JÖNKÖPING UNIVERSITY
School of Engineering

Enhancing video game experience with play time training and tailoring of virtual opponents

Using Deep Q-Network based Reinforcement Learning on a Multi-Agent Environment

PAPER WITHIN *AI Engineering*
AUTHORS: *Nishant Pillai, Roberto Giaconia*
TUTOR: *Lars Carlsson*
JÖNKÖPING *June 2023*

This exam work has been carried out at the School of Engineering in Jönköping in the subject area AI Engineering. The work is a part of the two-year Master of Science in Engineering programme. The authors take full responsibility for opinions, conclusions and findings presented.

Examiner:	Vladimir Tarasov
Supervisor:	Lars Carlsson
Scope:	30 credits
Date:	2023-06-08

Mailing address:	Visiting address:	Phone:
Box 1026 551 11 Jönköping	Gjuterigatan 5	036-10 10 00 (vx)

Abstract

When interacting with fictional environments, the users' sense of immersion can be broken when characters act in mechanical and predictable ways. The vast majority of AIs for such fictional characters, that control their actions, are statically scripted, and expert players can learn strategies that take advantage of this to easily win challenges that were intended to be hard. Games can also be too hard or too easy for certain players. Through the means of Reinforcement Learning, we propose a method to train adversaries in a simple environment for a game of tag from the PettingZoo library, exploring the possibility of such modern AIs to learn during the game. Our work aims towards a new concept of continuously learning AIs in video games, giving a framework to greatly increase adaptability of products to their users, and replayability of the challenges offered in them. We found that our solution allows the agents to learn during the game, but that more work should be done to achieve a model that tailors the behavior to the specific player. Nonetheless, this is an exploratory step towards more research on this new concept, which could have numerous applications in many genres of video games.

Contents

1 Introduction

Video games are a form of art and entertainment that allows the user to interact with a fictional environment. Like for other forms of art, the creators of these environments take inspiration from real-life to some degree. In some aspects of the production, the designers and developers in this industry have the tools to very closely imitate reality, from 3D rendering to spatial audio, from complex narratives to systems that make the characters react to the players' actions, making the plot more realistic and alive. While some of these tools and features are expensive to produce, the interest they are able to attract is high enough to encourage many actors of this industry to invest in them, in games often informally referred to as "AAA" or "triple A" titles [1], often featuring photo-realism. A popular example can be found in Figure 1.

Figure 1. A rendered frame from the Triple A title "Ghostwire: Tokyo"

One aspect that is still relatively limited in its ability to achieve realism is the Artificial Intelligence (AI) of Non-Player Characters (NPCs). Depending on the genre of video game, these NPCs could be secondary to the experience, interacting with the player to provide information or small challenges, or they can be the primary actors of the plot, working alongside the players or against them. Regardless of their importance, the AI of these NPCs is generally statically scripted. In many cases, learning the order of the actions of the opposing NPCs is the main way to beat the game. In some popular instances, expert players can manipulate the behavior of their adversaries by executing actions in a certain way. For instance, guides exist for the popular arcade game Pac-Man on the static AIs of the ghost characters [2].

Attempts to substitute these static AIs with modern solutions exist in the scientific literature (Bodas et al., 2018), but adoption in the industry is very limited since Machine Learning models can be seen as unreliable when put in an interactive environment for end customers. Moreover, a typical solution would be to train a model before the market release of the product and provide its final version to players. This requires a very large amount of game-play data and can result in a black-box for the developers, who might instead want to change the agent behavior depending on various conditions. Additionally, delivering a modern AI solution that does not learn anymore once released does not prevent players to find easy exploits, that break the illusion of real-life intelligent NPCs.

[1] Steinberg, Scott (2007). The definitive Guide: video game Marketing and PR (1st ed.). iUniverse. ISBN 978-0-59543-371-1.
[2] https://strategywiki.org/wiki/Pac-Man/Tips#Ghost_psychology

Although scripted and learned AIs can be viable solutions for many cases, a novel concept of continuously learning NPC should be studied, which could offer automatic tailoring of the behavior to individual players, reducing the risk for them to find permanent vulnerabilities of the algorithms, creating a new illusion of intelligence and realistic reactions to their actions. The ultimate objective will be to determine whether such a solution can be valuable for commercial products, but the first steps should still be made as, to the best of our knowledge, this possibility is currently very understudied.

For this purpose, in this study, we propose and test an environment that resembles a very simple video game and uses a modern AI solution to control opposing agents, while the player is emulated with various static scripts that implement different strategies. Modern AI solutions that have already been proposed in the game field were studied for the most part in board games, and their general objective has been to create very expert players of the given game, with the most notorious examples beating human masters in competitions (Silver et al., 2017). Our focus is, instead, on user experience and entertainment, and doing research towards new tools and solutions for the video game industry that allows it to include continuously learning AI for NPCs.

1.1 Aim and Scope

To introduce the concept of continuously learning AI for video game NPCs, the specific scope of this study is limited to this two-sided question: "Can AI for opponents in video games learn and tailor themselves to different player strategies during play time?".

Firstly, the AI will need to learn how to challenge the player inside the game environment. This means learning the rules and possibilities of the environment, that are going to be learned starting from scratch and while the game is running. Secondly, this solution must show that the result of the learning process depends on the specific strategy adopted by the player, resulting in a better overall score compared to agents that learn from many players or from individual players with different strategies.

In this study, the research question will be separated into two specific hypotheses, as a way to better define what is addressed by the experiments that we are going to report. The first hypothesis states: "Reinforcement Learning can make the AI controlling enemies in a video game learn to play during user play time". The second hypothesis states: "Training enemy agents with Reinforcement Learning against a specific player strategy will result in behaviors that are more effective against that strategy but less against others".

2 Related Work

In this section, we will first give a brief introduction to the theory behind Reinforcement Learning and the specific field where the work presented in this study is placed. Then, we write a summary of the main findings in the current scientific literature that inspired our work. Lastly, we include a basic description of the main tools used to implement our solution and conduct experiments.

2.1 Theory

Reinforcement Learning (RL) is a subset of Machine Learning that deals with decision-making problems in which an agent interacts with an environment over time, taking actions to maximize a variable called *reward*. The agent observes the state of the environment, which is a representation of its current condition. It includes all the information that the agent is able to observe, such as the location of objects or the state

of other agents, in order to make decisions. The agent may take the action found by the decision algorithm and then observe the state again. The strategy or rules that the agent uses to determine which action to take is called its policy. The environment provides feedback to the agent in the form of rewards or penalties based on the actions the agent takes. The reward is a scalar value that indicates how well the agent's action aligned with its goal. This process is one of the fundamental concepts of RL and it's called the agent-environment cycle (AEC)(Sutton & Barto, 2005). Some environments may have a terminal state when the objective of the game is reached. In such cases, time steps in the environment can be broken down into episodes and certain aspects of the environment maybe reset to the initial state.

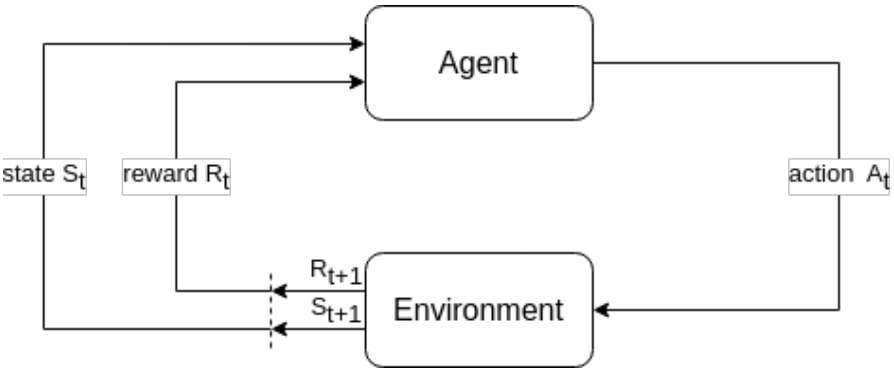

Figure 2. Agent-environment cycle (Sutton & Barto, 2005)

AEC is an iterative process in which the agent interacts with the environment to learn and improve its policy. The cycle begins with the agent observing the state of the environment and selecting an action. At each time step t, the agent observes a representation of the state at time step $S_t \in S$, where S is the set of possible states. The agent then selects an action $A_t \in A(S_t)$, where $A(S_t)$ is the set of actions available at state S_t. Finally, the cycle moves to the next time step and the environment transitions to a new state S_{t+1}. Based on its action, the agent receives a numerical reward $R_{t+1} \in \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers. In a multi-agent environment, one cycle in an episode refers to one AEC cycle when all agents have selected an action and the environment gives them respective rewards.

The objective of the agent is to maximize the cumulative reward it receives in the long run. This is called expected return G_t and is formulated as

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$

where T is the final time step. However, many environments may not have a terminal state and therefore have continuous rewards with no time limit. In this case, $T = \infty$ and the value of future rewards needs to be discounted. That way, a reward received in the future is worth less than a reward received immediately. The discount factor γ determines the extent to which future rewards are valued relative to immediate rewards. The expected discounted return G_t is formulated as

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{k+t+1},$$

where $0 \geq \gamma \geq 1$.

At each time step, the agent selects an action based on its policy π_t, which is a function that maps the agent's observation of the environment to an action to be taken. To maximize the expected return function G_t, the agent needs to find its optimal policy. To achieve this objective, an agent needs to explore the environment enough to know what paths and actions lead to better rewards. If the policy is set too early, the agent might not explore the environment completely and get stuck in a local maximum of rewards. To

avoid this, exploration-exploitation algorithms, like the Epsilon-greedy algorithm, are used. The Epsilon-greedy algorithm works by choosing between the optimal (greedy) action and a non-optimal (exploratory) action based on a probability value ε. So action taken at a time step is

$$A_t = \begin{cases} a_{\text{random}} & \text{with probability } \varepsilon \\ a_\pi & \text{with probability } 1-\varepsilon \end{cases}.$$

At each agent decision point, a random number is generated between 0 and 1. If this number is less than epsilon, then a random action is chosen otherwise the optimal action is chosen using the current policy. A higher value of epsilon will result in more exploration, while a lower value of epsilon will result in more exploitation. For a multi-agent environment, all the agents have their own policies and they choose their actions at the same time, while the rest of the AEC framework remains the same.

