Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Dynamics-aware Reinforcement Learning for Playing Tennis Games

Author:
Maytus Piriyajitakonkij

Supervisor:
Prof. Andrew Davison

Second Marker:
Dr. Antoine Cully

Submitted in partial fulfillment of the requirements for the MSc degree in Computing (AI and Machine Learning) of Imperial College London

September 2022

Abstract

Sports are straightforward for humans to play. However, none of the existing robot learning algorithms achieve the human-level abilities to play sports. In this project, we investigate the integration of dynamics prediction and control in AI agents for playing sports to make them more adaptable to dynamic changes. Our contributions are as follows. First, we develop a new learning environment called *Tennis2D* for developing reinforcement learning (RL) algorithms to play tennis. *Tennis2D* contains realistic dynamics and has dynamic adaptation challenges for single-agent and multi-agent RL. Second, we proposed a new dynamics prediction model called *Contrastive Dynamics Model (CDM)*. The model exploits contrastive representation learning to estimate unknown physical parameters and incorporates estimated parameters to improve prediction accuracy. Lastly, we proposed *Dynamics-aware RL (DynARL)* that utilises model-based prediction and model-free learning. *DynARL* is found to improve the generalisation performances of both in-distribution and out-of-distribution dynamics. The *DynARL*-trained agents are found to be more robust to windy circumstances.

Acknowledgments

My gratitude goes out to

- My family, especially Songpol and Mayvadee Piriyajitakonkij, for supporting and funding me to study this master's degree at Imperial College London.
- Andrew Davison, my supervisor, for introducing me to Robotics, for allowing me to explore my interests, for providing guidance that helped me improve my research, for the encouragement to keep me motivated when things did not turn out well, and for giving me the inspiration to continue doing AI research after finishing this master's degree.
- VISTEC colleagues: Theerawit Wilaiprasitporn, for introducing me to academic research, and for guiding me to complete my first journal paper in AI for sleep medicine; Nat Dilokthanakul, for introducing me to Reinforcement Learning, for inspiring me to do fundamental research, and for recommending the book: "Embodiment and the Inner Life: Cognition and Consciousness in the Space of Possible Minds," which shaped my thought about artificial intelligence and inspired me to do research in this field.
- The library buddies, especially Gréta Horváthová, Emir Şahin, and Charlize Yang, for giving me very useful feedbacks for improving my writing and for working together for several months in the library.
- All friends in the UK, for bringing me to good restaurants, for hanging out with me, and for never making me feel lonely.

Contents

Chapter 1

Introduction

How can we create an artificial intelligent (AI) agent to master sports?
Back in the day, two main approaches were proposed to develop AI: Symbolic AI and Machine Learning (ML). Symbolic AI was built based on human-interpretable abstractions, logic, and search [1]. Symbolic AI requires humans to design and put knowledge into it. ML, in contrast to symbolic AI, requires minimal human knowledge to build a model. Given the model data, the model can establish correlations between inputs and outputs automatically. The symbolic AI was the dominant approach for creating AI between the 1950s to 1990s [2]. Especially in 1996, the symbolic approach was used to design AI that won against the world chess champion [3]. The chess-playing AI was built based on a systematic search to find the optimal move. The search required a dynamics model of the chess game, i.e., relations of outcomes and moves. However, this approach becomes unfeasible for real-world robotic problems because they have more complex dynamics and a greater number of outcomes. It is impossible to search for all possible outcomes because the dynamics of the physical world are unknown. And even if the dynamics model was known, the symbolic AI system would have to search over continuous space, requiring too much computational resource. Moreover, Symbolic AI is found to work well only in well-controlled environments, and it cannot adapt to dynamic changes. For example, industrial robots in production lines are built using hand-engineered controllers, and they cannot operate in open environments [4]. Hence, designing symbolic representation to be a building block of the robot brain might be the wrong approach for achieving generalisable intelligent agents [5]. ML, however, does not suffer from these issues. It can learn patterns from observational data to construct an approximate dynamics model of the environment [6] and find optimal solutions without exhaustive search [7, 8]. Furthermore, ML can learn a useful representation that allows robots to operate and solve tasks in real-world environments [9]. Therefore, ML comes to solve a large-scale search and create adaptive behaviour for AI agents, and eventually, ML-based robots might master sports at the human level.

Reinforcement Learning (RL) is a class of ML algorithms that concerns the interactions between perceptions and actions, like the coordination of sensory and motor systems in humans. Thus, RL is a promising approach to create robots to master sports [10]. RL has two main elements: the *agent(s)* and the *environment*. The

agent learns by trial-and-error guided by rewards or punishments that the environment returns to it. The *agent* learns a *policy*, which can be considered as a function that maps the present *state* of the environment to an action that affects future environment states. The *policy* is learnt through an optimisation process to maximise the expected accumulative *reward* given by the environment in future states. RL has succeeded in achieving human-level performance in many games, such as backgammon [11], Go [12], Poker [13] and eSports [14, 15]. However, RL is susceptible to non-stationary environments [16, 17]. Dynamics changes in the environment cause nonstationarity. Even small changes in the dynamics can worsen RL's performance [18].

Unlike traditional RL agents, professional tennis players can play well in a non-stationary environment. The changes in racket mass and elasticity affect the optimal arm's angular velocity that hits the ball to reach the desired position. The change in drag force or a windy circumstance deviates ball trajectories. These nonstationarities are expected to be seen in a real-world tennis match, where a tennis player might use *meta-learning* [19] to deal with nonstationarities. Neuroscientific works investigate how humans deal with complex and non-stationary dynamics. They suggest that athletes master sports using a dynamics prediction model coupled with a control model [20, 21]. The ability to anticipate and control diverse dynamics settings might be the keystone for robot learning to master various kinds of sports.

In this project, we get inspired by the human cognitive abilities to play sports and to adapt to dynamic changes. We will investigate the integration of the RL model and the dynamics prediction model in robots to play tennis games. The key contributions of this project are explained below.

- In chapter 4, RL agents for playing tennis require an environment that represents realistic dynamics and has challenges for adaptation. Thus, we design and create a new learning environment for RL called *Tennis2D*. The environment is perfect for algorithmic design in an early stage because it contains the realistic dynamics of 2D motions. The environment covers some basic tasks: serve, counter, and smash in a single-player game and a two-player cooperative game. We also investigate several reward functions that are most suitable for proposed tasks. Several challenges, such as wind, changes in ball mass, and changes in drag constant, can be added to investigate how well robots adapt to these dynamics. Algorithms designed for our environment should be easily extended to a 3D simulation with a complex robot arm. To our best knowledge, none of the existing public RL environments has tennis tasks with realistic physics.

- In chapter 6, we specify a dynamics prediction model to be adaptable and generalisable in multiple dynamics. Therefore, we propose a new dynamics prediction method called *Contrastive Dynamics Model (CDM)*. It is designed to disentangle knowledge and physical parameters of the dynamics model using contrastive representation learning. The model training requires no ground

truth of physical parameters. The proposed method has better generalisation performance in both in-distribution and out-of-distribution in multiple-physical-parameter settings. Most existing works used contrastive learning for image, audio, and language data and found it constructs transferable representation for downstream tasks [22, 23, 24]. In addition to these works, we find that contrastive learning is also applicable to learning the representation of dynamical states.

- In chapter 7, we investigate how we can integrate the coordination of dynamics prediction and control to agents. We propose a new RL method called *Dynamics-aware RL (DynARL)* that is trained in a model-free setting. At the same time, it utilises the benefit of dynamics prediction as a model-based RL does. Unlike other model-based RL methods [25, 26, 27], we do not design a dynamics model that captures all observable dynamics of an environment, which is very difficult to train. Instead, we construct the dynamics model of a flying tennis ball. We augment the predicted future states of a flying tennis ball to an agent's input. *DynARL* improves both in-distribution and out-of-distribution generalisation performances in all proposed tasks from the baseline.

Chapter 2

Related Works

2.1 Model-free Reinforcement Learning

Reinforcement learning (RL) is commonly formalised with Markov Decision Processes (MDP). An agent receives the current state at time t; s_t, select an action a_t following a policy p_θ, and the environment returns reward r_{t+1} and the next state s_{t+1} to the agent. In a model-free setting, the RL agent is trained to learn a policy that maximises the expected sum of rewards from the environment. The agent does not have knowledge about how an environment works, i.e., not knowing what the outcome is after an action a is selected. The integration of neural networks as function approximators and RL achieves many complicated tasks. [28, 29] is the first work that proposed deep neural networks named DQN as an approximator of the action-value function (Q-value). Several methods have been proposed to mitigate overestimation [30], improve learning efficiency [31], and enhance the performance of DQN [32]. DQN can be trained to achieve tasks with a discrete action space. Then, an actor-critic architecture is proposed to learn continuous control tasks [33, 34, 35, 36, 37]. However, model-free algorithms normally require a large number of training data, and the learnt representation that is used to solve one task is difficult to transfer to solve another task. Moreover, traditional model-free RL cannot learn real-world robotics tasks very efficiently [38, 39, 10]. To mitigate these problems, unsupervised representation learning techniques [40, 41] and data augmentation [42, 43] are used to increase sample efficiency, to accelerate learning and to improve generalisation performance.

2.2 Model-based Reinforcement Learning

Model-based RL can be constructed in many different ways to solve different problems. This part will give examples of some state-of-the-art model-based RL.

PILCO [44]: The PILCO agent uses Gaussian Processes to model proprioceptive dynamics (i.e., parts of robot's body). It does not only represent the trasition dynamics of an environment, but it also represents the uncertainty of dynamics predictions. The policy evaluation is implemented using an approximate inference method, and

the policy improvement is implemented using policy gradient to maximise the expected return. The drawback of PILCO algorithm is that it cannot be scaled to a high-dimensional environment, because the dynamics model is constructed using Gaussian Processes.

PETS [25]: Uncertainty can be classified into two types: aleatoric uncertainty and epistemic uncertainty. Aleatoric uncertainty is inherently from randomness in data, such as noisy observation. Epistemic uncertainty is from the lack of knowledge of data distribution. Theoretically speaking, epistemic can be reduced to zero as the number of data increases to infinity. PETS algorithm represents two types of uncertainty in model-based RL problems by combining probabilistic neural networks and ensemble neural networks. It also uses a particle-based method to represent each state as a group of particles. During planning, the PETS agent uses Cross-Entropy Method (CEM)–a simple Model Predictive Control (MPC) algorithm. CEM uses some elite trajectories (those which achieve high returns) in the current iteration to construct probability distribution and then do re-sampling trajectories with this distribution in the next iteration of training. PET agent achieves great performance on several robot control tasks.

POPLIN [45]: Instead of randomly initialise a sequence of action in the MPC step, the POPLIN agent uses neural networks, which is learnt to propose the initial action sequence. Then, noise is added to the action sequence to generate diverse trajectories. Finally, it uses CEM to find the best sequence of action from these trajectories.

Model-Based RL With Model-Free Fine-Tuning (MBMF) [27]: Model-based RL is believed to be more sample efficient than model-free RL. However, model-free RL is often better in terms of task-specific performance than model-based RL with MPC. This work leverages the benefits of both model-based and model-free RL to the same agent. Firstly, the agent is trained with model-based RL and MPC. Then model-free's neural networks is trained to imitate model-based RL's policy using DAGGER [46]. Finally, the model-free's neural network is trained (fine-tuned) in model-free RL setting.

SimPLe [47]: A world model is trained to predict the video of Atari game. Then, this world model is used to generate artificial data for training model-free RL. Because a neural network simulator is much more time efficient than real simulator (Atari game). SimPLe accelerates model-free learning to reach the top performance faster, i.e., the world model reduces the number of times that the agent uses to interact with the real environment.

Dreamer [26, 48]: Similar to SimPLe, The Dreamer agent contains a world model which is trained to capture environment dynamics and predict the next states given action. The world model is variational autoencoders [49] with recurrent latent variables to capture the environment dynamics. Instead of rolling out a trajectory in observation pixel space, the Dreamer agent's policy is trained on the latent variables

using model-free RL, which has much less complexity than pixel observation and contains state abstractions beneficial to learning.

2.3 Dynamics Adaptation and Generalisation

One of the most challenging problems in model-based RL is that it is difficult or impossible to find a global solution that is applicable to the entire distribution. Rather than learning a single global solution, the model can learn an approximate global solution that is easily adapted to the task of interest, in the sense that the dynamics knowledge in the model can be fine-tuned and updated to minimise the prediction error for state-action space in the new task. [50, 51] propose recurrent updates in recurrent neural networks for adaptation to new data distributions. The algorithm stacks multiple episodes into one horizontal trial, and then it trains recurrent neural networks to act to maximise rewards along with this trial. Here we call it recurrent-based meta-learning. Recurrent-based meta-learning can only be applied to recurrent neural networks. [52] proposes model-agnostic meta-learning (MAML) that uses learning rules to find the initial parameters that can be adapted to multiple tasks and optimise the sum of losses in these tasks. MAML is also applied in model-free reinforcement learning and make agent adapted to perform new tasks in a few gradient updates. [53] uses recurrent-based meta-learning for multi-agent settings; the trained agent sees another agent as an observable state–i.e. a part of an environment–and the agent can adapt to collaborate with another agent, which is unseen during training. Other than model-free approaches, [17] proposes the use of gradient-based and recurrent-based meta-learning for model-based RL, and their methods enable agents to quickly adapt to dynamic changes in continuous control tasks with complex real-world dynamics. [16] uses variational inference to approximate unknown physical parameters, and approximated physical parameters are given as an auxiliary input of a learnt dynamics model. Then, the dynamic model can adapt to a new environment with unseen physical parameters and can make accurate predictions. [18] proposes a context encoder in which latent variables are used to predict both past and future states, and then context latent variables are added to a dynamics model, in addition to states and action, to predict future states. The context encoder allows the dynamic model to perform online adaptation. [54] utilises variational autoencoders for learning disentangled representation in RL environments, and they show that the agent, which uses the disentangled representation, can perform zero-shot adaptation to domain shifts.

2.4 Robotic Table Tennis

The real-world application that our model-based RL framework could be applied to in the future is robotic table tennis. Early research on robotic table tennis is based on calculating the virtual hit states (position and time) and controlling a paddle to hit the ball at the hit point at a particular time [55, 56, 57, 58]. Most of these works require a physics-based or learning-based dynamics model to predict a ball's

trajectory before planning. Recent works use end-to-end model-free RL, which is purely based on reward learning, explicit knowledge of ball dynamics is not required for the model-free RL approaches [59] proposes evolutionary search (ES) algorithm along and gated convolutional neural network (CNN) as a policy function to achieve nearly perfect ball-paddle hit rate. They find that ES is significantly better than Proximal Policy Optimization (PPO) [36] when using the small number of neural network parameters. Some reward shaping techniques are proposed to improve the success hit rate. [60] formulates table tennis problem as a single-step bandit using DDPG [34] and achieves a very good sample efficiency–training a real robot to achieve near-perfect success hit rate using only 200 episodes without pretraining. [61] proposes hybrid sim and real training. The physical robot arm is moving to hit the virtual ball; then, the physical robot arm is fine-tuned to hit the real ball. None of these works investigates nonstationarity in an environment.

2.5 Deep Learning for Dynamics Prediction

A World model can be considered as a function that transforms observed states into future states [62, 63, 64]. For a dynamics model, a function can be parameterised as force, torque, mass, and moment of inertia. Physicists symbolise physical parameters and state variables into mathematical equations (e.g. Newton's laws of motion). These symbolic models are very good at prediction in deterministic and simple scenarios such that there exist analytical solutions [65, 66]. However, when it comes to the complex real-world environment, symbolic physics models cannot deal with non-linear relations of object interactions and uncertainties. For instance, both tennis ball and tennis racket are not ideal rigid bodies, drag force is a non-linear function of the ball's velocity, and the friction coefficient between the ball and racket is unknown. Artificial neural networks (ANN), with their high representation power, are used to tackle some of these problems as function approximators. ANN can learn to represent non-linear dynamics and implicitly estimate unknown physical parameters [67, 68]. Physics prediction problems can be divided into two groups: passive and active learning. In a passive learning setup, ANN or other function approximators learn a function that maps current input states to future output states without directly interacting with an environment, i.e., all samples have been collected before the learning begins [68, 69, 70]. In an active learning setting, on the other hand, ANN is embodied as an agent interacting with an environment. Other than perceiving current states and predicting output states, the agent takes action that affects future environment states [67]. For example, the athlete agent determines the ball's dropped position and controls the racket to hit the ball to drop at that position. Active learning requires a policy that an agent uses to make decisions.

Chapter 3

Background

3.1 Reinforcement Learning

In the RL setting, an intelligent agent interacts with an environment, and it is trained to maximise the cumulative reward. The agent perceives the state of the environment at time t S_t. After an action A_t is performed by the agent, the environment state changes to the next state S_{t+1} and returns reward R_{t+1} to the agent. The agent has a policy π to decide what action A_t should be selected. The goal of RL is to find the optimal policy π that maximises cumulative reward. Most RL settings are formulated with Markov Decision Processes (MDP). MDP contains a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a transition dynamics function $\mathcal{T}(S_t + 1|S_t, A_t)$, a reward function $\mathcal{R}(S_t, A_t, S_{t+1})$, and a discount factor $\gamma \in [0, 1]$. The policy π is generally considered as a function that maps the state to the probability distribution of an action: $\pi(S_t) = p(A_t|S_t)$. The interactions between the agent and the environment generate a sequence of states, actions and rewards: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$ In general, the cumulative reward at time t is defined as the sum of discounted rewards G_t as in Equation 3.1

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \tag{3.1}$$

The optimal policy π^* is defined as Equation 3.2

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}_\pi[G] \tag{3.2}$$

State-Value Function $V_\pi(S)$ defined in Equation 3.3 indicates the expected return the agent will receive if the agent begins with state S and selects actions by following the policy π. In other words, the state-value function indicates how good it is when the agent is in state S and select an action following the policy π.

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{3.3}$$

Action-Value Function $Q(s, a)$ defined in Equation 3.4 indicates the expected future return when the agent select an action a in a state s and following a policy π.

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{3.4}$$

The greedy action a^* in a state s given the policy π is defined as in Equation 3.5

$$a^* = \underset{a}{\operatorname{argmax}}\, Q_\pi(s, a) \tag{3.5}$$

We can write the relationships between the optimal action-value function Q^* and optimal state-value function V^* as in Equation 3.6

$$Q^*(s, a) = \mathbb{E}[R_{t+1} + \gamma V^*(S_{t+1}) | S_t = s, A_t = a] \tag{3.6}$$

Where $V^*(s) = \max_\pi V_\pi(s)$ is the optimal state-value function of the optimal policy, and $Q^*(s, a) = \max_\pi Q_\pi(s, a)$ is the optimal action-value function of the optimal policy. The optimal state-value function for the state s is equal to the optimal action-value function for the state s. Because the action-value function is maximised when the greedy action a^* is selected. We can write the optimal action-value function $Q^*(S_t, A_t)$ in term of the next state's $Q^*(s', a')$ as in Equation 3.7.

$$Q^*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') | S_t = s, A_t = a] \tag{3.7}$$

Equation 3.6 and Equation 3.7 are called Bellman Optimality Equation for a state-value function and for an action-value function respectively. If the agent knows the true optimal action-value function, the agent can find the optimal policy π^* that maximises the expected return, by selecting greedy action (Equation 3.5) in every step.

