on a single policy distribution or action-value row, which should make learning slow.

Performance with self-play

We experimented with training the agent using self-play, after pre-training on the curriculum with dense reward. During self-play, we take copies/snapshots of the agent while learning, and set them up to be future opponents against the agent. Basically, the agent plays



(a) Elo **(b)** Episode length

Figure 5.25: Agent metrics during self-play, after pre-training on the curriculum.

against itself. Precise details on the self-play strategy adopted are present in Appendix A.4. Figure 5.25a presents the agent's Elo during self-play, while using the dense reward scheme. We utilize Elo, a performance metric often used for chess, since it is appropriate for two-player zero-sum games. We can note that Elo does increase over time, indicating that the agent is increasing its overall strength, at least within the pool of opponents that was generated including the in-game AI. However, the plateau from the 20000th episode to the 30000th also suggests that the agent can have trouble improving its behavior.

During self-play, it is also important to evaluate how long it takes the agent to complete episodes. Since the agent is playing against itself, it can be prone to stalling, as there is nothing discouraging the agent from doing that strategy if engaging with the opponent does not prove to be fruitful. However, as can be seen in Fig. 5.25b, this was generally not the case, with episodes terminating within a short number of time steps, and even decreasing in later stages of self-play. Only one seed suffered from stalling, causing the episode length spike at around the 20000th episode. This suggests the agent has adopted more aggressive behavior, and by visual analysis of the produced agents that is the case. This means that the agent can still be exploited by more deliberate opponents; the agent is not robust, and an opponent such as the `WhiffPunisher` is able to win it often. Therefore, it is important to constantly expose the agent to opponents that, although are not very successful by themselves, are very capable of exploiting bad habits of the agent. This is precisely the purpose of "exploiter" agents in [58]. Note that the agent was exposed to `WhiffPunisher` in the curriculum, before applying self-play. However, it is not enough to ensure the agent will keep being performant against `WhiffPunisher`.

Transfer learning

We expect that using an agent that was already trained to play against some opponent, or at least a similar one, should be able to adapt quicker to a new one. Figure 5.26 compares the win rate over time of three agents: two using a pre-trained action-value function and another using a randomly initialized one (using the default initialization in PyTorch). The setup was as follows: two reference agents were trained against the in-game AI for one million

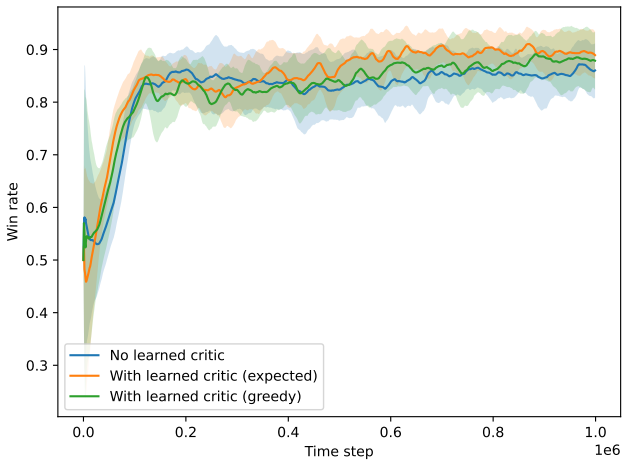


Figure 5.26: Difference in adaptation speed when using a pre-trained action-value function, in terms of win rate of the last 100 episodes against a self-play agent trained with the seed 0, with dense reward.

time steps, from which the action-value function was extracted. These reference agents were different in the opponent update style they assumed, with one using Expected Sarsa, and thus learning specifically for the in-game AI, and another assuming a greedy opponent. Then, three blank agents were initialized, but two of them with the previous action-value function of each reference agent, and were afterward trained to play against an agent trained with self-play with the seed 0. This self-play agent was trained using the regime evaluated in pag. 90, and was used as an opponent because it was the agent with the most training and was not deterministic like the curriculum opponents. The purpose of this evaluation is to check whether an action-value function trained on the in-game AI can be useful for adaptation to a different opponent. Note that the action-value function is still allowed to be adapted. From Fig. 5.26, we can notice that using a learned action-value function did not increase the learning speed. It did, however, allow slightly better performance in the long term, more specifically the function learned with Expected Sarsa updates. Contrary to what was expected, the action-value function learned assuming a greedy opponent was not very helpful. This could be due to the fact that the opponent acts similarly to the in-game AI or that it is easily exploitable, making differences in the learned action-value function not very meaningful.

5.3 Discussion

The results indicate that, overall, the adjustments implemented to tackle the specific domain of fighting games allow the agent to learn better by simplifying the task, but they are not strictly necessary to have good performance. Even then, with these adjustments the agent roughly matches the general purpose algorithm PPO, although without re-using training examples. In terms of parameters, we can note that always enforcing some entropy on both the policy and opponent model is beneficial.

In terms of opponent modeling, we can note that it did not help the agent in achieving the task. Not considering the opponent's immediate next action simplifies the task, and allows

the agent to learn much faster. Throughout the different adjustments and assumptions, we noticed that any configurations that most closely resembled indifference toward the agent's future action were the most successful. In fact, the best hyperparameterization of the agent found through tuning incentivizes the opponent model to have high entropy by having a high entropy coefficient. This is indicative of either a faulty opponent model, that enforcing level 1 reasoning on the agent is not useful, or that the opponent model is too myopic and only considers the immediate next action. Also, the opponent model only models low-level actions and not high-level strategies, which can constitute more useful predictions for the agent.

Since explicit opponent modeling does not aid the agent, how can we measure its uncertainty of the opponent without an explicit opponent model? Having an opponent model isolated from the agent's decision-making process does not make much sense conceptually. This suggests that one possibility is to disregard the opponent model completely and use a more general environment predictor. Instead of having the game model and opponent model separately, we have an environment model in charge of implicitly predicting both the game's dynamics and the opponent's actions. The agent's reaction time would then be a function of the environment model's uncertainty. However, it is unclear how uncertainty could be measured from environment model predictions. This environment model would be used merely to determine the reaction time and correct it.

Still, tackling reaction time proved to be more difficult than anticipated. Correcting the state delay while being performant and not spending too much time at inference proved to be difficult. Multi-step prediction models are the most computationally performant since fewer predictions are needed, without much loss in task performance. This is because in each state transition in fighting games very little information tends to change. Even with these predictive models however, the agent is not able to achieve as much performance as when reaction time is not used. The evaluation performed in Subsection 5.2.4 is also limited in the sense that win rate against the in-game AI is not necessarily indicative of good performance, as it is only a single opponent, and also in that the game model is learned in isolation. Learning all components at the same time is a more realistic configuration, which was not evaluated due to technical issues that were not resolved in time, pertaining mainly to how actions are chosen by the actor in corrected observations which are then analyzed when learning the critic.

In terms of learning in general, we can note that the opponents used for training are important. Ensuring there are opponents that are capable of exploiting bad habits of the agent, such as `WhiffPunisher`, is key to having a performant and *robust* agent. For instance, the agent can only effectively have special moves in its action space if it can pre-train with `WhiffPunisher`, in order to learn how to use them properly. Also, including a dense reward scheme did not strictly improve performance, but allowed much faster learning for the agent and avoided stalling, which is especially useful against some opponents such as `WhiffPunisher`. Still, adaptation between opponents takes a long time, and some opponents still prove to be much more difficult to learn than others, despite being deterministic and thus predictable in their behavior. As such, added to the imperfect handling of reaction time, the agent is not competent enough for human play.

CHAPTER 6

Conclusions and future work

Summary of the main points, and avenues for further exploration of the problem.

6.1 Conclusions

Artificial agents have the potential to benefit video games by serving as a learning tool for new players and by providing automatic feedback on game balancing. In this work, we addressed the fighting game genre, and explored a reaction time constraint mimicking that of human beings in order to ensure fair play, as it is an often ignored factor. This reaction time is dependent on the agent's uncertainty of the opponent's future actions. We utilized reinforcement learning (RL) to create an agent capable of autonomous learning, more specifically using the actor-critic algorithm, as well as supervised learning (SL) for learning predictors of the opponent and the game's dynamics. Each component was built using function approximation through artificial neural networks (ANNs). Reaction time delays the state provided to the agent, and correction of the delay was performed by having the agent build the current state from the delayed state, using the learned game dynamics model. Since reaction time is dependent on the opponent predictor, we also incorporated it into the RL algorithm.

We explored our method on the open-source game *FOOTSIES*, building a Gymnasium API for it. We found that explicitly considering the opponent in the RL algorithm is not fruitful and heavily complicates the problem for the agent. Other adjustments that were performed to appropriate the algorithm to the specific domain of fighting games improved learning speed, but most were not strictly necessary to have good performance. For correcting reaction time, multiple one-shot models targeting different reaction time values was the most computationally performant strategy, without significant loss in task performance. Considering performance of the agent, we found that the choice of opponents is exceedingly important, corroborating the conclusions taken from the literature review. The agent trained through self-play was still susceptible to being exploited by some simple hand-made opponents, suggesting cycles in strategy space were being followed.

Overall, we can note that the dynamic reaction time emulator needs to be rethinked. Utilizing an explicit, short-sighted opponent model does not constitute opponent modeling that is actually useful for the agent. There is an opportunity to heavily simplify the policy, critic and reaction time emulation architectures by not considering the opponent explicitly. Instead, we may experiment with modulating reaction time through other metrics such as general environment model prediction uncertainty. Other than reaction time, more importance needs to be put on the reward function and which opponents the agent is exposed to during training. We found the dense reward formulation necessary in having the agent learn the hand-made curriculum, and specific opponents are able to instill heavy strategy shifts in the agent, who otherwise would not be incentivized to act in any way other than aggressively. In summary, we should strive for heavy simplification of the RL algorithm, and focus instead on environment tuning in order to build a competent agent.

6.2 Limitations and future work

6.2.1 Planning

Due to time constraints, the solution was not evaluated on FightingICE. We opted for *FOOTSIES* mainly for its simplicity, to easily explore our method, and to build a Gymnasium API for it which otherwise would not have. FightingICE is also an appropriate testbed, with more complex state and action spaces. However, the integration of FightingICE with Python was still not perfect with gRPC communication being a recent feature, and its Gym API was outdated, so some tweaking was still required to interface with it. Still, FightingICE had the advantage of offering a vast variety of opponents, collected through the years in competition. In *FOOTSIES*, evaluation of the agent's performance was limited due to the lack of evaluation opponents, so it had to be done with the in-game artificial intelligence (AI), the hand-made curriculum and through self-play, with the latter two not being ideal since the curriculum's agents are deterministic and self-play is prone to generating non-diverse opponents. As reiterated in Chapter 3, evaluation against a large variety of opponents is required to properly assess an agent's performance. Additionally, FightingICE comes closer to the complexity of fighting games in various features, such as freedom of movement, action space and character variety.

Although work already exists corroborating the idea that, in fighting games, an agent learning to play as one character can generalize to learning to play as other characters [2], and similarly for playing *against* different characters [43], we still set out to test it in our work. However, since analysis became restricted to the *FOOTSIES* game due to limited time, we could not evaluate it since the game does not offer more than one character. As such, transfer learning was reduced to adaptation to different opponents.

Added to the difficulties in successfully implementing an RL algorithm for our problem, there were also times in which performance regression was noted after making some adjustments or additions. Unfortunately, this performance regression was often only observed later, making

it hard to find the source of the problem. In the future, automated testing should be performed in order to catch issues in time, which are hard to find in RL.