To learn the optimal policy, an agent needs to identify which states and actions are better than others, i.e. lead to better rewards. To accomplish this, a state-value function and an action-value function are defined. The state-value function is a function that estimates the expected return from a particular state under a given policy. It shows how good a state is and it is formulated as

$$V_\pi(s) = E[G_t|S_t = s].$$

Similarly, the action-value function shows how advantageous it is to take action a in state s under policy π. It is formulated as

$$Q_\pi(s,a) = E[G_t|S_t = s, A_t = a].$$

This is also known as the Q-value. It is a measure of the expected cumulative reward that an agent can obtain by taking a specific action in a given state. It represents the quality of a particular action in a particular state. Q-learning is a model-free RL algorithm that is based on using these Q-values to learn a policy (Watkins, 1989). In Q-learning, the agent learns to estimate the optimal Q-value for each state-action pair in the environment. The agent updates its Q-value estimates using the Q function,

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha(R_{t+1} + \gamma \times max_a(Q(S_{t+1},a)) - Q(S_t,A_t)),$$

which approximates the optimal action-value function q^*. This equation is derived from the Bellman equation of optimal action-value function (Sutton & Barto, 2005),

$$q^*(s,a) = E[R_{t+1} + \gamma \times max_{a_{t+1}} q^*(s_{t+1},a_{t+1})|S_t = s, A_t = a],$$

which expresses the recursive relationship between the optimal Q-value function at the current state-action pair and the maximum expected cumulative reward over all possible actions in the next state. Using this, a Q table is generated for all possible states and actions as shown in Figure 3 (a).

When the action space and environment state are finite and narrow, the learning process consists of populating a lookup table, called a Q-table, of expected reward values. However, this can be computationally expensive or impractical for environments with large state and action spaces. In such cases, a function approximator, like Artificial Neural Networks, can be used to replace the lookup table as shown in Figure 3 (b). This is the case for Deep Q-Networks, where a Deep Neural Network is used to learn the expected rewards.

Artificial Neural Networks (ANNs) are nets of threshold elements, inspired by animal brains. These combinations of connected elements work by feeding a set of input values and letting the first group, or layer, of elements determine an intermediate set of values, by applying a weighted sum of the input of every element and a function to the result, called *activation*. The intermediate result is in turn fed forward to the next layer. Once the final layer is reached in this process, the network returns the resulting vector of elements. These networks are normally self-organizing thanks to a learning algorithm (Amari, 1972), typically by providing a set of input-output couples and applying a back-propagation system to change the weight values. Many kinds of activation functions may be applied in the threshold elements. Some examples include a step function centered on the threshold value, a sigmoid function, and a rectified linear unit (He et al., 2015), which will be used in our experiments.

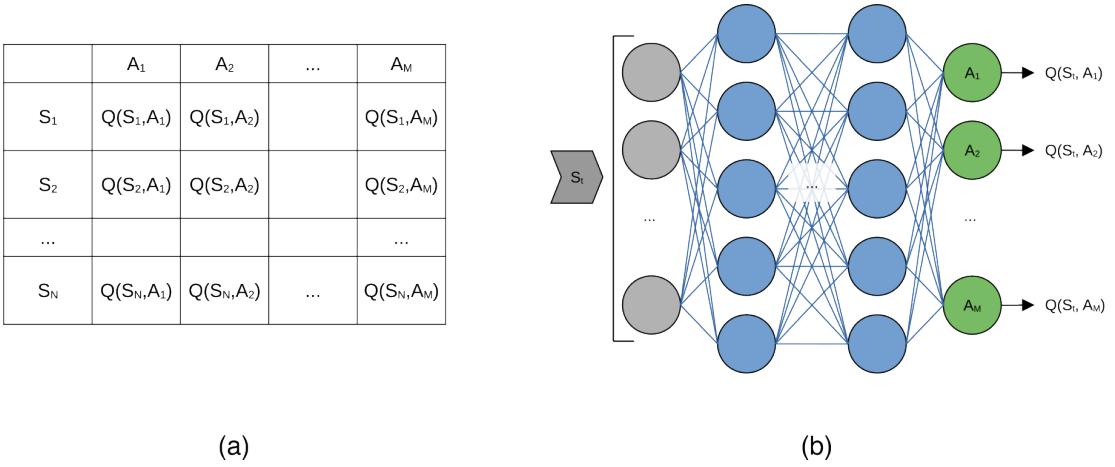

Figure 3. Q-table method (a) and Deep Q-Networks method (b)

Deep Q-Networks (DQN) is a powerful variant of the Q-Learning algorithm that combines deep neural networks with Q-Learning to approximate the optimal action-value function $Q(s,a)$ (Mnih et al., 2015). The neural network takes the current state of the environment as input and outputs the expected future reward for each possible action. The neural network is trained to minimize the difference between the predicted Q-values and the target Q-values, which are calculated using the Bellman equation with a target network that is periodically updated to improve stability. By using a deep neural network to approximate the Q-value, DQN is able to learn a more complex and high-dimensional representation of the environment, which allows it to solve more challenging tasks than Q-learning.

One of the challenges of using ANNs as an approximator is that the learning process can become unstable and constantly switch between different policies. This can happen because the neural network is constantly changing as the Q-value is updated, which can cause the target to shift and lead to feedback loops and overfitting. Two techniques that help to stabilise the learning process are the Experience replay method and the use of a target network.

Experience replay is a technique that stores a collection of past experiences of the agent, known as the replay buffer. The sequence of experience collected by an agent can be highly correlated because the current state of the environment and the action taken by the agent depend on the previous state and action. However, the training data for ANNs needs to be i.i.d. (independent and identically distributed) to prevent action values from diverging too much while training. An experience is formulated as

$$E_t = (S_t, A_t, R_{t+1}, S_{t+1}).$$

Such experiences are stored in the replay buffer of an agent. During training, instead of using the most recent experiences, DQN randomly samples batches from the replay buffer of the agent. This reduces the correlation between successive updates and stabilizes the learning process by preventing the agent from

overfitting to recent experiences (Mnih et al., 2015).

Another technique used to stabilize the training is the use of a target network. The target network provides a more stable and consistent target for the Q-value updates, which improves convergence and reduces fluctuations in the learned policy. It is a copy of the Q-network but with a fixed set of weights that are updated more slowly. Training a deep ANN requires minimizing the error estimated by the loss function by optimizing the weights. Huber loss function (Huber, 1964) was used leveraging on the predicted Q-value and the Q-value from the target network, which is treated as the "ground truth".

2.2 State of the Art

One of the main approaches for video game NPCs with modern AI is the use of ANNs trained using an Evolutionary Algorithm. Similarly to the Deep-Q Learning approach, the inputs of the ANN are the observation of the agent, and the outputs constitute the action selection. However, in this case, the weights are chosen and evolved with an open-ended algorithm that mimics genetic processes. In (Risi & Togelius, 2017), the authors discuss many advantages of this approach, called *Neuroevolution*. Some are shared with RL, such as the ability to apply the solution to a broad field of problems. For other aspects the advantages are exclusive to Neuroevolution, such as supporting a large action and state space. Several examples are then presented and discussed. In particular, most Neuroevolution solutions function by having the ANN learn the relationship between the current game state and possible actions, to then guess which action would lead to a better state for the agent. This works best on environments that give full information on the state to agents, such as in Chess, and many video game environments that also fit this description despite not being turn-based, like Pac-Man or the environment used for the experiments of our study. The authors also discuss different approaches for what kind of input should be given to the ANNs controlling the agents, such as sensors of relative position to other objects in the environment, which applies to our case study. As concluded in the review, the comparison of Neuroevolution to other solutions such as RL is an open question.

Another type of evolutionary technique for agents learning to act intelligently is Genetic Programming (GP). In GP, the behavior is scripted by code found with a Genetic Algorithm, rather than depending on a ANN. In (Smith & Heywood, 2019), GP was found to be a viable solution to create a competitive player for a complex video game. In particular, the authors used the Tangled Program Graphs framework, where "programs" link a specific action to the context, and the algorithm works by connecting groups of programs in a graph. RL was used to evaluate individual nodes of the graph. The base method was originally designed for games with complete state information, but the case of this study deals with partial observability. Thus, a memory model was needed, and in this case external indexed memory was selected, that is, a memory used with read and write instructions, and that is never completely reset during the learning process. The memory is also divided in short- and long-term sections, achieved by giving write operations a lower probability to use addresses of the long-term memory: the algorithm then learned to use the regions accordingly for short and long term information. This study shows a possible modern solution to create expert models of modern video game even at a high level of complexity. It requires a long training session: the authors let the algorithm run for about two weeks, although the environment was constrained to a single core at 4 GHz.

Learning for games is not necessarily done for just one objective, to create an optimized artificial player. Other research sub-fields focus on different reasons to train a game agent, which require different methods. Even when the objective is to create a better performing model, it is important to make a distinction between learning to play well against any generic opponent or learning to give a certain level of challenge to a specific user. The authors of (McIlroy-Young et al., 2022) argue, indeed, that training a generally optimized player does not always lead to the desired experience. Humans are not optimal, we have biases, and in a game like Chess, where predicting the opponent's moves is essential, understanding and adapting the prediction to the biases of a specific person leads to better accuracy. Furthermore, the results of their

study have the potential to create an educational tool for the human player. The specific method chosen for this experiment was conceptually simple: they used transfer learning from an existing popular chess AI solution, to tune the model enough to learn to predict specific players' choices from their past games, rather than assuming that the players act all similarly or optimally. The authors tested if the transfer learned models were actually recognizing just the specific player they were tailored for, by running the model to predict actions of other players, and they did find a decrease in accuracy. However, it should be considered that the authors only found an increase in accuracy, compared to the base version of the chess AI solution, when the number of games used to tune was more than 4,000.

