Developing Reinforcement Learning Algorithms for Robots to Aim and Pour Solid Objects

by

Haoxuan Li

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Yu Sun, Ph.D.
Shaun Canavan, Ph.D.
Yasin Yilmaz, Ph.D.

Date of Approval:
March 4, 2022

Keywords: Continuous Action, Actor-Critic, DQN, Function Approximation, Robot Manipulation

Acknowledgments

I would like to express my sincere gratitude to my advisor Dr. Yu Sun for advising and guiding me through this research. Without his support and patience, this thesis would not have been possible.

I also want to thank my committee members: Dr. Shaun Canavan, Dr. Yasin Yilmaz for providing insightful comments and encouragement.

Throughout all the difficulties under the social upheaval, my family gives me countless support and help review my draft. I could not have done this without anyone of you.

Table of Contents

List of Tables

List of Figures

Abstract

Pouring is one of the most commonly executed tasks in a variety of environments. There is less attention paid to pouring solid objects and avoiding spillage. Learning the dynamics for pouring solid objects can be a challenge because the collisions and static frictions between objects make their trajectories less predictable than liquid. Nonetheless, pouring solid objects is an important task in real life. In this work, we propose a solution to help robots aim and pour solid objects. The agents will learn how to interact with the environment and identify the optimal pouring trajectories, then manipulate the arm to aim at the target area by moving the end-effector in Inverse Kinematics (IK). The proposed solution is able to pour accurately into the target and minimize spillage. It also has the ability to generalize to unseen objects. Three reinforcement learning algorithms have been implemented, namely TD3, DDPG, and Double-DQN to address the proposed problem. Our study shows that the robot can learn to pour accurately by 1) using a continuous action space, and 2) avoiding overestimation in state-action value approximation.

Chapter 1: Introduction and Background

Pouring is one of the most frequent practices in humans' daily lives [1]. Pouring contents vary from liquid such as water, milk, to solid objects like ice cubes, and diced fruits. Most of the previous studies in robotic manipulation focus on pouring liquid or liquid-like objects with very limited spillage avoidance precautions. Manipulating robot arms to pour different amounts of solid objects in different sizes or shapes into the target area and avoid spillage remains a significant challenge. On the one hand, aiming requires precise movement, rigid movement will lower the chance of reaching the optimal pouring position and therefore fail the task with a higher probability. On the other hand, unlike liquid [2–6] or small rounded particles [7, 8], larger solid objects will collide with each other as well as the container. As the result, the trajectories become less predictable. What's more, the static friction between the objects results in rigid movements. In practice, solid objects often get poured out as bulk because the static friction prevents them from moving. Thus, pouring solid objects becomes a difficult task for the robot to solve.

Reinforcement learning has shown remarkable successes in many domains, such as gaming [9], Go [10], and robotics manipulation [11]. Reinforcement learning algorithms are able to efficiently explore the environment and learn how to solve the task without learning the environment dynamics. They do not suffer from high dimensional learning space as classic controllers do [2]. The state spaces for the reinforcement learning algorithms are also versatile, which avoid a certain modality becoming the training bottleneck [12, 13].

This work focuses on how to teach agents to aim and pour solid objects into the target area. More specifically, the problem is further divided into two: 1) pouring with visual feedback; 2) pouring without any visual feedback. A system without visual constraints,

such as lighting or camera resolution, would yields a more robust and versatile performance. We propose using reinforcement learning algorithms to control the robot arm. The agents receive feedback from the environment and learn the best way to pour and avoid spillage by changing the robot's pose and adjusting the end-effector's rotation velocity.

In the following sections, chapter 2 will give an overview of the previous work in order to understand the fundamental of the problem. Chapter 3 will describe the problem formulation for our novel task and define the systems. Chapters 4 and 5 will illustrate the performance of our approach as well as future developments.

Chapter 2: Literature Review

2.1 Related Work

Pouring has been a major topic in robotic manipulation and attracts the attention from researchers around the globe. A great number of solutions have been proposed. This paper mainly focuses on deep learning approaches. There are two major branches: one uses recurrent neural networks and the other uses reinforcement learning.

2.1.1 Deep Learning Methods

Many works aimed to improve robotic manipulation in pouring liquid or liquid-like contents using Recurrent Neural Network(RNN). There are several methods [2–6, 14] proposed by our researchers. Huang and Sun [2] proposed a method that used functional analysis of motion and dynamic time warping to generate manipulation trajectory. However, such a method is limited by the sequence length and dimensionality of the learning space. To address this issue, we moved on to use recurrent neural networks which are designed for processing sequences. [4] proposed to use LSTMs for processing the pouring sequence and predict the rotation speed given a certain state. *Chen et al.* [3] claimed that combining model predictive control (MPC) with LSTMs can better capture the behavior of liquid. The method models the rotation angle of the end-effector and predicts the volume at the next time step. Although the pouring accuracy had been improved significantly, there were two main issues: 1) lack of dynamic models; 2) low data efficiency. The lack of dynamics limits our work's ability to be extended to unseen containers and different liquids. However, learning them requires lots of quality data, which is another constraint. Most of the time was spent on data collection because it was hard to obtain the desired pouring trajectory, which

stopped us from fast convergence and fine-tuning [6]. To tackle the constraints, Huang and Wilches [6] combined the imitation learning method with recurrent neural networks to learn from human demonstration. Learning from experts helps the models understand the dynamics from a baseline rather than learning from scratch. Peephole LSTMs were used to resolve the exploding and vanishing gradients. The paper also proposed a method called generalization by self-supervised practicing (GSSP) in order to improve data efficiency. GSSP uses the actual outcomes instead of the desired outcomes to fine-tune the model. The model received rotation feedback as well as force sensor reading as the input and output of the rotation velocity. Although we successfully improved the performance, there were other constraints being introduced. Imitation learning requires intensive human labor and it is strict on the quality of the demonstration. More recent work from us also uses LSTM but pays attention to pouring solid objects in different shapes with a real robot [14]. We use an auto-encoder to process the visual feedback and try to pour at the exact amount declared by the user. We are able to achieve outstanding results such that the difference between final outcome and the user's input is less than 10% of the total pouring content. The proposed method outperforms the human baseline. However, there is room for improvement. Training the LSTMs and auto-encoder require a significant amount of data and time. Similar to other studies, in [14] we also ignore the spillage avoidance. A large container is used as the accommodation.

[12] utilized different visual feedback when training the network. They used a thermal camera to capture the stream of the liquid. The images were then passed to the CNN network for feature extraction. The hidden states were passed to the LSTM network to generate control signals. In [13], the Kalman filter was used to track the movement of the liquid and predict the upcoming movement.

A graph neural network is used in [7] for granular manipulation. It predicts the acceleration of the granular material particles and their trajectories. The graphs were created by reconstructing 3D point clouds from images. An autoencoder is used to map the granular material and container information to a latent state and output the acceleration of each

particle. They are able to pour the granular material into the desired distribution. However, the granularity of the materials such as ground coffee and small beans, is small and the motions are similar to liquid. The small granularity doesn't produce aggressive collisions. In addition, getting a graph representation of the material is very expensive in real life, as it requires a significant amount of computation and high-resolution sensors.

The works mentioned above tried to learn good dynamic models so that the exact movements of the poured objects can be predicted. However, learning the environment models requires a significant amount of time as well as quality data. In addition, RNN is known for its unstable hidden states. Even though LSTMs alleviates the exploding or vanishing gradient issue, it still often yields undesirable performance [15]. On the other hand, they heavily relied on human participation to define the dynamics and have a strict requirement on the vision sensor. The drawback of using vision sensors is that the quality of the feedback directly affects the performance. Issues such as lighting, resolutions, translucent objects, and processing time will greatly damage the performance. Accommodations have to be made for visual-intensive models.

2.1.2 Reinforcement Learning

Reinforcement learning is known for its ability in model-free learning regimes. Agents can learn how to solve the task without knowing the dynamics. *Do et al.* [11] proposed to use Deep Deterministic Policy Gradient (DDPG) to control a robot to pour water. They used (d, ω, h, s, r) as the state, where each letter stands for the relative distance between cups, rotation angle, filled level, spillage, filling rate, respectively. The action is a 3D vector where the first 2 controls the end-effector's position in 2D while the last one is the rotation angle. The setting can reflect the pouring status timely and is vision-moderated. The actor will select an action at each step based on the state input. The critic will then evaluate the quality of the state-action pair by using the Q value. The reward is predefined and given to the agents as the consequence of taking action a in state s. On the other hand, [8] proposed a

novel approach that used CNN to approximate the reward function and pour solid spheres. The learnable reward function allows the agents to learn how to solve the tasks without human participation. They used a top-down vision sensor to provide feedback and used the difference between frames to learn the movements. However, their method was limited by the resolution of the vision sensor.

The main focus of the above works is pouring liquid at a certain amount and do not have spillage avoidance precautions. Only [8] used solid spheres, however, the behavior of spheres is still close to the liquid because of the smooth and round surface. Non-spherical solid objects have very different properties and trajectories. The existing works won't be able to generalize to pour them, yet, humans often pour ice cubes and diced fruits in real life. Pouring solid objects accurately remains a less-popular task that is worthy of more attention.

Chapter 3: Methodology

3.1 Problem Overview and Formulation

The problem we try to solve is making the arm holding the cup that contains all the objects aim at the target container and pour all the objects into it without spillage. The objects will be cuboids in different sizes because the sharp edges and large surface area can result in a more aggressive collision compared to cylinders and spheres. What's more, the static frictions between objects will hold the clusters tight, making them hard to pour. By solving the harder pouring scenario we are able to transfer the solution to other primitives easily in the future.