6.2.2 Simplifications

The action space was simplified by including special moves as actions, to avoid the need of keeping memory for performing specific primitive action sequences. Utilizing the primitive action space, similar to how human beings play, is left to be analyzed. For this, a recurrent neural network (RNN), a convolutional neural network (CNN) like in [43] or attention mechanisms [107] may be used. Even with a simplified action space, the policy may need to be recurrent for better emulation of human behavior.

To implement a fair agent in modern fighting games, it may also be necessary to use the visual display as the environment state rather than a structured representation as we used. This is because visual ambiguity and occlusion, although not very frequent, can be important factors in these games, which is awkward to model in a structured state representation.

6.2.3 Reward

Another avenue worth exploring is a different reward formulation. In this work we considered the environment to be episodic, but we may also consider it to be continuous as in [101]. It is intuitive to consider each round/match in a fighting game independently and thus consider each round/match to be an episode. However, it is also valid to consider a continuous environment, and it may perhaps even be a better formulation. In fighting games, a human player usually plays multiple consecutive matches against another one in what is commonly called a "set", rather than play a single match. This is on purpose, to allow players to adapt to each other's strategies which requires some time. In a similar vein, matches in competitive rock-paper-scissors are not decided with a single match, but with multiple matches to allow adaptation [27]. Taking this into account, it might not be entirely correct to isolate each match. As such, we can utilize the continuous setting, in which we formulate a different RL objective: instead of maximizing the expected return, the agent maximizes the *average reward* per time step. Since the reward scheme we used rewards based on wins and losses, this means the agent is directly maximizing the metric which we are interested in: win rate. Therefore, it might be more appropriate to consider decision making to persist between rounds and matches, rather than be isolated, and use a continuous formulation of the problem. Additionally, including reward discounting in the continuous case has no effect on the objective [11, Section 10.3]. The policy gradient for continuous settings is potentially better formulated for our use in practice, since the policy objective $J(\theta)$ directly encodes our objective of maximizing win rate whereas the policy objective used in this work represents the value of the starting state of an episode, which is short-sighted.

It is also worth exploring different reward schemes. The two we experimented with are very straightforward, but we may want to instill different behaviors in the agent, extraneous to winning. For instance, we can encourage the agent to frequently approach the opponent in order to avoid stalling, which was notable against `WhiffPunisher`. This kind of incentivization can even be found in the *Guilty Gear* series as a game mechanic. In general, we may use more

intricate reward schemes to instill specific motivations in the agent, which have the potential to instill more human-like behaviors.

There are also reward scheme formulations useful for exploring. One problem we found in this work is the difficulty the agent has in efficiently exploring the environment. We found the need to include a curriculum of opponents to force the agent to learn the game appropriately. We can therefore experiment with reward shaping or intrinsic reward schemes focused on exploration [108], [109], such as curiosity [85], where the agent tries to reduce the uncertainty of their environment model, or novelty-based ones, where the agent tries to reach states it rarely visits [110].

Another useful form of intrinsic reward can be found in the MaxEnt RL framework, which lends itself theoretically well to the development of agents for fighting games, and may be studied in the future. This is because MaxEnt RL focuses on creating agents that not only achieve a given task but that do so *robustly*. That is, agents are resistant to perturbances in their environment that may compromise their performance. This is done by simply modifying the RL objective by including the entropy of the policy π as a form of intrinsic reward. Maximizing the entropy of the agent's policy jointly with reward allows the creation of robust agents [111], which makes sense for our problem since acting unpredictably in fighting games is particularly important. Additionally, it also aids in exploration since it discourages the agent from becoming deterministic. Note that including the entropy in the RL objective is different from including it in the policy's objective as we did with the entropy coefficient. Including the entropy in the RL objective encourages the agent to maintain high action entropy over *trajectories*, and not necessarily to keep high action entropy for *every single state*. Ensuring robustness can be useful in mitigating odd behaviors in the agent when facing unknown scenarios, which were noted in practice [2]. However, encouraging the policy to always maintain some randomness also has an impact on this work's formulation of reaction time, so if that constraint is to be included it should be tuned appropriately. Also, another hyperparameter is introduced, which dictates how much to care about the entropy of the policy compared to the reward. This poses a robustness-exploitation trade-off, which was discussed in Subsection 2.2.1 in the context of deviation from the Nash equilibrium, i.e. the most robust/unexploitable strategy, in order to exploit the current opponent.

6.2.4 Other approaches

One big problem of our solution is that it does not employ temporal abstraction. This problem is especially apparent since we are explicitly modeling the opponent's course of action, which is very short-sighted: we only consider which action the opponent is going to perform in this *instant*, and not consider whether they are going to perform some action in the near future. To overcome this inflexibility in how the agent reasons over extended periods of time, hierarchical reinforcement learning (HRL) approaches such as [112] may be studied. This is especially important in the *FOOTSIES* game since the action of performing a special move requires holding the attack button for 60 consecutive frames, during which the agent can still move back and forth. As such, while the agent has already decided to perform this

special move, it is also deciding how to move until it is performed, or it may even decide to quit performing the move altogether, as long as the attack button has not been held for the required duration. In our work, since performing a special move is an isolated action, the agent simply holds the attack button without making decisions during this period, making them considerably more vulnerable.

For environment modeling, which was solely used to correct the reaction time delay, we utilized ANNs that were tasked with the prediction of a specific number n of time steps ahead. If we wanted to make a prediction for m steps ahead, what we would do is chain k predictions together such that $kn = m$. This methodology of chaining small predictions is flawed however, as it only works in case we have a perfect environment model. In the case of an imperfect model, it is easy for the error to accumulate over time. In fact, in our work we had the need to constantly correct the model's prediction manually every time since the predictions' values easily diverged. In any case, prediction of the future is something the agent should technically be capable of handling by itself, since the environment model could be implicitly integrated into the agent's policy and value/action-value networks. Considering the environment model explicitly simply allows the decision making to be more transparent, as well as train the environment model independently of the agent. However, independent training poses the question of whether the environment model learns to predict the future in such a way that is useful to the agent. It might not be useful for the agent to predict the specific values of the environment state variables, but to predict other abstract concepts that may actually be useful for making good decisions. As such, for the task of handling reaction time, predicting the exact current time step might not be appropriate. One work utilized a recurrent environment model to tackle reaction time in such a manner with success [75], but was too computationally expensive to apply for the number of frames in the reaction time formulation considered in our work.

Adaptation in this work was attempted by explicitly training an opponent model and having the agent learn considering that model. However, it still proved too slow for few-shot learning, i.e. efficiently learn with little experience. The agent only learns using experience it obtains online while interacting with the environment, which is limited and would hardly generalize to unseen scenarios. To adapt better, the agent should be able to perform simulations of possible scenarios and learn directly from them, without having to wait to actually experience them, as in [113]. To achieve this, environment models need to be used, which implicitly include a model of the current opponent, or at least use a simulator such as the one in FightingICE to simulate relevant scenarios and learn from them without requiring actual experience with the current opponent. Otherwise, if the agent was to play against a human player, then it would require hundreds of matches for adaptation. This does not go in line with the way humans learn, who can easily predict and assess the results of their actions in different situations. In fact, fighting games tend to offer a "training mode" to allow players, on their own, to set up specific game scenarios in order to understand what they can do and what would happen in an actual match. Added to the training mode, games also offer a "replay mode", that allows players to watch recordings of past matches and scrutinize past scenarios so as

to understand them better, or to even interact within those recordings to see what would happen had either player acted differently. These modes are very important in supporting offline learning for human players, and so this kind of learning should be explored for faster adaptation in artificial agents as well.

6.2.5 Domain knowledge

In general, the solution relied heavily on domain knowledge and explicit instrumentation of different components that, we believed, aided the agent in completing the task. However, it may in fact be more beneficial to include the least amount of domain knowledge in the algorithm so that the agent is free to develop any abstractions it needs to solve the problem. The main reason for including domain knowledge, such as explicitly modeling the opponent, correcting the reaction time through environment modeling or skipping the hitstop/blockstop freeze, was to speed up agent learning and make its decision making more transparent. The issue is that these adjustments and modifications may not be ideal, or even correctly implemented, to achieve our goal. For example, in this work we define the opponent's action space to be each possible action in the *FOOTSIES* game, including special moves. In fighting games, however, we may perform opponent modeling at a higher level, predicting their overall strategy or, even worse, predicting they will perform an action but *delayed in time*, which requires reasoning over multiple time steps and, therefore, temporal abstraction. Not only does this hint at hierarchical formulations of the opponent's behavior, but it also means that opponent modeling should not be an isolated component, it should instead occur naturally as a solution to the problem at hand and exist to *serve* the solution, not merely to predict the opponent well [114]. In future work, we should minimize the amount of domain knowledge incurred in these systems and opt for more general solutions if possible. The inclusion of domain knowledge may have aided in simplifying the problem and achieving good performance, but it also makes the algorithm more complex and significantly more difficult to debug, which has large costs in development time. For instance, going to the trouble of ignoring hitstop/blockstop freeze as studied in Subsection 5.2.5 made the algorithm able to learn quicker but also more complex, with roughly the same performance in the end. Of course, having the agent learn high-level abstractions requires more capable models and a larger training budget, that are constrained for the application to fighting games which are meant to run on commodity hardware. This trade-off between incorporation of expert knowledge and task-agnostic learning methodologies is a tough balance to maintain, deserving of further study.

References

[1] R. Liu, «Creating human-like fighting game ai through planning», M.S. thesis, Carnegie Mellon University, Pittsburgh, PA, Dec. 2017.

[2] V. Firoiu, W. F. Whitney, and J. B. Tenenbaum, «Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning», arXiv, Tech. Rep., May 2017, arXiv:1702.06230 [cs] type: article. DOI: 10.48550/arXiv.1702.06230. arXiv: 1702.06230 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1702.06230 (visited on 06/05/2024).

[3] Y. I. Gingold, «From Rock, Paper, Scissors to Street Fighter II: Proof by construction», in *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, ser. Sandbox '06, ISBN: 1595933867, Boston, Massachusetts: Association for Computing Machinery, Jul. 2006, pp. 155–158, ISBN: 1595933867. DOI: 10.1145/1183316.1183339. [Online]. Available: https://doi.org/10.1145/1183316.1183339 (visited on 01/09/2024).

[4] K. Yu and N. R. Sturtevant, «Application of retrograde analysis on fighting games», in *2019 IEEE Conference on Games (CoG)*, Publisher: IEEE Computer Society ISBN: 9781728118840, vol. 2019-August, IEEE, Aug. 2019. DOI: 10.1109/cig.2019.8848062. (visited on 11/08/2023).

[5] T. Sugiyama, R. Takahashi, and M. Kanoh, «Can an advanced ai provide the same level of enjoyment as playing with human beings?», in *HCI International 2022 Posters*, C. Stephanidis, M. Antona, and S. Ntoa, Eds., Cham: Springer International Publishing, Jun. 2022, pp. 456–461, ISBN: 978-3-031-06417-3. DOI: 10.1007/978-3-031-06417-3_61.

[6] B. Soni and P. Hingston, «Bots trained to play like a human are more fun», in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, ISBN: 9781424418213, IEEE, Jun. 2008, pp. 363–369. DOI: 10.1109/ijcnn.2008.4633818. (visited on 12/24/2023).