Q-learning:
In most real-world cases, knowing the optimal action-value function is impossible because it requires a joint probability distribution of states, actions, and rewards. Instead, the agent has to estimate the optimal action-value function from experience, to find the optimal policy.
Q-learning algorithm estimates the optimal action-value function Q^* by the update defined as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{3.8}$$

Q-learning we described above, works for tabular settings where both state and action spaces are discrete. However, Q-learning is not applicable to continuous state space, because there are infinite possible states. Subsequently, the Q-table for keeping Q-values will have infinite size. Instead of keeping the action value for each state-action pair, we can approximate the action value using function approximators.

Deep Q-Network (DQN):
The Deep Q-learning algorithm utilises the representational power of deep neural networks to approximate the action-value function. DQN is neural networks (parameterised by θ) that receives an input state s (such as an image) and outputs an estimated action value $f_\theta(s) = Q(s, \cdot; \theta)$. Similar to tabular Q-learning, the objective of the training is to reach the Bellman Optimality Condition of action-value function

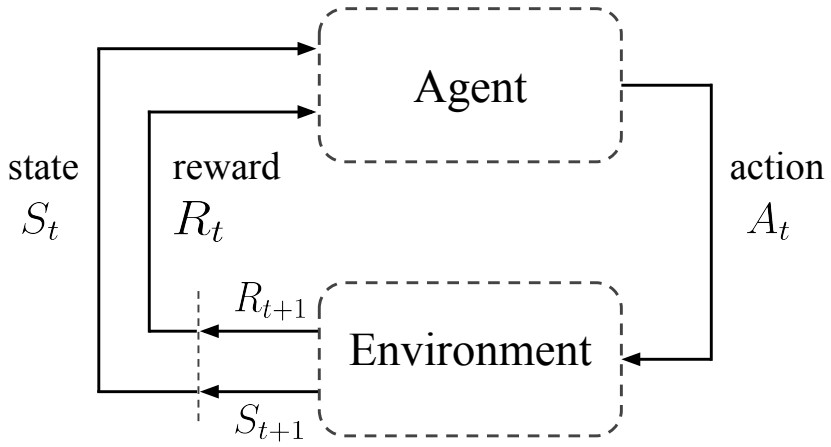

Figure 3.1: *Agent-Environment Interaction:* The diagram shows interaction between an agent and an environment.

as in Equation 3.7. In other words, we want the difference between the optimal action-value function $Q^*(\cdot)$ and the estimated action-value function $f_\theta(s)$ to be as less as possible. Hence, DQN can be trained by minimising the temporal difference loss:

$$\mathcal{L}(\theta_i) = \mathbb{E}_{s,a}[(y_i - Q(s, a; \theta_i))^2] \tag{3.9}$$

Target y_i is defined as:

$$y_i = \mathbb{E}_{s'}[r(s, a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1})] \tag{3.10}$$

where $r(\cdot)$ is reward function, i is iteration of training, θ_i is the current set of neural network parameters, and θ_{i-1} is the previous set of neural network parameters. We can see that if Bellman Optimality Equation satisfies (or mostly satisfies), loss $\mathcal{L}$ will be zeros (or be nearly zero).

3.2 Continuous control using Policy Gradient

DQN is designed for discrete action space. However, in this project, an agent is set to control two continuous variables: force and torque applied to a tennis racket. So DQN is not applicable in our setting. The authors of [33, 34] propose the use of policy gradient and an actor-critic architecture to tackle continuous control problems. An actor can be thought of as a function approximator of argmax over continuous space. In other words, the actor is designed to estimate a greedy action over continuous action space. A critic can be thought of as a function approximator of an action-value function, similar to DQN. The actor is a function parameterised as $\mu(s; \theta^\mu)$, which outputs an action given the state. The critic is a function parameterised as $Q(s, a; \theta^Q)$. The critic outputs an action-value given an action and a state.

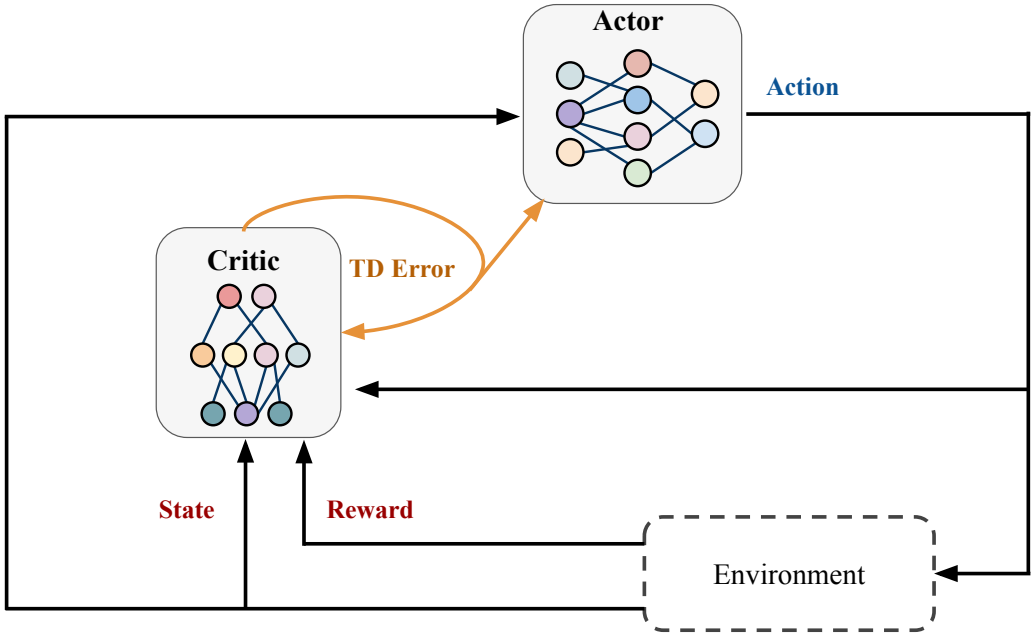

Figure 3.2: *An actor-critic architecture*

The critic is trained in the same way as DQN is trained, that is to minimise temporal difference square. At the same time, the actor is learnt to maximise the expected return:

$$\begin{aligned} \nabla_{\theta^\mu}\mathcal{L} &= \mathbb{E}_s[\nabla_{\theta^\mu} Q(s,a;\theta^Q)]|_{a=\mu(s;\theta^\mu)} \\ &= \mathbb{E}_s[\nabla_a Q(s,a;\theta^Q)|_{a=\mu(s)} \nabla_{\theta^\mu}\mu(s;\theta^\mu)] \end{aligned} \tag{3.11}$$

Equation 3.11 is called policy gradient which is the gradient of the policy's performance with respect to the policy parameters.

3.3 Model-Based Reinforcement Learning

Model-based reinforcement learning is the use of a dynamics model of an environment to incorporate planning. For example, Model Predictive Control (MPC) [71], which Here, we do not try to define strictly what model-based RL is, nor try to group model-based RL variants. Instead, we will explain crucial elements in model-based RL briefly.

3.3.1 Elements of model-based RL

Dynamics Prediction:

Dynamics prediction can be done by either known true dynamics or learnt dynamics model. For some situations, true dynamics is easy to model with physics equations, such as a rigid body motion. However, in many situations, we can only estimate

dynamics using learnt model. A model can predict dynamics in three different fashions. Forward model, backward model, and inverse model.

Forward model predicts a next state s_{t+1} from a current state s_t and selected an action a_t. It is natural for an agent to look into the future states and select an action to reach the desired goal. The most existing methods are forward models.
Backward model predicts a predecessor state s_{t-1} and an action $a_{t-1}$from a current state s_t. The agent which uses this kind of model will plan backward into the past from the desired final state.
Inverse model predicts an action a_t that leads a current state s_t to a particular future state s_{t+1}. Goal-conditioned reinforcement learning could be classified as inverse model [72].

Planning:

There are two main planning approaches: planning with gradient and planning with sampling. Planning with gradient requires a policy model to select an action, then the dynamics model predicts the next state and reward given the selected action. The policy model is then trained to maximise the expected accumulative rewards through the gradient. It works for only a differentiable environment, and the optimisation is non-convex. Therefore, It is possible for the policy model to get stuck at a local maximum during optimisation. In contrast to planning with gradient, planning with sampling does not require a policy model to select an action. But it still requires a dynamics model of an environment to roll out multiple state-action trajectories. These trajectories are generated by given random actions to the dynamics model input. Subsequently, the optimal sequence of actions is selected based on some of these trajectories, to maximise the expected return. In some cases, planning algorithms use state abstraction instead of raw observations. For example, [26] uses model-free reinforcement learning to learn a policy from latent variables, which are extracted by an encoder given raw observations.

In contrast to the existing methods, our dynamics model is for predicting the uncontrollable part of an environment, so the dynamics model does not require an action as an input to roll out multiple state-action trajectories. Our proposed algorithm is augmenting predicted future states of a flying tennis ball to an agent's input, and the agent is trained using model-free RL.

Chapter 4

Environment Design and Implementation

4.1 Tennis2D

We design and construct a new learning environment for RL named Tennis2D that has complex real-world dynamics. Tennis2D is built based on PyMunk[1]–2D physics simulation based on Python, and we use PyGame[2] for interacting and visualising the environment. We select PyMunk because it provides us with realistic dynamics for 2D motions, including a rotation in a 2D plane. It mitigates the difficulties of creating an RL environment.

The environment contains three parts: the ball, the racket(s), and the scene. The ball and the racket are dynamic objects that are movable. On the other hand, the floor, the net, and the ceiling are static objects. They cannot be moved, but they still have friction coefficient and elasticity that determine how they interact with other dynamic objects. We will explain the dynamic objects and their equations of motion below. Apart from the gravitational force, which has already been implemented in PyMunk, we added additional forces into the simulation manually.

The ball can bounce on the ground and the middle net. The ball's physical parameters are radius, mass, the moment of inertia, friction coefficient, and elasticity. Friction causes force and torque on the ball during the bounce. The drag force, of which magnitude is proportional to the square of the ball's speed, is applied to the ball when it is moving. Gravity is applied to all dynamic objects (the racket and the ball) in the scene. We can write the equation of motion of the flying tennis ball from Newton's second law of motion as in Equation 4.1.

$$\boldsymbol{F} = m_{ball}\frac{d^2\boldsymbol{x}_{ball}}{dt^2} = m_{ball}\boldsymbol{g} - K_{drag}|\boldsymbol{v}_{ball}|^2\hat{\boldsymbol{v}}_{ball} + \boldsymbol{w} \tag{4.1}$$

where m_{ball} is the ball mass, x_{ball} is the displacement vector of ball from the origin, $\boldsymbol{g}$

[1] `http://www.pymunk.org/en/latest/pymunk.html`
[2] `https://www.pygame.org/news`

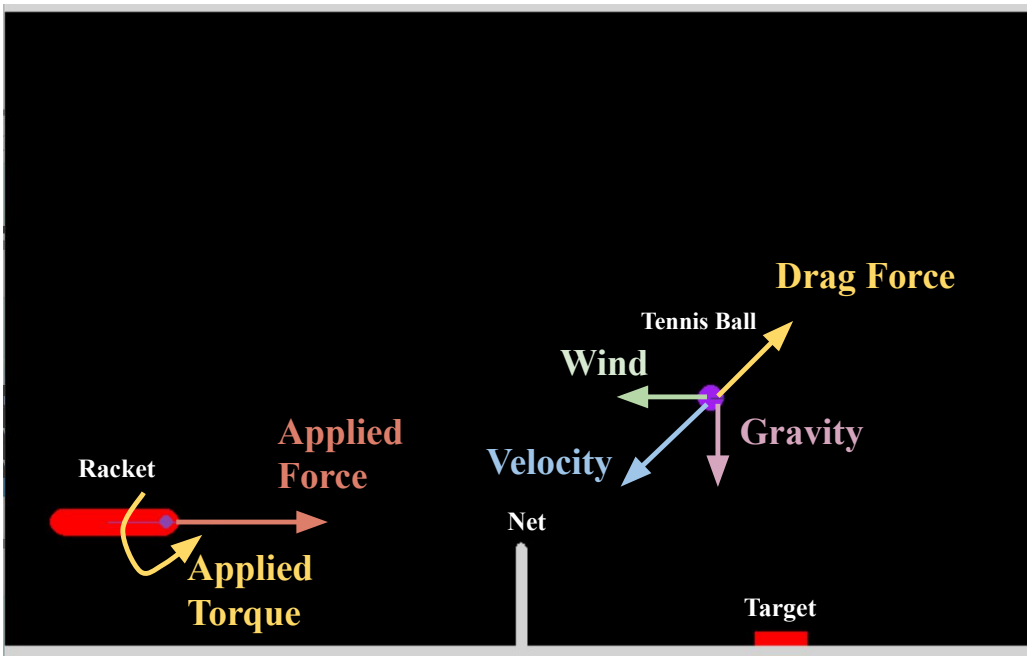

Figure 4.1: *Tennis2D:* The structure of the environment and free-body diagrams of the tennis ball and the racket, are shown in the figure.

is the gravitational acceleration (vector), K_{drag} is the drag force constant , v_{ball} is the ball velocity, and $\hat{\boldsymbol{v}}_{ball}$ is the unit vector of the ball velocity. $\boldsymbol{w}$ is the constant wind force applied to the ball. The Equation 4.1 can be rewritten as Equation 4.2.

$$\frac{d^2\boldsymbol{x}_{ball}}{dt^2} = \boldsymbol{g} - \frac{K_{drag}}{m_{ball}}|\boldsymbol{v}_{ball}|^2\hat{\boldsymbol{v}}_{ball} + \frac{\boldsymbol{w}}{m_{ball}} \tag{4.2}$$

Since the gravity is constant, the change of ball mass affects ball dynamics by the changes in the second and the third terms on the right-hand side of Equation 4.2. As ball mass increases, both drag force and wind force contribute less to the ball dynamics.

The racket(s) is constrained to move along a horizontal line, and it has a rotational joint at its tip. Horizontal force can be applied through the center of mass to accelerate the racket, and torque can be applied to rotate the racket. The rotation is also constrained because the rotational constraint leads to more sensible behaviour after the agent learns how to control the racket to hit the ball. The equation of motion of the racket is described below.

$$\frac{d^2x_{racket}}{dt^2} = \frac{F}{m_{racket}} \tag{4.3}$$

$$\frac{d^2\theta_{racket}}{dt^2} = \frac{T}{I_{racket}} \tag{4.4}$$

where x_{racket} is racket's horizontal position, θ_{racket} is racket's angle, m_{racket} is racket's mass, and I_{racket} is racket's moment of inertia. F and T are force and torque applied

to the racket, which is controlled by an agent. We found that constraints of angle and position have a huge impact on a learnt agent's behaviour. Without the angular constraint, an agent hits the ball by rotating the racket many times. Without the positional constraint, some learnt agents move to bounce the net to hit the ball.

State variables:

State variables refer to what an agent can observe from the environment. We assume that the scene, composed of the ground and the net, is static. Observing only dynamic parts is sufficient for the agent to accomplish tasks. Therefore, the state variables are a ball position, a ball velocity, a ball angular velocity, a racket position, a racket velocity, a racket angle, a racket angular velocity, and a target position (for single-agent tasks). The state variables for the multi-agent task are similar but include both agents' states and exclude a target position.

Control variables:

The agent can control two physical variables: torque (T) and horizontal force (F) applied to the racket. In other words, we can define the action space of the environment as $\mathcal{A} \subset \mathbb{R}^2$. Positive τ is defined as the clockwise direction. Positive F_x is defined as the right direction. The racket is constrained to move horizontally along the purple line and cannot be moved along the vertical axis. Force can be applied to accelerate the racket to move along the purple line, and torque can be applied to accelerate the racket to rotate. To prevent irregular phenomena such as a tennis ball tunneling through the net, both velocity and angular velocity are limited.

Next, we will design and have experiments with tasks and reward functions such that trained agents have sensible behaviour and achieve human-interpretable goals. For the single-agent setting, an agent is expected to hit the ball and make the ball fly toward the red target. For the multi-agent setting, agents have to play cooperatively to hold a game as long as possible. The game ends when a tennis ball drops on the ground for the second time, similar to a real tennis match.

4.2 Single-agent games

The environment with a single-agent setting has three tasks: serve, counter, and smash. All tasks have the same goal, and the agent has to hit the tennis ball to reach the target on the other side. In this chapter, we will investigate the difficulty of tasks as well as the most suitable reward function for the agent to achieve all tasks.

4.2.1 Tasks and Reward Functions

The impulse force used to shoot the ball is randomly selected with uniform distribution in both magnitude and direction, and the red target is randomly placed with uniform distribution.

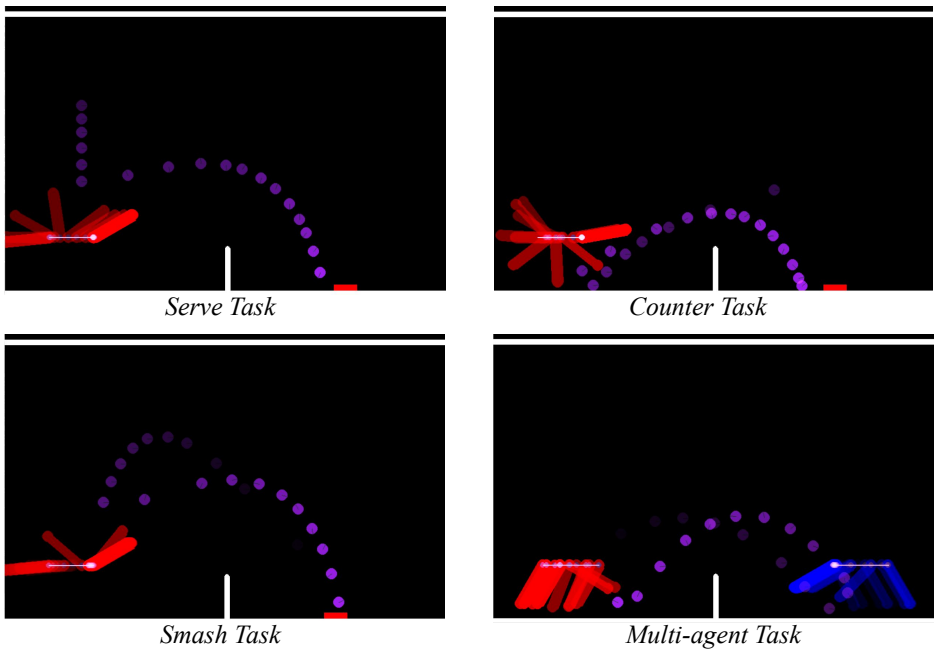

Figure 4.2: *Examples of agent behaviours and tennis ball trajectories of four tasks in Tennis2D.*

Task I: Serve

The first task is to serve the ball to reach the target or drop as close to the target as possible. Firstly, the ball is placed ahead of the racket with a slightly random initial state composed of the initial position and the impulse force. Figure 4.3 demonstrates how this task is done by the trained agent.

Task II: Counter

The counter task starts with the ball being shot from the right-hand side. The shooting simulates a serve from the right side by a virtual opponent. Similar to real tennis, the agent must wait for the ball to bounce to the floor for one time before hitting the ball. Otherwise, we consider the agent fails to hit the ball. The task requires the agent to hit the ball with the desirable racket's state, which contains position, angle, velocity, and angular velocity. The agent has to learn the desirable final racket's state and the amount of force and torque to be applied to the racket to reach that state. The desirable final state causes the ball to drop at the position as close to the target as possible. The impulse force used to shoot the ball and the initial state of the ball are randomly selected by the environment. Figure 4.4 demonstrates how this task is done by the trained agent. Figure 4.7 and Figure 4.8 show the distribution of the ball trajectories in this task.