The concept of learning towards a different objective than winning the game has also been explored. For example in (Bodas et al., 2018), using an RL solution. The authors define a new reward function based on a concept outside of the environment itself: player engagement. Using the environment for the popular game Pong, they assume that users will be more engaged when a point is contended for more time between them and the virtual opponent, before one of them scores. Their solution is shown to be able to keep individual players interested for longer, sometimes two to four times, than when playing against a generic agent. The authors of this study, like the ones of the study introduced at the previous paragraph, also decided to tune an existing model trained with the objective to win. The agents would then tailor their behavior with the new reward function during game play. This is done to prevent a loss in player interest that could happen during the initial part of the training, where the agent would explore possible actions randomly. In this kind of game, random behavior would result in an uninteresting game where the human player always wins. This shows that it is really important to consider the nature of the environment from a game design perspective to make an informed choice on the kind of learning dynamics.

Many video games do not simply create a virtual challenge for the user, they create a realistic or imaginary environment, they tell a story and present many characters. This creates an interest in making agents that act in a human-like fashion. The AI model should not simply try to win the game in an optimal way, and in (Zhao et al., 2020) a few methods are presented to achieve more human results: a modified A* algorithm that proved to be effective on a simplified version of a mobile game, a model-free RL algorithm in a multiplayer mobile game that features delayed rewards, a compressed DNN model that learns from game demonstrations in a Human-in-the-loop approach, a RL method with a complex reward system to make cooperative agents that complement a team of players to compete against a competent team of human players. The authors show and stress that the potential of current RL solutions used in benchmark environments does not translate directly on production examples, they need an extensive amount of tuning, hampering research efforts. A final example of solutions that models human player behavior is found in (Pfau et al., 2018), where algorithms are once again trained to predict and replicate the choices of human players, with an approach targeted towards game production in big companies. In general, this approach could be taken into consideration when designing NPCs that, in the narrative context of a game, are supposed to be already rather expert but still make mistakes.

In addition to the experience we collected from the literature about trained and tailored NPC AIs, it should be noted that our project deals with an environment that features not one agent but many, with the same goal. An example of an RL approach for such a task is found in (Grupen et al., 2022), where the authors focus on the effects of curriculum-driven learning on communication between the agents. In particular, since each agent could see the positions of the others, a behavior of implicit signaling emerged. While this does not constitute a deliberate communication between agents, it still allows for coordination, since the choices of one agent partially dictate actions for others. The training was decentralized, and this sparked the need for ways to stabilize learning. One way was the Velocity Ratio curriculum, meaning the speed of the pursuers has been decreased over time, from being the same as the evader, to lower values. They also introduced a Behavioral curriculum, consisting of a policy that collects single-agent experience in the environment, and then a standard exploration policy for the multi-agent one. When training is done in a centralized way, this problem should not arise since the ANN is joint or shared between the agents. Communication and coordination are still possible, the authors simply preferred decentralized learning for their task of reproducing more natural settings. Thus, we do not think that the learning policies should be included in our environment, as they could constitute a confounding factor while gathering results.

2.3 Tools

In our experiment, we use two famous frameworks: PyTorch to design, instantiate, train, and use the ANN, and PettingZoo to create and interact with the game environment.

PyTorch is, at the core, a library for dynamic tensor computations with automatic differentiation and allows for GPU acceleration, and proved to be very popular in Deep Learning research fields. It supports Python, following the trend of the scientific community to use free software and the Python programming language in particular. The library is updated to new Machine Learning trends often, thanks to the simple internal implementation. Finally, it implements its own multiprocessing solution and it supports interoperability with other frameworks (Paszke et al., 2019).

PettingZoo is a library designed to make Multi-Agent RL research easier and more standardized. The authors took great inspiration from a Single-Agent virtual environment framework that is very popular in research, Gym by OpenAI, and decided to replicate its API to promote wide adoption. As for the environments themselves, the key feature introduced with PettingZoo compared to other multi-agent solutions was the Agent Environment Cycle: agents actions, observations and rewards are dealt in a turn-based fashion. The library comes with many reference example environments (Terry et al., 2021).

Additionally, the Ray parallelization framework [3] is used to run multiple versions and instances of the experimental tasks. Our environment and learning algorithm are run on Ray Core instances. The Ray Tune features are also used for the hyper-parameter optimization that is going to be introduced in the next section.

3 Methodology

As mentioned in the introductory Subsection 1.1, our work will address two different hypothesis through our experiment. For the first hypothesis, our experiment limits the learning time to the time actually spent by a human player on the game. Existing solutions generally offer a pre-trained AI that can then be optimized to new goals or specific situations and users. Instead, we attempt to cut the pre-training step to allow for greater game designing freedom and potentially to improve the level of customization of the produced strategies.

When training virtual players of existing videogames, it can be possible to retrieve existing game play data, or to train agents against single-player game modes that were made available by the original developers, like in (Smith & Heywood, 2019). Instead, when developing a new game that features a learning agent, any training data needs to be generated by the developers, either by having human play-testers help tackle the initial pre-training of the agents, or by training the learning agents against traditional statically scripted ones. Training the model from scratch when the user starts playing could be a different methodology, that we want to focus on in this study.

The second hypothesis directly correlates to specific tests, very similarly to the final comparison tests in (McIlroy-Young et al., 2022) for chess, with the main difference being that our approach does not feature pre-training or transfer learning. A description of the environment and an explanation of how the experiments address the two hypotheses follow.

[3] https://docs.ray.io/en/latest/ray-overview/index.html#ray-framework

3.1 Experimental setup

We took inspiration from an environment of the PettingZoo python library and we modified it to better fit our needs and remove potential confounding factors. The original environment, called *Simple Tag*, is a top-down two-dimensional world with a continuous observation, discrete action space and discrete time. It consists of one good agent, that we are going to refer to as *player*, and three adversary agents, that we will call *enemies*. Their starting positions are randomized on the field at the start of every experiment, called *episode*. The field also features many obstacles. There were two obstacles in randomized positions in the original version of the environment, instead for our version we put nine obstacles in fixed positions. A rendered representation of our version of the environment is shown in Figure 4.

The objective for the player is to avoid being hit by the enemies, while the objective for the enemies is to hit by the player. The environment also has landmarks, represented by black circles, that agents cannot move over. The environment also has simulated physics for movements and collisions. The player can accelerate faster and has a higher maximum speed than the enemies. This is done to balance out the advantage of enemies outnumbering the player. All agents can have elastic collisions with each other or with the landmarks.

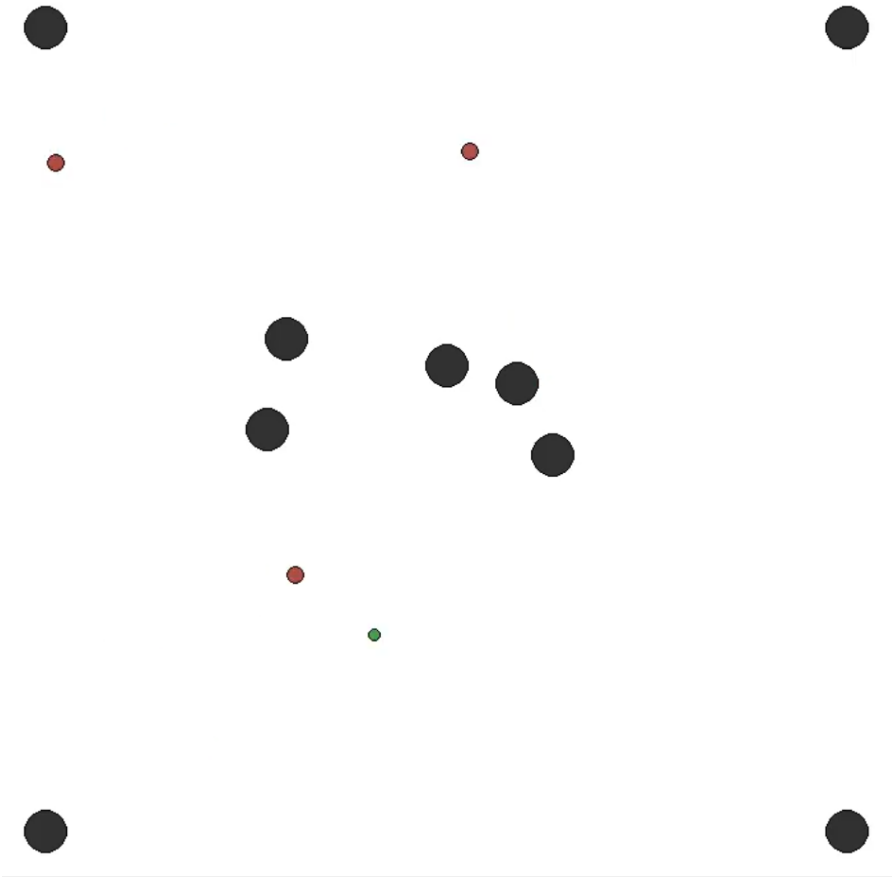

Figure 4. The custom environment. The player is the green circle, while the red circles are the enemies. The bigger black circles are landmarks, they are placed in the fixed positions shown here.

The only learning agents in our experiment are the enemies, and they are rewarded based on how close they are to the player. The player moves 30% faster than the enemies and is controlled by a static AI. Contrarily to the original environment, the experiments run until a specified number of cycles, and does not get interrupted by a condition. Even if the agents collide, the game continues to run.