[7] B. Hayles and D. Neal, «Designing AI for Competitive Games», San Francisco, Mar. 2016. [Online]. Available: https://gdcvault.com/play/1022992/Designing-AI-for-Competitive (visited on 06/05/2024).

[8] M. Ibrahim, P. Sweetser, and A. Ozdowska, «Tutorial Level Design Guidelines for 2D Fighting Games», in *Proceedings of the 18th International Conference on the Foundations of Digital Games*, ser. FDG '23, Publisher: Association for Computing Machinery ISBN: 9781450398565, Lisbon, Portugal: Association for Computing Machinery, Apr. 2023, pp. 1–11, ISBN: 9781450398558. DOI: 10.1145/3582437.3582470. [Online]. Available: https://dl.acm.org/doi/10.1145/3582437.3582470 (visited on 01/09/2024).

[9] R. e Silva Vieira, A. R. Tavares, and L. Chaimowicz, «Towards sample efficient deep reinforcement learning in collectible card games», *Entertainment Computing*, vol. 47, p. 100 594, Aug. 2023, Publisher: Elsevier, ISSN: 1875-9521. DOI: 10.1016/j.entcom.2023.100594. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1875952123000496 (visited on 11/28/2023).

[10] N. Sato, S. Temsiririrkkul, S. Sone, and K. Ikeda, «Adaptive fighting game computer player by switching multiple rule-based controllers», in *2015 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence*, Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781467396424, IEEE, Jul. 2015, pp. 52–59. DOI: 10.1109/acit-csi.2015.18. [Online]. Available: https://www.researchgate.net/publication/308604729_Adaptive_Fighting_Game_Computer_Player_by_Switching_Multiple_Rule-Based_Controllers (visited on 11/28/2023).

[11] R. S. Sutton and A. G. Barto, *Reinforcement learning, An introduction* (Adaptive computation and machine learning), Second edition. Cambridge, Massachusetts: The MIT Press, 2020, 526 pp., Literaturverzeichnis: Seiten 481-518, ISBN: 9780262039246.

[12] A. S. Polydoros and L. Nalpantidis, «Survey of Model-Based Reinforcement Learning: Applications on Robotics», en, *Journal of Intelligent & Robotic Systems*, vol. 86, no. 2, pp. 153–173, May 2017, ISSN: 1573-0409. DOI: 10.1007/s10846-017-0468-y. [Online]. Available: https://doi.org/10.1007/s10846-017-0468-y (visited on 06/05/2024).

[13] D. Silver, S. Singh, D. Precup, and R. S. Sutton, «Reward is enough», *Artificial Intelligence*, vol. 299, p. 103 535, Oct. 2021, Publisher: Elsevier, ISSN: 0004-3702. DOI: 10.1016/j.artint.2021.103535. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370221000862 (visited on 11/28/2023).

[14] C. Nota and P. S. Thomas, «Is the Policy Gradient a Gradient?», arXiv, Tech. Rep., Feb. 2020, arXiv:1906.07073 [cs, stat] type: article. DOI: 10.48550/arXiv.1906.07073. arXiv: 1906.07073 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1906.07073 (visited on 04/14/2024).

[15] M. Eppe, C. Gumbsch, M. Kerzel, P. D. H. Nguyen, M. V. Butz, and S. Wermter, «Intelligent problem-solving as integrated hierarchical reinforcement learning», en, *Nature Machine Intelligence*, vol. 4, no. 1, pp. 11–20, Jan. 2022, arXiv: 2208.08731v1 Publisher: Nature Research, ISSN: 2522-5839. DOI: 10.1038/s42256-021-00433-9. [Online]. Available: https://www.nature.com/articles/s42256-021-00433-9 (visited on 06/05/2024).

[16] M. Hutsebaut-Buysse, K. Mets, and S. Latré, «Hierarchical Reinforcement Learning: A Survey and Open Research Challenges», en, *Machine Learning and Knowledge Extraction*, vol. 4, no. 1, pp. 172–221, Feb. 2022, Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 2504-4990. DOI: 10.3390/make4010009. [Online]. Available: https://www.mdpi.com/2504-4990/4/1/9 (visited on 11/28/2023).

[17] G. L. Zuin, Y. P. Macedo, L. Chaimowicz, and G. L. Pappa, «Discovering combos in fighting games with evolutionary algorithms», in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16, Publisher: Association for Computing Machinery, Inc ISBN: 9781450342063, Denver, Colorado, USA: Association for Computing Machinery, Jul. 2016, pp. 277–284, ISBN: 9781450342063. DOI: 10.1145/2908812.2908908. [Online]. Available: https://dl.acm.org/doi/10.1145/2908812.2908908 (visited on 01/18/2024).

[18] I. Oh, S. Rho, S. Moon, S. Son, H. Lee, and J. Chung, «Creating pro-level ai for a real-time fighting game using deep reinforcement learning», *IEEE Transactions on Games*, vol. 14, no. 2, pp. 212–220, Jun. 2022, arXiv: 1904.03821 Publisher: Institute of Electrical and Electronics Engineers Inc., ISSN: 2475-1510. DOI: 10.1109/tg.2021.3049539. (visited on 11/28/2023).

[19] W. E. Hick, «On the Rate of Gain of Information», en, *Quarterly Journal of Experimental Psychology*, vol. 4, no. 1, pp. 11–26, Mar. 1952, ISSN: 0033-555X. DOI: 10.1080/17470215208416600. [Online]. Available: https://doi.org/10.1080/17470215208416600 (visited on 12/06/2023).

[20] R. Pavão, J. P. Savietto, J. R. Sato, G. F. Xavier, and A. F. Helene, «On Sequence Learning Models: Open-loop Control Not Strictly Guided by Hick's Law», en, *Scientific Reports*, vol. 6, no. 1, p. 23 018, Mar. 2016, Publisher: Nature Publishing Group, ISSN: 2045-2322. DOI: 10.1038/srep23018. [Online]. Available: https://doi.org/10.1038/srep23018 (visited on 12/30/2023).

[21] R. Vigo, C. A. Doan, J. Wimsatt, and C. B. Ross, «A context-sensitive alternative to hick's law of choice reaction times: A mathematical and computational unification of conceptual complexity and choice behavior», *Mathematics*, vol. 11, no. 11, 2023, ISSN: 2227-7390. DOI: 10.3390/math11112422. [Online]. Available: https://www.mdpi.com/2227-7390/11/11/2422.

[22] T. Komai, H. Kurokawa, and S.-J. Kim, «Human Randomness in the Rock-Paper-Scissors Game», en, *Applied Sciences*, vol. 12, no. 23, p. 12 192, Nov. 2022, Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 2076-3417. DOI: 10.3390/app122312192. [Online]. Available: https://www.mdpi.com/2076-3417/12/23/12192 (visited on 12/18/2023).

[23] E. Brockbank and E. Vul, «Recursive Adversarial Reasoning in the Rock, Paper, Scissors Game», *Cognitive Science*, Jun. 2020. DOI: 10.31234/osf.io/yxb2v. [Online]. Available: https://osf.io/yxb2v (visited on 11/29/2023).

[24] I. Guennouni and M. Speekenbrink, «Transfer of Learned Opponent Models in Zero Sum Games», en, *Computational Brain & Behavior*, vol. 5, no. 3, pp. 326–342, Sep. 2022, ISSN: 2522-087X. DOI: 10.1007/s42113-022-00133-6. [Online]. Available: https://doi.org/10.1007/s42113-022-00133-6 (visited on 06/05/2024).

[25] J. Sundvall and B. J. Dyson, «Breaking the bonds of reinforcement: Effects of trial outcome, rule consistency and rule complexity against exploitable and unexploitable opponents», *PLOS ONE*, vol. 17, no. 2, E. Konstantinidis, Ed., pp. 1–19, Feb. 2022, Publisher: Public Library of Science, ISSN: 1932-6203. DOI: 10.1371/JOURNAL.PONE.0262249. [Online]. Available: https://doi.org/10.1371/journal.pone.0262249 (visited on 01/18/2024).

[26] Z. Wang, B. Xu, and H.-J. Zhou, «Social cycling and conditional responses in the Rock-Paper-Scissors game», en, *Scientific Reports*, vol. 4, no. 1, p. 5830, Jul. 2014, arXiv: 1404.5199v1 Publisher: Nature Publishing Group, ISSN: 2045-2322. DOI: 10.1038/srep05830. [Online]. Available: https://doi.org/10.1038/srep05830 (visited on 12/10/2023).

[27] W. M. Czarnecki, G. Gidel, B. Tracey, *et al.*, «Real World Games Look Like Spinning Tops», arXiv, Tech. Rep., Jun. 2020, arXiv:2004.09468 [cs, stat] version: 2 type: article. DOI: 10.48550/arXiv.2004.09468. arXiv: 2004.09468 [cs.LG]. [Online]. Available: http://arxiv.org/abs/2004.09468 (visited on 06/05/2024).

[28] J. N. Towse, T. Loetscher, and P. Brugger, «Not all numbers are equal: Preferences and biases among children and adults when generating random sequences», *Frontiers in Psychology*, vol. 5, no. JAN, p. 19, Jan. 2014, Publisher: Frontiers Media SA, ISSN: 1664-1078. DOI: 10.3389/FPSYG.2014.00019. [Online]. Available: https://www.frontiersin.org/journals/psychology/articles/10.3389/fpsyg.2014.00019 (visited on 12/10/2023).

[29] «SAMURAI SHODOWN OFFICIAL WEBSITE | SNK». en. (May 20, 2024), [Online]. Available: https://www.snk-corp.co.jp/us/games/samuraishodown (visited on 05/20/2024).

[30] «MODE | TEKKEN 8». (May 20, 2024), [Online]. Available: https://tk8.tekken-official.jp/en/mode/offline.php (visited on 05/20/2024).

[31] «Figure Player». en. (Dec. 2023), [Online]. Available: https://www.ssbwiki.com/index.php?title=Figure_Player&oldid=1847186 (visited on 06/05/2024).

[32] F. Lu, K. Yamamoto, L. H. Nomura, S. Mizuno, Y. Lee, and R. Thawonmas, «Fighting game artificial intelligence competition platform», in *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*, ISSN: 2378-8143, Oct. 2013, pp. 320–323. DOI: 10.1109/GCCE.2013.6664844. [Online]. Available: https://ieeexplore.ieee.org/document/6664844 (visited on 11/14/2023).

[33] I. Khan, T. V. Nguyen, X. Dai, and R. Thawonmas, «Darefightingice competition: A fighting game sound design and ai competition», in *2022 IEEE Conference on Games (CoG)*, arXiv: 2203.01556 Publisher: IEEE Computer Society ISBN: 9781665459891, vol. 2022-August, IEEE, Aug. 2022, pp. 478–485. DOI: 10.1109/cog51982.2022.9893624. (visited on 12/09/2023).

[34] M.-J. Kim and K.-J. Kim, «Opponent modeling based on action table for MCTS-based fighting game AI», in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, ISSN: 2325-4289, Aug. 2017, pp. 178–180. DOI: 10.1109/CIG.2017.8080432. [Online]. Available: https://ieeexplore.ieee.org/document/8080432 (visited on 11/28/2023).

[35] G. T. Lam, D. Logofătu, and C. Bădică, «A novel real-time design for fighting game ai», *Evolving Systems*, vol. 12, no. 1, pp. 169–176, 2021, ISSN: 1868-6486. DOI: 10.1007/s12530-020-09351-4. [Online]. Available: https://doi.org/10.1007/s12530-020-09351-4.