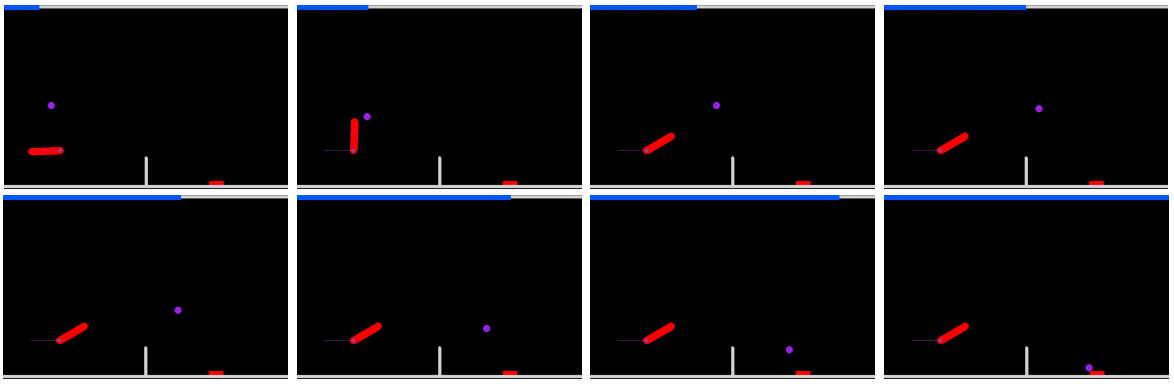

Figure 4.3: *Serve Task:* The ball is shot up ahead of the racket, and a learnt agent controls the racket to rotate in the clockwise direction to hit the ball to reach the red square target on the right-hand side. Example images are stacked in chronological order from left to right and top to bottom. Time is indicated by the blue bar on the top of the images.

Task III: Smash

Similar to the counter task, the ball is firstly shot from the right-hand side. Instead of being shot downward, the ball is shot in the upward direction, and the agent needs to counter-hit by smashing the ball. The task requires the countered ball to be dropped at the position as close to the target as possible. Figure 4.5 demonstrates how this task is done by the trained agent. Figure 4.9 and Figure 4.10 show the distributions of the ball trajectories in this task.

Reward Function:

We will investigate three reward functions, and we will find what reward shapes the most desirable behaviour. The reward will be given to an agent if the ball falls on the right side (the same side as the square target side). A learning episode ends at the time the ball drop on the right side. A reward function is designed to have the highest value when the ball hits the target, i.e., the distance between the ball and the target is close to zero. The simplest sensible reward function is the linear function of the proximity from the ball to the target as in Equation 4.5. The second reward is the polynomial function of the proximity from the ball to the target as in Equation 4.6. The authors of [61] found empirically that the exponent of $\frac{3}{4}$ gives the best results among other exponents, so we decide to use the same exponent. To make the reward gradient smoother with respect to the distance function, we propose the third reward as the radial basis function of the distance from the ball to the target as in Equation 4.7.

Linear Reward Function:

$$r = 1 - dist(x_{target}, x_{drop}) \tag{4.5}$$

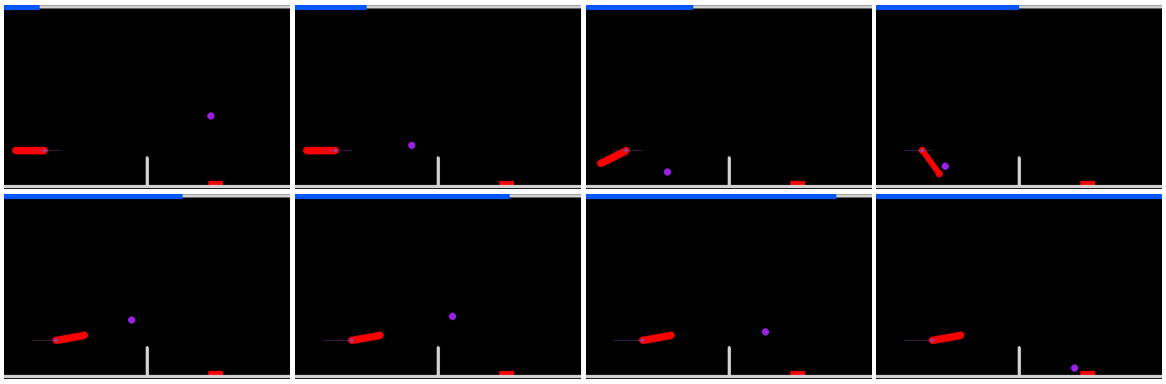

Figure 4.4: *Counter Task:* The ball is shot from the right-hand side, and a learnt agent controls the racket to rotate in the counter-clockwise direction to counter-hit the ball and make it fly toward the red target. Example images are stacked in chronological order from left to right and top to bottom. Time is indicated by the blue bar on the top of the images.

Polynomial Reward Function:

$$r = (1 - dist(x_{target}, x_{drop}))^{\frac{3}{4}} \tag{4.6}$$

Radial Basis Reward Function (RBF):

$$r = K \exp(-dist(x_{target}, x_{drop})^2) \tag{4.7}$$

where $dist(x_{target}, x_{drop})$ is the normalised Euclidean distance function between x_{target} and x_{drop}, x_{target} is the x-position of the target and x_{drop} is the x-position where the ball is drop on the right hand side.

4.2.2 Experiments and Results

Implementation Detail: We use a DDPG-trained agent that receives state variables as inputs of both actor and critic networks, with the discount factor of 1.0 and the target update factor (τ) of 0.01. The critic's additional input is the output action from the actor. Rather than using only the current state, the neural networks are provided with 3 past states. The state inputs of neural network models are $X = [S_t, S_{t-1}, S_{t-2}, S_{t-3}]$, where S_t is the state variable at time t. All neural network models are fully-connected and has three hidden layers with the sizes of $[512, 256, 128]$. All agents are trained with $1,000$ episodes, the exploration is done by Ornstein–Uhlenbeck process, and the later episode has smaller exploration noise. The replay buffer size is $10,000$. The optimisation is done using Adam [73] with the learning rate of 0.0001, and we use a batch size of 128.

In addition to average return, three human-interpretable metrics are proposed to justify tasks and reward functions, hit rate (HR), net pass rate (NPR), and success

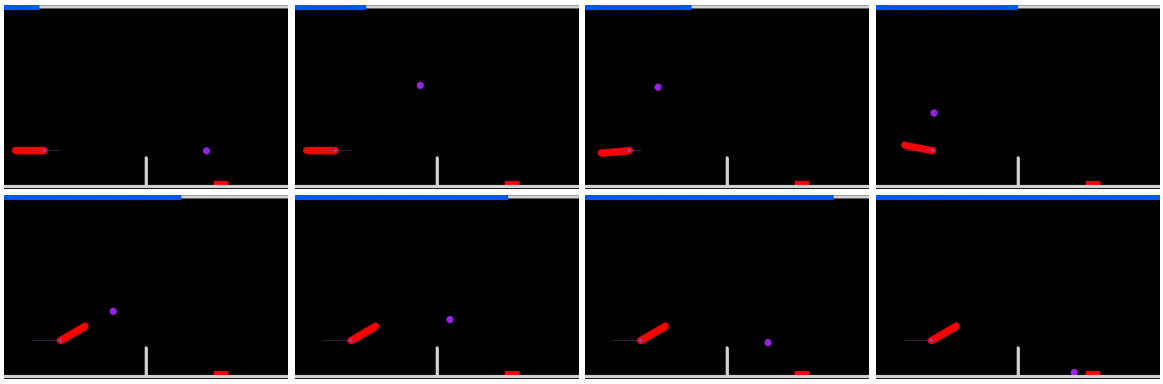

Figure 4.5: *Smash Task:* The ball is shot upward from the right-hand side, and a learnt agent controls the racket to rotate in the clockwise direction to smash the ball and make it fly toward the red target. Example images are stacked in chronological order from left to right and top to bottom. Time is indicated by the blue bar on the top of the images.

rate (SR).

Hit rate (HR) is defined as the proportion of episodes in which the agent manages to hit the ball, regardless of how the ball move after that.
Net Pass Rate (NPR) is defined as the proportion of episodes in which the ball flies and passes the net to the right-hand side of the scene.
Success Rate (SR) is defined as the proportion of episodes in which the ball reaches the red target. The reward for success episode does not need to be 1, because the target has a size, and the distance between the drop position and the target position is calculated from the centers of masses of the ball and the target.

The results are shown in Table 4.1. Overall, the radial basis function is found to be the best among other reward functions in all tasks. The radial basis function has a smoother surface than others, and this might lead learning to converge to the better local optimum. Serve task is clearly the easiest one to get a nearly perfect net pass rate and has the highest success rate among other tasks. Serve task is not challenging enough. For the remaining parts, we will create algorithms that improve both Counter and Smash tasks. We select the Radial Basis function as the reward function for the remaining experiments.

4.3 A multi-agent game

In the same way as in single-agent games, the impulse force used to shoot the tennis ball is randomly selected from the uniform distribution. One agent starts to hit the ball to reach the other side. Then, the other agent has to counter-hit the ball. Two agents have to play cooperatively to hold the game as long as possible, and the game ends when the tennis ball bounces on the floor for the second time. One agent observes another agent as a part of an environment. Each agent has its neural

Table 4.1: Performance of RL agents trained with different reward functions. The results are from 5 models with different training random seeds, and each model is evaluated on 1000 episodes, **HR** is Hit Rate (%) defined as the proportion of episodes that the agent manages to hit the ball, **NPR** is Net Pass Rate (%) defined as the proportion of episodes that the ball passes the net, and **SR** is Success Rate (%) defined as the proportion of episodes that the ball reaches the target. A left-side number is the mean over 5 runs from 5 random seeds, and a right-side number in a bracket (...) is the standard deviation over five runs.

Reward Functions	Serve		
	HR	NPR	SR
Linear	100.0 (0)	100.0 (0)	84.1 (7.6)
Polynomial	100.0 (0)	100.0 (0)	91.3 (4.3)
RBF	100.0 (0)	100.0 (0)	**95.5** (1.4)

Reward Functions	Counter		
	HR	NPR	SR
Linear	99.9 (0.2)	93.8 (2.6)	20.3 (13.0)
Polynomial	**100.0** (0)	89.1 (8.4)	21.3 (20.0)
RBF	99.8 (0.3)	**94.7** (2.4)	**37.0** (7.1)

Reward Functions	Smash		
	HR	NPR	SR
Linear	100.0 (0)	81.4 (2.6)	33.3 (5.6)
Polynomial	100.0 (0)	80.3 (2.2)	33.9 (3.4)
RBF	100.0 (0)	**83.0** (1.8)	**44.6** (7.5)

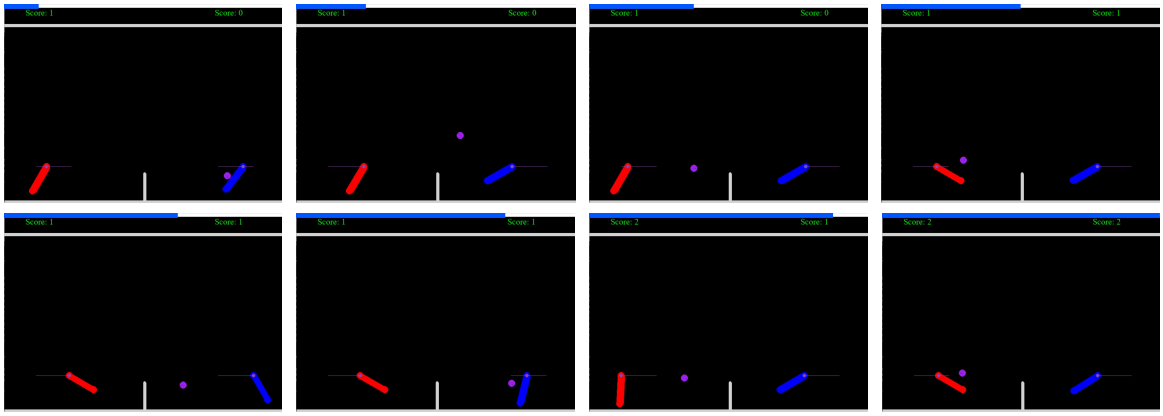

Figure 4.6: *Multi-agent Task:* The ball is shot upward from either the right-hand or left-hand side, and a learnt agent controls the racket to rotate in the counter-clockwise (for the left agent) direction to counter the other agent. The other agent does try to hit the ball in the same way. Two agents have to play with each other in a cooperative way to hold the ball as long as possible. Example images are stacked in chronological order from left to right and top to bottom. Time is indicated by the blue bar on the top of the images.

networks (both actor and critic networks), and their neural networks are trained separately and simultaneously. Agents are trained to play together using DDPG algorithm, the same as our single agent is trained. An example of how agents play together is shown in Figure 4.2 and Figure 4.6.

4.3.1 Reward Function

The ultimate goal of our multi-agent cooperative game is to hold the game as long as possible. How well the agents play in a game can be quantified by either the total time steps or the total number of ball hits by both agents. In the long episode, the agents might hit the tennis ball more than 10-20 times before the episode ends. If the reward is given to both agents at the final time step of the long episode, the agents might not learn something useful because the reward function is very sparse.

We propose two reward functions that encourage agents to play the game for as long as possible. Every time an agent makes a successful hit, a reward is given to it, to make the reward function dense.

Drop Reward: The agent is rewarded as $+1$, when the agent successfully hits the ball and the ball is dropped near the other agent. The region for rewarding is 120 pixels distance from the racket's centre. This number comes from $1.5 \times L_{racket}$, where L_{racket} is the length of the tennis racket.

Counter-hit Reward: An agent who hits the ball to its opponent's side will be rewarded as $+1$ after the opponent agent successfully receives the ball.

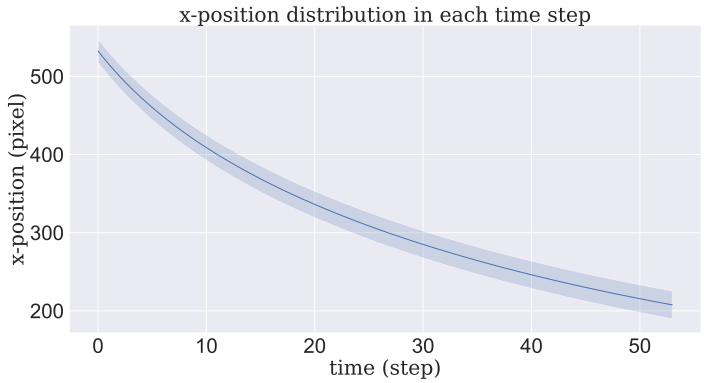

Figure 4.7: The figure shows the mean of x-position of the ball trajectories which are used in *Counter* task in each time step. Light-blue band represents standard deviation of x-position.

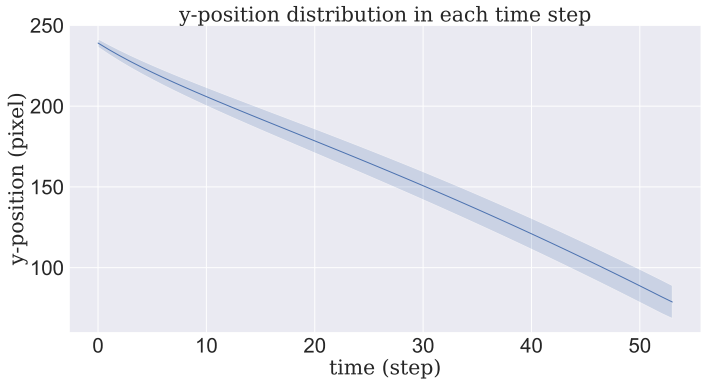

Figure 4.8: The figure shows the mean of y-position of the ball trajectories which are used in *Counter* task in each time step. Light-blue band represents standard deviation of y-position.

4.3.2 Experiments and Results

Implementation Detail: All models are trained with $2,000$ episodes using the DDPG algorithm, with the discount factor of 1.0 and the target update factor (τ) of 0.001. Each agent has its own fully-connected neural networks (an actor and a critic) with three hidden layers, and with the sizes of $[512, 256, 128]$, and has its own replay buffer with the size of $10,000$. All neural network models receive the left agent's state, the right agent's state, and the tennis ball's state, similar to a single agent's input but adding another agent's state. The critic's additional input is the output action from the actor. The agents are designed to receive the past states, so the state inputs of neural network models are $X = [S_t, S_{t-1}, S_{t-2}, S_{t-3}]$, where S_t is state variable at time t. The exploration is done by Ornstein–Uhlenbeck process, and the later episode has smaller exploration noise. The optimisation is done using Adam [73] with the learning rate of 0.0001, and we use a batch size of 128.

We defined two metrics that can inform how well the agents play cooperatively on

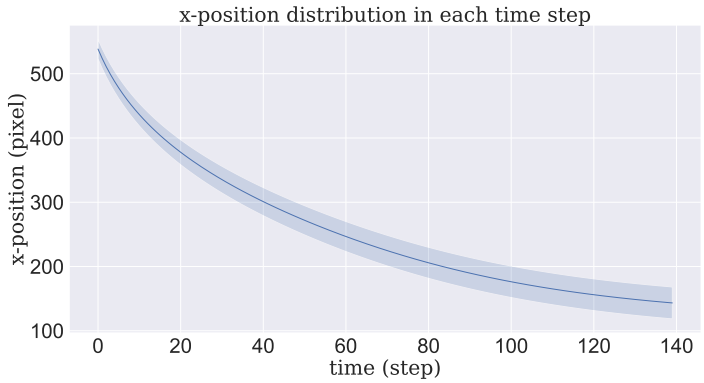

Figure 4.9: The figure shows the mean of x-position of the ball trajectories which are used in *Smash* task in each time step. Light-blue band represents standard deviation of x-position.

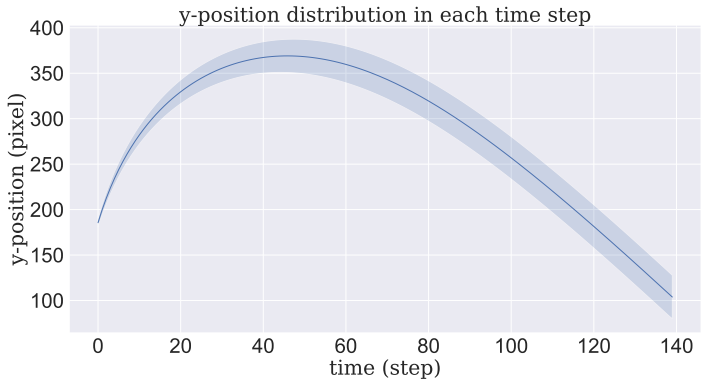

Figure 4.10: The figure shows the mean of y-position of the ball trajectories which are used in *Counter* task in each time step. Light-blue band represents standard deviation of y-position.

average.

Average Hits is defined as the average of ball hits by two agents that is calculated over evaluation episodes. This is our main metric that represents the overall performance of the agents. And this metric is the same as *Counter-hit* reward function. *Average Hits* is large if and only if the agents hold the ball for a very long time without fail.
Max Hits is defined as the maximum number of ball hits by two agents that is calculated from all evaluation episodes and all models with different training random seeds. The metric can indicate that some models are overfitting to some episodes if *Max Hits* is much higher than the *Average Hits*.
Min Hits is defined as the minimum number of ball hits by two agents that is calculated from all evaluation episodes and all models with different training random seeds.