After every action, the environment produces a reward value that is going to be used both within the RL algorithm and as a metric. By analysing the change in average rewards over time we will address the first hypothesis. We will run the test several times in the same environment, sometimes changing the setting for the static AI of the player, from straightforward ones to more complex behavior. These different settings are described in the next subsection. We will focus on two aspects to qualitatively relate the results to the player engagement, that affects how long a user is going to play. For the Pong agent in (Bodas et al., 2018), pre-training was an important addition to keep the player interested during the initial phases of learning. Our aim is to not require any pre-training, so we will focus on the comparison between the average rewards in the initial exploratory phase and those reached after a relatively small number of cycles. We will also analyze the average reward improvement speed before it reaches its plateau. Contrarily to our static AI player, a human player will also learn to play the game during play-time, so an initial random behavior can be accepted as long as the improvement comes fast enough to keep the player interested. Both of those measures are arbitrary and are mostly presented as conceptual benchmarks.

For the second hypothesis, another kind of test is provided. For each of the static player strategies, a model is trained for a long session, that allowing the enemies to score high rewards. Each model is then frozen and tested against the other player strategies, on rotation. The distribution of the rewards obtained in these tests will be analyzed to compare the results on different strategies than the one each model was trained on. We will make use of Welch's t-test to study statistical significance as it is more robust to the violation of the assumption of normal distribution compared to other tests (Colas et al., 2022).

The reward distributions are not strictly Gaussian, as we will see in the results section, and we identify this as a potential threat to the validity of a positive answer to the second hypothesis. Nonetheless, if we reach a positive conclusion, we will compare our findings with the chess experiment in (McIlroy-Young et al., 2022); since the two environments are fundamentally different, this comparison will be qualitative.

While the topic of our work is training NPCs while they are facing human players, our tests will be run with simulated player characters. We neither possess nor aim to obtain data about actual human gameplay experience in this work. Our environment is also a simple benchmark, compared to the complexity of environments and rules normally found in video games. Thus, our work functions as a proof of concept, and our positive findings need to be then tested on actual game scenarios.

3.2 AI Algorithms

The AI for the enemies is a shared neural network model trained with Deep Q-Learning. At each step, each agent observes the environment and uses the state information on the network to infer an action. It then collects the reward which is used to update the network weights during training. The optimization step of the network takes into account a memory stream containing observations, actions, and rewards from all enemies together. The agents are thus controlled by the same mind, producing inferences simply based on different observation points.

The player features a number of different AIs, from simple to complex. For every experiment, the environment assigns a specific AI setting. Four static settings were designed and implemented for our study, and they are called *Random*, *Evasive*, *Hiding* and *Shifty*.

The Random AI simply selects one action using a random number generator, without using the observed stare. The Evasive AI is straightforward: it calculates the average of the directional vectors to the enemies and moves in the opposite direction. The enemies simply need to learn how to surround or corner the player. The Hiding AI will try to use the group of obstacles at the center of the environment as a separation between the player and the enemies. This is very similar to the Evasive AI but it uses radial coordinates, so the enemies might learn to do an additional step of translation of coordinates, or they might find ways to confuse the player movement by finding strategic flaws in the algorithm. The Shifty AI will try to use

hard-coded positions (close to obstacles, especially in between them) to decide where to move. This way the movement is less dependent on enemies positions. Enemies must devise clever strategies, increasing the need for collaboration or coordinated actions.

These static AIs are further discussed in the following subsections, and their algorithms are shown in the Appendix.

3.2.1 Evasive Player

This algorithm tries to make the play get away from other elements, giving different levels of importance to enemies, landmarks and the four sides of the environment area. The importance, calculated separately for the x and y coordinates, is defined as

$$I = \sum_{e \in E} \frac{1}{p_e} + 0.1 \sum_{l \in L} \frac{1}{p_l} + 5 \sum_{w \in W} \frac{1}{p_w + 0.01\,\mathrm{sgn}(d_w)},$$

where p_e, p_l and p_w are the positions of an enemy, a landmark or a side (or wall) of the environment, relative to the player. This way, their sign is considered. E, L and W are the sets of enemies, landmarks and walls respectively.

These two measures of importance are used to determine if it is more important to move horizontally or vertically. The coordinate with the highest absolute value is taken into account, and the action returned by the algorithm is the one that makes the player move away from the elements in that direction. For example, if I of the x coordinate has a higher absolute value than the one of the y coordinate, the player will move horizontally. If the value is negative, the player will move towards positive x, which means to the right.

3.2.2 Hiding Player

The map contains various landmarks around the center, and the objective of this player is to separate itself from the enemies using that group of landmarks. This is achieved by using a polar coordinate system, with the center of the map as the focal point. Positions of all the agents are thus converted to polar coordinates.

For each enemy, the radial direction away from it is calculated, so if the enemy is closer to the focal point than the player, the direction is away from the focal point, and vice versa. For the tangential movement, the direction that would put the player π radians away from the enemy is considered.

A measure of importance between the two polar coordinates is calculated by dividing the two distances by their maximum values (π for the tangential movement, and $2\sqrt{2}$ for the radial one), and then calculating the inverse of the result, for each enemy. Additionally, a measure of closeness is calculated for each enemy by using the inverse of the Euclidean distance (back in the Cartesian space) of them to the player. Each enemy radial and tangential importance measures are multiplied by their closeness.

The resulting set of 6 values are used to get a value of radial direction and another of tangential direction, by summing the radial values and the tangential values. We are interested in the rate between the two and their sign. A vector containing the two sums is created, normalized and divided by the maximum of the absolute values, thus one will equal ± 1 and the other will be lower in absolute value.

The two values are then multiplied by the maximum values (π and $2\sqrt{2}$) times 0.3. Then, they are summed to the current polar coordinates of the player to get a target polar position. This is then converted

to Cartesian coordinates, and the player will be instructed to choose the action that brings it closest to it.

However, in case the player's radial position is over 0.98, the player will randomly be instructed to instead use the center of the field as the target position. The probability for this to happen is set at the current radial position divided by 2.

3.2.3 Shifty Player

This strategy consists in using specific positions that a human player would often use: in between the landmarks and at the sides of the environment area.

Firstly, the distance between the player and the closest enemy is calculated. The player will fall back to Evasive behavior, on a cycle-by-cycle basis, with a probability given by that minimum distance, the closer the enemy, the more likely the player will try to simply evade. Otherwise the main Shifty algorithm is processed.

The specific positions are hard-coded: three positions in between the five landmarks that are close to each others, to create some sort of tunnel between them, and a position at center of each of the four sides of the environment, for a total of 16 positions.

At each cycle, the positions are sorted by distance to the player. First, the closest three and the furthest two are selected and evaluated. For every position, the distance to its closest enemy is taken into consideration, and the position with the highest of these distances is selected as the target position.

If, however, the distance to the closest enemy is too low, below 0.5, the evaluation is re-done for all 16 positions rather than the selected five. So the target will simply be the position that is currently furthest away from its closest enemy.

The end result is a human-like evasive behavior, that prefers specific positions rather than the one that is technically furthest away from all enemies.

3.2.4 Random Player

This strategy is straightforward: at each cycle, the player randomly chooses an action between staying still and moving in one of the four directions, with each action having the same probability.

3.3 Training agents

Agents are trained using DQN. Each experiment is made of an environment that runs for a predefined number of episodes with a fixed number of cycles in each episode. Every experiment has its own DQN network training separately to other experiments. There are two types of experiment conducted:

- *Long-term* tests, where the experiment runs for 100 episodes with 1000 cycles in each episode, making a total of 100,000 time steps.
- *Play-time* tests, where the experiment runs for one episode with 5000 cycles, making a total of 5000 time steps.

Colas et al., 2022 recommends that the sample size should be greater than 20 in order to show statistical significance. Therefore, 50 instances of a experiment were run in parallel using multi-thread processing

achieved with the Ray library.

Two identical ANNs, policy network and target network, are built for each experiment. The three enemies in the environment are trained and controlled by these networks. The ANN is built on PyTorch, and it is a sequence of fully-connected layers. The input to the network is the observation of a specific enemy, and the output is one of the five possible actions (no action or move in one of four directions). There are three hidden fully-connected layers with 64 neurons each, making a total of five layers with the input and output layer. There are 10,885 trainable parameters in each neural network. The input layer of the ANN takes 34 values, that constitute the observed state produced by the environment for each agent. The first two values are the current horizontal and vertical velocities of the agent. The 2nd and 3rd values are the horizontal and vertical positions of the agent. The following 18 values are the relative positions of the nine landmarks, given in couples of Cartesian coordinates. The next six values are the relative positions of the other three agents. Finally, the last six values are the velocities of those three other agents.

For every cycle, an action is selected using the policy network or from a random action sample using Epsilon-greedy algorithm. The probability of choosing a random action peaks at the start of the episode when 90% of the episode is left and ends when 3% of the episode is left, decreasing exponentially throughout the episode. The environment rewards the agents based on the action chosen. The reward value is made of two elements, distance penalty and collision reward. Originally, the reward system designed in the PettingZoo environment only accounted for collisions between the adversaries and the player. However, with that reward system learning any significant strategy in play time would be difficult because of the sparsity of the rewards, making the learning process slower. Distance penalty was added to mitigate this issue. For each cycle, agents receive a penalty equal to the distance between the player and closest enemy to it. Please note that the player position is forced to stay between x and y values of -1 and +1, however enemies can move outside of that range. A reward value of -1.0 means that the closest enemy is away from the player as much as half the size of a side of the map. The collision reward is given to enemies when one of them collides with the player, for that cycle only. Additionally, the episode does not end when a collision happens, instead rewards are continuously given based on the distance between adversaries and good agent.

The latest experience E_t is then pushed into the replay buffer consisting of 384 samples. Transitions are sampled from the replay buffer into batches of 128 and they are used to calculate the loss using the target network which provides the target Q value for each action. Then, one pass of back-propagation is performed on the policy network using the calculated loss with a learning rate (α) of 1.86e-4. Thereafter, a soft update is also performed on the target network with a learning rate (τ) of 0.0098.