We further divide the pouring task into non-visual and visual settings. They are proposed in order to study the importance of visual assistance in pouring solid objects. Since visual feedback in real life has many uncertainties, such as resolution, visibility, and frame rate, etc., our agents will become more robust when there are fewer constraints.

3.1.1 Markov Decision Process in Pouring Solid Objects

The pouring process is formulated as the Markov decision process (MDP). MDP is defined by a tuple $M = (S, A, P_a, R_a, \gamma)$, where S is a set of states, A is a set of actions, P_a(s,s') is the probability that state s at time t transits to the next state s' by taking action $a \in A$. In our case, P is deterministic: given a state s and action a, the next state s' can be calculated using Inverse Kinematics (IK). R_a(s, s') is the reward for each state transition and $\gamma \in [0, 1]$ is the discount factor controlling for a trade-off between immediate and future rewards.

Instead of using the whole 3D space, we limit the linear action space to a vertical plane that cuts the receiving container to half, as shown in Figure 3.1. It allows the robot to

move the pouring cup up-and-down and back-and-forth, which is sufficient for aiming. We define the z axis as the vertical axis and the y axis as the horizontal axis. Linear movement in both the vertical and horizontal directions is limited in the range of $[-0.1\,m, 0.1\,m]$ from the center. The arm can't move outside of the boundary. The size and center of the plane are selected based on our experience. The agent will control the cup's horizontal and vertical displacement $(\Delta y, \Delta z)$. We limit the displacement range at each step to be within $[-0.1\,m, 0.1\,m]$ to prevent big sudden movement. The agents don't control the joints directly such that trajectories and velocities to achieve target positions are calculated using IK. The speeds have an upper limit of 3.14 rad/s.

The visual and non-visual pouring processes have different rotation control signals. In the non-visual setting, the agent without visual feedback cannot control the pouring velocity and the pouring velocity is fixed to -0.4 rad/s. On the contrary, the agents with visual feedback can also control the pouring velocity by $\Delta\omega$, the change in angular velocity at each step lies within $[-0.1\,rad/s, 0.1\,rad/s]$. The initial velocity is -0.4 rad/s and the speed's upper limit is 3.14 rad/s. Each action will only be executed once per step.

State s describes the end-effector's current pose and pouring content's location. The pose of the end-effector is defined as (y, z, ω, θ). The first two terms are the end-effector's world coordinates, (ω, θ) describes the angular velocity and rotation angle. θ is limited to [-2.8, 0] radian, where 0 is the initial position and -2.8 is the maximum rotation angle. Clockwise rotation is indicated by a negative sign. Agents should not rotate counter-clockwise over the initial position. Content location refers to where the objects are once they leave the pouring container. This information is described using hidden states whose weights are learned from the visual inputs. Coordinates are not used because it is computationally expensive to maintain a list of precise coordinates. The future state s' is the consequence of the robot taking action a at state s and is conditionally independent of all previous states. Thus, our state transitions meet the Markov property [16]. Specific states and rewards will be given in sections 3.3 and 3.4.

3.1.2 Reinforcement Learning in Pouring Solid Objects

The model-free merit helps RL solve many difficult tasks that RNN and CNN failed to tackle. We choose to use DDPG [17] and TD3 [18] from the Actor-Critic family and Double-DQN [19] from Q-learning, which are all model-free algorithms. To avoid confusion, we refer to models as general settings and agents as specific networks. The Actor-Critic system is the combination of the policy network and the value network. Double-DQN only contains the value network, aka the Q network. One special characteristic of the Actor-Critic system is that the policy network (actor) will receive judgment(Q value) from the value network (critic) and use that judgment as the loss to update actors' weights. The value network produces the Q value based on the observation at time t, as shown in Equation 3.1. For TD3, there is a twin critics system. It outputs 2 candidate Q values. The smaller one is selected and becomes the actual Q value to avoid overestimation [18].

To update the critic or the Q network, we use Equation 3.3. Here, the TD target is calculated, which is an approximation of the future rewards. Then the squared difference between the currently observed Q and the calculated TD target is used to perform backpropagation. The Q value is used as the loss for the actors to perform gradient ascent using Equation 3.2. Target networks are introduced to calculate the TD targets and avoid bootstrapping. For each actor and critic in TD3 and DDPG, there will be a target network. Double-DQN also has a target Q network.

$$Q^{\mu}(s_t, a_t) = \mathop{\mathbb{E}}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, a_{t+1}))] \tag{3.1}$$

$$\nabla_{\theta^{\mu}} J = \mathbb{E}_{s \sim p}[\nabla_{\theta^{\mu}} Q(s, a|\theta^{Q})|_{s=s_t, a=\mu(s_t|\theta^{\mu})}] \tag{3.2}$$

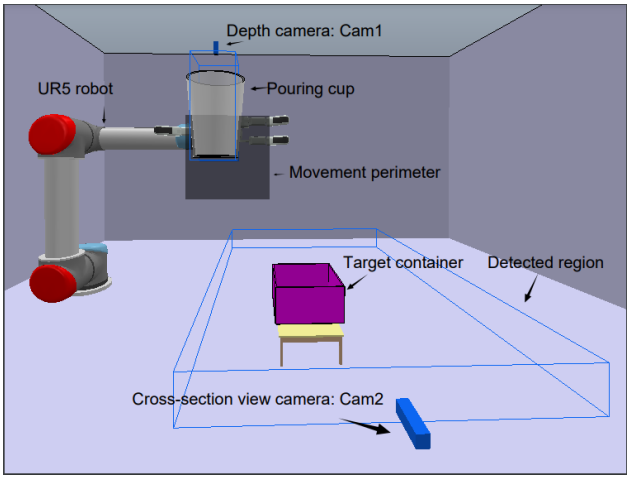
$$L(\theta^{Q}) = \mathbb{E}\left[\left(Q(s_t, a_t) - \left(r(s_t, a_t) + \gamma Q^{\mu}\Big(s_{t+1}, a_{t+1}\Big)\right)\right)^2\right] \tag{3.3}$$

3.2 Environment Setup

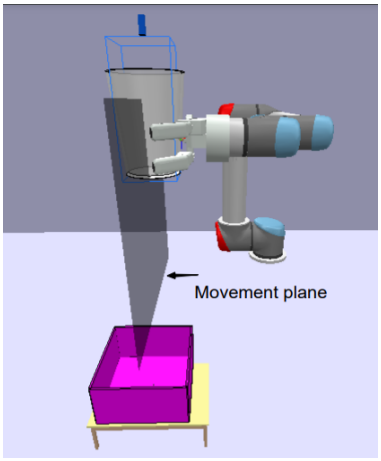
We used CoppeliaSim [8] for our data collection, model training, and testing. Figure 3.1 shows the setup. The configuration is used to replicate the objects' properties in real life.

The pouring part is composed of a pouring container controlled by a UR5 robot arm. The pouring cup has an upper diameter of 143 mm, a lower diameter of 100 mm and 200 mm of height which jointly moves with the end-effector. A depth camera is mounted on the wrist of the robotic arm, and it is 35 cm above the cup with its z-axis aligning with the centerline of the cup. The camera moves with the cup during pouring and returns a depth map that shows the structure of the pouring content. Another cross-section view camera locates on the side. It is 12.5 cm higher and 32.5 cm in the direction of pointing out of the screen away from the target box. The camera detection area is indicated by the blue-wire box. We call the depth camera Cam1 and side-view camera Cam2. The target box is shown in purple. It has a square base with a length of 15 cm. The height of the box is 8 cm. A force sensor is placed under the box to measure the weight of the poured contents. The box center is located 41 cm below and 21 cm to the right relative to the source cup.

The movement perimeter is marked by the black square. The center of the perimeter is placed 10 cm lower and to the right relative to the source cup's center. The numbers are chosen based on our heuristic. The center of the pouring cup is allowed to move within the perimeter. At the beginning of each episode, the arm will have a random initial position controlled by joints marked in red, namely joint2 and joint3. Each joint will have $\pm 5^\circ$ of freedom for the initial rotation. These two joints are chosen because they are relatively close to the robot's base and move parallel to the yz plane. A bigger range of movement can be achieved by using these joints and therefore making the exploration diverse. Simulated cubes have side lengths of 25, 19, and 16 mm and a mass of 14.345 grams. The objects' coefficient of friction is set to 0.06, to mimic the ice cube's friction in real life [20]. The outer gray walls bound the entire pouring scene. The simulation runs at 60 Hz.



(a) Environment setup



(b) Movement plane

Figure 3.1: Pouring scene in the simulator

3.3 Visual Models

3.3.1 MDP Elements

The observation space for visual models is defined as follow:

Θ(t): Rotation angle at time t, $-2.8\,rad < \theta < 0\,rad$

ω(t): Angular velocity of the pouring cup, $|\omega| < 3.14\,rad/s$

Δx: Displacement between the pouring cup and the target, $|\Delta x| < 0.14\,m$

s: Euclidean distance between the pouring cup and the target, $0.1\,m < s < 0.35\,m$

ν: Depth map from the pouring cup

ζ: Cross-section view for the falling stream

Models can choose to use only the depth image or cross-section view, or they can use both. The output is: $(\Delta y, \Delta z, \Delta\omega)$, where $-0.1 < \Delta y, \Delta z, \Delta\omega < 0.1$. Each letter represents the change in y coordinate, z coordinate and the rotation velocity respectively. The initial rotation velocity is set to -0.4 rad/s. The architecture for visual models is shown in Figure 3.2. For Double-DQN, it will only have the critic part and one Q value. TD3 and DDPG

have both the actor and critic. For TD3 and DDPG, the actions can be any real number in [-0.1,0.1] while DQN needs to select the actions from a predefined set. The numbers are chosen based on our heuristic. Details of the discrete actions will be shown in Appendix A.