Table 4.2: Performance of multi-agent RL trained with different reward functions. The results are from 5 runs from 5 random seeds, and each model is evaluated on 100 episodes, **Average Hits** indicates the average times of hitting the ball by both agents in one episode. **Max Hits** is the maximum number of hits by both agents observed during evaluation. A right-side number in a bracket (...) is the standard deviation over five running seeds.

Reward Functions	Multi-agent		
	Average Hits	Max Hits	Min Hits
Drop	6.81 (4.6)	**108**	0
Counter-hit	**11.1** (3.9)	33	0

In Table 4.2, *Counter-hit reward* seems to return better behaviour of agents, with almost two-fold higher *Average Hits* and slightly lower standard deviation. For *Drop Reward*, *Max Hits* is relatively much higher than *Average Hits*, which means some models trained by this reward function are overfitting to some episodes, and they do not perform well in overall since the *Average Hits* is low. Therefore, we decide to use *Counter-hit reward* for training models in the remaining multi-agent experiments.

Chapter 5

Single-Agent Planning Styles

Tennis players do not need to think about new actions (torque and force vectors) in every environment step. Instead, they swing the racket using a consistent amount of force and torque [74] that should be described by a few action vectors corresponding to keyframes or key events. Motivated by how humans play tennis, we investigate three problem formulations: full-episode planning, half-episode planning, and single-step planning, for the single-agent tasks. Full-episode planning is what we have done in the last part. The agent receives an input state and selects action in every environment step after the ball is shot from the right-hand side. In half-episode planning, the agent begins to observe and perform an action in every time step after the ball passes the net. In single-step planning, we formulate it as bandit optimisation. The agent observes a single state, and the agent selects only one action (both force and torque). Subsequently, the same action will be selected in every remaining step. Note that single-step planning can be implemented in full-episode and half-episode fashions, and we will investigate both of them. Both half-episode planning and single-step planning should reduce the search space for solutions.

5.1 Results

As mentioned earlier, we evaluate different planning styles for two challenging tasks: *Counter* and *Smash*. We will call an agent which is trained in the full-episode setting as a full-episode agent. The other agents which are trained in other settings are called in the same ways. All agents are trained with the same setting as in subsection 4.2.2

Figure 5.1 shows learning dynamics of four RL agents in *Counter* task. The full-episode agent (*FE* agent) achieves the highest expected return after learning converges. The half-episode agent (*HE* agent) has the second highest expected return. Interestingly, single-step-full-episode agent (*SFE* agent), while single-step-half-episode agent (*SHE* agent) can still accomplish the task but with poor performance. With two to four hundred training episodes, the *FE* agent significantly outperforms the *HE* agent.

Figure 5.2 shows learning dynamics of four RL agents in *Smash* task. Both *FE* and *HE* agents achieve the highest expected return after learning converges, while both *SFE* and *SFE* agents are not converged. With two hundreds training episodes, the *FE* agent significantly outperforms the *HE* agent.

Table 5.1 shows the evaluation results of both *FE* and *HE* agents performance on *Counter* and *Smash* tasks, and the *FE* agent has better performance than the *HE* agent based on most of the metrics, including average return (AR). The only exception is that the *HE* agent has a slightly higher success rate (SR) than the *FE* agent. The difference in SR is not significant.

In *Counter* task, the agent always fails to hit the ball to reach the target if the distance between the ball and the target at bounce time is greater than 440 (The width of the game screen is 800), as shown in Figure 5.3. The agent does not achieve the final goal (a ball reaching a target) if it hits the ball with insufficient racket velocity. The agent fails to return a ball if it hits the ball with too low a position (indicated by bounce angle). It is worth noting that the ball's bounce velocity is in proportion to the racket's velocity. A racket's velocity is a linear combination between the rotating point's velocity and the multiplication of angular velocity and distance from the rotating point to the contact point.

In *Smash* task, the agent mostly achieves the final goal if it hits the ball with sufficient speed and a small bounce angle (around -10 to 10 degree). The agent fails if it hits the ball with a large positive and negative bounce angle. Furthermore, the agent has a chance to achieve the final goal regardless of the distance between a ball and a target at bounce time.

5.2 Discussion

We find empirically that the single-step setting does not work in both tasks. The possible reason is that the agent has only one chance to find single values of torque and force that move a racket to hit a ball to the desired target. The action vector must lead the racket to reach the perfect hit position and hit velocity to return the ball to the other side. Although the loss of a neural network is near zero, the neural network prediction will have an error. The neural network cannot correct the error because it has only one chance to predict the optimal action. In contrast to single-step planning, multi-step planning allows an agent to select multiple actions so that the agent does not need to select the perfect action in the first stance. The agent has multiple chances to adjust the swing in every environmental step such that the agent makes sure that the racket will hit the ball to the desired target.

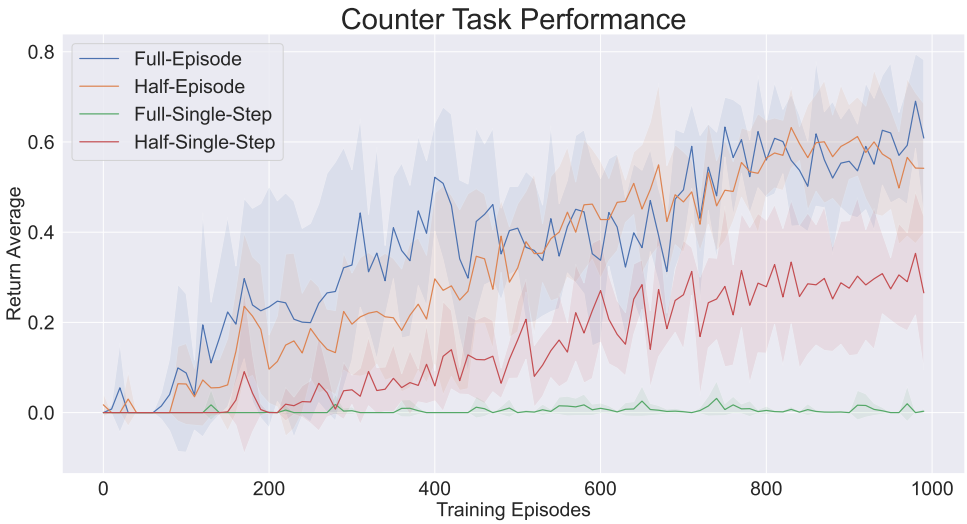

Figure 5.1: The figure shows the average of the expected return over 10 runs for 4 different planning styles. Performance is evaluated on *Counter Task*. *Full-Episode:* the agent uses full-episode planning, *Half-Episode:* the agent uses half-episode planning, *Full-Single-Step:* the agent uses full-episode planning along with single-step action selection, *Half-Single-Step:* the agent uses half-episode planning along with single-step action selection.

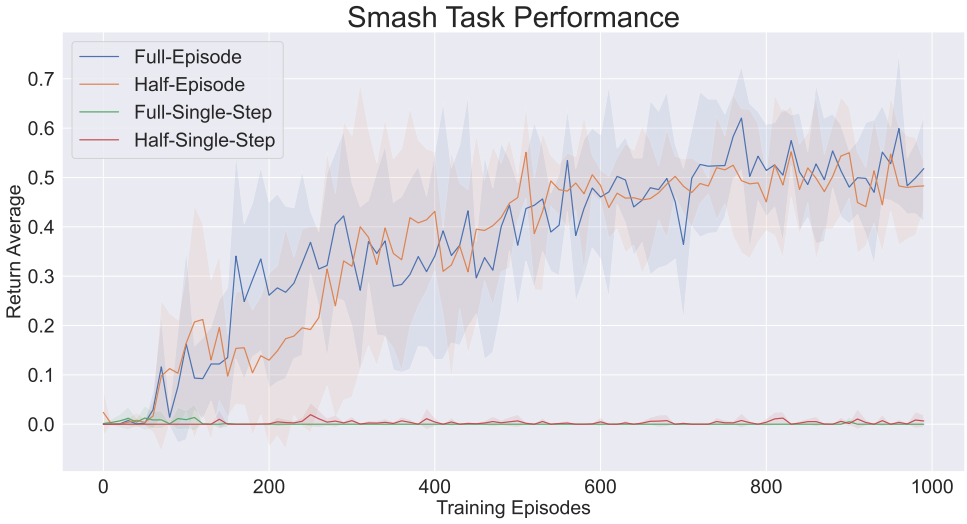

Figure 5.2: The figure shows the average of the expected return over 10 runs for 4 different planning styles. Performance is evaluated on *Smash Task*. *Full-Episode:* the agent uses full-episode planning, *Half-Episode:* the agent uses half-episode planning, *Full-Single-Step:* the agent uses full-episode planning along with a single-step action selection, *Half-Single-Step:* the agent uses half-episode planning along with single-step action selection.

Table 5.1: Performance of Full-Episode (FE) and Half-Episode (HE) agents in *Counter* and *Smash* tasks. The results are average over 5 training random seeds, and each model is evaluated on 1000 episodes, **HR** is Hit Rate (%) defined as the proportion of episodes that the agent manages to hit the ball, **NPR** is Net Pass Rate (%) defined as the proportion of episodes that the ball passes the net, **SR** is Success Rate (%) defined as the proportion of episodes that the ball reaches the target, and **AR** is average return over all episodes determined by radial basis reward function (Equation 4.7)

Methods	Counter			
	HR	NPR	SR	AR
Full-Episode	100.0 (0)	**93.8** (2.8)	43.4 (8.9)	**0.74** (0.04)
Half-Episode	100.0 (0)	80.3 (10.9)	**44.0** (11.7)	0.66 (0.10)

Methods	Smash			
	HR	NPR	SR	AR
Full-Episode	99.9 (0.1)	**80.9** (2.5)	36.0 (2.9)	**0.58** (0.03)
Half-Episode	99.9 (0.1)	78.7 (3.0)	**36.7** (3.1)	0.57 (0.03)

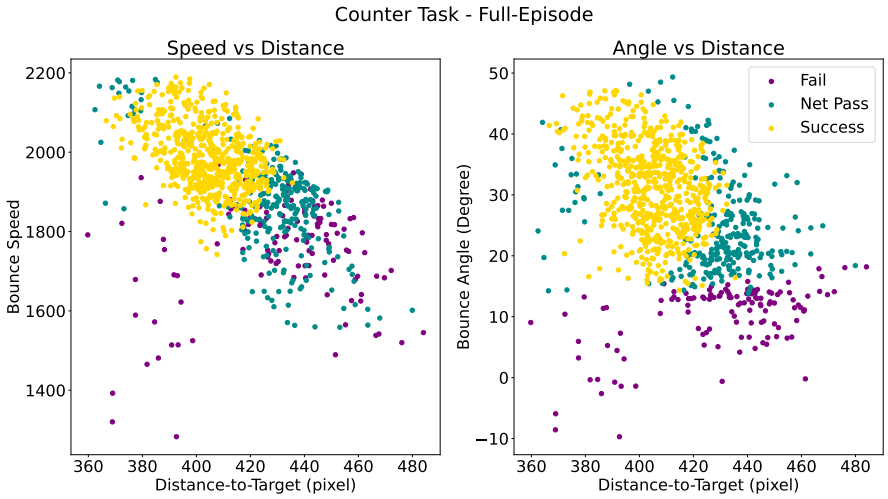

Figure 5.3: The figure shows the bounce velocity of a ball hit by a trained agent versus the distance between the ball and the target at hit time from each evaluation episode. The agent is trained to accomplish *Counter task* in the Full-Episode setting. Speed is the magnitude of velocity, and the angle is the velocity vector's angle with respect to the horizontal line. Each point in the figure represents one episode. The data are grouped into three classes: Fail, Net Pass, and Success. **Fail:** An agent manages to hit a ball, but the ball does not fly over and pass the net. **Net Pass:** A ball flies over and passes the net, but the ball does not reach a target. **Success:** A ball reaches a target.

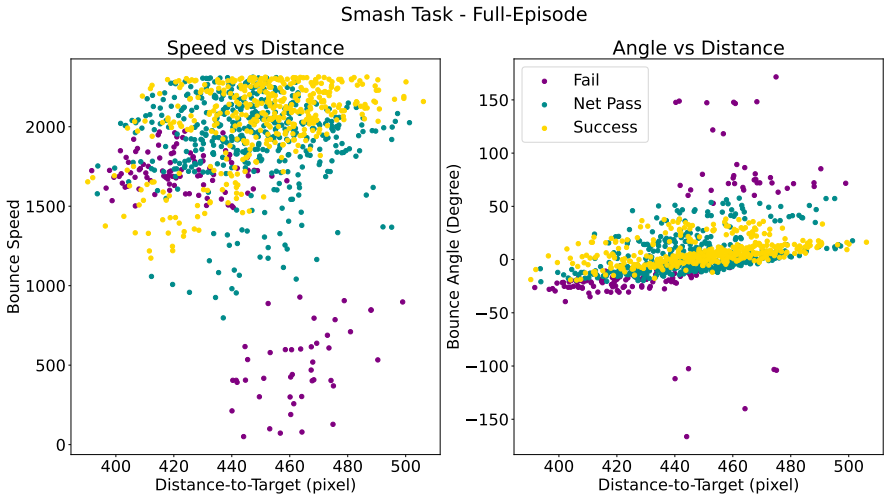

Figure 5.4: The figure shows the bounce velocity of a ball hit by a trained agent versus the distance between the ball and the target at hit time from each evaluation episode. The agent is trained to accomplish *Smash task* in the Full-Episode setting. Speed is the magnitude of the velocity, and the angle is the velocity vector's angle with respect to the horizontal line. Each point in the figure represents one episode. The data are grouped into three classes: Fail, Net Pass, and Success. **Fail:** An agent manages to hit a ball, but the ball does not fly over and pass the net. **Net Pass:** A ball flies over and passes the net, but the ball does not reach a target. **Success:** A ball reaches a target.

Chapter 6

Trajectory Prediction

6.1 Motivation

In the real world, physical parameters such as wind force, drag force, and ball mass can change. Humans are still capable of predicting tennis ball trajectories with different physical parameters. For example, considering the extreme case, the tennis player, who switches to playing badminton, might respond to the shuttlecock very quickly by moving to the projected optimal hit point and smashing the shuttlecock back to her opponent. The human anticipation of physical event outcomes is generic such that it can be applied to many situations with different physical parameters. In this chapter, we want to create a dynamics model with the generality of dynamics prediction like humans. The problem we want to solve is building a model that can make predictions conditioned on physical parameters under the assumption that these parameters are unknown.

6.2 Problem Formulation

The structure of dynamics models can be decomposed into two parts: knowledge and parameter. The parameter model estimates physical parameters from observations, and the knowledge model makes the predictions of future observations conditioned on estimated physical parameters. Here we use neural networks as function approximators for the dynamics model. The knowledge and parameter models receive the current and previous states of a tennis ball. The knowledge model is trained to predict the next state of the tennis ball with supervised learning. The parameter model is expected to estimate physical parameters from input states. In a simulation, it is easy to predict physical parameters using supervised learning, as the true physical parameters are given by a simulator. However, we do not know physical parameters in real-world settings. For this reason, our parameter model is not trained to predict physical parameters directly. Instead, it is trained to capture the invariant property of latent outputs z given input states from the same physical parameters.

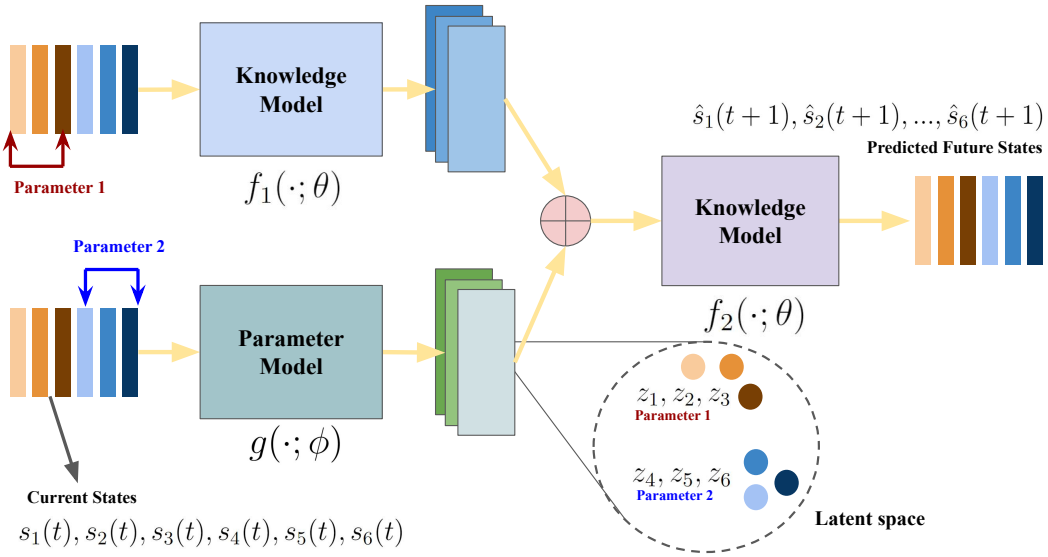

Figure 6.1: *Contrastive Dynamics Model (CDM)* is parameterised by neural networks: θ for the knowledge model and ϕ for the parameter model. Suppose we have a batch size of 6. Both knowledge and parameter models receive the same inputs $s_1(t), ..., s_6(t)$, where $s_1(t), s_2(t), s_3(t)$ (orange ones) share the same physical parameter, and also for $s_4(t), s_5(t), s_6(t)$ (blue ones). The knowledge model has an intermediate layer such that the output in this layer is fused with the output of the parameter model z. The latent variables $z_1, ..., z_6$ are trained to be close to each other in the group with the same physical parameter by contrastive representation learning.

Suppose we have two input states s_1 and s_2, corresponding to two trajectories. If s_1 and s_2 share the same physical parameters, the parameter model's outputs, z_1 and z_2, should be the same ideally. The transformation that returns unchanged outputs from different inputs is *invariant*. On the other hand, if s_1 and s_2 have different physical parameters, z_1 and z_2 should be different. Here we will relax the invariant condition such that z_1 and z_2 from the same parameters can have a small distance instead of zero distance. Contrastive representation learning [22, 23] is one approach to construct latent representation z that contains this property. In the next part, we will give a brief explanation of how contrastive learning has been developed through the lens of information theory. Subsequently, we will explain our proposed algorithm based on the contrastive learning framework to predict tennis ball trajectories with different physical parameters.