3.4 Hyperparameter Optimization

The performance of a DQN agent can be highly sensitive to the hyperparameters used, such as the learning rate, discount factor, batch size, and exploration rate. Selecting inappropriate values for these hyperparameters can lead to sub-optimal performance of agents. By performing hyperparameter optimization (HPO) we can systematically explore the hyperparameter space and find the set of hyperparameters that result in the best performance.

There are a total of ten hyperparameters available. The architecture of the model is defined by two hyperparameters, *Number of layers* and *Number of neurons*. The input and output of the neural network remain the same but the depth of the network and the size of each layer in the network (i.e. number of neurons in each layer) is defined by these two hyperparameters. ANNs with three to six layers were tested with 16 to 80 neurons in each layer. The search space for the hyperparameter number of neurons was in steps of 16. The best architecture found was a three-layer network with 64 neurons in each layer. There are two hyperparameters for Experience Replay, *Replay memory*, which is the size of the replay buffer, and *Batch size*, which is the number of samples in each batch sampled from the replay buffer for training

the ANN. The search spaces for both were done in integer steps of 128. There are three hyperparameters related to exploration-exploitation strategy, *Episode end*, *Episode start*, and *Episode decay*. *Episode start* defines at what percentage of an episode the agent starts exploring, while *Episode end* defines when it stops. *Episode decay* how quickly the exploration rate decreases over time during an episode. Finally, there are three hyperparameters for the RL algorithm, γ, which, is the discount factor for the expected discounted return, τ, which is the learning rate for soft updates to the target network and α, which is the learning rate of the ANN. Table 1 shows the intervals and the values for the hyperparameters found after HPO.

Hyperparameter	**Intervals**	**Value**
Number of layers	[3, 6]	3
Number of neurons	[16, 80]	64
Batch size	[128, 512]	128
Replay memory	[128, 512]	384
Episode start	[0.8, 0.9]	0.896
Episode end	[0.02, 0.05]	0.034
Episode decay	[150, 2000]	1150
γ	[0.8, 0.99]	0.942
α	[1e-4, 1e-2]	1.86e-4
τ	[0.005, 0.01]	0.0098

Table 1

Intervals and results of HPO

The algorithm for sampling values for HPO is Nondominated Sorting Genetic Algorithm II (Deb et al., 2002). This algorithm was chosen because the search space mentioned in Table 1 is very large. Using other techniques like Bayesian search or Grid search are computationally intensive for such large search spaces and genetic algorithms are preferred over other sampling algorithms (Ozaki et al., 2020). Figure 5 shows the top 100 best combinations of hyperparameters found. The axes of the plot correspond to the values of hyperparameters that were varied during the hyperparameter search. The color of the lines represent the average reward obtained from a DQN agent trained using the specific combination of hyperparameters.

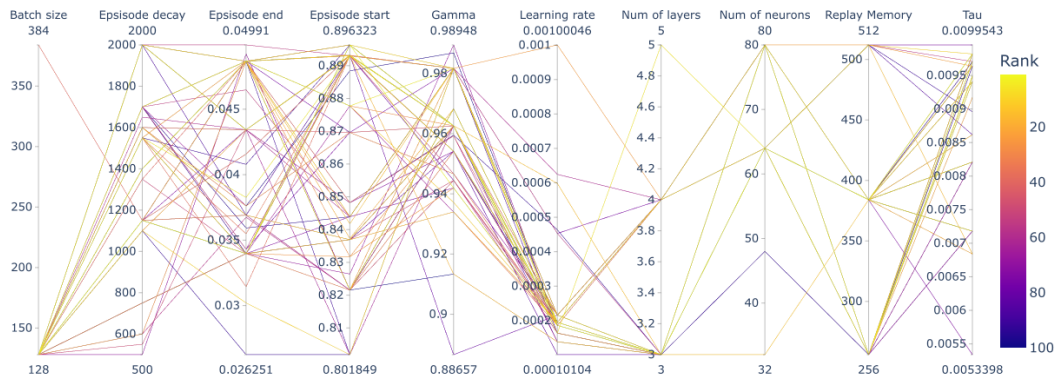

Figure 5. HPO sweep

The hyperparameters used for DQN agents mentioned in the sections 3.3, 3.5 and 4 were found through hyperparameter optimization.

3.5 Statistical Model

As previously mentioned, we test the second hypothesis by training the ANN model on the three statically scripted strategies plus the Random one and by then running the trained models against all other strategies.

As mentioned, we device two kinds of training types: the *play-time* training and the *long-term* training. The former is supposed to show what is the result of learning from a single episode, that can be compared to a human player trying the game for the first time, while the former is comparable to a user playing the game for a longer time on multiple sessions, and this helps to show the trend of the training and to maximize the efficiency of the algorithm.

The two kinds of training are achieved by changing the number of episodes and cycles for the environment: one episode of 5000 cycles for play-time and 100 episodes of 1000 cycles for long-term. The two training types are used in separate tests for both the hypotheses. An example setup for the second hypothesis follows.

For this example, we want to test on the Evasive strategy. We start by training a model on the Evasive strategy. Once the training is done, we freeze the ANN weights and we test the model against the Evasive strategy and the other strategies. We produce a number of samples, each sample is the average reward of a single episode of play. Then we test if the distribution or samples from the test against the Evasive strategy is the same distribution (null hypothesis) as any of the others, or if the rewards against any of the others come from a distribution with a lower mean value.

More specifically, the distributions of samples are obtained by training ten independent models for each strategy and testing each of the networks against all strategies for five independent episodes each. Thus, each matching of models and strategies creates a distribution of 50 values. For the test phase, the length of an episode is set to 500 cycles.

In total, for the play-time test, we run 16 matches, resulting in 16 distributions of samples. For the long-term test, an additional strategy setting is used in the training, called *Multiple*, which trains against a different strategy at every episode cycling between the aforementioned four. So the total number of matches for this test is 20.

Finally, for Welch's t-test, we will consider the null hypothesis to be rejected if the p-value is less than a significance value of 0.05.

4 Results

In this section, we will present the results of the experiments that were executed to address the research questions and hypotheses. An analysis of the results will be presented in the next section. All the results will show average values of the metric, which is the reward value also used in the RL algorithm.

The first hypothesis was tested by letting the models start from scratch and recording their rewards over a relatively short game, to notice when the RL algorithm reaches an improvement over its initial exploratory phase. The concept of time in the environment is expressed in cycles, which are hard to correlate to a human player's perception of time. The results from the play-time training experiments are shown in Figure 6. Each player static strategy has been used to train 100 independent models. The plots show the curve of the mean values of rewards and their standard error. Please note that for each model the array of 5000 reward values has been reduced to 50 values, by calculating the average of 100 cycles for each of the 50 points. This is used as a smoothing function for the curve, to improve readability.

Human players that are satisfied with the initial experience provided might continue playing, so it is

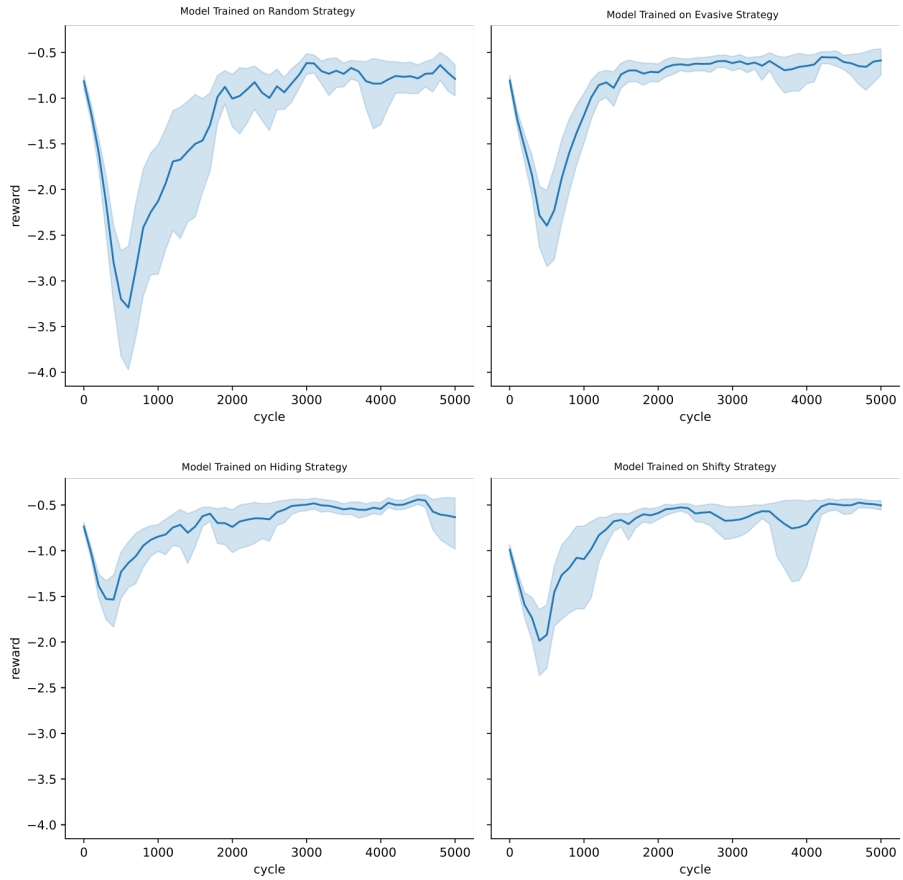

Figure 6. Average rewards of 400 play-time training phases between four player strategy settings, with per-experiment smoothing. During the initial 600 cycles, the enemies are mostly exploring the environment and its rules, then they quickly reach stability around values between -1.0 and -0.5

crucial to also test the long-term improvement of the enemy AI. This is shown in the long-term test, the results of which are presented in Figure 7. The plot shows the curve of the average rewards of each episode and the standard error. For each player static strategy, ten models have been trained over 100 episodes of 1000 cycles (so 20 times the time of the play-time test). Results are consistent with the play-time test, as after five episodes all models reach an average reward value between -1.0 and -0.5. The models that are playing against the Random strategy setting show an extra improvement that was mostly connected to an increased number of collisions since the player AI does not try to get away from the enemies like for the other three strategies. The Multiple strategy is also shown to perform better, but please keep in mind that exponential smoothing was applied to the plot, so the values depend also on the peaks produced on episodes with the Random strategy.

