Imitation Learning with Pixel-Wise Robotic End Effector Action Parametrization

Dian Wang

A thesis presented for the degree of
Master of Science
in Computer Science

Khoury College of Computer Sciences
Northeastern University
Boston, Massachusetts

Advised by
Robert Platt
Associate Professor of Khoury College of Computer Sciences

November 2019

Abstract

Pixel-wise parameterization action space in robotic end effector control provides an efficient framework for robotic reinforcement learning because of its low sample complexity. However, the high dimensional action space caused by the nature of this paradigm leads to inefficient exploration, especially in sparse reward scenario. This work tries to settle this problem by applying imitation learning algorithms to this paradigm. An optimal value function $V^*(s)$ and an expert policy $\pi^e(s)$ implemented in a full state simulator are utilized to facilitate different imitation learning algorithms, including both value based approaches and policy based approaches. This work additionally explores the performance of different variations of Fully Convolutional Network architectures, and propose an architecture applying Dynamic Filter and Input Pyramid components on top of Fully Convolutional Network. The experimental evaluation provides the comparison of different imitation learning approaches on four different simulated tabletop manipulation environments, as well as the evaluation of the advantage of the proposed architecture in contrast with three different baseline architectures.

Contents

Chapter 1

Introduction

The application of reinforcement learning on robotics faces a major challenge, potential large action space. There are two major categories of action space in robotic reinforcement learning literature, joint control (e.g. [7]) and end effector control (e.g. [26]). End effector control is typically more efficient than end-to-end joint control (e.g., from image to joint speed) because of the lower sample complexity. In end effector control literature, many researchers select the delta motion of the end effector (e.g., [9, 21, 22]) as the action space due to the potential high dimensional space in absolute position control.

Zeng *et al.* introduced a pixel-wise parameterization of robot end effector action space [26]. In this paradigm, the model takes input of an image of the scene, and outputs a Q map with the same size of the input image. Each image pixel corresponds to an end effector action with a specific motion primitive executed on the 3D location of that pixel in the scene. An advantage of this paradigm is that Fully Convolutional Neural Network (FCN) [12] can be applied to represent not only the one-to-one correspondence between a pixel in the image of the workspace and an action for the end effector but also the abstraction of different action categories. Furthermore, this method decreases the gap between simulator and reality because the observation is purely a scene image that is easy to reproduce in the real-world. Though this method works well in some short-horizon tasks like pushing-grasping [25] and picking-tossing [24], multi-step tasks (e.g., building a specific structure with toy blocks) might be challenging because of the unavailable successful trajectory caused by the large action space.

Another challenge in robotic reinforcement learning is the long action execution time. Learning from the real-world is typically infeasible since reinforcement learning algorithms normally require hundreds of thousands of episodes to train a policy. One alternative is using simulators, i.e. V-REP [16] and Pybullet [3]. However, due to the necessity of computing the physical contacts, the time needed for each action is still nonnegligible, so that the training time for a policy in some simulated robotic environment is still significantly longer than training policy in a video game environment.

In robotics literature, researches prefer to train a policy in a simulator and then transfer the learned policy to real robots. In real-world setup, the agent usually only has access to an image of the scene, so most of the policies are trained through a simulated image in simulators as well. However, in simulators, the full state of the environment is accessible, which potentially enables us to implement either an optimal value function V^* or an expert policy π^e to bootstrap the learning process using Imitation Learning algorithms.

1.1 Research Questions

The considerations above lead to the following research questions:

Research Question 1 *Can imitation learning algorithms be applied to pixel-wise parameterization action space paradigm in robotics?* The first question is if imitation learning algorithms can bootstrap the training process in the pixel-wise parameterization action space paradigm. With pixel-wise parameterization, the action space is dramatically bigger than other types of action space. Therefore, imitation learning algorithms might need to be modified to be integrated into this paradigm.

Research Question 2 *Which imitation learning algorithm works better in this paradigm?* There are many options for imitation learning algorithms that could be potentially applied. Finding the one that not only has better performance in terms of average returns but also has faster learning speed is essential due to the prolonged action execution time issue addressed above.

Research Question 3 *What network architecture works better in the pixel-wise parameterization action space paradigm?* The pixel-wise parameterization action space is relatively new, and not much work has been done in this field.

It is not clear what type of network architecture variation will have better performance.

1.2 Contributions

This work explores the use of imitation learning methods to bootstrap the learning process in the pixel-wise parameterization action space paradigm. Specifically, this work utilizes an optimal value function V^* and an expert policy π^e implemented in the simulator. V^* is used in value based methods as the target of training, and π^e is used both as the target of training in policy based methods and as guidance in the exploration process. This work compares multiple imitation learning algorithms, including both value based methods and policy based methods in four different robotic tabletop manipulation environments implemented in Pybullet [3]. This work also explores the performance of different neural network variations based on a primary Fully Convolutional Neural Network (FCN) [12].