To make sense of the observations, reward functions for teaching the agents 'what' and 'how' are developed. There are two parts in the reward functions: the movement phase and the settlement phase. The settlement phase contains our primary goal: to encourage the agents to pour as many objects into the target as possible. While movement phase is our secondary goal that teaches the agents to move smoothly and efficiently.

Movement phase contains the following rewards and penalties:

- -2β
- $-0.01 * \sqrt{\Delta x}$
- $-0.01 * \|\Delta\omega\|$
- $-0.01 * \omega$

each parameter will be updated at the beginning of each step and do not accumulate. β is the penalty for each time the arm tries to move outside of the boundary. Based on our experience, the optimal position for pouring should be within the perimeter. It is a waste of time to position the arm 1 meter away from the target since it would be impossible to pour into the target container from that distance. Limiting the workspace helps the agents learn faster by only exploring the meaningful space. There are two possible directions to move, hence $beta \leq 2$. If the target position's y coordinate is not in the perimeter, β will be increased by 1, the same rule for z. Δx is the displacement of the end-effector at each step. The middle two penalties regularize the movement of the arm such that it should move as smoothly as possible so that the hardware won't be damaged. The last penalty punishes counter-clockwise rotation and encourages rotating clockwise which has a negative angular velocity. Rotating counter-clockwise is only useful when the agents try to: 1) recover from aggressive

actions, or 2) break the static friction between objects. However, agents can achieve the same goals by adjusting the clockwise rotation speed. In summary, the movement rewards punish counter-clockwise rotation and aggressive changes, and encourage fast clockwise rotation in order to finish pouring in a reasonable amount of time.

Settlement phase contains the following rewards and penalties:

- -20Ω
- $+10\mathrm{I}$
- $+100$ if $\frac{\Omega}{total} < 0.05$
- -200 if $t > 80$

where Ω is the number of objects poured outside of the target, I is the number of objects poured inside of the target. The bonus is given if the number of outliers is less than 5% of the total objects. Finally, the timeout penalty is applied if the episode is longer than 80 steps. The episode will be terminated and a -200 penalty will be applied. The purpose of the settlement phase reward is to encourage our agents to pour more objects into the target box as quickly as possible.

To differentiate the primary and secondary goals, they are put in different magnitudes. The movement reward is calculated every step while the settlement phase's rewards will only be calculated once after the episode is terminated. The flag *Done* is used for marking the end of the episode. It is determined by two conditions: 1) if the cup has rotated more than 100°, or 2) the remaining weight in the pouring cup is 0. When either condition is satisfied, the agents will terminate the episode and set the *Done* flag to *True*, and calculate the settlement phase rewards.

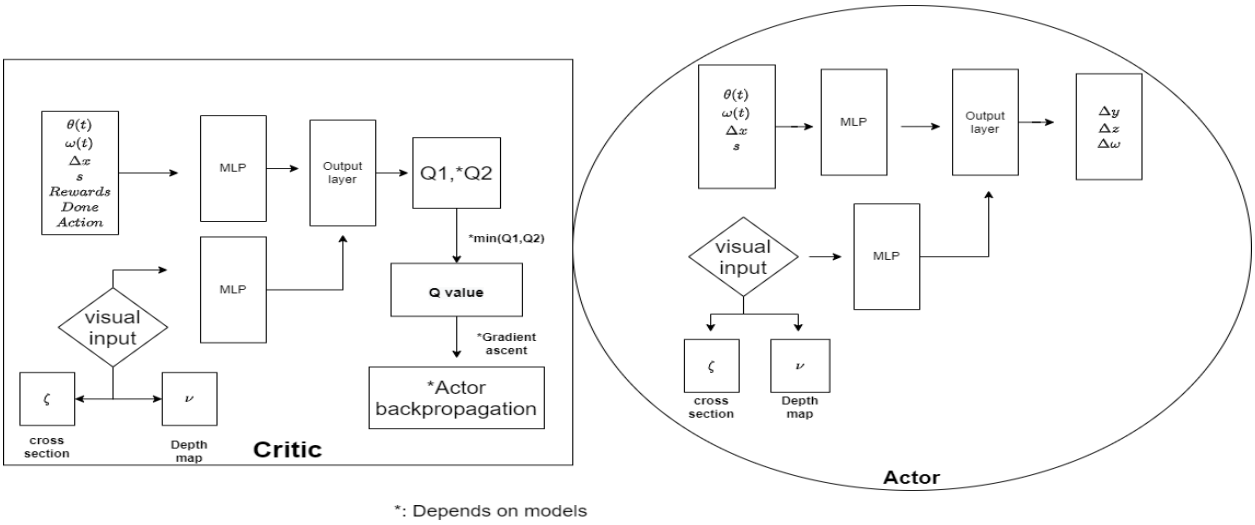
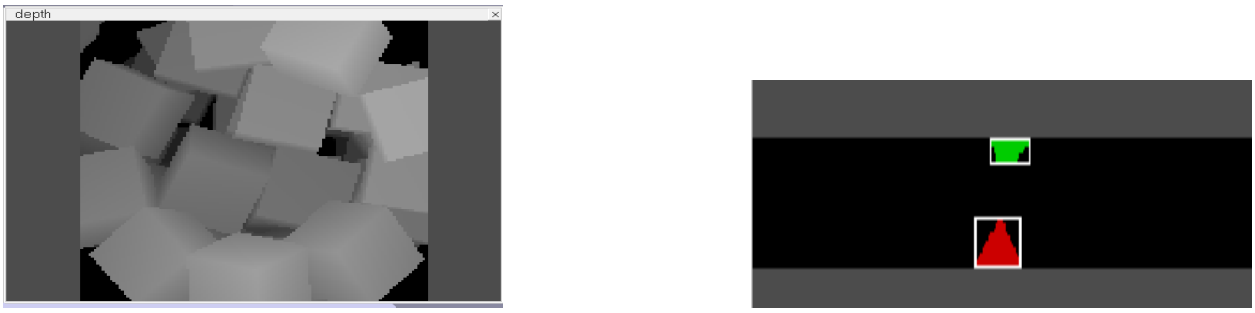


Figure 3.2: Architecture of visual models



(a) Example of the depth map

(b) Example of the cross-section view

Figure 3.3: Examples of visual inputs

3.3.1.1 Depth Map

The depth image will reveal how contents are layered in the pouring cup. The camera will return a 128*128 image. Since we are only interested in how the objects are layered in the pouring cup, the feature to extract is the depth at each pixel. The values are scaled from 0 to 1. Distance 0 is the camera and 1 is the farthest space that the camera can detect. An adaptive average pooling layer from Pytorch is used to resize the image to a 16*16 array. To speed up the training, convolutional layers or transformers are not used, which are overkill

for this simple task. Finally, the 16*16 array will be fed to two fully connected layers and added to the output layer. The agents should learn the hidden states that map the structural information and robot pose to the number of outliers. For instance, if lots of objects are stacked on one side of the cup, the agents should expect that the contents get poured at once due to the static friction, and therefore apply a different pouring strategy.

3.3.1.2 Cross Section

The other option is to have a side view of the pouring scene as the input. A cross-section is chosen over the whole side-view for the following reasons: 1) smaller images can be processed faster; 2) a full-size image that includes the entire pouring scene may confuse the agents; and 3) reinforcement learning algorithms are sensitive to noise and environmental changes. Using a cross-section to capture the falling stream should give the agents sufficient information to guess the whole trajectory and adjust the arm's pose and rotation velocity in order to avoid spillage.

The camera will return a 256*32 image whose center is aligned with the target container's center. We divide the 256-pixel width into 7 sections horizontally. Each section can contain at most 3 horizontally-placed cuboids. To detect objects' location, a bounding box detection algorithm is run on each frame. The algorithm will return a tuple containing the information about each bounding box's $\{x, w\}$, where x is the x coordinate of the center of the bounding box and w is the width. The x coordinate is then used to index which section the objects are. A heat map scaled from 0 to 1 is used to describe objects' location, where 0 means no bounding box is detected in the slot, and 1 means the bounding box's width is greater than or equal to the section length, which indicates there are multiple objects falling through that certain section. The final output will be a normalized 1D array that has 7 elements, which describe the distribution of the objects in the cross-section.

We ignore the height of the box because that doesn't contribute to our location prediction. x, w are in the camera coordinate system and we don't convert the camera coordinate system

into the world coordinate system. The 7 sections do the job by indicating the objects' relative position to the center of the target container. The example in Figure 3.3 returns two bounding boxes in the middle section. The value of that section is 1 and the rest are 0. The two cubes will land in the target because the camera's center aligns with the target container. If there are any objects in the first or last two sections, that means they won't land in the target.