After the models were trained on the long-term test, their weights were frozed to run the final test, that addresses the second hypothesis. The ten models produced by lerning long-term against each strategy setting were used against all four strategies for five episodes of 500 cycles, producing a total of 50 average reward values for each test, and 20 tests. The distributions of these reward values are shown in the violin plots in Figure 8. Each plot contains the results of the models that were trained on each of the five strategy settings, and each violin plot shows the distribution of average rewards of those models against the four strategies. A comparison between the distributions has been performed using the one-sided Welch's t-test and the results are shown in Table 2. Each cell of the table is the comparison of two distributions: the first one is the distribution of rewards obtained by a model trained on the strategy indicated on the left of its row and tested on that same strategy, the second one is the distribution of rewards obtained by that same

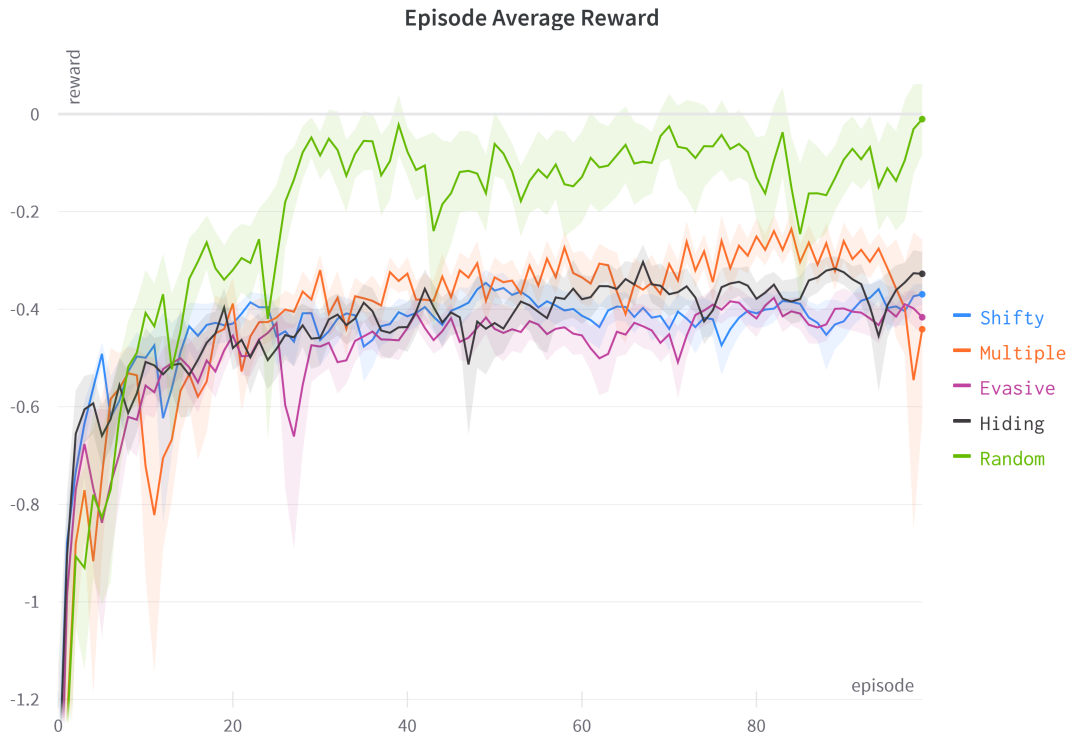

Figure 7. Average rewards of 50 long-term training phases between five player strategy settings, with exponential moving average smoothing. All models reach a reward of -0.5 before the 20th episode (20'000 cycles), and then grow to around -0.4, with the exception of the models learning against the random strategy which reach an average of -0.1 after 30 episodes.

model but tested on the strategy indicated at the top of its column.

The Welch's t-test to compare the generic model trained on the Multiple setting has been run differently, as the intention is to demonstrate if learning from a specific player strategy achieves better scores in the tests against that strategy than a generic model would do. Results from this are shown in Table 3. Each cell is the comparison between two distributions: the rewards obtained by a model trained on the strategy indicated on the left and tested on that same strategy, and the rewards obtained by the generic model trained on multiple and tested on the same strategy indicated on the left.

The same test was also run on models trained for one episode of 5000 cycles (the play-time settings), and the results of this test are shown in Figure 9 and Table 4.

Table 2

p-values of the Welch's t-tests after long-term training. The null hypothesis is considered rejected for this experiment when the p-value is lower than 0.05, which are the cells coloured in gray.

	Random	Evasive	Hiding	Shifty
Random	-	9.5e-12	3.5e-7	2.8e-10
Evasive	0.99	-	0.84	0.19
Hiding	0.64	6.0e-3	-	0.13
Shifty	0.99	0.08	1.0	-

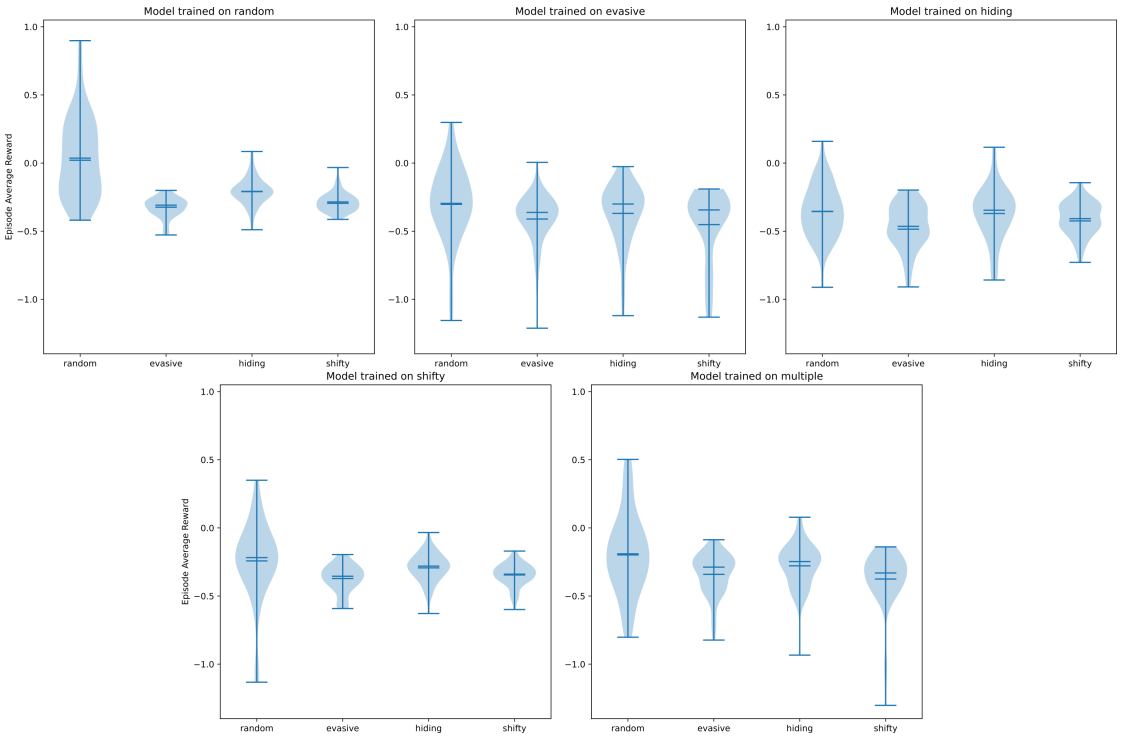

Figure 8. Distributions of average rewards from 50 episodes per test. Ten models have been trained long-term with the five strategy settings, and each model has been tested against all four strategies for five episodes each.

Table 3

p-values of the Welch's t-tests after long-term training comparing models trained on a specific strategy to the model trained on all strategies (the Multiple setting).

	vs Multiple
Random	0.02
Evasive	0.7
Hiding	1.0
Shifty	0.4

5 Discussion

In this section the results will be analyzed to address the research questions and respective hypotheses, introduced in Subsection 1.1.

5.1 Evaluation of the setup

The experimental setup proved to be satisfactory for the purposes of our project. The main challenge has been designing a reward system that would allow the RL algorithm to learn quickly while still letting it reach its own strategic choices. For our Welch's t-test, we identify a possible validity threat, as the violin plots in Figures 8 and 9 indicate that not all distributions present a shape resembling the normal distribution.