[36] Z.-t. Tang, R.-q. Liang, Y.-h. Zhu, and D.-b. Zhao, «Intelligent decision making approaches for real time fighting game», *Control Theory and Technology*, vol. 39, no. 6, pp. 969–985, Jun. 2022, Publisher: South China University of Technology, ISSN: 1000-8152. DOI: 10.7641/CTA.2022.10995. (visited on 11/08/2023).

[37] D. Silver, A. Huang, C. J. Maddison, *et al.*, «Mastering the game of Go with deep neural networks and tree search», en, *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/nature16961. [Online]. Available: https://doi.org/10.1038/nature16961 (visited on 06/05/2024).

[38] A. Palmas, «DIAMBRA Arena: A New Reinforcement Learning Platform for Research and Experimentation», arXiv, Tech. Rep., Oct. 2022, arXiv:2210.10595 [cs] version: 1 type: article. DOI: 10.48550/arXiv.2210.10595. arXiv: 2210.10595 [cs.LG]. [Online]. Available: http://arxiv.org/abs/2210.10595 (visited on 06/05/2024).

[39] J. Schrittwieser, I. Antonoglou, T. Hubert, *et al.*, «Mastering Atari, Go, chess and shogi by planning with a learned model», en, *Nature*, vol. 588, no. 7839, pp. 604–609, Dec. 2020, arXiv:1911.08265 [cs, stat], ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4. [Online]. Available: https://doi.org/10.1038/s41586-020-03051-4 (visited on 06/05/2024).

[40] I. P. Pinto and L. R. Coutinho, «Hierarchical Reinforcement Learning With Monte Carlo Tree Search in Computer Fighting Game», *IEEE Transactions on Games*, vol. 11, no. 3, pp. 290–295, Sep. 2019, Publisher: Institute of Electrical and Electronics Engineers Inc., ISSN: 2475-1510. DOI: 10.1109/TG.2018.2846028. [Online]. Available: https://ieeexplore.ieee.org/document/8378245 (visited on 11/28/2023).

[41] D.-W. Kim, S. Park, and S.-i. Yang, «Mastering Fighting Game Using Deep Reinforcement Learning With Self-play», in *2020 IEEE Conference on Games (CoG)*, ISSN: 2325-4289, vol. 2020-August, Aug. 2020, pp. 576–583. DOI: 10.1109/CoG47356.2020.9231639. [Online]. Available: https://ieeexplore.ieee.org/document/9231639 (visited on 11/28/2023).

[42] M. Lanctot, J. Schultz, N. Burch, *et al.*, «Population-based Evaluation in Repeated Rock-Paper-Scissors as a Benchmark for Multiagent Reinforcement Learning», arXiv, Tech. Rep., Oct. 2023, arXiv:2303.03196 [cs] version: 2 type: article. DOI: 10.48550/arXiv.2303.03196. arXiv: 2303.03196 [cs.GT]. [Online]. Available: http://arxiv.org/abs/2303.03196 (visited on 06/05/2024).

[43] H. Du and R. Jóźwiak, «Representation of Observations in Reinforcement Learning for Playing Arcade Fighting Game», en, in *Digital Interaction and Machine Intelligence*, C. Biele, J. Kacprzyk, W. Kopeć, J. W. Owsiński, A. Romanowski, and M. Sikorski, Eds., Cham: Springer Nature Switzerland, 2023, pp. 45–55, ISBN: 9783031376498. DOI: 10.1007/978-3-031-37649-8_5.

[44] H. Liang and J. Li, «A study on the agent in fighting games based on deep reinforcement learning», *Mobile Information Systems*, vol. 2022, P. K. R. Maddikunta, Ed., pp. 1–8, Jul. 2022, Publisher: Hindawi Limited, ISSN: 1574-017X. DOI: 10.1155/2022/9984617. (visited on 11/28/2023).

[45] P. Alexander, *Esports for Dummies*, en. Newark: John Wiley & Sons, Incorporated, 2020, ch. 5, pp. 87–92, 1307 pp., Description based on publisher supplied metadata and other sources., ISBN: 9781119650560. [Online]. Available: https://www.dummies.com/article/home-auto-hobbies/sports-recreation/esports/esports-the-basics-of-the-fighting-game-genre-270300/ (visited on 06/05/2024).

[46] S. Yoon and K.-J. Kim, «Deep q networks for visual fighting game ai», in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781538632338, IEEE, Aug. 2017, pp. 306–308. DOI: 10.1109/cig.2017.8080451. (visited on 11/28/2023).

[47] Y. Yu, «Towards sample efficient reinforcement learning», in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, ser. IJCAI-2018, International Joint Conferences on Artificial Intelligence Organization, Jul. 2018, pp. 5739–5743. DOI: 10.24963/ijcai.2018/820. [Online]. Available: https://doi.org/10.24963/ijcai.2018/820.

[48] R. Everett, «Learning Against Non-Stationary Agents with Opponent Modelling & Deep Reinforcement Learning», in *31st Conference on Neural Information Processing Systems (NIPS 2017)*, Long Beach, CA, USA, 2017. [Online]. Available: https://www.semanticscholar.org/paper/Learning-Against-Non-Stationary-Agents-with-%26-Deep-Everett/8b1288f8d930daea8184e7528b53946791bc767c (visited on 06/05/2024).

[49] E. Halina and M. Guzdial, «Diversity-based Deep Reinforcement Learning Towards Multidimensional Difficulty for Fighting Game AI», arXiv, Tech. Rep., Nov. 2022, arXiv:2211.02759 [cs] version: 1 type: article. DOI: 10.48550/arXiv.2211.02759. arXiv: 2211.02759 [cs.LG]. [Online]. Available: http://arxiv.org/abs/2211.02759 (visited on 06/05/2024).

[50] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine, «Diversity is All You Need: Learning Skills without a Reward Function», arXiv, Tech. Rep., Oct. 2018, arXiv:1802.06070 [cs] type: article. DOI: 10.48550/

arXiv.1802.06070. arXiv: 1802.06070 [cs.AI]. [Online]. Available: http://arxiv.org/abs/1802.06070 (visited on 04/19/2024).

[51] T. Chen, F. Richoux, J. M. Torres, and K. Inoue, «Interpretable Utility-based Models Applied to the FightingICE Platform», in *2021 IEEE Conference on Games (CoG)*, ISSN: 2325-4289, vol. 2021-August, Aug. 2021, pp. 1–8. DOI: 10.1109/CoG52621.2021.9619121. [Online]. Available: https://ieeexplore.ieee.org/document/9619121 (visited on 11/28/2023).

[52] C. Nimpattanavong, W. Choensawat, and K. Sookhanaphibarn, «Action prediction of ai bot in fightingice by using deep learning model», in *2022 IEEE 4th Global Conference on Life Sciences and Technologies (LifeTech)*, Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781665419048, IEEE, Mar. 2022, pp. 259–262. DOI: 10.1109/lifetech53646.2022.9754922. (visited on 11/29/2023).

[53] N. D. T. Tri, V. Quang, and K. Ikeda, «Optimized Non-visual Information for Deep Neural Network in Fighting Game», in *Proceedings of the 9th International Conference on Agents and Artificial Intelligence - Volume 1: ICAART*, Publisher: SciTePress ISBN: 9789897582202, INSTICC, vol. 2, SciTePress, 2017, pp. 676–680, ISBN: 978-989-758-220-2. DOI: 10.5220/0006248106760680. (visited on 11/28/2023).

[54] T. Van Nguyen, X. Dai, I. Khan, R. Thawonmas, and H. V. Pham, «A deep reinforcement learning blind ai in darefightingice», in *2022 IEEE Conference on Games (CoG)*, arXiv: 2205.07444 Publisher: IEEE Computer Society ISBN: 9781665459891, vol. 2022-August, IEEE, Aug. 2022, pp. 632–637. DOI: 10.1109/cog51982.2022.9893718. (visited on 11/28/2023).

[55] A. Cherukuri and F. G. Glavin, «Balancing the performance of a fightingice agent using reinforcement learning and skilled experience catalogue», in *2022 IEEE Games, Entertainment, Media Conference (GEM)*, Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781665461382, IEEE, Nov. 2022. DOI: 10.1109/gem56474.2022.10017566. (visited on 11/28/2023).

[56] J. P. Q. Tomas, N. J. R. Aguas, A. N. De Villa, and J. R. G. Lim, «Developing an adaptive ai agent using supervised and reinforcement learning with monte carlo tree search in fightingice», in *Proceedings of the 2021 4th International Conference on Computational Intelligence and Intelligent Systems*, ser. CIIS '21, Tokyo, Japan: Association for Computing Machinery, Apr. 2022, pp. 31–36, ISBN: 9781450385930. DOI: 10.1145/3507623.3507629. [Online]. Available: https://doi.org/10.1145/3507623.3507629.

[57] Y. Takano, H. Inoue, R. Thawonmas, and T. Harada, «Self-Play for Training General Fighting Game AI», in *2019 Nicograph International (NicoInt)*, Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781728140216, Jul. 2019, pp. 120–120. DOI: 10.1109/NICOInt.2019.00034. [Online]. Available: https://ieeexplore.ieee.org/document/8949184 (visited on 11/28/2023).

[58] O. Vinyals, I. Babuschkin, W. M. Czarnecki, *et al.*, «Grandmaster level in StarCraft II using multi-agent reinforcement learning», en, *Nature*, vol. 575, no. 7782, pp. 350–354, Nov. 2019, ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. [Online]. Available: https://doi.org/10.1038/s41586-019-1724-z (visited on 04/19/2024).

[59] M.-J. Kim and C. W. Ahn, «Hybrid fighting game AI using a genetic algorithm and Monte Carlo tree search», in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '18, Publisher: Association for Computing Machinery, Inc ISBN: 9781450357647, Kyoto, Japan: Association for Computing Machinery, Jul. 2018, pp. 129–130, ISBN: 9781450357647. DOI: 10.1145/3205651.3205695. [Online]. Available: https://doi.org/10.1145/3205651.3205695 (visited on 11/08/2023).

[60] I. Gajardo, F. Besoain, and N. A. Barriga, «Introduction to Behavior Algorithms for Fighting Games», in *2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*, Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781728131856, Nov. 2019, pp. 1–6. DOI: 10.1109/CHILECON47746.2019.8988008. [Online]. Available: https://ieeexplore.ieee.org/document/8988008 (visited on 11/08/2023).

[61] M.-J. Kim, J.-H. Lee, and C. W. Ahn, «Genetic Optimizing Method for Real-time Monte Carlo Tree Search Problem», in *The 9th International Conference on Smart Media and Applications*, ser. SMA 2020, Publisher: Association for Computing Machinery ISBN: 9781450389259, Jeju, Republic of Korea: Association for Computing Machinery, Nov. 2021, pp. 50–51, ISBN: 9781450389259. DOI: 10.1145/3426020.3426030. [Online]. Available: https://doi.org/10.1145/3426020.3426030 (visited on 11/08/2023).

[62] N. Q. Vu, «Building a strong fighting game player», M.S. thesis, Japan Advanced Institute of Science and Technology, Sep. 2016. [Online]. Available: http://hdl.handle.net/10119/13735.