However, using the cross-section view alone doesn't reveal the structural information nor the size of the objects. The camera only returns bounding boxes, which are always rectangles. In order to compensate for the lack of structural information of the objects without using another vision sensor, we include a new term, κ, which is the object's surface to volume ratio. The term is used to reduce bias and help the agents identify what object is being poured. As shown in Figure 3.4, κ will be passed to two fully connected layers with RELU activation function. The output will be combined with the hidden states from the observation and then pass to the final activation function to produce the result. κ is processed separately for two reasons. First, κ will be the only constant value if it is mixed with the observation. It is likely that the network treats the value as noise and learns nothing from it. Secondly, using a separate network to learn the representation of an object's shape can help each network focus on the right thing. There will be one network processing the observation while the other one giving instructions on how to deal with objects in different sizes.

κ is fed to the actor rather than the critic. If the actor has more information about the pouring setup, it should be able to output a better action that suits the objects. The critic will eventually learn how κ affects the action by learning from the reward and producing proper Q values. Feeding the value to actor also lowers the risk of overfitting the critic since our models are relatively simple and the critic has already received many inputs.

The above change is only applied to the TD3 algorithm because it is the only algorithm that can solve the pouring task with consistent performance. A stable baseline is vital when

we want to evaluate our implementation. For better readability, we call the model using the cross-section visual input with κ a 'structural encoding' model.

In a quick summary, using the depth map solely as input doesn't give us direct information about the objects' location. Using the cross-section alone as input doesn't give us direct information about the shape of the objects. A scalar κ is used as the substitution for structural information. Here are the four models defined in this section.

- Only use depth image, aka Cam1 model
- Only use cross-section image, aka Cam2 model
- Use both depth and cross-section image, aka Cam1 + Cam2 model
- Structural encoding model

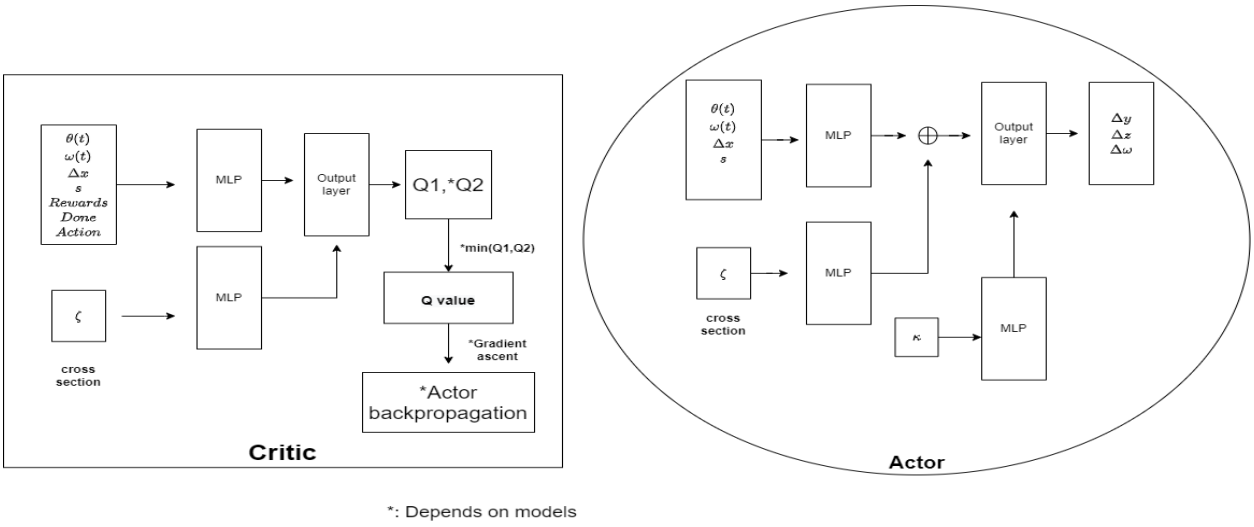


Figure 3.4: Architecture of structural encoding model

3.3.2 Actor-Critic in Visual Models

TD3 and DDPG are in the family of Actor-Critic which is the combination of value-based method and policy-based method. The critic is implemented using the value-based method.

Given the state-action pair, it will output a Q value to determine how good the pair is. The hidden layers in both actor and critic use RELU as the activation function. The critic's output layer uses linear. The policy-based method refers to an agent's actor who learns the policy function that maps states to actions, i.e., given a state, it will output the action distribution. For TD3 and DDPG, instead of giving a distribution, the network will produce deterministic values as the actions. This is done by the tanh activation function in the ouput layer. The actions are then scaled down to continuous numbers in [-0.1,0.1].

3.3.3 Double-DQN in Visual Models

DDQN is relatively simple, yet very powerful. As discussed earlier, it performs well in many different environments. In our work, DDQN only contains the left part in Figure 3.2, which is the critic and only outputs one Q value. It doesn't contain anything related to the actor. At each step, it receives state inputs and outputs a Q* which is the maximum value from the state-action pairs. The agents will use Q* as the index to select the action from the pool. The actions for DDQN are the combination of all possible values in $(\Delta y, \Delta z, \Delta\omega)$. We show the actions in Appendix A.

3.4 Non-visual Models

3.4.1 MDP Elements

The observation space for non-visual models is defined as follow:

Θ(t): Rotation angle at time t, $-2.8\,rad < \theta < 0\,rad$

ω(t): Angular velocity of the pouring cup, $|\omega| < 3.14 rad/s$

Δx: Displacement between the pouring cup and the target, $|\Delta x| < 0.14\,m$

s: Euclidean distance between the pouring cup and the target, $0.1\,m < s < 0.35\,m$

The models only output $(\Delta y, \Delta z)$. The action space is also bounded for the same reason mentioned earlier. Instead of giving different rotation velocities, it is fixed to -0.4 rad/s throughout the training.

To better explain the change in non-visual models, we define four stages in the pouring motion. There will be an initial stage, where the objects are put into the source container and the arm moves to a random initial pose; the ready stage, where the agents move to this optimal position before starting to pour; the intermediate stage, when the cup is rotating; and the final stage, where the movement is finished and settlement rewards are calculated.

Unlike training with vision, where agents will have information of objects' location according to the visual feedback, the blind agents no longer have information of the intermediate states, i.e., where the objects are. There is no element in our observation that can represent the objects' location without using vision sensors. The state only contains the pose of the robot. On the other hand, the settlement rewards are calculated based on objects' location which is missing in the non-visual model's observation, hence the settlement rewards don't reflect the consequence of taking action a in state s. Only the movement phase rewards are still affected by our actions. The task becomes finding the best ready position that would result in the best performance, in short, a search problem. Similar to a maze, at the beginning of the game, the agent has no idea where the exit is and whether the current path will lead to the exit. It has to try enough times until the solution is found. Our agents in the pouring task don't know where the optimal pouring position is. Practices are needed so that the agents can map a certain position to an expected number of outliers.

With γ and the replay buffer, agents are able to solve this problem. Firstly, agents will learn the best way to move the arm with movement phase rewards. After multiple attempts, there will be enough steps stored in the buffer. The agents can both 'see the future' and look up the history by calculating TD targets using γ and sampling from the buffer. Ideally, the agents would have a mapping from pouring positions to the numbers of outliers. In other words, after practicing enough times, the agents should learn how many outliers to expect if pouring is performed at a certain position. The position with the smallest number of outliers will become the ready state for pouring.

To match the change in the state space, we reduce the dimensionality of the action space for non-visual models and convert the problem into finding the optimal pouring position. Therefore, non-visual models only output $(\Delta y, \Delta z)$. We also include κ to compensate for the lack of structural information, for the same reason explained in section 3.3.1.2. More thorough work can be done in the future to include $\Delta\omega$ in non-visual models by including objects' location in non-visual observation.

The movement phase here only covers the out-of-bound and smooth penalties. Settlement phase rewards are exactly the same as the visual models.

- -2β
- $-0.01 * \sqrt{\Delta x}$

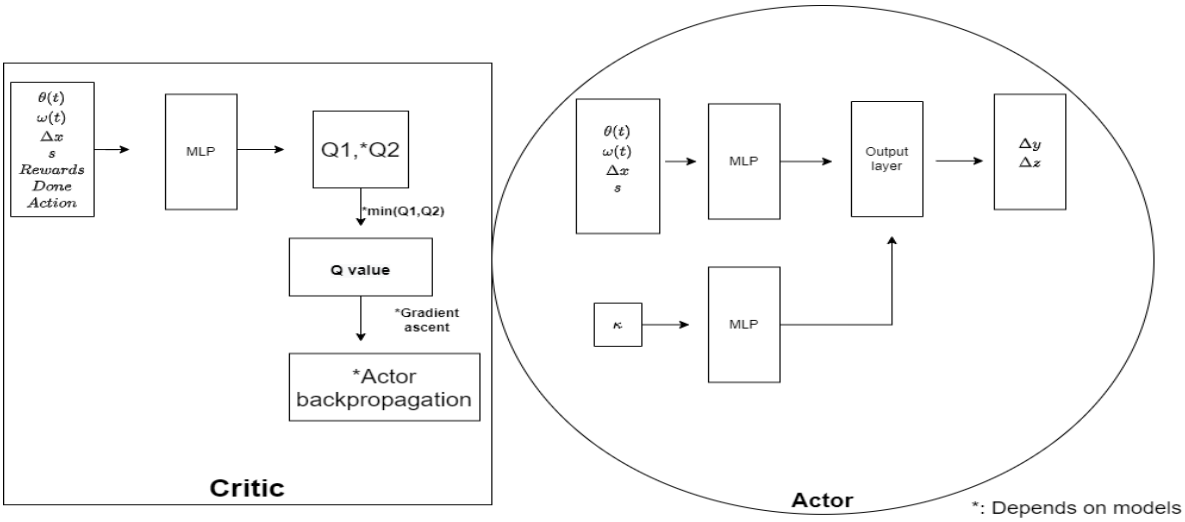


Figure 3.5: Architecture of non-visual models

3.4.2 Actor-Critic in Non-visual Models

The networks here share a similar architecture with the structural encoding model. κ is also included to provide the structural information. There are no visual inputs to the networks. The model will now output $(\Delta y, \Delta z)$.