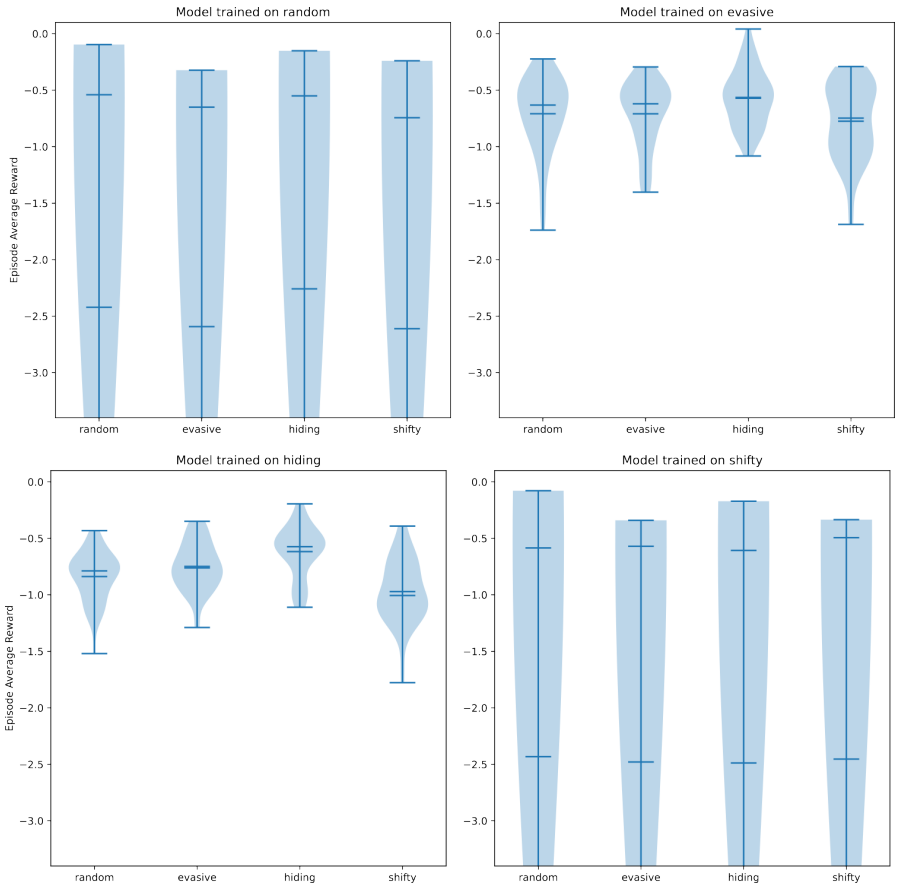

Figure 9. Distributions of average rewards from 50 episodes per test. Ten models have been trained on play-time settings with the four strategies, and each model has been tested against all four strategies for five episodes each.

5.2 First hypothesis

For the first hypothesis, the experiment on play-time training should be considered. The rewards obtained during the training were shown in Figure 6. The curve shows the general trend of the mean value of the rewards during the first-ever training phase of a model that started from random weights. The starting value depends on the starting positions of the player character and the enemy characters, and it corresponds to the distance between the player and its closest enemy. After the initial decrease in rewards, which depends on the random behavior of the agents in the exploratory phase, the value then grows and reaches rewards that are generally higher than the starting value. This shows that a slight improvement was achieved during a short game, on average, against all the implemented static strategies.

Table 4

p-values of the Welch's t-tests after play-time training. The null hypothesis is considered rejected on the cells coloured in gray.

	Random	Evasive	Hiding	Shifty
Random	-	0.44	0.55	0.44
Evasive	0.5	-	0.99	0.15
Hiding	1.9e-6	2.1e-3	-	3.1e-10
Shifty	0.51	0.49	0.49	-

To have an idea of the speed at which the enemies learn to act in a way that can be interesting to the player, we can notice that the number of cycles needed for the reward to return to the starting value is generally 1.5 to three times the cycles that it takes for the exploration phase to reach the negative peak. The enemies were in proximity of the player agent for most of the time, and we consider this to be satisfactory for human players to keep their interest in the game for a short introduction to it. While the user learns to move in the environment, the NPCs also act in random ways to explore the reward rules.

If the play time is extended by the human player, we previously stated that it is crucial that the algorithm is able to continuously learn and improve over many games against the same player. This was tested in the experiment on long-term training, the achieved rewards of which are draw in in Figure 7. The results show an upward trend throughout the experience, for all three static strategies as well as the random setting and the generic multiple setting that cycles between the four other settings after every episode.

These two results show that, in this environment, it was possible to accept the first hypothesis: a modern AI solution for video game opponents can learn to play against a user during their play time, and RL can be a viable method.

5.3 Second hypothesis

For the second part of the research question, we are interested in the ability of our solution to tailor enemy behavior to the specific user, in order to achieve better results. Each learned model was thus tested against all four strategies, both after a long-term training (Figure 8) and after a short play-time training (Figure 9). The distributions showed in the violin plots are further compared using one-sided Welch's t-tests, with results shown in Tables 2 and 4. For our hypothesis to be accepted, the null hypotheses should have been rejected on a large majority of the tests, with perhaps the exception of the Random strategy column of those tables. This was not the case, a positive tailoring could only be observed when training against the Random setting for long-term and in the Hiding setting for short play-time.

Furthermore, our environment did not show the existence of an advantage on training against a specific strategy rather than training against all strategies for the same number of total episodes, as demonstrated by the results of the Welch's t-tests in Table 3. Thus, the second hypothesis of our project is considered rejected, and we could not show that our solution tailors itself to the specific player to achieve better results. All the training experiments achieved models that obtained very similar rewards when put against the other strategies.

Since the second hypothesis is rejected, the comparison to the experience on tailored chess agents by (McIlroy-Young et al., 2022) is very simple: tuning an existing expert model should currently be preferred to learning a new model from scratch if the objective is to achieve a tailored behavior that can better address a player's weaknesses and habits. However, the nature of our environment, which is simple and presents continuous states, allowed us to show that pre-training is not a necessity to feature modern AI opponents in video games.

We believe this experience paves the way for more research that can potentially change the way video game development companies design the AI of NPCs, from static and repetitive behaviors that reduce the replayability of a product, to modern solutions that help maintain the player perceived immersion in the realism of the virtual environment.

It should be noted that the environment and rules of our experiments are very simple. While the observation space is normal, the number of possible actions to choose from is very limited, and the actions themselves are straightforward. Games normally have special actions, that might be available on certain conditions. So our results are mainly relevant for cases in which enemies have a simple logic. In other words, AIs for "boss" characters should be studied separately. Lastly, our environment did not feature a win or lose condition, that terminates a match.

6 Conclusions

This project was designed to assess the possibility for modern AI solutions to replace static scripting for the behavior of fictional characters in video games. The different development approach would be useful and necessary if two important results were to be achieved: the ability for those NPCs to learn to play by themselves and to then tailor their behavior to the specific player, to increase the level of realism and immersion for the user. Through the experiments presented in this report, we were able to achieve a positive answer for the first aspect, while more work should be put into tailoring.

Our experiment used a RL solution to have the opponents of a simple game of tag in a two-dimensional environment learn to approach the simulated player, which would feature various statically scripted strategies instead. We found that training the agents from scratch generally led to intelligent action within a time frame comparable to human play time, both in the case of a short match and when the player continues to try the game in a series of matches. The simple environment was not suited, instead, to show the ability for the agents to tailor themselves to specific player strategies, since training on many strategies rather than a single one led to similar results. Even training on a different strategy did not guarantee lower results.

This study shows that there is a possibility for research in this direction: so far most studies in video game AI focused on training player agents or opponents in games for two equal players, we instead put our attention to an example that is very present in the video game market of one human player against many artificial opponents that are individually weak.

For instance, games that feature hordes of undead characters (such as zombies) are normally designed to let the individual monsters simply walk towards the player character, in a slow and easy game of tag where the player has the advantage of using weapons. With this study, the possibility to improve the experience, by having more intelligent and learning hordes of opponents instead, is explored.

Our work, however, concentrated on an environment with very straightforward rules, especially on the number of actions that the agents were required to learn to use, and on the non-existent winning condition.

To deal with more complex situation, it is not possible to say, to the best of our knowledge, if it is advisable or possible to make the NPCs learn from scratch during user play time. More experiments should be made on other kinds of environments that still feature one or few player characters and many opposing ones that have lower individual capabilities. Furthermore, modern games often feature different challenges where some or all rules change: another study should be made on when and how it is possible to transfer some of the knowledge acquired on one challenge to the others. Lastly, an exhaustive study should be made on the predictability of the results and explainability of the models, especially since the learning phase happens after a product has been released and tried by the users: this would be crucial to help the video game development industry have confidence in the adoption of these solutions.

7 Acknowledgements

We would like to express our gratitude to the Department of Computing at Jönköping University and the Jönköping AI Lab for providing a remote server. This resource provided us the ability to design and perform extensive tests for our research that greatly increased the value of our work.

Many thanks to Florian Westphal and Patrick Gabrielsson for providing us access to the machine and for their technical assistance in maintaining the server. Finally, our gratitudes also goes Lars Carlsson and Vladimir Tarasov for their guidance in the design and plan of the project.

References

Amari, S.-I. (1972).
Learning patterns and pattern sequences by self-organizing nets of threshold elements.
IEEE Transactions on Computers, *C-21*(11), 1197–1206.
https://doi.org/10.1109/T-C.1972.223477

Bodas, A., Upadhyay, B., Nadiger, C., & Abdelhak, S. (2018).
Reinforcement learning for game personalization on edge devices.
2018 International Conference on Information and Computer Technologies (ICICT), 119–122.
https://doi.org/10.1109/INFOCT.2018.8356853

Colas, C., Sigaud, O., & Oudeyer, P.-Y. (2022).
A hitchhiker's guide to statistical comparisons of reinforcement learning algorithms.

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002).
A fast and elitist multiobjective genetic algorithm: Nsga-ii.
IEEE Transactions on Evolutionary Computation, *6*(2), 182–197.
https://doi.org/10.1109/4235.996017

Grupen, N. A., Lee, D. D., & Selman, B. (2022). Multi-agent curricula and emergent implicit signaling.
Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems, 553–561.

He, K., Zhang, X., Ren, S., & Sun, J. (2015).
Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.
2015 IEEE International Conference on Computer Vision (ICCV), 1026–1034.
https://doi.org/10.1109/ICCV.2015.123

Huber, P. J. (1964). Robust estimation of a location parameter. *Annals of Mathematical Statistics*, *35*, 492–518.

McIlroy-Young, R., Wang, R., Sen, S., Kleinberg, J., & Anderson, A. (2022).
Learning models of individual behavior in chess.
Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining.
https://doi.org/10.1145/3534678.3539367

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015).
Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.
https://doi.org/10.1038/nature14236

Ozaki, Y., Nomura, M., & M., O. (2020).
Hyperparameter optimization methods: Overview and characteristics.
IEICE Transactions on Information and Systems, *J103-D*(9).
https://doi.org/10.14923/transinfj.2019JDR0003

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019).
Pytorch: An imperative style, high-performance deep learning library.
In *Proceedings of the 33rd international conference on neural information processing systems*.
Curran Associates Inc.