6.3 Contrastive Learning

Contrastive representation learning is used in unsupervised learning setting to learn useful representations for downstream tasks, such as speech recognition, image classification, visual reinforcement learning, and natural language processing [22, 23, 24, 75]. Suppose we have two random variables corresponding to the different tennis trajectories s_1 and s_2 but share the same physical parameter. The original

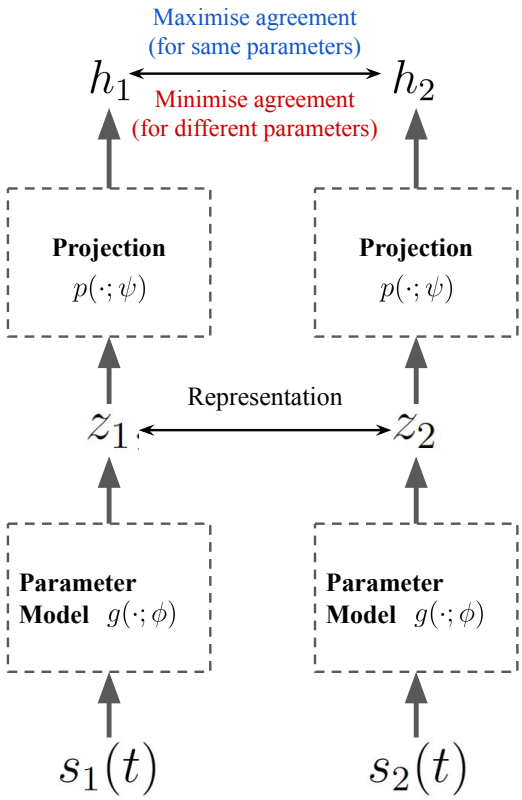

Figure 6.2: *Contrastive Representation Learning:* Two input states s_1 and s_2 are randomly sample from datasets. Parameter model transforms input states into latent variables z_1 and z_2. Latent variables are transformed into h_1 and h_2 by a projection head $p(\cdot;\psi)$ which are used to calculate contrastive loss. Note that the projection head $p(\cdot;\psi)$ is used only for calculating contrastive loss during training and is not used for predicting the next state at the inference time.

motivation behind contrastive learning is to learn the function g that maps a state variable s in to the latent representation z in such a manner that mutual information between z_1 and z_2 as in Equation 6.1 maximally preserves [22]. In our case, we suppose that $s_1 = [s_1(0), s_1(1), ...S_1(T)]$ and$s_1 = [s_2(0), s_2(1), ...S_2(T)]$ can be any trajectories from the same physical parameter π and their initial conditions $s_1(0)$ and $s_2(0)$ are independent. We can write the data generating processes of trajectories as probabilistic graphical models in Figure 6.3. As the initial states are independent, the only mutual information in $z_1 = g(s_1)$ and $z_2 = g(s_2)$ is the information of their physical parameter, which means the representation z should contain the complete information of the physical parameter; if mutual information $I(z_1; z_2)$ is maximised. The mutual information $I(z_1; z_2)$ is defined on probability density function $P(\cdot)$ as in Equation 6.1.

$$I(z_1; z_2) = \sum_{z1,z2} P(z_1, z_2) \log \frac{P(z_1|z_2)}{P(z_1)} \tag{6.1}$$

We model the density ratio as,

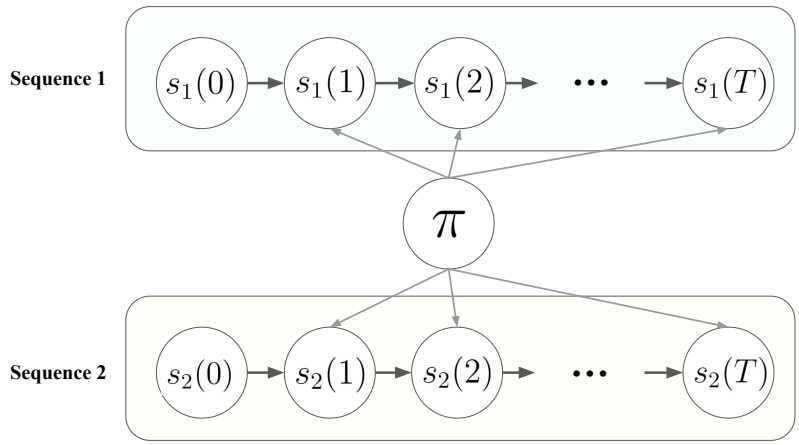

Figure 6.3: *Data generating process:* Two state sequences (ball's trajectories in our case) are generated from their initial states $S_1(0)$ and $S_2(0)$. The distribution of next state $S(t+1)$ is conditioned on both current state $S(t)$ and the physical parameter π.

$$\frac{P(z_1|z_2)}{P(z_1)} \propto \exp(h_1^T h_2) \tag{6.2}$$

where $h = p(z;\psi)$, note that $p(\cdot;\psi)$ is the projection function in Figure 6.2, not a probability density function $P(\cdot)$.
We now know how to model the density ratio term in mutual information definition, the next thing is to maximise it. The authors of [22] propose *InfoNCE* loss $\mathcal{L}_N$ to maximise the lower bound of mutual information in Equation 6.1 by minimising the loss $\mathcal{L}_N$,

$$\mathcal{L}_N = -\mathbb{E}_{z1,z2}\left[\log \frac{\frac{P(z_1|z_2)}{P(z_1)}}{\sum_{z_i \in \mathcal{Z}} \frac{P(z_i|z_2)}{P(z_i)}}\right] \tag{6.3}$$

In the paper, they prove that,

$$I(z_1; z_2) \geq \log(N) - \mathcal{L}_N \tag{6.4}$$

Minimising the loss $\mathcal{L}_N$ is equivalent to maximising the lower bound of mutual information between z_1 and z_2. Other than the information-theoretic perspective, we can think of minimising $\mathcal{L}_N$ as minimising the distance between positive samples z_1 and z_2, which share the same physical parameters, and implicitly maximising the distance between negative examples. Next, we will explain how we use *InfoNCE* loss ($\mathcal{L}_N$) to train our proposed *CDM*.

6.4 Contrastive Dynamics Model

As we mentioned earlier, the dynamics model can be decomposed into two parts: knowledge and parameter. An overview of our model architecture is shown in Figure 6.1. The knowledge and parameter models receive input state $s(t)$, which can

Algorithm 1 Contrastive Learning for Dynamics Prediction

Input: arrays of state $\boldsymbol{S}$ and $\boldsymbol{S}^{next}$ with the size of $M \times N \times P$
▷ $\boldsymbol{S}^{next}$ is true next state of $\boldsymbol{S}$
▷ M is the number of datasets D, each dataset contains states from the same physical parameter
▷ N is the number of samples per dataset
▷ P is the dimensions of state

for sampled batch $\boldsymbol{S} = \{\{S_{m,n}\}_{n=1}^{N} \in D_m\}_{m=1}^{M}$ **do**
  $z_{m,n} = g(S_{m,n}; \phi)$ ▷ Parameter model transforms inputs to latent variables
  $h_{m,n} = p(z_{m,n}; \psi)$ ▷ latent variables are projected to calculate contrastive loss

  ▷ The first loss is prediction error $\mathcal{L}_{mse}$

  $\mathcal{L}_{mse}(\theta, \phi) = \sum_{m,n} \left\| (S_{m,n}^{next} - S_{m,n}) - f_2(f_1(S_{m,n}; \theta), z_{m,n}; \theta) \right\|^2$

  ▷ The second loss is contrastive loss $\mathcal{L}_N$

  Define: $sim_{Bilinear}(m_1, m_2, n_1, n_2) = h_{m_1,n_1}^T h_{m_2,n_2} / \tau$
  Define: $sim_{L2}(m_1, m_2, n_1, n_2) = -\left\| h_{m_1,n_1} - h_{m_2,n_2} \right\|^2$

  Define: $C(m, n_1, n_2) = -\log\left(\frac{\exp(sim(m,m,n_1,n_2))}{\sum_{l_1=1}^{M}\sum_{l_2=1}^{M}\sum_{p=1}^{N} \mathbb{1}_{[l_1 \neq m \vee l_2 \neq m \vee p \neq n_1]} \exp(sim(l_1,l_2,n_1,p))}\right)$

  $\mathcal{L}_N(\phi, \psi) = \frac{1}{MN} \sum_{m}^{M} \sum_{n_1}^{N} \sum_{n_2}^{N} \mathbb{1}_{[n_1 \neq n_2]} C(m, n_1, n_2)$

  Update $f(\cdot; \theta)$: $\theta \leftarrow \theta - \nabla_\theta \mathcal{L}_{mse}(\theta, \phi)$
  Update $g(\cdot; \phi)$: $\phi \leftarrow \phi - \nabla_\phi \mathcal{L}_{mse}(\theta, \phi) - \nabla_\phi \mathcal{L}_N(\phi, \psi)$
  Update $p(\cdot; \psi)$: $\psi \leftarrow \psi - \nabla_\psi \mathcal{L}_N(\phi, \psi)$
end for
return Knowledge model $f(\cdot; \theta)$ and Parameter model $g(\cdot; \phi)$

contain multiple frames. The parameter model $g(\cdot;\phi)$ transforms the input state to the latent variable z, and the latent variable z is fed into the knowledge model at the intermediate layer. The knowledge model is trained to minimise the mean square error between the next state prediction and the true next state. The parameter model is designed to extract physical parameters from state observations, and it is trained with contrastive learning approach–i.e., to maximise the mutual information between z_1 and z_2, which share the same physical parameter. The projection head $p(\cdot;\psi)$is for modeling probability density ratio as we explained in the previous part.

The implementation detail of our algorithm is explained in detail in Algorithm 1. We found that predicting the difference between the current and next state (i.e., $s(t+1) - s(t)$ gives much better results than predicting the next state, which is the same finding as in [17]. Therefore, the MSE loss we use is,

$$\mathcal{L}_{mse}(\theta,\phi) = \sum_{s\in\mathcal{D}} \|(s(t+1) - s(t)) - \delta\hat{s}(t)\|^2 \tag{6.5}$$

where $\delta\hat{s}$(t) is prediction from f's output. Other than bilinear similarity function $sim_{Bilinear}(i,j) = h_i^T h_j$, we also investigate on another similarity function as negative L2 square $sim_{L2}(i,j) = -\|h_i - h_j\|^2$.

6.5 Experiment

We hypothesise that our contrastive learning framework can improve prediction accuracy in multiple-physical-parameter settings. To test this hypothesis, we evaluate three dynamics models (DM).

Vanilla DM is the model with the same architecture as in Figure 6.1, but the contrastive loss is ablated during training. That means this model is trained with only prediction loss ($\mathcal{L}_{mse}$).
Bilinear CDM is the model in Figure 6.1 and it is trained with both prediction loss ($\mathcal{L}_{mse}$) and contrastive loss ($\mathcal{L}_N$) as explained in Algorithm 1. The similarity function used in this model is the Bilinear function.
L2CDM is a similar model as **Bilinear CDM**, and the only difference is the similarity function. This model uses a negative L2 square function instead of a Bilinear function.

Implementation Detail: All models here have the same architectures and are trained with the same setting. The batch size is 125, which contains 5 samples per physical parameter and 25 physical parameters. Regarding the Figure 6.1, f_1 and g are fully-connected neural networks with the hidden sizes of $[256, 128]$. f_1 has the 64 output units and g has 32 output units. f_2 is a fully-connected neural network with the hidden sizes of $[128, 64]$. The projection head p is a fully-connected neural network with the hidden sizes of $[64, 32]$ and has 16 output units. All hidden layers use ReLU as the activation function. The models are trained for 200 epochs using Adam [73]

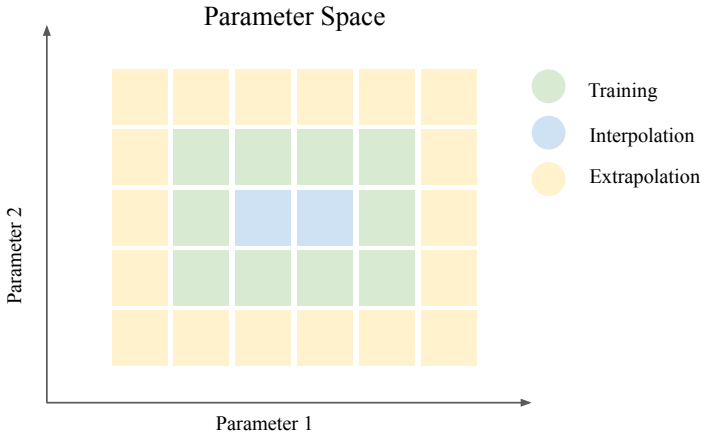

Figure 6.4: *Parameter space:* The trajectory datasets are classified into three groups: training, interpolation and extrapolation. Training datasets are used to train dynamics model, interpolation and extrapolation datasets are unseen during training.

optimiser with the learning rate of 0.0001.

We want the learnt dynamics models to predict trajectories with different physical parameters, including trajectories that the model has not been trained with before. Hence, we create three different evaluation datasets: Training, Interpolation, and Extrapolation datasets. Here we can define a physical parameter as a 2D vector. Each physical parameter has two elements: wind force $\boldsymbol{W}$ and ball mass m, which describes the equation of motion in Equation 4.1. Considering physical parameters in Euclidean space $\mathbb{R}\times\mathbb{R}$, the Interpolation dataset contains trajectories with physical parameters that are surrounded by training physical parameters, and the Extrapolation dataset contains trajectories with physical parameters that surround training physical parameters as shown in Figure 6.4. We generate datasets in which each physical parameter has its corresponding 100 tennis ball trajectories.

Training Dataset contains trajectories from 25 different physical parameters, including trajectories which have zero wind force and original ball mass as in subsection 4.2.2 and chapter 5. Training Dataset is only for evaluation, so it is not the same as the dataset we use to train dynamics models. Trajectories in this dataset are generated from the same physical parameters as the data that we use to train the models.

Interpolation Dataset contains trajectories from 4 different physical parameters, which have small wind forces in either left or right direction. Ball mass values are slightly different (either higher or lower value) from the original ball mass value used in subsection 4.2.2 and chapter 5.

Extrapolation Dataset contains trajectories from 16 different physical parameters, all of them have higher values of wind force than those values in Training and Interpolation datasets. Each ball mass in this dataset is either larger or smaller than the original ball mass.

6.6 Results

A ball's state consists of its position and velocity. Therefore, we separate predictions of position and velocity. The evaluation metric we use is the root mean square error (RMSE). The game's screen in Figure 4.3 has resolution of 800×600 pixels. RMSE for position prediction has unit of *pixels*, and RMSE for velocity prediction has the unit of *pixels per second*. For position prediction, *L2CDM* has the lowest RMSE among other models in all datasets. *Vanilla DM* is the second best in Training and Interpolation Datasets, while *Bilinear CDM* is the second best for Extrapolation Dataset. However, *Bilinear CDM* performs worst in Training and Interpolation Datasets. In overall, all prediction errors (RMSE) are around 9 pixels for all models. For velocity prediction, the difference in RMSE between each model in each dataset is not significant, and, all models have RMSE of around 45.34 to 45.54. Hence, our contrastive learning framework does not improve velocity predictions.

Other than the prediction task, the other goal is to make the model implicitly estimate physical parameters. Thus, we investigate what representation z each model learns. Figure 6.5 and Figure 6.6 show t-SNE plots [76] of the latent representations z in *L2CDM* and *Vanilla DM* respectively. *L2CDM* can disentangle trajectories with different physical parameters in both Training and Extrapolation Datasets. Especially the latent variables for Training Dataset are separated into about 5 clusters. While *Vanilla DM* cannot disentangle trajectories with different physical parameters in Training Dataset, it can slightly disentangle trajectories in the Extrapolation Dataset as cool tone and warm tone samples are separated.

6.7 Discussion

6.7.1 Result Analysis

Experimental results suggest that L2 contrastive loss improves position prediction accuracy. However, it only improves 0.1 pixel for in-distribution (Training and Interpolation Datasets) and 0.2 pixel for out-of-distribution (Extrapolation Dataset) from the original dynamics model. Since the improvement is small, it is possible that the model trained with $\mathcal{L}_{mse}$ already learns to predict trajectories conditioned on different physical parameters. We train models using 25 different parameters and with more than 100,000 training samples. With these large data, it might not be too difficult for *Vanilla DM* to find an optimal solution that could extract physical parameters and incorporate these parameters into the knowledge model to adjust predictions conditioned on their physical parameters implicitly. The bottom image of Figure 6.6 supports this argument. If a pair of trajectories (s_1, s_2) has a large enough distance in physical parameter space, their corresponding latent variables (z_1, z_2) can appear in the different clusters. Furthermore, velocity prediction errors are high, and the contrastive learning method does not reduce these errors. It is something that could be investigated in future studies. For the pure trajectory prediction task, we can reduce both position and velocity prediction errors by reducing

temporal step size, i.e., increasing the temporal sampling rate for state perception. However, to incorporate trajectory prediction into RL, we need the same temporal step size. Reducing temporal step size causes RL to learn slower because there are more frames per episode, and many frames in an episode are not important for RL to solve tasks. For this reason, we select a temporal step size of 67 milliseconds (15 Hz of the temporal sampling rate), which is reasonable for many existing RL tasks [28, 29].

6.7.2 Limitations

Although we found that adding the invariant property to the parameter model improves prediction by a small margin, we still do not know if the latent variables contain physical parameter information. It is possible that the latent variables are just for clustering the different physical parameters, not for estimating their values. Future investigation is necessary to confirm the alignment between latent variables and physical parameters. Correlation between latent variables and their corresponding physical parameters can be used to test if they are linearly aligned. For nonlinear relation, Deep Canonical Correlation Analysis [77] could be used to test the alignment.

6.7.3 Future works

Our generative model assumption of state trajectories is only conditioned on physical parameters. The assumption is not applicable to control problems where external actions can affect state trajectories. Future works could investigate the action-conditioned version of our *Contrastive Dynamics Model* and test it on many kinds of real-world dynamics such as robot walking, robot object manipulation, and drone control.

6.7.4 Connection to other works

The authors of [16] use variational inference to find latent variables z, with a similar generative assumption to the one in Figure 6.3, for the estimation of physical parameters, and their method allows the dynamics model adapted and generalised to new physical parameters. In contrast to our work, we do not assume latent distribution as an explicit probability density function, so we cannot use variational inference to find the posterior distribution $p(z|s)$ directly. We make the assumption that contrastive learning can construct invariance property in the transformation. The other similar work is *Noether networks* [70], they use a meta-learning framework to construct invariant transformation by neural networks that extract conserved quantity from physical observations (e.g., conserved energy, momentum, or angular momentum). *Noether networks*, which entails invariance property, is found to improve long-horizon (large time step) prediction accuracy. Unlike *Noether networks*, we assume our observations are generated from multiple physical parameters, and both energy

Table 6.1: Evaluation results from three dynamics models on three evaluation datasets. Left numbers are average of denormalised root mean square errors (RMSE) (in the unit of *pixels* for position prediction and the unit of *pixels per second* for velocity prediction) and bracketed numbers are standard deviations of RMSE. The results are calculated from 10 models with different training random seeds.

	Training		Interpolation		Extrapolation	
	Position	Velocity	Position	Velocity	Position	Velocity
Vanilla DM	9.28 (0.42)	46.34 (0.22)	9.25 (0.39)	46.53 (0.23)	9.60 (0.67)	45.42 (0.14)
Bilinear CDM (Ours)	9.38 (0.50)	46.35 (0.34)	9.35 (0.49)	46.54 (0.33)	9.58 (0.57)	45.41 (0.32)
L2CDM (Ours)	**9.17 (0.28)**	46.35 (0.36)	**9.14 (0.28)**	46.53 (0.40)	**9.40 (0.30)**	45.42 (0.36)

and momentum of the tennis ball are not conserved because non-conservative external forces (drag and wind forces) are exerted on the ball. In our case, the conserved variables are physical parameters, not physical quantities.