[63] D. E. Zambrano Huertas and J. S. Díaz Salamanca, «Deep reinforcement learning for optimal gameplay in street fighter III: A resource-constrained approach», Fundación Universitaria de Ciencias de la Salud, Tech. Rep., Aug. 2023. [Online]. Available: https://hdl.handle.net/1992/70987.

[64] M. Zohaib, «Dynamic Difficulty Adjustment (DDA) in Computer Games: A Review», en, *Advances in Human-Computer Interaction*, vol. 2018, no. 1, p. 5 681 652, 2018, Publisher: Hindawi Limited, ISSN: 16875907. DOI: 10.1155/2018/5681652. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2018/5681652 (visited on 12/02/2023).

[65] J. Moon, Y. Choi, T. Park, J. Choi, and J.-H. H. an d Kyung-Joong Kim, «Diversifying dynamic difficulty adjustment agent by integrating player state models into monte-carlo tree search», *Expert Systems with Applications*, vol. 205, p. 117 677, Nov. 2022, Publisher: Pergamon, ISSN: 0957-4174. DOI: 10.1016/j.eswa.2022.117677. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417422009757 (visited on 11/28/2023).

[66] Z. Wu, K. Li, E. Zhao, *et al.*, «L2E: Learning to Exploit Your Opponent», arXiv, Tech. Rep., Feb. 2021, arXiv:2102.09381 [cs] type: article. DOI: 10.48550/arXiv.2102.09381. arXiv: 2102.09381 [cs.LG]. [Online]. Available: http://arxiv.org/abs/2102.09381 (visited on 06/05/2024).

[67] J. R. Bezerra, L. F. W. Góes, and A. R. da Silva, «Development of an autonomous agent based on reinforcement learning for a digital fighting game», in *2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, Publisher: IEEE Computer Society ISBN: 9781728184326, vol. 2020-November, IEEE, Nov. 2020, pp. 1–7. DOI: 10.1109/sbgames51465.2020.00017. (visited on 11/28/2023).

[68] X. Neufeld, S. Mostaghim, and D. Perez-Liebana, «HTN fighter: Planning in a highly-dynamic game», in *2017 9th Computer Science and Electronic Engineering (CEEC)*, Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781538630075, Sep. 2017, pp. 189–194. DOI: 10.1109/CEEC.2017.8101623. [Online]. Available: https://ieeexplore.ieee.org/document/8101623 (visited on 11/28/2023).

[69] R. Brown and C. Guinn, «Developing game-playing agents that adapt to user strategies: A case study», in *2014 IEEE Symposium on Intelligent Agents (IA)*, Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781479944897, IEEE, Dec. 2014, pp. 51–56. DOI: 10.1109/ia.2014.7009458. (visited on 11/28/2023).

[70] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, «Trust Region Policy Optimization», arXiv, Tech. Rep., Apr. 2017, arXiv:1502.05477 [cs] type: article. DOI: 10.48550/arXiv.1502.05477. arXiv: 1502.05477 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1502.05477 (visited on 06/05/2024).

[71] C. Finn, P. Abbeel, and S. Levine, «Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks», arXiv, Tech. Rep., Jul. 2017, arXiv:1703.03400 [cs] type: article. DOI: 10.48550/arXiv.1703.03400. arXiv: 1703.03400 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1703.03400 (visited on 06/05/2024).

[72] D. Charles, «Enhancing gameplay: Challenges for articifical intelligence in digital games.», in *Proceedings of DiGRA 2003 Conference: Level Up*, Tampere: DiGRA, Jan. 2003.

[73] S. Milani, A. Juliani, I. Momennejad, *et al.*, «Navigates Like Me: Understanding How People Evaluate Human-Like AI in Video Games», in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. CHI '23, arXiv: 2303.02160, Hamburg, Germany: Association for Computing Machinery, Apr. 2023, pp. 1–18, ISBN: 9781450394215. DOI: 10.1145/3544548.3581348. [Online]. Available: https://dl.acm.org/doi/10.1145/3544548.3581348 (visited on 06/05/2024).

[74] B. Gonçalves, «The Turing Test is a Thought Experiment», en, *Minds and Machines*, vol. 33, no. 1, pp. 1–31, Mar. 2023, ISSN: 1572-8641. DOI: 10.1007/s11023-022-09616-8. [Online]. Available: https://doi.org/10.1007/s11023-022-09616-8 (visited on 06/05/2024).

[75] V. Firoiu, T. Ju, and J. Tenenbaum, «At Human Speed: Deep Reinforcement Learning with Action Delay», arXiv, Tech. Rep., Oct. 2018, arXiv:1810.07286 [cs] type: article. DOI: 10.48550/arXiv.1810.

07286. arXiv: 1810.07286 [cs.AI]. [Online]. Available: http://arxiv.org/abs/1810.07286 (visited on 06/05/2024).

[76] K. Yuda, M. Mozgovoy, and A. Danielewicz-Betz, «Creating an affective fighting game ai system with gamygdala», in *2019 IEEE Conference on Games (CoG)*, Publisher: IEEE Computer Society ISBN: 9781728118840, vol. 2019-August, IEEE, Aug. 2019. DOI: 10.1109/cig.2019.8848031. (visited on 11/28/2023).

[77] K. Yuda, S. Kamei, R. Tanji, R. Ito, I. Wakana, and M. Mozgovoy, «Identification of Play Styles in Universal Fighting Engine», arXiv, Tech. Rep., Aug. 2021, arXiv:2108.03599 [cs] type: article. DOI: 10.48550/arXiv.2108.03599. arXiv: 2108.03599 [cs.AI]. [Online]. Available: http://arxiv.org/abs/2108.03599 (visited on 06/05/2024).

[78] F. Lu, C. Ken Choy, and R. Thawonmas, «A fighting game ai with evolutionary strategy and imitation learning in opportunity maximization and sensible maneuvering tactic», in *77th National Conference Lecture Collection*, vol. 2015, Mar. 2015, pp. 101–102. [Online]. Available: http://id.nii.ac.jp/1001/00164212/.

[79] M. Ishihara, S. Ito, R. Ishii, T. Harada, and R. Thawonmas, «Monte-Carlo Tree Search for Implementation of Dynamic Difficulty Adjustment Fighting Game AIs Having Believable Behaviors», in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, ISSN: 2325-4289, vol. 2018-August, Aug. 2018, pp. 1–8. DOI: 10.1109/CIG.2018.8490376. [Online]. Available: https://ieeexplore.ieee.org/document/8490376 (visited on 11/08/2023).

[80] M. R. F. Mendonça, H. S. Bernardino, and R. F. Neto, «Simulating Human Behavior in Fighting Games Using Reinforcement Learning and Artificial Neural Networks», in *2015 14th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, ISSN: 2159-6662, vol. 0, IEEE Computer Society, Nov. 2015, pp. 152–159, ISBN: 978-1-4673-8843-6. DOI: 10.1109/SBGames.2015.25. [Online]. Available: https://ieeexplore.ieee.org/document/7785852 (visited on 06/05/2024).

[81] S. Miyashita, X. Lian, X. Zeng, T. Matsubara, and K. Uehara, «Developing game ai agent behaving like human by mixing reinforcement learning and supervised learning», in *2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781509055043, IEEE, Jun. 2017, pp. 489–494. DOI: 10.1109/snpd.2017.8022767. (visited on 11/28/2023).

[82] C. Arzate Cruz and J. A. Ramirez Uresti, «HRLB^2: A Reinforcement Learning Based Framework for Believable Bots», en, *Applied Sciences*, vol. 8, no. 12, p. 2453, Dec. 2018, Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 2076-3417. DOI: 10.3390/app8122453. [Online]. Available: https://www.mdpi.com/2076-3417/8/12/2453 (visited on 11/28/2023).

[83] D.-W. Kim, S.-Y. Park, and S.-i. Yang, «Reusing Agent's Representations for Adaptation to Tuned-environment in Fighting Game», in *2021 International Conference on Information and Communication Technology Convergence (ICTC)*, ISSN: 2162-1233, vol. 2021-October, Oct. 2021, pp. 1120–1124. DOI: 10.1109/ICTC52510.2021.9620988. [Online]. Available: https://ieeexplore.ieee.org/document/9620988 (visited on 11/28/2023).

[84] M. Stetter and E. W. Lang, «Learning Intuitive Physics and One-Shot Imitation Using State-Action-Prediction Self-Organizing Maps», en, *Computational Intelligence and Neuroscience*, vol. 2021, no. 1, p. 5 590 445, 2021, arXiv: 2007.01647 Publisher: Hindawi Limited, ISSN: 1687-5273. DOI: 10.1155/2021/5590445. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2021/5590445 (visited on 12/16/2023).

[85] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, «Curiosity-driven exploration by self-supervised prediction», arXiv, Tech. Rep., May 2017, arXiv:1705.05363 [cs, stat] type: article. DOI: 10.48550/arXiv.1705.05363. arXiv: 1705.05363 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1705.05363 (visited on 04/19/2024).

[86] G. Papoudakis, F. Christianos, A. Rahman, and S. V. Albrecht, «Dealing with non-stationarity in multi-agent deep reinforcement learning», arXiv, Tech. Rep., Jun. 2019, arXiv:1906.04737 [cs, stat] type: article. DOI: 10.48550/arXiv.1906.04737. arXiv: 1906.04737 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1906.04737 (visited on 06/04/2024).

[87] R. F. Prudencio, M. R. O. A. Maximo, and E. L. Colombini, «A survey on offline reinforcement learning: Taxonomy, review, and open problems», *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–, 2024, arXiv: 2203.01387 Publisher: Institute of Electrical and Electronics Engineers Inc., ISSN: 2162-2388. DOI: 10.1109/tnnls.2023.3250269. (visited on 12/24/2023).

[88] Z. Liang, J. Cao, S. Jiang, D. Saxena, and H. Xu, «Hierarchical Reinforcement Learning with Opponent Modeling for Distributed Multi-agent Cooperation», in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, ISSN: 2575-8411, vol. 2022-July, Jul. 2022, pp. 884–894. DOI: 10.1109/ICDCS54860.2022.00090. [Online]. Available: https://ieeexplore.ieee.org/document/9912260 (visited on 06/05/2024).

[89] Y. Wen, Y. Yang, R. Luo, J. Wang, and W. Pan, «Probabilistic Recursive Reasoning for Multi-Agent Reinforcement Learning», arXiv, Tech. Rep., Mar. 2019, arXiv:1901.09207 [cs, stat] type: article. DOI: 10.48550/arXiv.1901.09207. arXiv: 1901.09207 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1901.09207 (visited on 06/05/2024).

[90] C. F. Camerer, T.-H. Ho, and J.-K. Chong, «A cognitive hierarchy model of games», *The Quarterly Journal of Economics*, vol. 119, no. 3, pp. 861–898, Aug. 2004, ISSN: 1531-4650. DOI: 10.1162/0033553041502225. eprint: https://academic.oup.com/qje/article-pdf/119/3/861/5461769/119-3-861.pdf. [Online]. Available: https://doi.org/10.1162/0033553041502225.

[91] D. Batzilis, S. Jaffe, S. Levitt, J. A. List, and J. Picel, «Behavior in strategic settings: Evidence from a million rock-paper-scissors games», *Games*, vol. 10, no. 2, p. 18, Apr. 2019, Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 2073-4336. DOI: 10.3390/g10020018. [Online]. Available: https://www.mdpi.com/2073-4336/10/2/18/htm (visited on 01/11/2024).