3.4.3 Double-DQN in Non-visual Models

DDQN has a similar change. First, it doesn't have visual inputs. Secondly, κ is fed to the Q networks since there are no actors in DDQN. The rest of the implementation is the same as the counterpart in visual models. The actions for DDQN are the combination of all possible values in $(\Delta y, \Delta z)$. We show the actions in Appendix A.

3.5 Implementation and Training

This section will explain in detail how our models are implemented and trained. General training procedures are displayed in Algorithm 3.1 and 3.2. In each episode, we will randomize the object's amount as well as the size of the object in order to help our model generalize to different pouring scenarios. The objects will be dropped from different heights into the pouring cup in order to create different structures. The random seed is set to 0 for training and the current time for testing. To counter the impact of network latency, we stack 10 frames or 0.1667s into one step. A step will start with the end-effector in its original position, end when it arrives at the target position. This technique forces our agents to make the most efficient choice at each step. It lowers the requirement of communication bandwidth and input precision and contributes to a more robust model [21,22]. Rotation velocity is set to 0 when the arm tries to reach the target position derived from $(\Delta y, \Delta z)$. The latency for moving the arm is usually less than 3 frames. At each step within the episode, the agent will use the most recent observation stored in $\mathcal{V}$ to select actions. After the action is executed, the state will transit to a new state and be ready for the next step. Rewards are calculated based on the current state and action. All the MDP elements are stored in the replay buffer for TD learning. The agents will randomly sample from the buffer and perform backpropagation. The general hyper-parameters are shown in Table 3.1.

3.5.1 TD3 Implementation

For all the hidden layers we use RELU as the activation function. The output layer in the critic is linear and the actor uses tanh. This setting is applied to all the architectures shown in Figures 3.2, 3.4, and 3.5. There are twin critics in TD3, whose weights are updated separately. They will produce 2 Q values using Equation 3.1 and select the smaller one as the actual Q value. The output layer for the actor uses tanh, it outputs a deterministic real number between -1 and 1 rather than a distribution. The actions are scaled down to [-0.1, 0.1] to avoid sudden movements. Layers' weights are initialized with a uniform distribution. Layer normalization is applied to each hidden layer. There are two different learning rates for our actors and critics, where the learning rate for actors is 0.000025 and 0.00025 for the critics. Critics are updated every step while actors only get updated every two steps. By using a larger learning rate for critics, we shall have the latest and most accurate representation of the environment learned by critics. A lower update frequency and a smaller learning rate are applied to the actors to avoid overestimation. Soft-update is used for updating the target networks' weights and alleviate bootstrapping. As shown in Equation 3.4, θ is the normal network and θ^t is the target network. Each network has different initial weights. OU action noise [17] is used for exploration. The noise will have a clip value of 0.05 to avoid the noises overwhelming the actions. The hyper-parameters are listed in Table 3.2.

$$\theta^t = \tau * \theta + (1 - \tau) * \theta^t \tag{3.4}$$

3.5.2 DDPG Implementation

DDPG is very similar to TD3 because TD3 is developed based on DDPG [18]. The hyper-parameters for DDPG are shown in Table 3.3. Its weights are also initialized by a uniform distribution and uses OU action noise for exploration. However, DDPG doesn't have a twin critic system and the actors are updated in every step.

3.5.3 DDQN Implementation

Double-DQN only has the Q networks and therefore only one learning rate. The weights are initialized in the same way as above. DDQN uses the Epsilon Greedy Algorithm for exploration. The exploration rate will decay with training. The hyper-parameters are shown in Table 3.4.

Table 3.1: Summary of general hyper-parameters

Batch size	64
Epoch	1000
Early stopping patient	200
Replay buffer size	1000000
Discount factor γ	0.99
Optimizer	Adam

Table 3.2: Summary of hyper-parameters for TD3

Actor's learning rate α	0.000025
Critic's learning rate β	0.00025
Soft update factor τ	0.01
TD3 clip parameter	0.05
Actor's update frequency	2
Continuous action space	[-0.1,0.1]

Table 3.3: Summary of hyper-parameters for DDPG

Actor's 'earning rate α	0.000025
Critic's learning rate β	0.00025
Soft update factor τ	0.01
Continuous action space	[-0.1,0.1]

Table 3.4: Summary of hyper-parameters for DDQN

Learning rate λ	0.00025
Update frequency	2
Epsilon greedy exploration decay	0.001
Epsilon greedy exploration range	[0.01,1]

Algorithm 3.1 Main loop for training

Require: $\mathcal{I} = \{30, 35, 40, 45, 50\}$: initial number of objects
Require: $\mathcal{S} = \{0.025, 0.019, 0.016\}$: side length in meter
Require: $\mathcal{W} = [-5, 5]$: random rotation angles in degree
Require: $\mathcal{D} = \{\}$: empty replay buffer

for $episode = 1, 2, ..., n$ **do**
  Drop $n_{init} \sim \mathcal{U}(\mathcal{I}, \mathcal{S})$ objects in pouring container
  Sample $\omega_1, \omega_2 \sim \mathcal{U}(\mathcal{W})$
  Apply the random rotation ω_1, ω_2 to joint 2 and joint3
  Done = False
  Initialize the observation vector $\mathcal{V}$
  while Done != True **do**
    $a \leftarrow$ action selected by agents using $\mathcal{V}$
    Apply a and update $\mathcal{V}$ ▷ Done flag is updated here
    Store $\mathcal{V}$ into $\mathcal{D}$
    if $\mathcal{D}$ has enough samples **then**
      Randomly sample a batch $B \sim (\mathcal{D})$
      Update networks weights using B
    end if
  end while
end for

Algorithm 3.2 Environment step function

```
1: procedure STEP(action)
2:     Done = False, all reward = 0
3:     Initialize an empty observation vector 𝒱
4:     Update 𝒱 and Done flag
5:     if Done!=True then
6:         Apply action
7:         Calculate the movement reward
8:     else
9:         Calculate the settlement reward
10:        Terminate the episode
11:    end if
12:    Reward = movement reward + settlement reward
13:    return 𝒱, Reward, Done
14: end procedure
```

Chapter 4: Experiments and Evaluation

4.1 Training Performance

The performances of the agents are evaluated by comparing the rewards they obtained and also the number of outliers. We want to maximize the reward and minimize the number of outliers. The number of outliers is defined as objects that are not in the target box when the episode ends. This information is obtained by using a force sensor. The ideal reward is around 250. We define a pouring trial as a success if the number of outliers is less than 5% of the total objects. Each algorithm in each setting is trained for 7 times. The average performances of the agents are aggregated for fair comparison and are shown in each entry in the tables. All the data in images are normalized.

To better understand different algorithms' upper limits in aiming and pouring solid objects, the 'top 5' category is included. This category will pick the best 5 agents from each algorithm who have the smallest numbers of outliers. Some agents failed to converge towards the global optima and diverge to the opposite by pouring everything out of the target container. The failure is usually caused by a poor exploration direction. In TD3 and DDPG use OU action noise [17], and DDQN uses the Epsilon greedy algorithm for exploration [9]. A little tweak in the hyper-parameters or random seed will result in totally different learning outcomes. Therefore, this category gives us some insight into the upper limits of the algorithms.

4.1.1 Training with Visual Input

There are four visual models, namely Cam1 models, Cam2 models, Cam1 + Cam2 models, and the structural encoding model. Cam1 and Cam2 refer to the depth camera and the

cross-section view camera, as shown in Figure 3.1. Structural encoding model's architecture is discussed in Section 3.3.1.2. We observe a distinct performance difference between the types of visual inputs also types of algorithms. Figure 4.1, 4.2, and 4.3 illustrate the learning progresses. Table 4.1 compares the performances using the mean and standard deviation of the number of outliers.

At the type of algorithms level, TD3 performs the best. DDPG has a close performance if we were to use 'Top 4', which gives us 2.13 for the average number of outliers in 'Using Cam2' category. The performance of TD3 is superior because of the two techniques it adopted. The twin critics and different update frequencies help reduce the overestimation. DDQN failed to learn how to aim and pour. As training goes on, the performance decays. This suggests that random initial weights in the DDQN outperform the learned weights. Visual models will output 3 values as the actions, $(\Delta y, \Delta z, \Delta\omega)$, which gives us a 27-dimension discrete action space. [23] studies the DQN's learning capacity. They use Atari environment as the benchmark and the conclusion is DQN is likely to fail when the action space is greater than 18. The size of the action space here is bigger than the suggested maximum number. The simple Double Q networks failed to learn the large discrete action space. However, the experience gives us some insights into how to improve DDQN. [24] stacks the actions so that the output will only have one distribution rather than n different distributions. The pouring pose can be processed similarly so that we can make the action space concise.