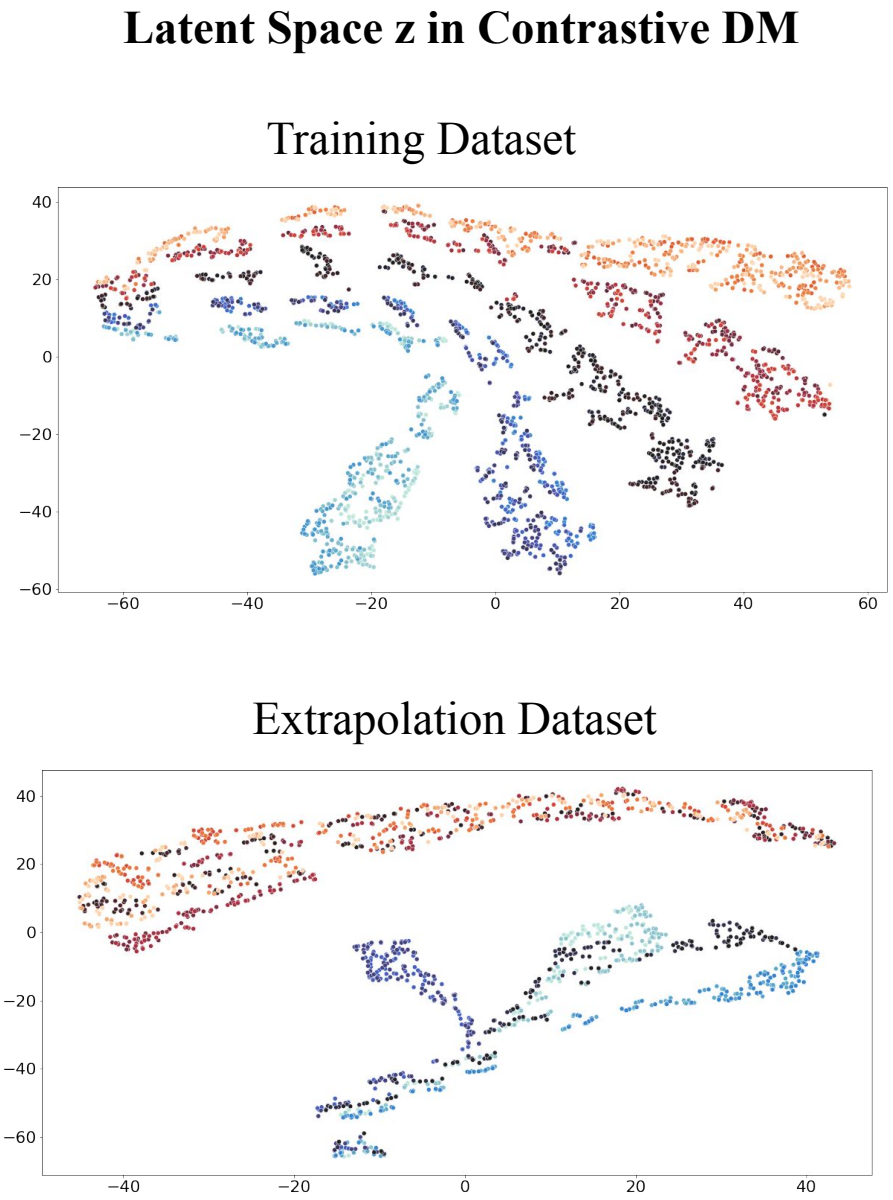

Figure 6.5: t-SNE plot of latent variable z in *L2CDM*. There are 25 colours represent 25 different physical parameters in Training Dataset, and 16 colours for Extrapolation Dataset. A pair of samples with similar colours has small distance in parameter space, while a pair of samples with contrast colours (e.g., light blue and dark red) has large distance in parameter space

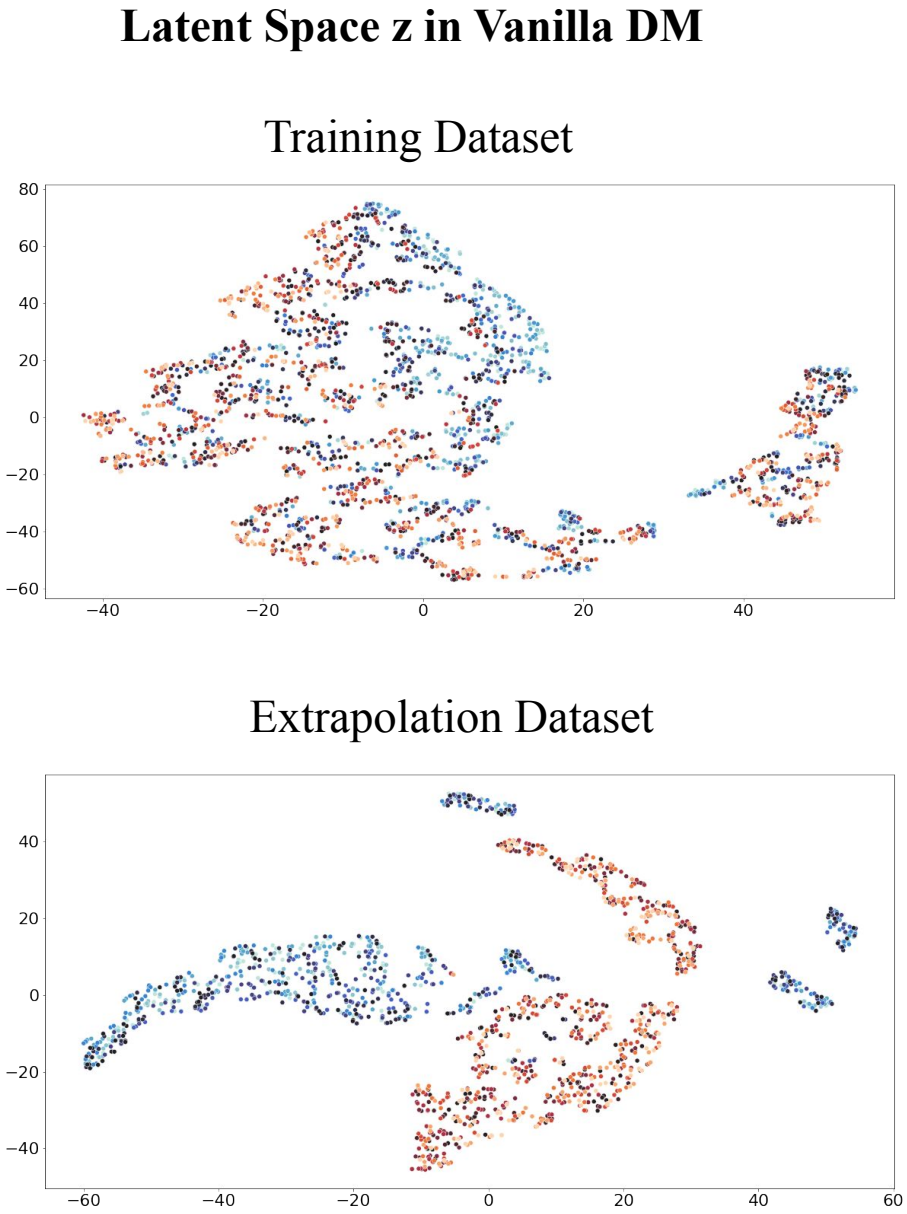

Figure 6.6: t-SNE plot of latent variable z in *Vanilla DM*. There are 25 colours represent 25 different physical parameters in Training Dataset, and 16 colours for Extrapolation Dataset. A pair of samples with similar colours has small distance in parameter space, while a pair of samples with contrast colours (e.g., light blue and dark red) has large distance in parameter space

Chapter 7

Dynamics-aware RL

7.1 Motivation

Model-based RL is thought to be more robust than model-free RL because model-based RL captures and utilises dynamics knowledge of an environment. However, model-based RL often has high prediction errors in complex dynamics, and this error causes model-based RL to have poor performance. While model-free RL can achieve high performance without explicit dynamics predictions, it is susceptible to unseen environment dynamics. Here we combine the advantages of model-free and model-based approaches by augmenting predicted future states of the environment to model-free RL's input. The whole environment's dynamics are difficult to accurately predict since it contains many nonlinear relationships, such as the interaction between ball and ground, the interaction between ball and racket, and falling ball dynamics. Instead of predicting the whole dynamics, we create the dynamics model learnt to predict only a flying tennis ball's trajectory where the ball's dynamics can be described by a single differential equation from Newton's second law of motion. Similarly, a tennis player knows where a ball will be dropped and decides how much she should swing a racket to hit the ball for flying and landing on another side.

7.2 Methods

We propose *Dynamics-aware RL (DynARL)*, an actor-critic-based RL model that receives predicted future states of a flying tennis ball as an auxiliary input as shown in Figure 7.1. The predicted future states come from a learnt dynamics model which has been trained prior to RL training in chapter 6. The dynamics model is not further trained during RL training. All models in this experiment are trained in DDPG setting, which has been explained earlier in chapter 3, only one different part is that the actor and critic have additional inputs.

7.2.1 Evaluation Settings

We use two evaluation settings for both single-agent and multi-agent tasks.

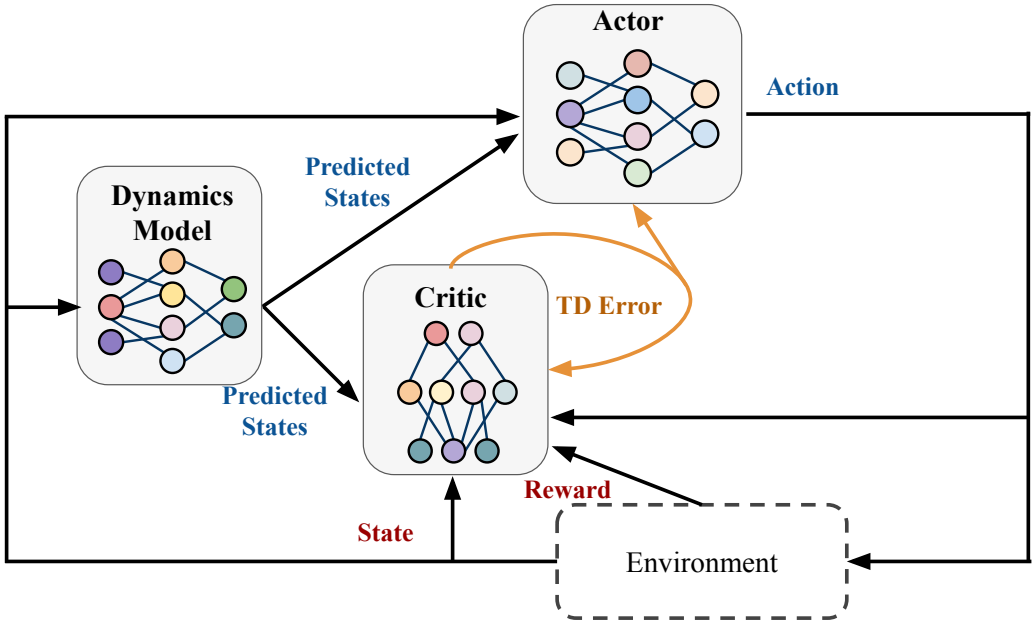

Figure 7.1: *DynARL Architecture*

In-distribution Dynamics: Models are evaluated with a single physical parameter which is the same as the parameter used for training all RL models in chapter 4.
Out-of-distribution Dynamics: For single-agent tasks, the physical parameter, a vector containing tennis ball mass and wind magnitude, is randomly selected in every episode. For the multi-agent task, wind magnitude is randomly selected in 100 environment steps, while the tennis ball mass remains the same. In other words, the flying ball's dynamics function can be changed instantly during the same episode in the multi-agent task. Therefore, the multi-agent task is very challenging compared to the single-agent tasks.

Trajectory distributions of the flying tennis ball of both training and out-of-distribution dynamics are shown in Figure 7.7, Figure 7.8, Figure 7.9 and Figure 7.10.

7.2.2 Models

We will investigate on performances of three DynARL models

Vanilla RL is the normal actor-critic model trained in DDPG setting.
DynARL + DM is dynamics-aware RL which has the dynamics model trained with $\mathcal{L}_{mse}$ in chapter 6.
DynARL + L2CDM is dynamics-aware RL which has the dynamics model trained with $\mathcal{L}_{mse}$ and $\mathcal{L}_N$ in chapter 6. The similarity function is $L2$
DynARL + True Dynamics is dynamics-aware RL with true dynamics model from simulation.
Nonstationary RL is the normal actor-critic model as *Vanilla RL*, but it is trained

using all physical parameters in the out-of-distribution dynamics evaluation. This model can be thought of as the upper-bound model for *Vanilla RL* when it is trained with the same physical parameters as the physical parameters of the out-of-distribution dynamics evaluation. The model is only evaluated with out-of-distribution dynamics.

7.3 Experiment

The hypothesis of this experiment is that augmenting future prediction improves RL model's performance in seen and out-of-distribution dynamics. If the hypothesis is proved to be true, we also want to know how the prediction accuracy of a dynamics model affects RL model's performance. Although the assumption of knowing true dynamics is not applicable in a real-world setting, we still use *DynARL + True Dynamics* to find the upper-bound performance of dynamics-aware RL when it has perfect or zero-error predictions. The comparison between *DynARL + DM / CDM* and *DynARL + True Dynamics* will answer us whether DynARL need near-perfect dynamics model to reach good performance.

All models are trained with the same setting as in subsection 4.2.2 and subsection 4.3.2. All models are also trained with the same physical parameter as the physical parameter used in subsection 4.2.2 and subsection 4.3.2. The only exception is *Nonstationary RL*, which is trained with the physical parameters in the *out-of-distribution Dynamics* evaluation. Learnt dynamics models (of *DynARL + DM* and *DynARL + L2CDM*) make the first step prediction conditioned on true current state and then they use the prediction as inputs to predict the further state. We use 4 predicted future states as input for all DynARL models because we have found that it is the best in our preliminary experiments. We train each type of model for 5 times using different random seeds. In each training, the best model is saved from being evaluated.

7.3.1 Results

The detailed results of *Counter*, *Smash* and *Multi-agent* tasks are shown in Table 7.1, Table 7.2, and Table 7.3 respectively. For in-distribution dynamics evaluation, DynARL models perform slightly better than the vanilla RL models for *Counter* and *Smash* tasks, and they perform much better than the vanilla RL model for *Multi-agent* task. DynARL significantly outperform *Vanilla RL* in out-of-distribution dynamics settings, especially for *Counter* task. However, for the out-of-distribution dynamics settings of *Smash* and *Multi-agent* tasks, only *DynARL + True Dynamics* outperforms *Vanilla RL*. The examples of agent's behaviours and ball's trajectories in both success and failure cases are shown in Figure 7.5 and Figure 7.6. Agent fails to return the ball when the wind force is too strong and blows the ball to the right side. Lastly, DynARL does not accelerate learning convergence compared to vanilla RL as shown in Figure 7.2, Figure 7.3, and Figure 7.4.

Table 7.1: *Counter Task*'s performance metrics (See subsection 4.2.2) of proposed DynARL models and vanilla RL evaluated on in-distribution and out-of-distribution dynamics. Each performance metric in the tables is from averaging models' performances from 5 different training random seeds. The performance of each model is calculated from 1000 evaluation episodes. Standard deviations of performances are shown in brackets. The best one is bold, and the second best one is underlined.

Counter Task	In-distribution Dynamics			
	HR	NPR	SR	Average Return
Vanilla RL	99.9 (0.1)	93.6 (4.6)	40.2 (15.3)	0.74 (0.04)
DynARL + DM (Ours)	99.2 (0.2)	90.4 (5.9)	49.7 (10.0)	0.74 (0.04)
DynARL + L2CDM (Ours)	**100** (0)	**95.7** (3.8)	48.3 (14.8)	**0.77** (0.04)
DynARL + True Dynamics (Ours)	**100** (0)	92.8 (5.6)	**53.2 (3.5)**	0.76 (0.04)

Counter Task	Out-of-distribution Dynamics			
	HR	NPR	SR	Average Return
Vanilla RL	**99.7** (0.4)	33.5 (12.1)	13.9 (4.6)	0.23 (0.08)
DynARL + DM (Ours)	98.7 (2.6)	51.3 (24.1)	18.8 (9.0)	**0.34** (0.16)
DynARL + L2CDM (Ours)	92.5 (6.5)	48.4 (15.0)	**19.2 (4.7)**	0.33 (0.09)
DynARL + True Dynamics (Ours)	99.4 (0.7)	**58.2** (10.8)	17.4 (4.6)	**0.34** (0.07)
Nonstationary RL	99.3 (0.3)	83.7 (3.1)	31.9 (2.0)	0.57 (0.02)

Table 7.2: *Smash Task*'s performance metrics (See subsection 4.2.2) of proposed DynARL models and vanilla RL evaluated on in-distribution and out-of-distribution dynamics. Each performance metric in the tables is from averaging models' performances from 5 different training random seeds. The performance of each model is calculated from 1000 evaluation episodes. Standard deviations of performances are shown in brackets. The best one is bold, and the second best one is underlined.

Smash Task	In-distribution Dynamics			
	HR	NPR	SR	Average Return
Vanilla RL	100 (0)	82.9 (1.5)	42.0 (6.4)	0.62 (0.04)
DynARL + DM (Ours)	100 (0)	81.9 (1.3)	41.0 (5.9)	0.61 (0.03)
DynARL + L2CDM (Ours)	100 (0)	84.4 (2.0)	45.9 (7.9)	0.64 (0.04)
DynARL + True Dynamics (Ours)	**100** (0)	**92.4** (4.1)	**50.7** (9.6)	**0.70** (0.06)

Smash Task	Out-of-distribution Dynamics			
	HR	NPR	SR	Average Return
Vanilla RL	90.8 (3.9)	32.8 (2.9)	10.6 (1.0)	0.20 (0.02)
DynARL + DM (Ours)	**93.0** (3.0)	30.0 (2.5)	7.9 (1.9)	0.17 (0.03)
DynARL + L2CDM (Ours)	92.4 (3.9)	32.6 (3.4)	8.5 (2.0)	0.18 (0.03)
DynARL + True Dynamics (Ours)	92.2 (4.2)	**48.8** (12.6)	**17.1** (5.6)	**0.31** (0.09)
Nonstationary RL	90.6 (3.8)	36.1 (10.6)	9.0 (3.5)	0.20 (0.06)

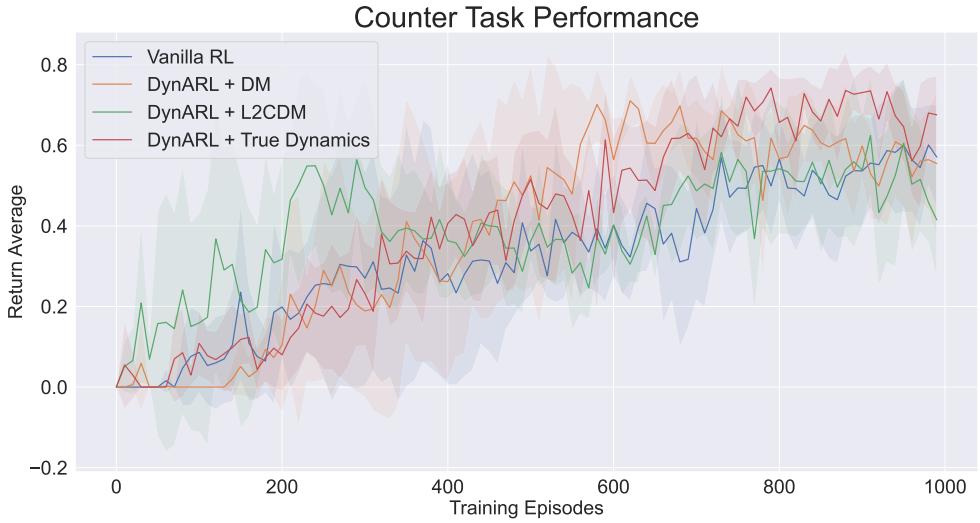

Figure 7.2: *Learning Dynamics of RL:*The figure shows the average of episode returns over 5 runs for vanilla RL and DynARL models during training.