Pfau, J., Smeddinck, J. D., & Malaka, R. (2018). Towards deep player behavior models in mmorpgs.
Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play, 381–392. https://doi.org/10.1145/3242671.3242706

Risi, S., & Togelius, J. (2017). Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, *9*(1), 25–41. https://doi.org/10.1109/TCIAIG.2015.2494596

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., & Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, *550*(7676), 354–359. https://doi.org/10.1038/nature24270

Smith, R. J., & Heywood, M. I. (2019). Evolving dota 2 shadow fiend bots using genetic programming with external memory. *Proceedings of the Genetic and Evolutionary Computation Conference*, 179–187. https://doi.org/10.1145/3321707.3321866

Sutton, R. S., & Barto, A. G. (2005). Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, *16*, 285–286.

Terry, J., Black, B., Grammel, N., Jayakumar, M., Hari, A., Sullivan, R., Santos, L. S., Dieffendahl, C., Horsch, C., Perez-Vicente, R., et al. (2021). Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, *34*, 15032–15043.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards* (Doctoral dissertation). King's College, Cambridge United Kingdom.

Zhao, Y., Borovikov, I., de Mesentier Silva, F., Beirami, A., Rupert, J., Somers, C., Harder, J., Kolen, J., Pinto, J., Pourabolghasem, R., Pestrak, J., Chaput, H., Sardari, M., Lin, L., Narravula, S., Aghdaie, N., & Zaman, K. (2020). Winning is not everything: Enhancing game development with intelligent agents. *IEEE Transactions on Games*, *12*(2), 199–212. https://doi.org/10.1109/TG.2020.2990865

Appendix

The code for this work is also available at Customized MAE Agents on GitHub

A Player AIs algorithms

The following algorithms, provided in Python3, are executed by the player agent in each cycle. They take the observation space as input and return one action from the following list: *no action*, *move up*, *move left*, *move down*, *move right*.

A.1 Evasive Player

This player's intention is to get away from the enemies or from everything as quick as possible.

```
def evasive_player(state):
  # Getting sum of inverse relative positions
  x_diffs = 0
  y_diffs = 0
  # for landmarks
  for i in range(4, 22, 2):
    x_diffs += 0.1 / state[0][i].item()
  for i in range(5, 23, 2):
    y_diffs += 0.1 / state[0][i].item()
  # for adversaries
  for i in range(22, 28, 2):
    x_diffs += 1 / state[0][i].item()
  for i in range(23, 29, 2):
    y_diffs += 1 / state[0][i].item()
  # for walls
  x_diffs += 5 / (1.01 - state[0][2].item())
  x_diffs += 5 / (-1.01 - state[0][2].item())
  y_diffs += 5 / (1.01 - state[0][3].item())
  y_diffs += 5 / (-1.01 - state[0][3].item())

  # Checking larger difference and moving in opposite direction
  action = ACTIONS["no_action"]
  if abs(x_diffs) > abs(y_diffs):
    if x_diffs > 0:
      action = ACTIONS["move_left"]
    elif x_diffs < 0:
      action = ACTIONS["move_right"]
  elif abs(x_diffs) < abs(y_diffs):
    if y_diffs > 0:
      action = ACTIONS["move_down"]
    elif y_diffs < 0:
      action = ACTIONS["move_up"]

  return action

```

A.2 Hiding Player

This player's intention is to use the group of obstacles as separation from the enemies, so that the enemies are forced to circumnavigate the obstacles. It uses a conversion of Cartesian coordinates to and from polar ones from the center of the field.

```
def hiding_player(state):
  self_pos = cart_to_polar(state[0][2].item(), state[0][3].item())
  enemies_pos = [
    cart_to_polar(  # Note we need the absolute cartesian position here
      state[0][i].item() + state[0][2].item(),
      state[0][i + 1].item() + state[0][3].item(),
    )
    for i in range(22, 28, 2)
  ]

  # Get directions away from each enemy
  radial_dirs = [(self_pos[0] - enemy_pos[0]) for enemy_pos in enemies_pos]
  tangential_dirs = []
  for enemy_pos in enemies_pos:
    rel_pos = enemy_pos[1] - self_pos[1]
    if rel_pos < 0:
      rel_pos += 2 * np.pi
    tangential_dirs.append(rel_pos - np.pi)

  # Getting the importance between radial escape and tangential escape
  # The idea is that p should escape radially when enemies close radially,
  # and should escape tangentially when enemies close tangentially
  radial_importances = 1 / (np.array(radial_dirs) / 2.83)
  tangential_importances = 1 / (np.array(tangential_dirs) / np.pi)

  # Enemy importance based on distance
  enemy_importances = []
  for i in range(3):
    enemy_importances.append(
      1 / np.sqrt(state[0][i + 22].item() ** 2 + state[0][i + 23].item() ** 2)
    )
  radial_importances *= enemy_importances
  tangential_importances *= enemy_importances

  final_dir = (np.sum(radial_importances), np.sum(tangential_importances))
  final_dir /= np.linalg.norm(final_dir)
  final_dir /= np.max(np.abs(final_dir))

  # Now we define a target position for the player
  target_polar = (
    self_pos[0] + final_dir[0] * 2.83 * 0.3,
    self_pos[1] + final_dir[1] * np.pi * 0.3,
  )
  if self_pos[0] > 0.98 and random.random() < self_pos[0] * 0.5:
    target_polar = (0, target_polar[1])
  target_cart = polar_to_cart(target_polar[0], target_polar[1])

```

```
48    # And cartesian distance for the player to target
49    dist_x = target_cart[0] - state[0][2].item()
50    dist_y = target_cart[1] - state[0][3].item()
51
52    # Now we move towards that target
53    action = ACTIONS["no_action"]
54    if abs(dist_x) > abs(dist_y):
55      if dist_x > 0:
56        action = ACTIONS["move_right"]
57      elif dist_x < 0:
58        action = ACTIONS["move_left"]
59    elif abs(dist_x) < abs(dist_y):
60      if dist_y > 0:
61        action = ACTIONS["move_up"]
62      elif dist_y < 0:
63        action = ACTIONS["move_down"]
64
65    return action
```

A.3 Shifty Player

This player chooses a desired position on the map based on the situation, and moves towards it while still avoiding enemies.

```
1
2   def shifty_player(state):
3     """
4     For this code, it is important to know the landmark fixed positions:
5       [[-0.9 , -0.9 ],
6        [ 0.9 , -0.9 ],
7        [-0.9 ,  0.9 ],
8        [ 0.9 ,  0.9 ],
9        [-0.4 ,  0.  ],
10       [-0.36,  0.2 ],
11       [ 0.  ,  0.14],
12       [ 0.16,  0.1 ],
13       [ 0.24, -0.06]]
14    """
15
16    # Run the Evasive algorithm depending on how close enemies are
17    min_dist = min(
18      [
19        np.linalg.norm(np.array((state[0][i].item(), state[0][i + 1].item())))
20        for i in range(22, 28, 2)
21      ]
22    )
23    if random.random() > min_dist:
24      return evasive_player(state)
25
26    # Fixed target positions for this player to use
27    positions = [
```

```
    (0, -0.9),  # sides
    (0, 0.9),
    (-0.9, 0),
    (0.9, 0),
    (-0.38, 0.1),  # tunnel between landmarks [-0.4 , 0.] and [-0.36, 0.2]
    (-0.41, 0.09),
    (-0.35, 0.11),
    (-0.18, 0.17),  # between [-0.36, 0.2] and [ 0. , 0.14]
    (-0.2, 0.23),
    (-0.16, 0.11),
    (0.8, 0.12),  # between [0. , 0.14] and [0.16, 0.1]
    (0.9, 0.06),
    (0.7, 0.18),
    (0.19, 0.02),  # between [0.16,  0.1] and [0.24, -0.06]
    (0.27, -0.14),
    (0.11, 0.18),
  ]

  # Sorting positions by distance to the player
  distance_sorting = np.argsort([
    np.linalg.norm(
      np.array(pos) -
      np.array((
        state[0][2].item(),
        state[0][3].item()))
      )
      for pos in positions
    ])

  # The player first checks for these 5 positions: 3 closest and 2 furthest
  current_closest = [positions[i] for i in distance_sorting[:3]]
  current_distant = [positions[distance_sorting[-2]], positions[distance_sorting[-1]]]
  first_check = current_closest + current_distant

  # Now out of these let's check the distance from each to the closest enemy
  min_enemy_dists = [
    min(
      np.linalg.norm(np.array(pos) - np.array(
        (
          state[0][i].item() + state[0][2].item(),
          state[0][i + 1].item() + state[0][3].item(),
        )
      ))
      for i in range(22, 28, 2)
    )
    for pos in first_check
  ]

  # Now if the maximum among those distances is high enough, let's move there
  if np.max(min_enemy_dists) > 0.5:
    target = first_check[np.argmax(min_enemy_dists)]
  # Otherwise, re-check using every position, move to the furthest away from the enemies
  else:
```

```
    target = positions[
      np.argmax(
        [
          min(
            np.linalg.norm(np.array(pos) - np.array(
              (
                state[0][i].item() + state[0][2].item(),
                state[0][i + 1].item() + state[0][3].item(),
              )
            ))
            for i in range(22, 28, 2)
          )
          for pos in positions
        ]
      )
    ]

  # Now that the target position is acquired, let's just move there
  # Distance for the player to target
  dist_x = target[0] - state[0][2].item()
  dist_y = target[1] - state[0][3].item()

  # Now we move towards that target
  action = ACTIONS["no_action"]
  if abs(dist_x) > abs(dist_y):
    if dist_x > 0:
      action = ACTIONS["move_right"]
    elif dist_x < 0:
      action = ACTIONS["move_left"]
  elif abs(dist_x) < abs(dist_y):
    if dist_y > 0:
      action = ACTIONS["move_up"]
    elif dist_y < 0:
      action = ACTIONS["move_down"]

  return action
```