[92] «GGPO | Rollback Networking SDK for Peer-to-Peer Games». en. (May 20, 2024), [Online]. Available: https://www.ggpo.net (visited on 05/20/2024).

[93] A. Ehlert, «Improving Input Prediction in Online Fighting Games», eng, BA thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2021, p. 62. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-296341 (visited on 06/05/2024).

[94] Z. Tang, Y. Zhu, D. Zhao, and S. M. Lucas, «Enhanced Rolling Horizon Evolution Algorithm With Opponent Model Learning: Results for the Fighting Game AI Competition», *IEEE Transactions on Games*, vol. 15, no. 1, pp. 5–15, Mar. 2023, arXiv: 2003.13949 Publisher: Institute of Electrical and Electronics Engineers Inc., ISSN: 2475-1510. DOI: 10.1109/TG.2020.3022698. [Online]. Available: https://ieeexplore.ieee.org/document/9190073 (visited on 06/05/2024).

[95] J. Ansel, E. Yang, H. He, *et al.*, «PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation», in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf.

[96] V. Mnih, A. P. Badia, M. Mirza, *et al.*, «Asynchronous Methods for Deep Reinforcement Learning», in *International Conference on Machine Learning*, arXiv:1602.01783 [cs] version: 2 type: article, arXiv, Jun. 2016. DOI: 10.48550/arXiv.1602.01783. arXiv: 1602.01783 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1602.01783 (visited on 04/19/2024).

[97] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, «Proximal Policy Optimization Algorithms», arXiv, Tech. Rep., Aug. 2017, arXiv:1707.06347 [cs] type: article. DOI: 10.48550/arXiv.1707.06347. arXiv: 1707.06347 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1707.06347 (visited on 04/19/2024).

[98] S. Huang, A. Kanervisto, A. Raffin, W. Wang, S. Ontañón, and R. F. J. Dossa, «A2C is a special case of PPO», arXiv, Tech. Rep., May 2022, arXiv:2205.09123 [cs] type: article. DOI: 10.48550/arXiv.2205.09123. arXiv: 2205.09123 [cs.LG]. [Online]. Available: http://arxiv.org/abs/2205.09123 (visited on 06/04/2024).

[99] R. Liang, Y. Zhu, Z. Tang, M. Yang, and X. Zhu, «Proximal Policy Optimization with Elo-based Opponent Selection and Combination with Enhanced Rolling Horizon Evolution Algorithm», in *2021 IEEE Conference on Games (CoG)*, ISSN: 2325-4289, vol. 2021-August, Aug. 2021, pp. 1–4. DOI: 10.1109/CoG52621.2021.9619146. [Online]. Available: https://ieeexplore.ieee.org/document/9619146 (visited on 11/08/2023).

[100] S. Huang and S. Ontañón, «A Closer Look at Invalid Action Masking in Policy Gradient Algorithms», *The International FLAIRS Conference Proceedings*, vol. 35, May 2022, arXiv:2006.14171 [cs, stat], ISSN: 2334-0762. DOI: 10.32473/flairs.v35i.130584. [Online]. Available: http://dx.doi.org/10.32473/flairs.v35i.130584 (visited on 04/19/2024).

[101] Y. Zhu and D. Zhao, «Online Minimax Q Network Learning for Two-Player Zero-Sum Markov Games», *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 3, pp. 1228–1241, Mar. 2022, ISSN: 2162-2388. DOI: 10.1109/TNNLS.2020.3041469. [Online]. Available: https://ieeexplore.ieee.org/document/9292435 (visited on 06/05/2024).

[102] R. Nijhawan, «Motion extrapolation in catching», en, *Nature*, vol. 370, no. 6487, pp. 256–257, Jul. 1994, ISSN: 1476-4687. DOI: 10.1038/370256b0. [Online]. Available: https://doi.org/10.1038/370256b0 (visited on 03/31/2024).

[103] S. Nath, M. Baranwal, and H. Khadilkar, «Revisiting State Augmentation methods for Reinforcement Learning with Stochastic Delays», in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, ser. CIKM '21, arXiv:2108.07555 [cs, eess, math], Virtual Event, Queensland, Australia: Association for Computing Machinery, Oct. 2021, pp. 1346–1355, ISBN: 9781450384469. DOI: 10.1145/3459637.3482386. [Online]. Available: https://doi.org/10.1145/3459637.3482386 (visited on 04/09/2024).

[104] M. Towers, J. K. Terry, A. Kwiatkowski, *et al.*, *Gymnasium*. [Online]. Available: https://github.com/Farama-Foundation/Gymnasium.

[105] J. Hoffstadt and P. Cothren, *Hoffstadt/DearPyGui*, original-date: 2020-05-28T17:12:57Z, May 2024. [Online]. Available: https://github.com/hoffstadt/DearPyGui (visited on 05/11/2024).

[106] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, «Stable-Baselines3: Reliable Reinforcement Learning Implementations», *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021, ISSN: 1533-7928. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html (visited on 06/05/2024).

[107] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, «Attention Is All You Need», arXiv, Tech. Rep., Aug. 2023, arXiv:1706.03762 [cs] type: article. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762 [cs.CL]. [Online]. Available: http://arxiv.org/abs/1706.03762 (visited on 04/28/2024).

[108] M. Pîslar, D. Szepesvari, G. Ostrovski, D. Borsa, and T. Schaul, «When should agents explore?», arXiv, Tech. Rep., Mar. 2022, arXiv:2108.11811 [cs] version: 2 type: article. DOI: 10.48550/arXiv.2108.11811. arXiv: 2108.11811 [cs.LG]. [Online]. Available: http://arxiv.org/abs/2108.11811 (visited on 06/05/2024).

[109] M. Yuan, «Intrinsically-Motivated Reinforcement Learning: A Brief Introduction», arXiv, Tech. Rep., Jun. 2022, arXiv:2203.02298 [cs] type: article. DOI: 10.48550/arXiv.2203.02298. arXiv: 2203.02298 [cs.LG]. [Online]. Available: http://arxiv.org/abs/2203.02298 (visited on 04/19/2024).

[110] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, «Exploration by Random Network Distillation», arXiv, Tech. Rep., Oct. 2018, arXiv:1810.12894 [cs, stat] type: article. DOI: 10.48550/arXiv.1810.12894. arXiv: 1810.12894 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1810.12894 (visited on 04/19/2024).

[111] B. Eysenbach and S. Levine, «Maximum Entropy RL (Provably) Solves Some Robust RL Problems», arXiv, Tech. Rep., May 2022, arXiv:2103.06257 [cs] type: article. DOI: 10.48550/arXiv.2103.06257. arXiv: 2103.06257 [cs.LG]. [Online]. Available: http://arxiv.org/abs/2103.06257 (visited on 04/19/2024).

[112] R. Chunduru and D. Precup, «Attention Option-Critic», arXiv, Tech. Rep., Jan. 2022, arXiv:2201.02628 [cs] type: article. DOI: 10.48550/arXiv.2201.02628. arXiv: 2201.02628 [cs.LG]. [Online]. Available: http://arxiv.org/abs/2201.02628 (visited on 06/05/2024).

[113] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap, «Mastering Diverse Domains through World Models», arXiv, Tech. Rep., Apr. 2024, arXiv:2301.04104 [cs, stat] type: article. DOI: 10.48550/arXiv.2301.04104. arXiv: 2301.04104 [cs.AI]. [Online]. Available: http://arxiv.org/abs/2301.04104 (visited on 04/19/2024).

[114] S. Nashed and S. Zilberstein, «A survey of opponent modeling in adversarial domains», *Journal of Artificial Intelligence Research*, vol. 73, pp. 277–327, Jan. 2022, ISSN: 1076-9757. DOI: 10.1613/jair.1.12889. [Online]. Available: https://dl.acm.org/doi/10.1613/jair.1.12889 (visited on 04/19/2024).

[115] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, «Optuna: A next-generation hyperparameter optimization framework», in *The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2623–2631.

APPENDIX A

Additional content

Extra details about the work, omitted from the main content.

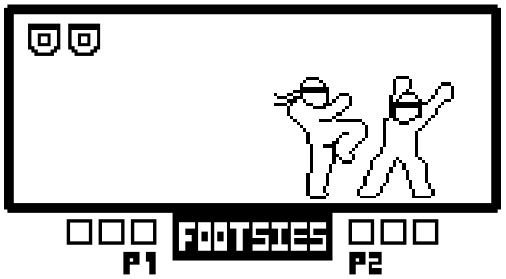
A.1 The *FOOTSIES* environment

FOOTSIES is a simple fighting game that captures the essence of the genre. Players are able to move throughout the screen, attack and perform special moves. If a player is walking backward when they are about to be hit, they block the attack/special move. When an attack hits or is blocked, the player can choose to attack again to immediately perform a special move in sequence. However, if a special move is blocked, then the player becomes highly vulnerable, and open to attacks from the opponent without being able to perform anything, including blocking. As such, it is important that when the opponent receives an attack, the player determines whether the opponent was hit or not before proceeding with the special move, which involves reaction. This practice is colloquially known as "hit confirming".

Contrary to usual fighting games however, *FOOTSIES* has a slightly different objective: to hit the opponent with any special move. Additionally, no time limit is present. Each player has three points of health, which depletes every time an attack or special move is blocked. Once a player blocks an attack or special move with zero health, they become vulnerable for a very long period of time, making them unable to act before the opponent does, which usually results in a loss since the opponent will be able to attack without the player being able to block. Therefore, although the objective is not merely to deplete the opponent's health to zero, reducing their health directly aids in achieving the goal.

A.1.1 State space

Figure A.1 presents the structure of the game's states. It is important to think about how these variables are provided as input to the artificial neural networks (ANNs). There are three discrete variables: health, the player's move and its current frame. The player's health can be considered discrete, which would mean a situation in which player 1 has three health



Default

	Player 1	Player 2
Health	2	0
Position	0.85	3.03
Move	B_ATTACK	GUARD_BREAK
Move frame	3	0

Normalized

	Player 1	Player 2
Health	0.66	0
Position	0.18	0.66
Move	6	13
Move frame	0.14	0

Figure A.1: *FOOTSIES*'s state structure. Player 1 is on the left, and player 2 is on the right. Player 1's health is at the top-left of the screen, while player 2's is at the top-right. Position corresponds to the player's horizontal position, with zero being the center and the right side of the screen having positive values. The move each player is performing has a set duration, and the move frame indicates how many frames/time steps it has been since the move started.

would be completely distinct from a situation in which they have two. However, with this formulation we do not take into account the order relation between different health values such as $2 < 3$. For this reason, and to allow more generality considering usual fighting games feature much larger health ranges, we treat health as a continuous variable. The move frame variable has the same problem, and as such we treat it as continuous as well. Additionally, we transform it so that it is normalized to a number between zero and one by dividing it by the total duration of the respective move. Otherwise, we would have different ranges of values for different moves, which can make training of ANNs harder. Therefore, the player's move is the only variable we consider discrete, for which we perform one-hot encoding on the move's integer identifier. Finally, the health and position variables are normalized to the ranges $[0, 1]$ and $[-1, 1]$ respectively, as illustrated in Fig. A.1.