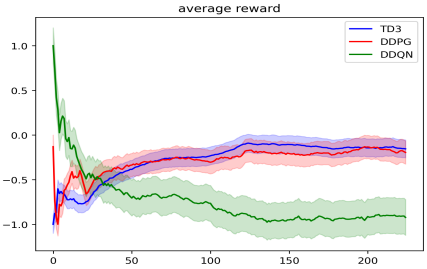
At the type of visual inputs level, the models solely using Cam1 or Cam2 barely learned. The models used both Cam1 and Cam2 had a even worse performance. Only the 'top 5' category in 'Using Cam2' for TD3 provides successful results. On the other hand, structural encoding model has overwhelming success. The number of outliers is a magnitude smaller than the others. A comparison between losses is given in Figure 4.4. This chart reveals whether the learning progress was smooth or had reached convergence or not. When using Cam1 alone, the loss didn't reduce with training because the agents were having difficulties in learning the visual representations. In each update, the gradient was large. For model using

Cam2 and structural encoding model, the speed of convergence was faster and the gradients were decreasing steadily because agents had information about the location and shape of the objects and were able to learn how to create good pouring motions. Nevertheless, the histogram of the loss itself does not reveal the agent's performance. A smooth and decreasing loss only means the agents are converging to a certain saddle point. By combining the loss histogram with the outcomes in Table 4.1 we can then confirm that our agents can't efficiently solve the task when using Cam1 which is the depth camera. Models using both Cam1 and Cam2 converged to some local optimal, which led to an undesired performance. The structural encoding model was able to learn how to solve the task efficiently.

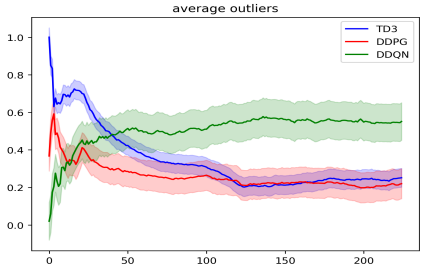
There are two major concerns when using Cam1. Firstly, the depth information reveals some insights into how the objects are layered in the source container but there is no location information about the objects. For human experts, they have the skill to deduct possible trajectories given the pouring content's structure. However, such a skill is too hard for the agents to learn. Secondly, the filtered information overfits the agents. Compared to other inputs, such as θ, ω, which are just scalars, the depth map is a 16*16 matrix flattened into a 1D array, which results in 256 elements. The difference in size forces the agents to focus more on the depth image while learning nothing from it, which makes critics fail to converge to the global optimal. The incorrect Q values then mislead the actors. As the consequence, the whole system won't be able to actively learn how to aim and pour. On the other hand, Cam2 returns a 7-element array, which correctly locates the relative position of the objects. κ is a scalar that describes the shape of the object. Both of them efficiently accomplished their tasks, which are representing the objects' location and the size of the objects. Hence, structural encoding model dexterously utilized the states and learned the optimal pouring trajectories.

Table 4.1: Mean (μ) and standard deviation (σ) of the number of outliers for visual models

		DDPG	TD3	DDQN
Using Cam1	μ	17.32	**13.15**	35.65
	σ	**9.2**	9.4	10.53
Using Cam2	μ	16.48	**7.94**	25.74
	σ	4.06	**2.6**	6.98
Using Cam1 and Cam2	μ	30.4	**21.09**	21.68
	σ	6.53	**5.4**	5.44
Structural encoding	μ	-	**1.53**	-
	σ	-	**1.62**	-
Top 5				
Using Cam1	μ	15.1	**12.62**	24.23
	σ	**9.2**	9.39	9.54
Using Cam2	μ	12.81	**1.7**	24.49
	σ	3.42	**1.6**	6.93
Using Cam1 and Cam2	μ	21.87	**17.43**	18.55
	σ	6.89	**5.04**	5.29
Structural encoding	μ	-	**1.17**	-
	σ	-	**1.1**	-

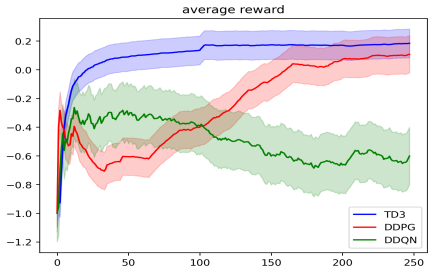


(a) Average reward

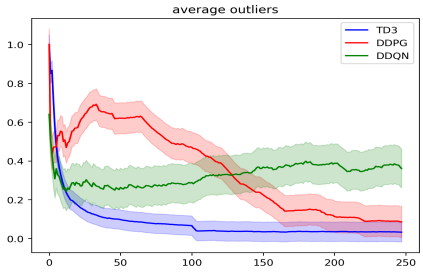


(b) Average number of outliers

Figure 4.1: Training with Cam1

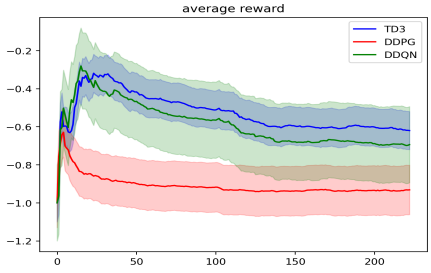


(a) Average reward

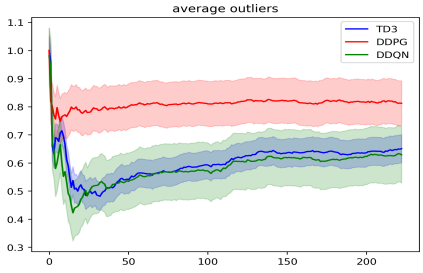


(b) Average number of outliers

Figure 4.2: Training with Cam2



(a) Average reward



(b) Average number of outliers

Figure 4.3: Training with Cam1+Cam2

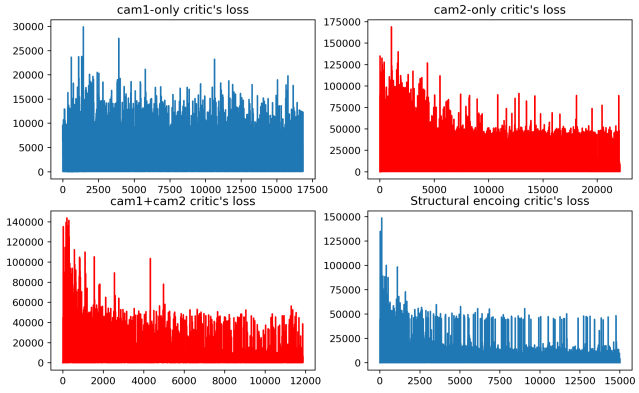


Figure 4.4: Critics' losses for different visual models

4.1.2 Training without Visual Input

We performed an ablation study on whether using κ as one of the inputs. Figure 4.5 and 4.6 demonstrate the training performance of non-visual models. In Table 4.2, their performances are compared using the mean and standard deviation of the number of outliers.

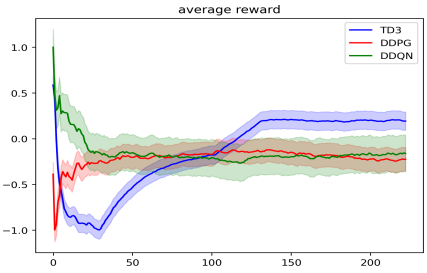
At the algorithm level, TD3 performs the best across different settings. It converges to the ideal reward much faster than the rest. The small standard deviation also indicates TD3 is more stable than the other two. DDPG has a moderate performance. The model is able to pour the majority of the content into the target but it barely reaches our definition of success. DDQN failed to learn how to pour, meaning the numbers of outliers are always greater than 5%. However, it managed to reduce the number of outliers to some extent.

Even though Double-DQN is a great improvement from DQN, the algorithm still suffers from rigid actions and overestimation. The target Q network in DQN relieves overestimation and bootstrapping to some extent but the problems still have significant impacts on the output. Additionally, DQN outputs a Q value for each action, meaning the final action is not deterministic. Given the same state, the action could be different throughout the time. Thus, DQN never reaches our definition of success. In non-visual models, the size of the action space is 9 such that $(\Delta y, \Delta z)$ are the combination of $\{\pm 0.02, 0\}$, which is less than 18 and DQN should work. Yet, it failed to aim and pour accurately. On the other hand, TD3 and DDPG produce promising results. The continuous action space helps agents move the arm to the optimal ready position smoothly and accurately. The twin critics in TD3 contribute to relieving the overestimation issue even further. It always selects the smaller Q value for the actor to perform gradient ascent, which helps regularize the actor's weights.

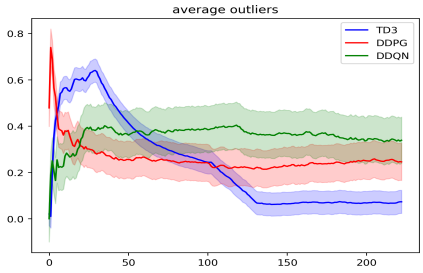
There is a significant improvement when using κ. It not only contributes to fewer outliers but also reduces the standard deviation. By introducing the structural information to our networks, the agents are able to produce better results even without visual assistance.