7.4 Discussion

7.4.1 Result Analysis

Some *DynARL* models do not outperform the vanilla model, probably because their dynamics models are not accurate enough for some tasks. The *Counter* task might not require agents to have very accurate dynamics prediction because all DynARL models have similar performances. On the other hand, only *DynARL + True Dynamics* models improve task performances. The results imply that *Smash* and *Multi-agent* tasks require very accurate dynamics models to enhance generalisation performance. Due to time and computational resource limits, we did not tune the dynamics models to reach maximum accuracies. However, we can consider *DynARL + True Dynamics* is a proof-of-concept, such that if predictions are sufficiently accurate, the augmentation of predicted future states always improves generalisation performance.

Furthermore, some multi-agent DynARL models are overfitting some episodes as the Max Hits scores are much higher than the Average Hits scores. The agents might find the optimal policies specific to some trajectory distributions so that they can perform many hundred hits without fail. On the other hand, the agents always fail to hit in some situations, and they get zeros scores from these episodes. One way to mitigate this problem is using another variant of DDPG called the *Soft Actor-Critic* algorithm, which maximises the expected returns along with a policy's entropy. The *Soft Actor-Critic*'s policy would have more diverse behaviours, and less overfitting [37].

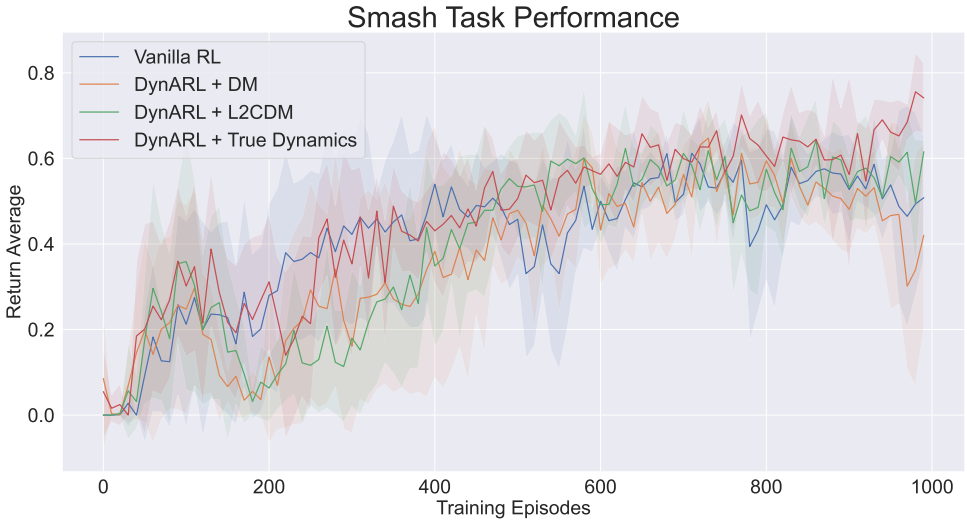

Figure 7.3: *Learning Dynamics of RL:* The figure shows the average of episode returns over 5 runs for vanilla RL and DynARL models during training.

7.4.2 Limitations and Future Works

DynARL is very task-specific. Only predicted future states of an uncontrollable object are augmented to a model-free RL model, not all objects. Furthermore, it is designed to predict the simple dynamics of a flying tennis ball, not its interactions with the ground or with the net. Future works could include symbolic graph representations to predict interactions between objects as in [78, 79, 80]. To make the algorithm applicable to all tasks, we might augment the predicted future states of the entire environment. However, the predicted future states of controllable parts require a sequence of actions. For example, the next position of the racket depends on the force applied to it in the current state. So there will be a great number of outcomes in predicted future states. Many of the predicted outcomes might not be useful to solve tasks if they are from trivial or random sequences of actions. Future works should consider designing a model that can select useful sequences of actions for predicting future outcomes, and the selection might be implemented using attention [81, 82, 83]. Lastly, Contrastive Dynamics Model (CDM) is pretrained on a static dataset before assembling to an RL agent. However, the trajectory distribution in the CDM's training dataset is not the same as the trajectory distribution in a replay buffer for RL. This might cause the prediction accuracy during executing RL to be lower than the accuracy evaluated on static test datasets. Future works could consider a way to fine-tune CDM during RL training to make CDM more aligned with the new distribution in the replay buffer [84].

7.4.3 Connection to other works

Most of the existing model-based RL works concern the predictions of the entire observations [17]. They do not consider controllable and uncontrollable parts of an

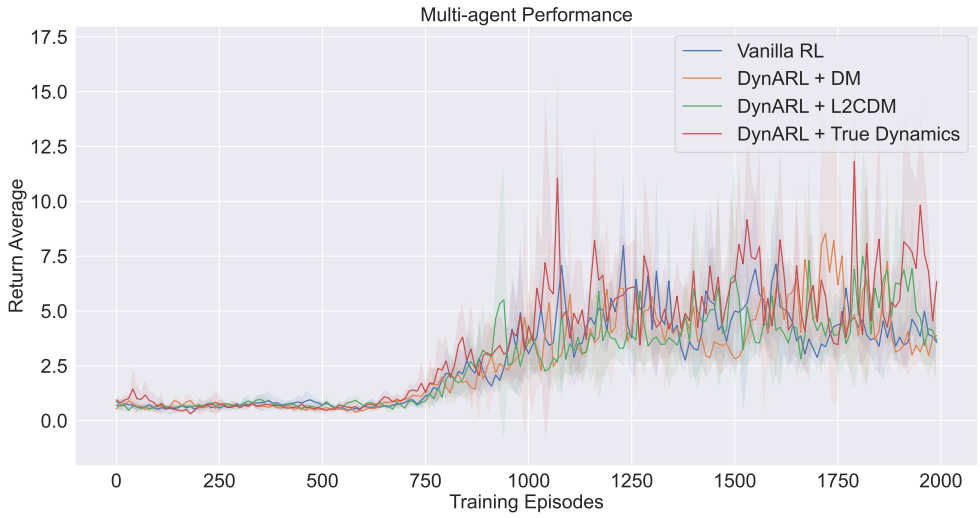

Figure 7.4: *Learning Dynamics of multi-agent RL:* The figure shows the average of episode returns over 5 runs for vanilla multi-agent RL and dynamics-aware multi-agent RL models during training.

environment separately. Hence, the predictions have to be either conditioned on sampled sequences of actions for sampling-based planning [27, 17, 25] or conditioned on a policy function for gradient-based planning [44, 85, 26, 48]. The most similar works to ours might be *I2A* [86] and *DreamerV2* [48]. In *I2A* [86], like our work, they augmented *context* to a policy network, and the *context* is the encoded representation from multiple predicted trajectories. The predicted trajectories are conditioned on the policy network. In contrast to our work, we augment the single trajectory of the uncontrollable part, which is not affected by the policy. *DreamerV2*'s dynamics model is trained to predict future states and has latent variables with which the model-free RL is trained. Like our work, their approach combines model-based predictions with model-free learning, which is trained to find the optimal policy. In contrast to our work, we train model-free RL using raw observations, and our dynamics model is not used to simulate virtual experiences.

Table 7.3: *Multi-agent*'s performance (see subsection 4.3.2) of proposed DynARL and vanilla RL models evaluated on in-distribution and out-of-distribution dynamics. The performance metric in the tables is from averaging models' performances from 5 different training random seeds. The performance of each model is calculated from 100 evaluation episodes. Standard deviations of performances are shown in brackets. The best one is bold, and the second best one is underlined.

Multi-agent	In-distribution Dynamics		
	Average Hits	Max Hits	Min Hits
Vanilla RL	15.4 (10.2)	177	0
DynARL + DM (Ours)	27.4 (9.8)	256	0
DynARL + L2CDM (Ours)	**76.3** (93.6)	**1679**	0
DynARL + True Dynamics (Ours)	<u>70.9</u> (34.9)	<u>798</u>	0

Multi-agent	Out-of-distribution Dynamics		
	Average Hits	Max Hits	Min Hits
Vanilla RL	11.3 (6.2)	<u>135</u>	0
DynARL + DM (Ours)	<u>14.0</u> (5.5)	123	0
DynARL + L2CDM (Ours)	9.9 (3.4)	79	0
DynARL + True Dynamics (Ours)	**23.7** (6.9)	**247**	0
Nonstationary RL	12.5 (5.9)	168	0

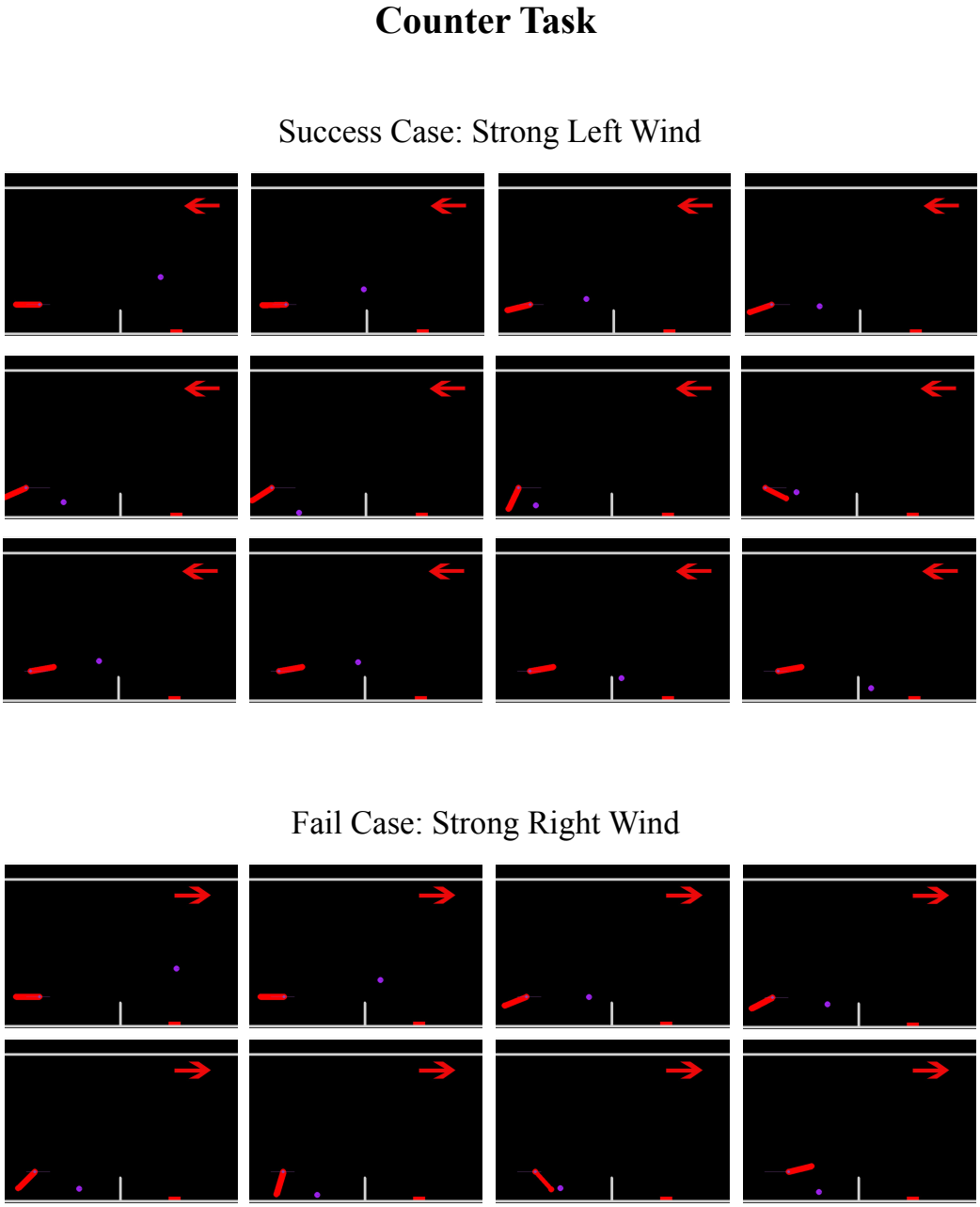

Figure 7.5: The figure shows trajectories and agent's behaviours in *Counter Task*. Top image is when strong wind force applied to the ball in the left direction, the agent succeeds in both hitting the ball and returning the ball to another side. Left image is when strong wind force applied on the right direction to the ball, the agent succeeds in hitting the ball but fails to return the ball to another side. The chronological order is left to right and top to bottom.

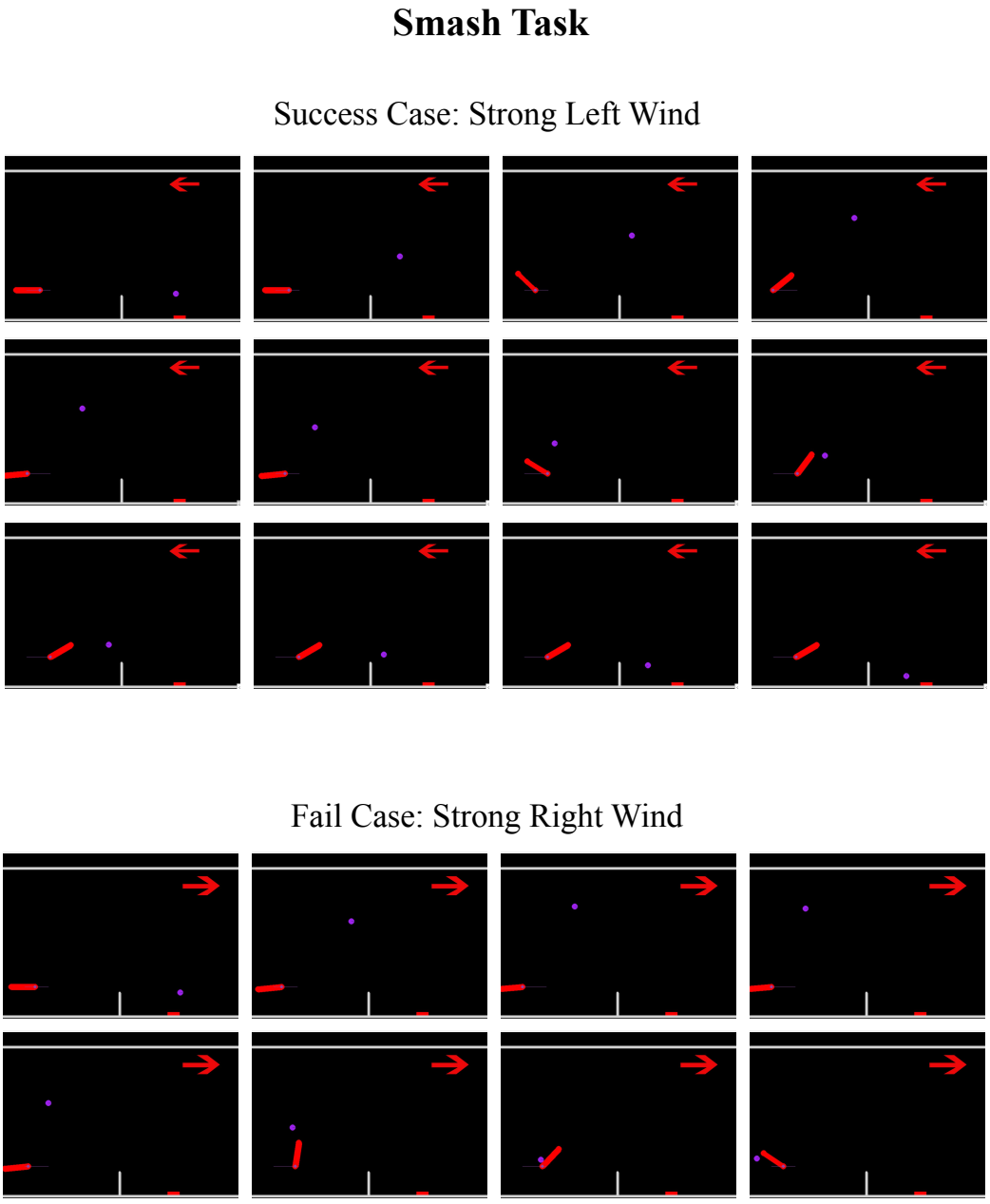

Figure 7.6: The figure shows trajectories and agent's behaviours in *Smash Task*. Top image is when strong wind force applied to the ball in the left direction, the agent succeeds in both hitting the ball and returning the ball to another side. Left image is when strong wind force applied on the right direction to the ball, and the agent fails to hit the ball. The chronological order is left to right and top to bottom.

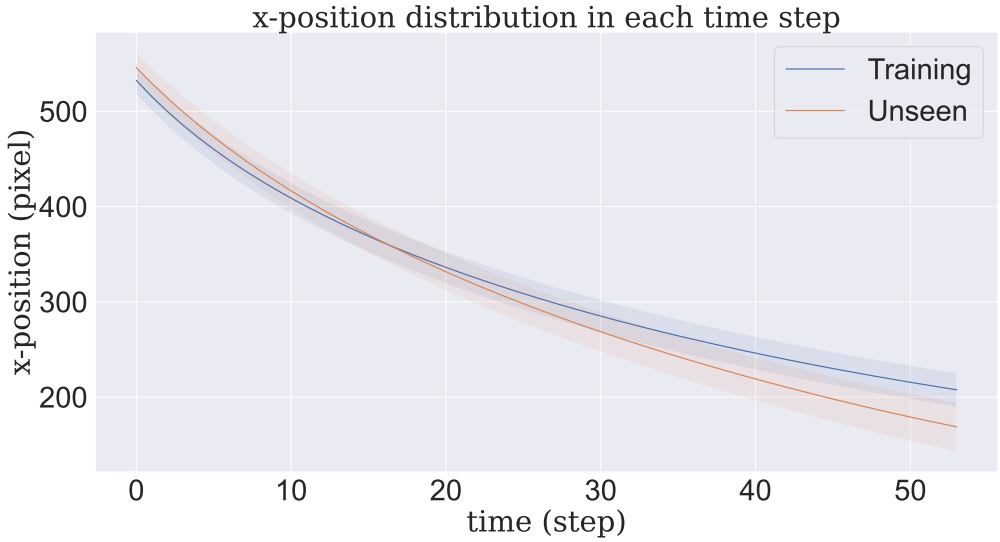

Figure 7.7: The figure shows the mean of x-position of the ball trajectories which are used in *Counter* task in each time step. Light bands represent standard deviation of x-position.