A.1.2 Action space

A single action in *FOOTSIES* is a triple of three primitive actions: "left", "right" and "attack". Figure A.2 presents all possible moves that can be performed using the primitive actions. Different combinations of primitive actions and sequences of those combinations yield different moves. The `DASH_BACKWARD` and `DASH_FORWARD` moves are not considered special moves, despite requiring a sequence of actions. The special moves `N_SPECIAL` and `B_SPECIAL` require holding the attack primitive action for at least 60 frames, during which any move from the top row can be performed. The action space we consider is simplified: instead of having the agent act using the primitive actions, we have the agent directly perform the desired move by executing the shown action sequences. As such, the total duration of a simplified action becomes the resulting move's duration in addition to the primitive action sequence that was needed to perform that move. Do note that even though for special moves a player can perform any of the moves on the top row while holding the primitive attack, we have the agent merely stand still throughout the duration. Decision making while attempting either special move would have been better handled by a hierarchical policy, which we leave for future work.

A final note worth mentioning is that, for this work, we remove the introductory time

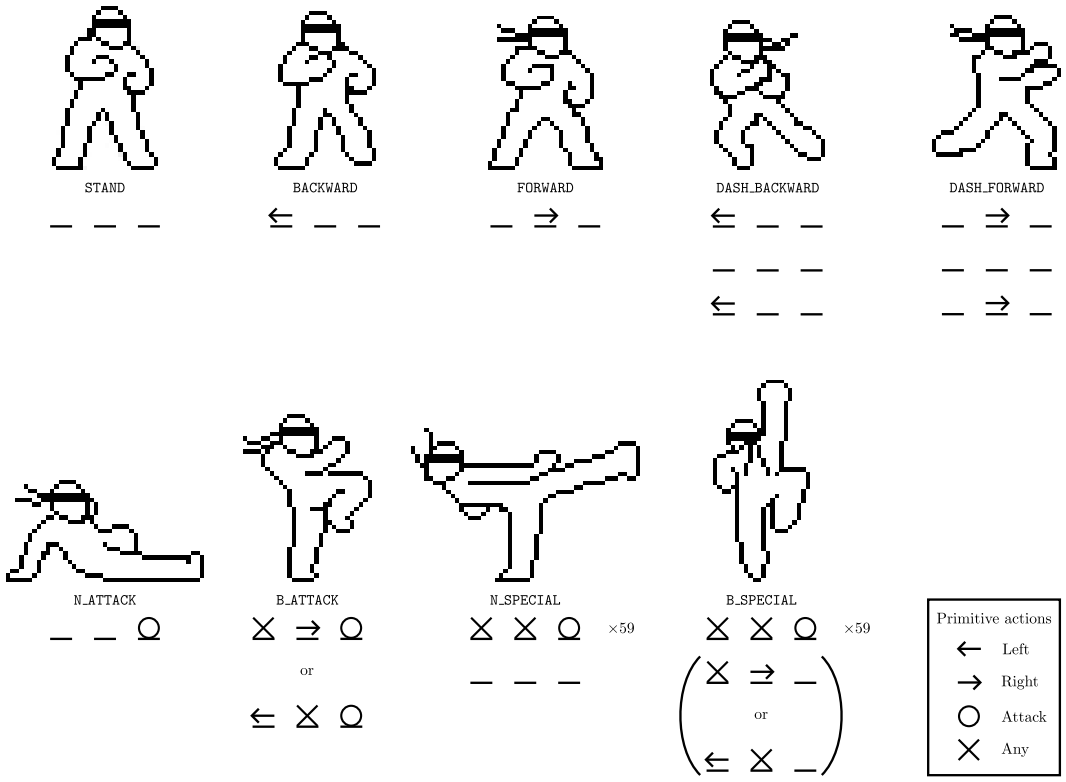


Figure A.2: *FOOTSIES*'s action space. There are three primitive actions and nine total moves that can be performed from combinations and sequences of those actions. A full action is a tuple of each primitive action. An empty space in the tuple stands for no input. The "any" action means the respective primitive action is either performed or not. Note that technically input detection for the `DASH_FORWARD` and `DASH_BACKWARD` actions is slightly more intricate than what is shown, but the general intention is shown for brevity.

before each round starts, and so the game starts immediately. During this introductory time, no player can act, but they can still perform combinations and sequences of primitive actions in order to execute one of the moves in Fig. A.2 as soon as the game starts. Since we consider a simplified action space, we ignore this initial period.

A.2 Gradient-based optimization

In this section we briefly introduce the mechanism by which we update the ANNs. ANNs are parameterized functions f_θ, with a set of parameters θ that we can change in order to represent different functions. In Fig. 2.4 for instance, the nine connection weights are the parameters θ. Let's consider the following function as an example

$$f_\theta(x) = \sigma(mx + b),$$

with $\theta = \{m, b\}$, $m, b \in \mathbb{R}$ and σ being the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Then, we can have the network approximate some ideal function f. For that, we can utilize *gradient-based optimization*. Basically, we have some numerical objective $J(\theta)$ that is

evaluated for some specific function f_θ. This objective is itself a function of the network's parameters, that returns some metric that we wish to either maximize or minimize. The goal is to find a set of parameters θ^* that minimizes, or maximizes depending on how we frame the problem, the objective $J(\theta)$ according to

$$\theta^* = \arg\min_\theta J(\theta), \tag{A.1}$$

in the case of minimization, with maximization utilizing the arg max operator instead. One common example of such an objective, mainly for supervised learning (SL) tasks, is to have $J(\theta)$ be the error/distance between the predictions of f_θ and the actual values of f, for each x. In this case, we want to find the function f_θ that minimizes $J(\theta)$. In these situations, we may also call the objective $J(\theta)$ the "cost" or "loss", since it is a value we wish to minimize. For instance, $J(\theta)$ could be the squared error to emphasize large differences, as in

$$J(\theta) = \int (f_\theta(x) - f(x))^2 dx. \tag{A.2}$$

With a clear numerical objective, we can then update the parameters θ utilizing the gradient of J with respect to θ, according to

$$\nabla J(\theta) = \left[\frac{\partial J(\theta)}{\partial \theta_1}, \frac{\partial J(\theta)}{\partial \theta_2}, \ldots, \frac{\partial J(\theta)}{\partial \theta_n} \right], \tag{A.3}$$

in case we have n parameters in θ. The gradient $\nabla J(\theta)$ essentially indicates, for each parameter θ_i, the partial derivative of $J(\theta)$ with respect to θ_i, which in turn indicates in which direction should θ_i change in order to increase our objective $J(\theta)$ the most, assuming the other parameters are fixed. For instance, if $\frac{\partial J(\theta)}{\partial \theta_1} = -1.5\theta_2$, then it means decreasing the value of θ_1 will increase the value of $J(\theta)$ as long as θ_2 is positive, with the magnitude of this increase being $1.5\theta_2$. If we have this gradient, then we can perform gradient *descent* to minimize $J(\theta)$ (with gradient *ascent* being for the case we maximize $J(\theta)$), using the update rule

$$\theta \leftarrow \theta - \alpha \nabla J(\theta), \tag{A.4}$$

with each parameter θ_i being updated according to their partial derivative, and α being the learning rate, a value between 0.0 and 1.0 indicating how large should the update be. With large learning rates we are able to learn faster, but it may also make learning more unstable, so a trade-off is made here. This update rule is analogous to (2.8). Note that we subtract in (A.4) since the nudges indicated by $\nabla J(\theta)$ are meant to maximize $J(\theta)$, so we need to go in the opposite direction. We then apply (A.4) iteratively until convergence.

However, in practice, we cannot have an exact value for $\nabla J(\theta)$, both because we do not have all the data to even compute $J(\theta)$ in (A.2) (if we did, we would not need to get an approximate function f_θ) and because we cannot evaluate $\nabla J(\theta)$ on all the data. In practice, we only use some example input-output pairs $(x, f(x))$, and then have the ANN iteratively learn with these examples. These examples are not exhaustive, but we assume to be representative enough of the function we wish to approximate. As such, (A.2) becomes

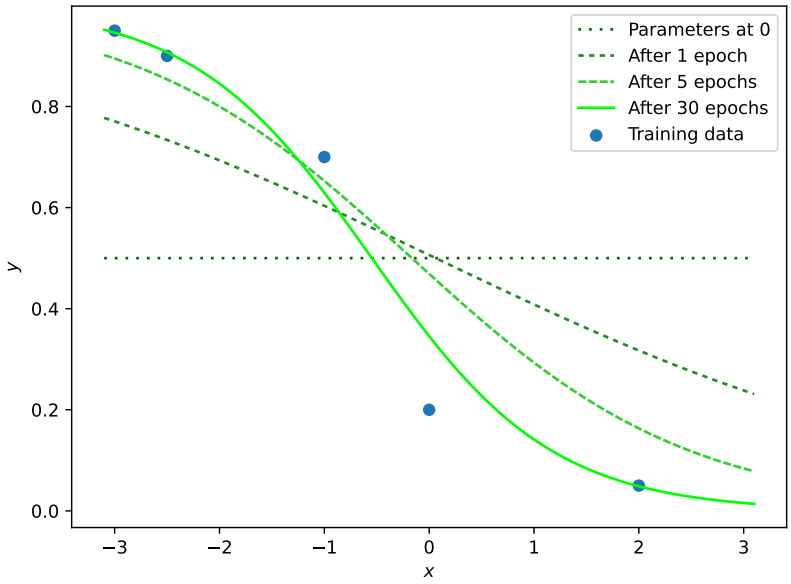


Figure A.3: Training of $f_\theta(x) = \sigma(mx + b)$ using gradient descent, mean-squared error and $\alpha = 0.3$ on example training data.

$$\tilde{J}(\theta) = \frac{1}{N} \sum_{i}^{N} (f_\theta(x_i) - f(x_i))^2, \tag{A.5}$$

which is also known as the mean-squared error. We then use the gradient of this approximate objective for updating the ANN. We can also have the ANN learn with an approximation of $\nabla J(\theta)$ calculated not from the entirety of the examples before each update, but from only a subset (*mini-batch* gradient descent/ascent) or from a single example (*stochastic* gradient descent/ascent). Figure A.3 shows the previously defined function f_θ being updated to approximate a function defined by the points, which serve as examples of the ideal function, using stochastic gradient descent and mean-squared error for 30 epochs. An "epoch" stands for a whole run over the training examples. Therefore, each of the five examples was seen 30 times in total. Algorithm 1 shows how this can be performed simply using the PyTorch library.

Prime examples of gradient-based learning are the policy gradient (PG) methods introduced in Subsection 2.1.3. There, the objective $J(\theta)$ represents the expected discounted return since the beginning of the episode, which the agent wishes to maximize. However, $\nabla J(\theta)$ cannot be computed in practice since we cannot evaluate it on all states, so we need to settle for an approximation, which can be obtained using the many PG methods that exist. Gradient-based optimization is essential in learning all components used in this work.

```python
# Import PyTorch
import torch as T

# Define the function with its parameters
m = T.nn.Parameter(T.tensor(0.0))
b = T.nn.Parameter(T.tensor(0.0))

def f(x):
    return T.sigmoid(m * x + b)

# Create the optimizer, which will handle parameter updates
optim = T.optim.SGD([m, b], lr=3e-1)

# Define the training data
train = [
    (-3, 0.95),
    (-2.5, 0.9),
    (-1, 0.7),
    (0, 0.2),
    (2, 0.05),
]

# Train
for epoch in range(30):
    for x, y in train:
        optim.zero_grad()       # reset the internally stored gradient
        j = (f(x) - y) ** 2     # compute the objective
        j.backward()            # compute the gradient
        optim.step()            # update the parameters
```

Code 1: Training of $f_\theta(x) = \sigma(mx + b)$ on example training data using PyTorch.