Table 4.2: Mean (μ) and standard deviation (σ) of the number of outliers for non-visual models

		DDPG	TD3	DDQN
With κ	μ	**3.81**	5.86	19.43
	σ	4.04	**2.12**	12.44
Without κ	μ	11.13	**9.67**	13.82
	σ	4.35	**3.53**	5.98
Top 5				
With κ	μ	3.63	**1.37**	15.47
	σ	3.94	**1.51**	10.79
Without κ	μ	3.09	**1.49**	11.84
	σ	**3.1**	3.86	5.98

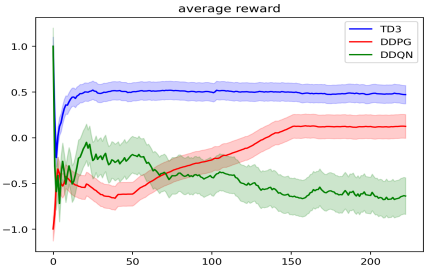


(a) Average reward

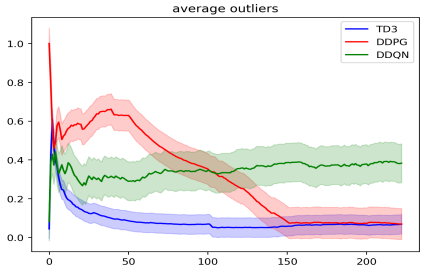


(b) Average number of outliers

Figure 4.5: Training without visual or ratio



(a) Average reward



(b) Average number of outliers

Figure 4.6: Training without visual but using ratio

4.2 Evaluation

The same pouring scene built in CoppeliaSim is used for testing. A different initial number of objects $\mathcal{I}' = \{32, 36, 43, 46, 53\}$ and the size of the object $\mathcal{S}' = \{0.027, 0.020, 0.014\}$ are used as the testing setting. Training setting discussed in algorithm 3.1 is also used during evaluation. The purpose is to have an expressive comparison of how well our models generalize to unseen objects. Random seed is set to current time to provide variations. The seed not only affects the starting position, but also the noise. We only tested the following models: 1) cross-section models; 2) structural encoding model; 3) non-visual models use κ. Furthermore, in order to show the best performance, we selected the agents that have the smallest number of outliers as the candidates. There are two reasons for only choosing the three models. Firstly, they have the best performance and greater chances to generalize to unseen objects. Secondly, due to time and resources constraints, attentions are paid to only the implementations that can solve the problem. Each agent ran for 200 episodes for evaluation.

4.2.1 Results of Visual Models

Table 4.3 illustrates the result. In Cam2 models, DDPG generalized better to unseen objects. TD3 has a larger performance decay when shifts from training to testing setting. To evaluate the performance using more readable terms, we define a total success (TS) when there is no outlier. If the number of outliers is smaller than 5% of the total pouring contents, it's called an acceptable success (AS). Three algorithms have a tight match in models using Cam2. However, the standard deviation of DDQN is almost 5 times larger than the others, which indicates DDQN is not stable and there should be many failures (outliers $> 50\%$).

Surprisingly, the structural encoding model yields outstanding results even with unseen objects, the numbers of outliers are 1.34 and 1.45, for testing and training setting respectively. It not only has the lowest number of outliers, but also the highest number in TS and AS. However, this is a very task-specific solution. If the given object differs a lot from

cuboid, the learned hidden states will not be able to correctly represent the shape of the object since the hidden layers are trained specifically on cuboids. The friction and collision properties from spheres and cylinders are quite different than cuboids. The agents will use biased weights and therefore produce sub-optimal performance. Table 4.4 shows the result when applying structural encoding model to pour cuboids, spheres, and cylinders. We can see compared to the third and fourth rows in Table 4.3, the number of outliers increased significantly. Nonetheless, our model still avoids spillage to some extent. An unbiased metric for describing an object's shape can be developed in order to extend our work to various objects.

Table 4.3: Mean (μ) and standard deviation (σ) of the number of outliers for visual models

		DDPG	TD3	DDQN
Cam2 with testing setting	μ	**3.08**	4.07	8.08
	σ	**2.68**	3.31	11.81
	0 outliers	32	**35**	**35**
	outliers $< 5\%$	**136**	119	94
Cam2 with training setting	μ	2.11	**1.99**	5.87
	σ	**1.98**	2.01	9.88
	0 outlier	55	**59**	36
	outliers $< 5\%$	162	**162**	137
Structural encoding with testing setting	μ	-	**1.45**	-
	σ	-	**1.48**	-
	0 outlier	-	**78**	-
	outliers $< 5\%$	-	**178**	-
Structural encoding with training setting	μ	-	**1.34**	-
	σ	-	**1.48**	-
	0 outlier	-	**75**	-
	outlier $< 5\%$	-	**184**	-

4.2.2 Results of Non-visual Models

For non-visual models, there is one rare agent from Double DQN, that has an average number of outliers of 0.59. During the testing, its performance is also robust across different

settings as shown in Table 4.5. However, we trained around 50 agents for each algorithm and there is only one such agent exists for Double DQN. The average performance for DDQN is shown in Table 4.2,which is around 20 outliers.

The performance of non-visual models are noticeably better than visual models. Nonetheless, it is not fair to directly compare the performance between visual and non-visual models since they have different states and action spaces. In term of the ability of solving the task, non-visual models can aim and pour accurately even with unseen objects. Our proposed method is able to accomplish the search task by mapping a pouring position to an expected number of outliers and always starting pouring from the optimal ready position. The ability to perform the task without visual inputs makes our models robust.

To evaluate the capability of generalize to objects in other shapes, we also applied non-visual models to pour cubes, cylinders and spheres. Similar to the structural encoding model, we also observed a significant drop in performance.

Table 4.4: Mean (μ) and standard deviation (σ) of the number of outliers for pouring objects in different shapes

	μ	σ	Top 5 μ	Top σ
Structural encoding	10.12	3.79	5.12	3.22
Non-visual model with κ	12.62	9.39	11.62	8.44

Table 4.5: Mean (μ) and standard deviation (σ) of the number of outliers for non-visual models

		DDPG	TD3	DDQN
Testing setting	μ	1.51	1.3	**0.59**
	σ	1.69	1.4	**0.15**
	0 outlier	75	65	**133**
	outliers $< 5\%$	**183**	180	170
Training setting	μ	1.25	1.26	**0.26**
	σ	1.34	1.43	**0.82**
	0 outlier	61	72	**145**
	outliers $< 5\%$	**188**	182	184

Chapter 5: Discussion

5.1 Limitation and Future Work

Although the work has demonstrated a promising path in robotic aiming and pouring, especially there aren't many prior works, there are some areas we could not have a chance to explore due to the time constraint. First, we didn't explore the impact of different pouring containers. The falling trajectory would be different when using a bowl vs a pitcher. RNN and imitation learning are used in our lab's previous work [6] to learn how to pour liquid using different containers. We could adopt a similar method in order to generalize the aiming problem to different containers. Secondly, κ is used to provide structural information and yields great results. But it is a very specific parameter. It won't generalize well when the object's shape differs a lot from the training ones. If we want to not only pour cuboid-like objects, but also flat objects (fried eggs), polyhedrons (nuts), we will need to develop a new metric for better generalization. Furthermore, the underlying environment is the same across different settings, e.g., gravity, friction, etc., the models should be able to efficiently learn how to manipulate the arm with offline learning. The main advantage of offline learning is that models can generalize well to unseen tasks because the data is task-agnostic. In addition, offline learning will save us some computations from rendering the simulation. Last but not least, no target pouring amount was defined in our current work. The robot will pour everything out for each episode. A possible future study can be done focusing on how to pour the exact amount specified by humans.

5.2 Conclusion

In this work, we present a solution to help manipulate robots to aim and pour solid objects. The proposed method is based on reinforcement learning. The agents receive feedback from the environment at each step and output the end-effector's movement. Our proposed method can work effectively with and without visual inputs and avoid spillage when pouring cubes in different sizes.

References

[1] David Paulius, Yongqiang Huang, Jason Meloncon, and Yu Sun. Manipulation motion taxonomy and coding for robots. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5596–5601, 2019.

[2] Yongqiang Huang and Yu Sun. Generating manipulation trajectory using motion harmonics. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4949–4954, 2015.

[3] Tianze Chen, Yongqiang Huang, and Yu Sun. Accurate pouring using model predictive control enabled by recurrent neural network. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7688–7694, 2019.

[4] Yongqiang Huang and Yu Sun. Learning to pour. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7005–7010. IEEE, 2017.

[5] Juan Wilches, Yongqiang Huang, and Yu Sun. Generalizing learned manipulation skills in practice. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9322–9328, 2020.

[6] Yongqiang Huang, Juan Wilches, and Yu Sun. Robot gaining accurate pouring skills through self-supervised learning and generalization. *Robotics and Autonomous Systems*, 136:103692, 2021.

[7] Neea Tuomainen, David Blanco-Mulero, and Ville Kyrki. Manipulation of granular materials by learning particle interactions. *arXiv preprint arXiv:2111.02274*, 2021.

[8] Homayoun Moradi, Mehdi Tale Masouleh, and Behzad Moshiri. Robots learn visual pouring task using deep reinforcement learning with minimal human effort. In *2021 9th RSI International Conference on Robotics and Mechatronics (ICRoM)*, pages 504–510, 2021.

[9] 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.

[10] 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.

[11] Chau Do, Camilo Gordillo, and Wolfram Burgard. Learning to pour using deep deterministic policy gradients. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3074–3079. IEEE, 2018.

[12] Connor Schenck and Dieter Fox. Visual closed-loop control for pouring liquids. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2629–2636. IEEE, 2017.

[13] Chau Do and Wolfram Burgard. Accurate pouring with an autonomous robot using an rgb-d camera. In *International Conference on Intelligent Autonomous Systems*, pages 210–221. Springer, 2018.