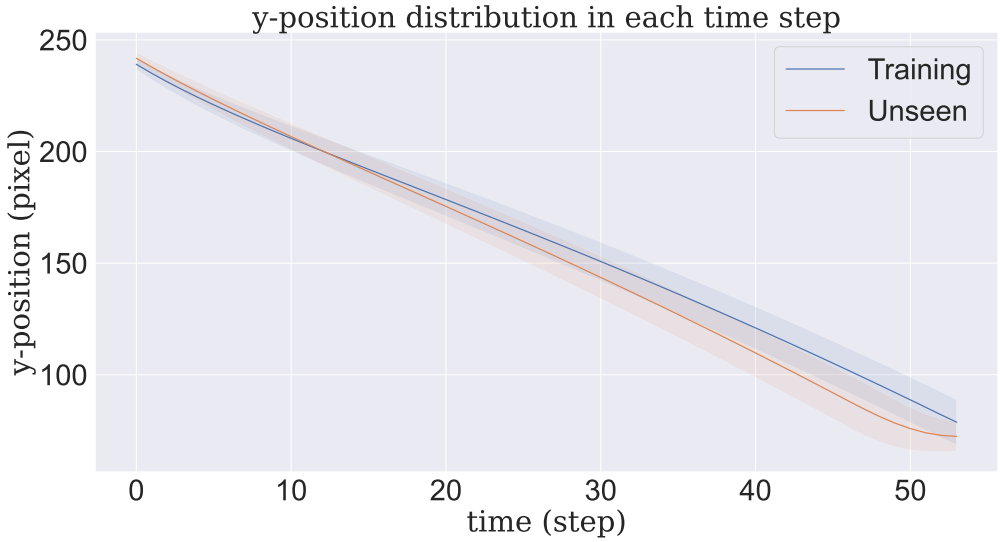

Figure 7.8: The figure shows the mean of y-position of the ball trajectories which are used in *Counter* task in each time step. Light bands represent standard deviation of y-position.

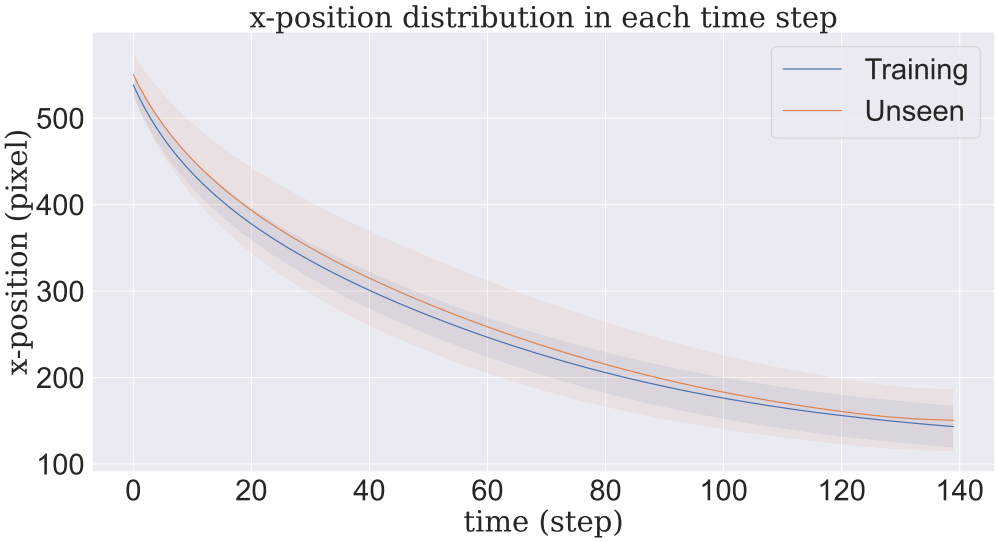

Figure 7.9: The figure shows the mean of x-position of the ball trajectories which are used in *Smash* task in each time step. Light bands represent standard deviation of x-position.

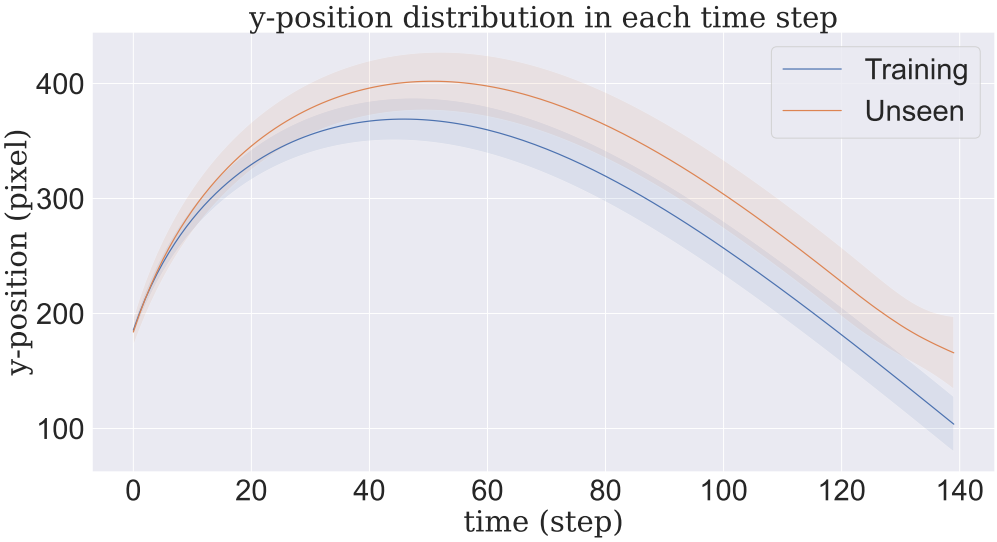

Figure 7.10: The figure shows the mean of y-position of the ball trajectories which are used in *Smash* task in each time step. Light bands represent standard deviation of y-position.

Chapter 8

Conclusion

Firstly, we design and develop *Tennis2D*, a new learning environment for training intelligent agents to play tennis. We investigate five reward functions for solving three single-agent tasks and one cooperative multi-agent task. *Tennis2D* is suitable for early-stage algorithmic design, as it contains realistic dynamics and nonstationarities. Secondly, we propose a new dynamics prediction method called *Contrastive Dynamics Model (CDM)*, designed to model knowledge and parameters separately. Recently, contrastive learning has been widely used for the learning representations of visual, auditory, and language data, leading to many successful applications. Thus, in this project, we explore the use of contrastive learning for learning the representation of dynamical state data. Our *CDM* improves the dynamics prediction accuracy in multiple dynamics. Lastly, we create a new RL algorithm for playing tennis games called *Dynamics-aware RL (DynARL)* that combines the advantages of model-free and model-based learning. *DynARL* outperforms a model-free RL baseline in in-distribution and out-of-distribution dynamics. Although it is a task-specific algorithm, *DynARL* is simple and easy to train, requiring minimal computational resources.

8.1 Ethical, Legal and Societal Considerations

All of our works here are at the fundamental level of algorithms. It has no concern about using human data and has no use of dangerous equipment. The open-source code for contrastive learning is used under the MIT license, and we do not have a copyright issue. All models are trained in single CPU machines, and the models have very low complexity. The training processes require small computational resources, so they have a low carbon footprint and cause minimal environmental damage.

This project has no direct effect on society because we develop algorithms at the fundamental level. They are designed based on simplified simulations and very specific to tasks. Our proposed algorithms are not applicable in the real world without significant modifications. However, the persistent improvements of fundamental algorithms can bring about many ethical and societal considerations. Eventually, very advanced algorithms will be created, and it will lead robots to learn much more

complicated tasks and operate in challenging environments. Autonomous intelligent robots might lead to catastrophe if they are used for unethical purposes. For example, AI-assisted weaponised robots were used to assassinate the Iranian general [87] and scientist [88]. Further regulations are required to moderate these considerations.

Bibliography

[1] Paul Smolensky. Connectionist ai, symbolic ai, and the brain. *Artificial Intelligence Review*, 1(2):95–109, 1987. pages 1

[2] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010. pages 1

[3] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002. pages 1

[4] Frank Guerin and Paulo Ferreira. Robot manipulation in open environments: New perspectives. *IEEE transactions on cognitive and developmental systems*, 12 (3):669–675, 2019. pages 1

[5] Rodney A Brooks. Intelligence without representation. *Artificial intelligence*, 47(1-3):139–159, 1991. pages 1

[6] Qiuming Zhu. Hidden markov model for dynamic obstacle avoidance of mobile robot navigation. *IEEE Transactions on Robotics and Automation*, 7(3):390–397, 1991. pages 1

[7] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996. pages 1

[8] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11): 1238–1274, 2013. pages 1

[9] Laura Smith, Ilya Kostrikov, and Sergey Levine. A walk in the park: Learning to walk in 20 minutes with model-free reinforcement learning. *arXiv preprint arXiv:2208.07860*, 2022. pages 1

[10] Julian Ibarz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*, 40 (4-5):698–721, 2021. pages 1, 4

[11] Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995. pages 2

[12] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016. pages 2

[13] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017. pages 2

[14] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019. pages 2

[15] Kai Arulkumaran, Antoine Cully, and Julian Togelius. Alphastar: An evolutionary computation perspective. In *Proceedings of the genetic and evolutionary computation conference companion*, pages 314–315, 2019. pages 2

[16] Suneel Belkhale, Rachel Li, Gregory Kahn, Rowan McAllister, Roberto Calandra, and Sergey Levine. Model-based meta-reinforcement learning for flight with suspended payloads. *IEEE Robotics and Automation Letters*, 6(2):1471–1478, 2021. pages 2, 6, 38

[17] Anusha Nagabandi, Ignasi Clavera, Simin Liu, Ronald S Fearing, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Learning to adapt in dynamic, real-world environments through meta-reinforcement learning. *arXiv preprint arXiv:1803.11347*, 2018. pages 2, 6, 35, 47, 48

[18] Kimin Lee, Younggyo Seo, Seunghyun Lee, Honglak Lee, and Jinwoo Shin. Context-aware dynamics model for generalization in model-based reinforcement learning. In *International Conference on Machine Learning*, pages 5757–5766. PMLR, 2020. pages 2, 6

[19] Jane X Wang, Zeb Kurth-Nelson, Dharshan Kumaran, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Demis Hassabis, and Matthew Botvinick. Prefrontal cortex as a meta-reinforcement learning system. *Nature neuroscience*, 21(6):860–868, 2018. pages 2

[20] Geert JP Savelsbergh, A Mark Williams, John Van Der Kamp, and Paul Ward. Visual search, anticipation and expertise in soccer goalkeepers. *Journal of sports sciences*, 20(3):279–287, 2002. pages 2

[21] Kielan Yarrow, Peter Brown, and John W Krakauer. Inside the brain of an elite athlete: the neural processes that support high achievement in sports. *Nature Reviews Neuroscience*, 10(8):585–596, 2009. pages 2

[22] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018. pages 3, 31, 32, 33

[23] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020. pages 3, 31

[24] Michael Laskin, Aravind Srinivas, and Pieter Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. In *International Conference on Machine Learning*, pages 5639–5650. PMLR, 2020. pages 3, 31

[25] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *Advances in neural information processing systems*, 31, 2018. pages 3, 5, 48

[26] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019. pages 3, 5, 12, 48

[27] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018. pages 3, 5, 48

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. pages 4, 38

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015. pages 4, 38

[30] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016. pages 4

[31] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015. pages 4

[32] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016. pages 4

[33] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014. pages 4, 10

[34] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015. pages 4, 7, 10

[35] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016. pages 4

[36] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. pages 4, 7

[37] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018. pages 4, 46

[38] Stephen James, Andrew J Davison, and Edward Johns. Transferring end-to-end visuomotor control from simulation to real world for a multi-stage task. In *Conference on Robot Learning*, pages 334–343. PMLR, 2017. pages 4

[39] Stephen James, Michael Bloesch, and Andrew J Davison. Task-embedded control networks for few-shot imitation learning. In *Conference on robot learning*, pages 783–795. PMLR, 2018. pages 4

[40] Michael Laskin, Aravind Srinivas, and Pieter Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. In *International Conference on Machine Learning*, pages 5639–5650. PMLR, 2020. pages 4

[41] Andrea Dittadi, Frederik Träuble, Manuel Wüthrich, Felix Widmaier, Peter Gehler, Ole Winther, Francesco Locatello, Olivier Bachem, Bernhard Schölkopf, and Stefan Bauer. Representation learning for out-of-distribution generalization in reinforcement learning. *arXiv e-prints*, pages arXiv–2107, 2021. pages 4

[42] Misha Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. Reinforcement learning with augmented data. *Advances in Neural Information Processing Systems*, 33:19884–19895, 2020. pages 4

[43] Denis Yarats, Ilya Kostrikov, and Rob Fergus. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. In *International Conference on Learning Representations*, 2020. pages 4

[44] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472. Citeseer, 2011. pages 4, 48

[45] Tingwu Wang and Jimmy Ba. Exploring model-based planning with policy networks. *arXiv preprint arXiv:1906.08649*, 2019. pages 5

[46] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011. pages 5

[47] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019. pages 5

[48] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020. pages 5, 48

[49] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. pages 5

[50] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016. pages 6

[51] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl ˆ2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016. pages 6

[52] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017. pages 6

[53] Rujikorn Charakorn, Poramate Manoonpong, and Nat Dilokthanakul. Learning to cooperate with unseen agents through meta-reinforcement learning. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1478–1479, 2021. pages 6

[54] Irina Higgins, Arka Pal, Andrei Rusu, Loic Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, and Alexander Lerchner. Darla: Improving zero-shot transfer in reinforcement learning. In *International Conference on Machine Learning*, pages 1480–1490. PMLR, 2017. pages 6

[55] Fumio Miyazaki, Michiya Matsushima, and Masahiro Takeuchi. Learning to dynamically manipulate: A table tennis robot controls a ball and rallies with a human being. In *Advances in Robot Control*, pages 317–341. Springer, 2006. pages 6

[56] Katharina Mülling and Jan Peters. A computational model of human table tennis for robot application. In *Autonome Mobile Systeme 2009*, pages 57–64. Springer, 2009. pages 6

[57] Katharina Muelling, Jens Kober, and Jan Peters. Learning table tennis with a mixture of motor primitives. In *2010 10th IEEE-RAS International Conference on Humanoid Robots*, pages 411–416. IEEE, 2010. pages 6

[58] Katharina Mülling, Jens Kober, and Jan Peters. A biomimetic approach to robot table tennis. *Adaptive Behavior*, 19(5):359–376, 2011. pages 6

[59] Wenbo Gao, Laura Graesser, Krzysztof Choromanski, Xingyou Song, Nevena Lazic, Pannag Sanketi, Vikas Sindhwani, and Navdeep Jaitly. Robotic table tennis with model-free reinforcement learning. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5556–5563. IEEE, 2020. pages 7

[60] Jonas Tebbe, Lukas Krauch, Yapeng Gao, and Andreas Zell. Sample-efficient reinforcement learning in robotic table tennis. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4171–4178. IEEE, 2021. pages 7

[61] Dieter Büchler, Simon Guist, Roberto Calandra, Vincent Berenz, Bernhard Schölkopf, and Jan Peters. Learning to play table tennis from scratch using muscular robots. *IEEE Transactions on Robotics*, 2022. pages 7, 17

[62] James F Allen and Johannes A Koomen. Planning using a temporal world model. In *Proceedings of the Eighth international joint conference on Artificial intelligence-Volume 2*, pages 741–747, 1983. pages 7

[63] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018. pages 7

[64] Yutaka Matsuo, Yann LeCun, Maneesh Sahani, Doina Precup, David Silver, Masashi Sugiyama, Eiji Uchibe, and Jun Morimoto. Deep learning, reinforcement learning, and world models. *Neural Networks*, 2022. pages 7

[65] Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020. pages 7

[66] Silviu-Marian Udrescu, Andrew Tan, Jiahai Feng, Orisvaldo Neto, Tailin Wu, and Max Tegmark. Ai feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity. *Advances in Neural Information Processing Systems*, 33:4860–4871, 2020. pages 7

[67] Pulkit Agrawal, Ashvin V Nair, Pieter Abbeel, Jitendra Malik, and Sergey Levine. Learning to poke by poking: Experiential learning of intuitive physics. *Advances in neural information processing systems*, 29, 2016. pages 7

[68] Damian Mrowca, Chengxu Zhuang, Elias Wang, Nick Haber, Li F Fei-Fei, Josh Tenenbaum, and Daniel L Yamins. Flexible neural representation for physics prediction. *Advances in neural information processing systems*, 31, 2018. pages 7

[69] Yunzhu Li, Toru Lin, Kexin Yi, Daniel Bear, Daniel Yamins, Jiajun Wu, Joshua Tenenbaum, and Antonio Torralba. Visual grounding of learned physical models. In *International conference on machine learning*, pages 5927–5936. PMLR, 2020. pages 7

[70] Ferran Alet, Dylan Doblar, Allan Zhou, Josh Tenenbaum, Kenji Kawaguchi, and Chelsea Finn. Noether networks: meta-learning useful conserved quantities. *Advances in Neural Information Processing Systems*, 34, 2021. pages 7, 38

[71] James B Rawlings. Tutorial overview of model predictive control. *IEEE control systems magazine*, 20(3):38–52, 2000. pages 11

[72] Benjamin Eysenbach, Tianjun Zhang, Ruslan Salakhutdinov, and Sergey Levine. Contrastive learning as goal-conditioned reinforcement learning. *arXiv preprint arXiv:2206.07568*, 2022. pages 12

[73] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. pages 18, 22, 35

[74] Jia-Wen Yam, Jing-Wen Pan, and Pui-Wah Kong. Measuring upper limb kinematics of forehand and backhand topspin drives with imu sensors in wheelchair and able-bodied table tennis players. *Sensors*, 21(24):8303, 2021. pages 25

[75] Benjamin Eysenbach, Tianjun Zhang, Ruslan Salakhutdinov, and Sergey Levine. Contrastive learning as goal-conditioned reinforcement learning. *arXiv preprint arXiv:2206.07568*, 2022. pages 31

[76] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008. pages 37

[77] Galen Andrew, Raman Arora, Jeff Bilmes, and Karen Livescu. Deep canonical correlation analysis. In *International conference on machine learning*, pages 1247–1255. PMLR, 2013. pages 38

[78] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. *Advances in neural information processing systems*, 29, 2016. pages 47

[79] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pages 8459–8468. PMLR, 2020. pages 47

[80] Rishi Veerapaneni, John D Co-Reyes, Michael Chang, Michael Janner, Chelsea Finn, Jiajun Wu, Joshua Tenenbaum, and Sergey Levine. Entity abstraction in visual model-based reinforcement learning. In *Conference on Robot Learning*, pages 1439–1456. PMLR, 2020. pages 47

[81] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014. pages 47

[82] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. *Advances in neural information processing systems*, 28, 2015. pages 47

[83] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. pages 47

[84] Bryan Lim, Luca Grillotti, Lorenzo Bernasconi, and Antoine Cully. Dynamics-aware quality-diversity for efficient learning of skill repertoires. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 5360–5366. IEEE, 2022. pages 47

[85] Nikhil Mishra, Pieter Abbeel, and Igor Mordatch. Prediction and control with temporal segment models. In *International Conference on Machine Learning*, pages 2459–2468. PMLR, 2017. pages 48

[86] Sébastien Racanière, Théophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. *Advances in neural information processing systems*, 30, 2017. pages 48

[87] Stephanie Nebehay. U.n. expert deems u.s. drone strike on iran's soleimani an 'unlawful' killing, 2020. URL `https://www.reuters.com/article/us-usa-iran-un-rights-idUSKBN2472TW`. pages 55

[88] Stephen Farrell. Iranian nuclear scientist killed by one-ton automated gun in israeli hit: Jewish chronicle, 2021. URL `https://www.reuters.com/article/us-iran-nuclear-scientist-idUSKBN2AA2RC`. pages 55