A.3 Evaluation hyperparameters

This section presents the hyperparameters used for the algorithms evaluated in Chapter 5. Tuning was performed on the subset of hyperparameters that we deemed important, by having the agents maximize win rate against *FOOTSIES*'s in-game artificial intelligence (AI) over 100K time steps with dense reward, using the Optuna library [115]. Unless specified otherwise, during evaluation, the sparse reward scheme was used and special moves were removed, since in a preliminary round of results collection we found those to be the most performant options. Additionally, during the opponent's decision skipping periods, we assume their current action is their last valid one. For fairness, the ANN architecture was the same for all algorithms. We use two hidden layers with a leaky Rectified Linear Unit (ReLU) activation function for all components, with 128 neurons for each hidden layer in the critic and 64 neurons for each hidden layer in all other components. From manual experimentation, we found the need to have more parameters for the critic than for the other components.

Tables A.1, A.2, A.3 and A.4 present the hyperparameters for our solution, Proximal Policy Optimization (PPO), Advantage Actor Critic (A2C) and Deep Q-Network (DQN), respectively. Emphasized parameters were tuned, and all unspecified parameters in the Stable Baselines3 implementations were left to their default values[1]. For DQN, we experiment with

[1]Default values on library version 2.1.0.

Component	Parameter	Value
Actor	Alternative advantage formula	True
	Action masking	False
	Actor entropy coefficient	0.036
	Actor learning rate	0.090
	Actor gradient norm clipping	0.500
Critic	*Critic learning rate*	0.050
	Target network update interval	Every 1000 updates
Actor-critic	Opponent decision skip	Last valid action
	Agent decision skip	True
	Discount factor	1.0
	Skip hitstop/blockstop	True
	Opponent update style	Expected Sarsa
Opponent model	Recurrent	True, with context reset at "end"
	Recurrent model hidden state	64
	Dynamic loss weights	False
	Opponent model learning rate	0.040
	Opponent model entropy coefficient	0.420
Reaction time	Observation correction method	Multiple models (15, 20, 25 steps)
	Game model method	Differences
	Game model learning rate	0.001

Table A.1: Our solution's best hyperparameters, after 1008 tuning trials.

Parameter	Value
Learning rate	0.011
GAE λ	0.584
Entropy coefficient	0.022
Value function coefficient	0.245
Discount factor	1.0

Table A.2: PPO's best hyperparameters, after 332 tuning trials.

the discount factor due to the smaller number of hyperparameters to tune compared with the other algorithms. Through tuning, a discount factor close to zero was used, which made the algorithm myopic despite this being the most successful configuration.

Parameter	Value
Learning rate	0.030
GAE λ	0.453
Entropy coefficient	0.098
Value function coefficient	0.967
Discount factor	1.0

Table A.3: A2C's best hyperparameters, after 481 tuning trials.

Parameter	Value
Discount factor	0.001
Learning rate	0.042
Polyak averaging coefficient	0.627

Table A.4: DQN's best hyperparameters, after 803 tuning trials.

A.4 Self-play

The opponent pool for self-play was built using snapshots of the agent taken once every 2000 games, to allow the agent plenty of time to learn. The agent switches to a new opponent of the pool every 100 games, to avoid overfitting to a particular one and making learning other opponents harder. Through sampling, it is possible that the newly chosen opponent is the same one that was just used. We additionally allow the current opponent to be immediately skipped if the agent has a win rate of 80% over 20 consecutive games, to speed up training when the agent is playing against worse opponents. The maximum number of opponents in the pool is 10, and *FOOTSIES*'s in-game AI is always included as a choice when sampling a new opponent, not counting toward the 10 opponents. As such, there are effectively 11 opponents to choose from when the pool is full. Whenever a new snapshot is made and the opponent pool is full, the oldest snapshot is discarded.

For the opponent selection strategy, we use fictitious self-play (FSP), choosing any opponent from the pool with uniform probability. We could have employed prioritized fictitious self-play (PFSP) as in [58], [99], which takes some performance metric such as Elo into account when selecting opponents, for instance by prioritizing opponents against which the agent has more difficulty winning. Since we were not sure the Elo calculations were being correctly performed among the pool, and so were not sure the Elo values would be accurate, we opted for the simpler strategy of FSP, which still provides benefits over simple self-play. Therefore, we use Elo merely as an evaluation metric.

The default Elo of any opponent is 1200. At the beginning of training, the opponent pool is empty, so the in-game AI and the agent have 1200 Elo each. During training, the Elo of the agent and the current opponent are updated at the end of every game. In order to have more accurate values for both the agent and the whole opponent pool as well, every 100 games we have the agent play 20 test games against each opponent that has ever been used in training, including opponents that had been discarded previously. Elo is updated at the end of each of

these test games. This way, we build an opponent pool where every opponent has played at least once with any other opponent or the agent. Note that this would not be the case if the opponent pool had been initialized with opponents, or we used more opponents other than the in-game AI; in that case, we would need those opponents at the time of initialization to play among themselves in order to have correct Elos before training.

A.5 Curriculum learning

To improve the agent's training and also to perform evaluation, we created a curriculum of rule-based opponents. These opponents are completely deterministic, and are merely meant to impose certain behaviors on the agent in an attempt for it to understand how to play the game. The opponents are the following, in order of increasing difficulty:

1. `Idle`: opponent that just stays still;
2. `Backer`: opponent that walks backward, which causes it to block all attacks automatically;
3. `NSpammer` and `BSpammer`: opponent that only attacks, and walks forward for a moment before each attack, with an opponent for each type of normal attack;
4. `NSpecialSpammer` and `BSpecialSpammer`: opponent only walks backward and performs special moves, with an opponent for each type of special move;
5. `WhiffPunisher`: highly defensive opponent that not only attempts to perform "whiff punishing", i.e. attack the agent when they miss an attack, but also attacks the agent whenever they leave themselves vulnerable, such as when they perform a special move and the opponent blocks;

The agent then learns the curriculum by playing against each opponent in sequence. When used for evaluation, we allow the agent to skip the current opponent if they take too long to achieve over 70% win rate over the previous 100 episodes against it. This was done to ensure agents do not get stuck into a specific opponent in the curriculum, since we set a fixed number of time steps for learning. We do this except when forming a pre-trained agent, for self-play for instance, where we allow the agent to take as long as necessary to achieve the win rate threshold.

A.6 Analysis tool

To aid in debugging, a tool for data visualization and troubleshooting was developed using DearPyGui, a toolkit for building performant graphical user interfaces using Python [105]. We found DearPyGui to be a good choice due to its ease of development and built-in plotting functionality. Such a tool is especially useful considering the difficulty in debugging reinforcement learning (RL) algorithms, as it is often hard to understand the source of a given issue, or to know if there is even an issue at all. Figure A.4 shows a screenshot of this tool. It allows checking the output of each component, as well as manipulating the environment state, either manually or by saving and loading previous states. In addition to this, other environment metrics such as reward, win rate and the actions of each player are also shown.

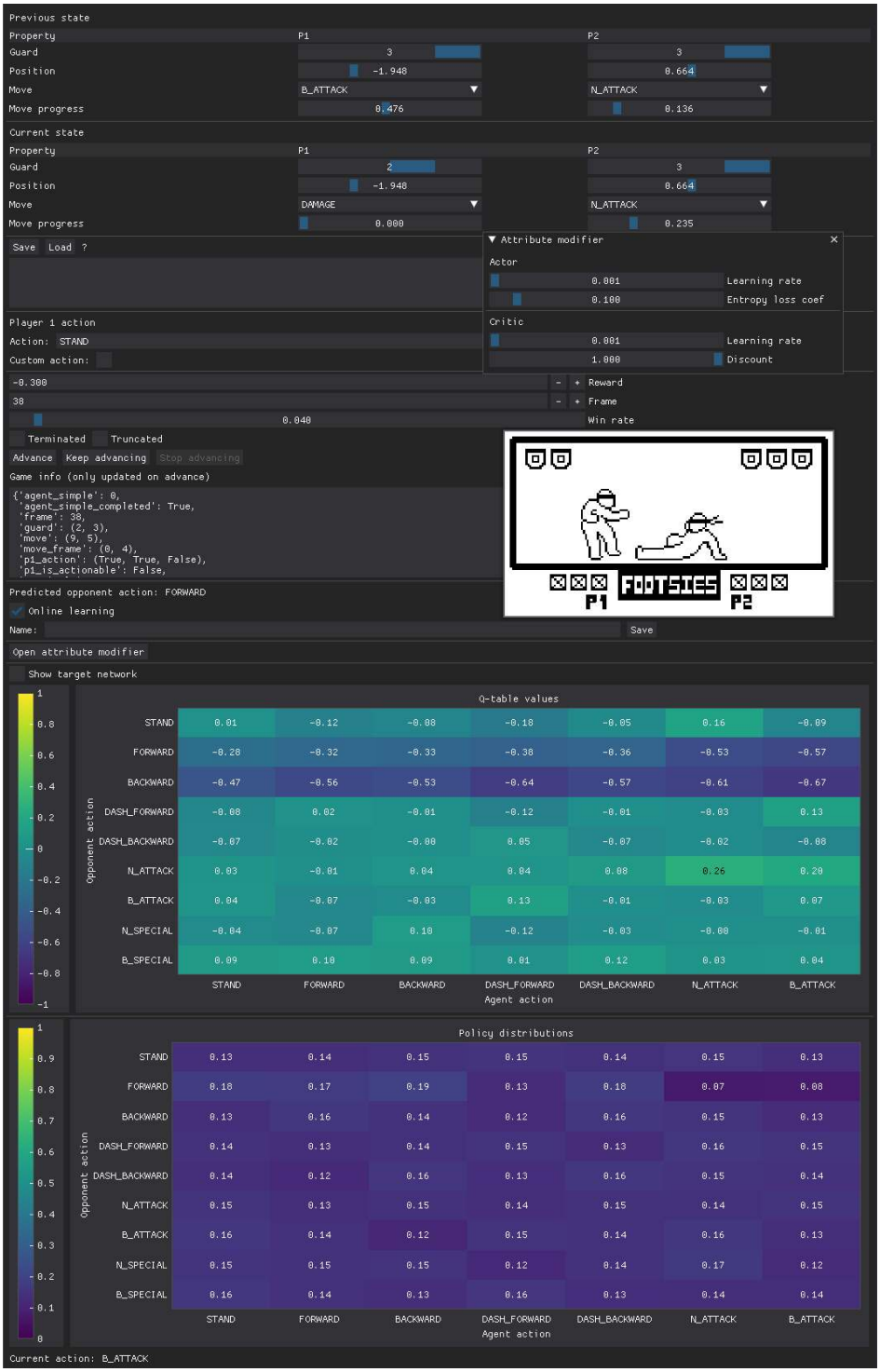


Figure A.4: Screenshot of the analysis tool, along with a view of the current game state. At the top are the structure of the previous and current environment states, with the current state being editable. Then, metrics about the interaction such as reward are shown, along with controls for advancing the environment. Afterward, the outputs of each component are shown along with some custom controls, such as the attribute modifier for the actor-critic component which allows changing hyperparameters at runtime. Of note, the matrix output of the action-value function and policy are shown, considering each opponent action. The agent's action space has excluded special moves, hence why there is a reduced number of columns.