[14] Juan Wilches, Haoxuan Li, and Yu Sun. Accurate robotic pouring of solid objects. *Manuscript Submitted for Publication*, 2022.

[15] 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.

[16] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, 6(5):679–684, 1957.

[17] 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.

[18] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

[19] 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.

[20] Engineering ToolBox. Friction coefficients for some common materials and materials combinations. 2004.

[21] Rinu Boney, Jussi Sainio, Mikko Kaivola, Arno Solin, and Juho Kannala. Realant: An open-source low-cost quadruped for research in real-world reinforcement learning. *arXiv preprint arXiv:2011.03085*, 2020.

[22] Ayush Jain, Andrew Szot, and Joseph J Lim. Generalization to new actions in reinforcement learning. *arXiv preprint arXiv:2011.01928*, 2020.

[23] Zhiheng Zhao, Yi Liang, and Xiaoming Jin. Handling large-scale action space in deep q network. In *2018 International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pages 93–96, 2018.

[24] Shuo Yang, Wei Zhang, Ran Song, Jiyu Cheng, and Yibin Li. Learning multi-object dense descriptor for autonomous goal-conditioned grasping. *IEEE Robotics and Automation Letters*, 6(2):4109–4116, 2021.

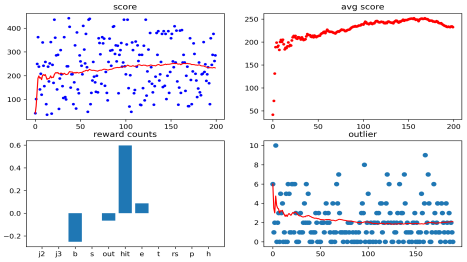
Appendix A: Supplemental Information

A.1 Double DQN Action Space

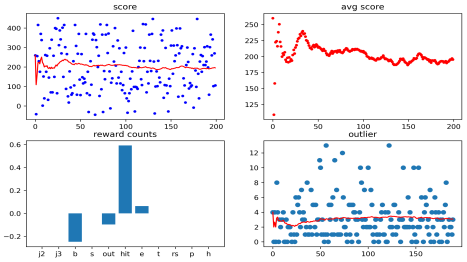
For non-visual models, they will output $(\Delta y, \Delta z)$. We choose $\pm 0.02, 0$ as the discrete actions. We will select action a from [-0.02, 0.02], [-0.02, -0.02], [-0.02, 0], [0, 0.02], [0, -0.02], [0, 0], [0.02, 0.02], [0.02, -0.02], [0.02, 0].

For visual models, they will output $(\Delta y, \Delta z, \Delta \omega)$. We choose $\pm 0.02, 0$ as the discrete actions for $(\Delta y, \Delta z)$, ± 0.05 for $\Delta \omega$). We will select action We will select action a from [-0.02, -0.02, -0.05], [-0.02, -0.02, 0], [-0.02, -0.02, 0.05], [-0.02, 0, -0.05], [-0.02, 0, 0], [-0.02, 0, 0.05], [-0.02, 0.02, -0.05], [-0.02, 0.02, 0], [-0.02, 0.02, 0.05], [0, -0.02, -0.05], [0, -0.02, 0], [0, -0.02, 0.05], [0, 0, -0.05], [0, 0, 0], [0, 0, 0.05], [0, 0.02, -0.05], [0, 0.02, 0], [0, 0.02, 0.05], [0.02, -0.02, -0.05], [0.02, -0.02, 0], [0.02, -0.02, 0.05], [0.02, 0, -0.05], [0.02, 0, 0], [0.02, 0, 0.05], [0.02, 0.02, -0.05], [0.02, 0.02, 0], [0.02, 0.02, 0.05]

A.2 Testing Results

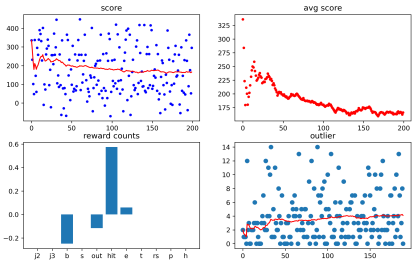


(a) Testing DDPG with training setting

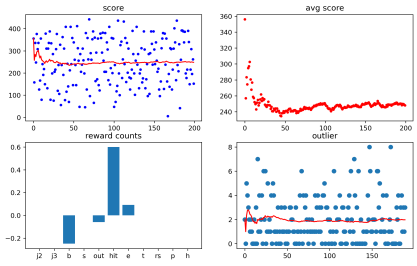


(b) Testing DDPG with testing setting

Figure A.1: Testing results of visual DDPG models

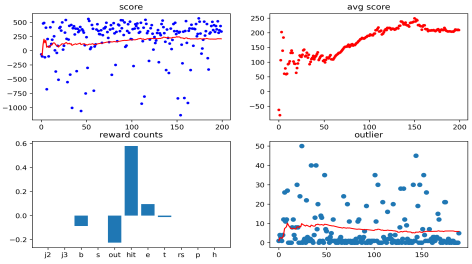


(a) Testing TD3 with testing setting

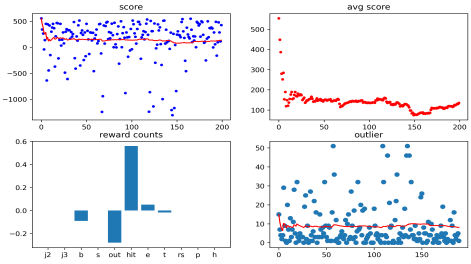


(b) Testing TD3 with training setting

Figure A.2: Testing results of visual TD3 models

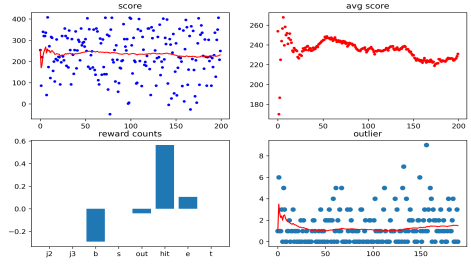


(a) Testing DDQN with testing setting

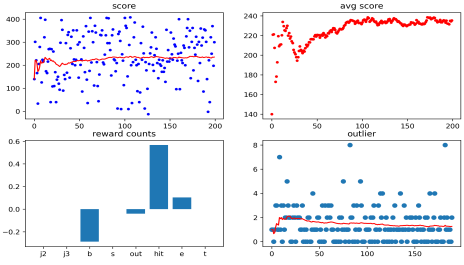


(b) Testing DDQN with training setting

Figure A.3: Testing results of visual DDQN models

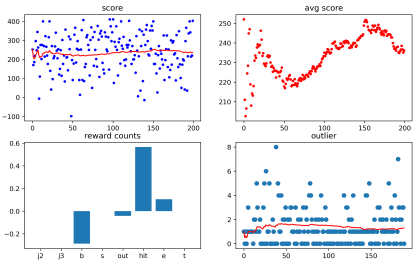


(a) Testing DDPG with training setting

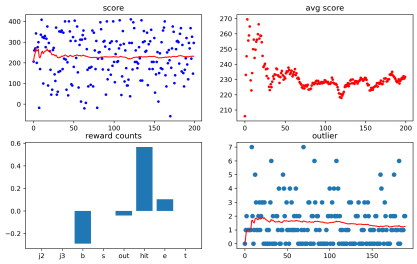


(b) Testing DDPG with testing setting

Figure A.4: Testing results of non-visual DDPG models

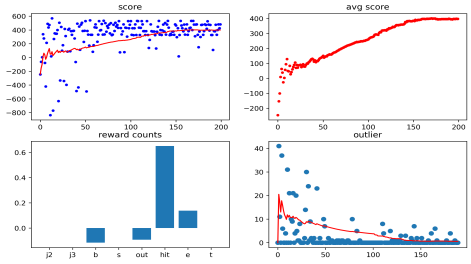


(a) Testing TD3 with testing setting

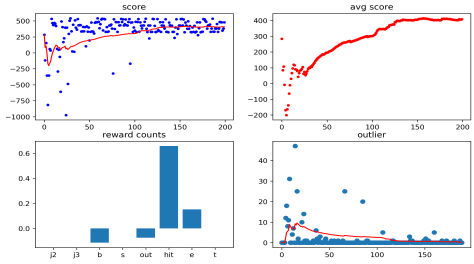


(b) Testing TD3 with training setting

Figure A.5: Testing results of non-visual TD3 models

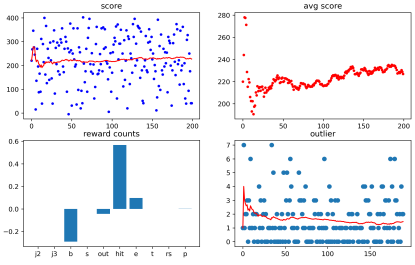


(a) Testing DDQN with testing setting

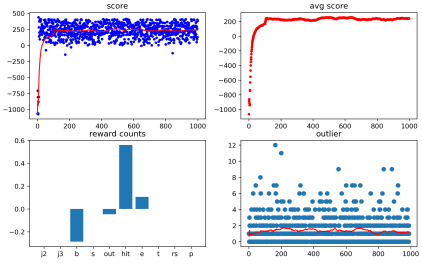


(b) Testing DDQN with training setting

Figure A.6: Testing results of non-visual DDQN models



(a) Testing structural encoding model with testing setting



(b) Testing structural encoding model with training setting

Figure A.7: Testing results of structural encoding model