1.3 Thesis Outline

The remainder of this thesis is structured as follows.

Chapter 2 introduces the background of reinforcement learning, starting from Markov Decision Process (MDP), a mathematical model used in this thesis to represent the problems. Then some traditional dynamic programming and temporal difference learning methods for solving MDP are introduced. In addition, chapter 2 also introduces some traditional approaches of imitation learning.

Chapter 3 introduces the proposed approach of this thesis. It starts with a problem formulation, including state representation, action representation, and reward shaping. Then, five algorithmic learning approaches are presented. This chapter ends with an introduction to the network architecture proposed by this thesis.

Chapter 4 reports the experimental evaluation of this thesis. It starts with an introduction of four different environments being used in this thesis, followed by two subsections of different experiments. The first experiment evaluates the

different approaches in this thesis, and the second experiment evaluates the performance of the proposed architecture compared with some baseline network architectures.

Finally, Chapter 5 concludes this thesis by summarizing the results. Furthermore, it points to directions for future research.

Chapter 2

Background

This chapter leads the introduction of the background of this thesis. Section 2.1 introduces the mathematical background of reinforcement learning, including MDP, traditional dynamic programming methods for solving MDP, and the most important algorithm category in reinforcement learning, temporal difference learning. Section 2.2 introduces 4 different traditional approaches in imitation learning.

2.1 Reinforcement Learning

Reinforcement learning is one of the underlying machine learning paradigms alongside supervised learning and unsupervised learning (Fig. 2.1). Reinforcement learning differs from supervised learning and unsupervised learning in that no labeled input and output pair is directly presented. The concentration of reinforcement learning is on sequential decision making in some environments such that the maximum cumulative rewards are collected. In a typical reinforcement learning scenario, in each time step, an agent will take an action in the environment, and get feedback of reward and a representation of the state (Fig. 2.2). The goal of the agent is to gather the maximum cumulative reward (i.e., return).

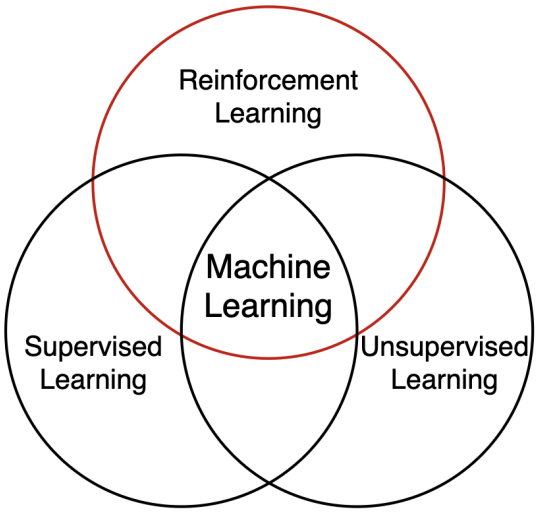


Figure 2.1: Three basic machine learning paradigms

2.1.1 Markov Decision Process

The problems in reinforcement learning are often formulated as a Markov Decision Process (MDP) [1]. Markov Decision Process is a discrete time stochastic control process, providing a mathematical framework for modeling a decision making problem.

Markov Property

In Markov Decision Process, the environment is fully observable and satisfies Markov property:

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_1, ..., s_t) \tag{2.1}$$

where s_t is the current state and s_{t+1} is the next state. The current state captures all relevant information from the history, and the next state is independent of the history given the current state.

MDP

A Markov Decision Process is a 4 tuple (S, A, P, R):

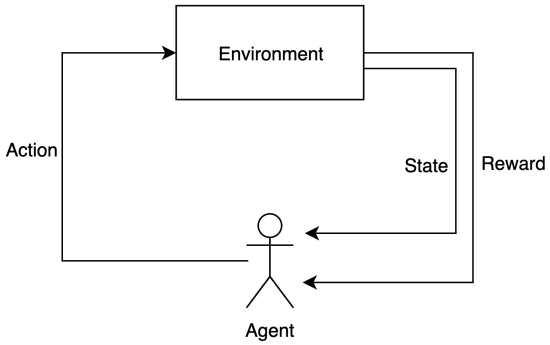


Figure 2.2: An agent take an action in the environment, and get feedback of reward and a representation of the state

- S: state space.
- A: action space.
- $P(s_{t+1}|s_t, a_t)$: transition dynamics, the probability distribution of next state s_{t+1} given the current state action pair (s_t, a_t).
- $R(s_t, a_t, s_{t+1})$: reward function, the immediate reward r_t after executing action a_t in state s_t and transitioned to state s_{t+1}. It could also be formulated as $R(s_t, a_t)$ or $R(s_{t+1})$, depending on the problem.

Policy

A policy $\pi(s)$ is defined as a distribution over actions given states:

$$\pi(a|s) = P(a_t = a|s_t = s) \tag{2.2}$$

A policy fully defines the behavior of an agent.

Return

Return is defined as the cumulative discounted reward G:

$$G_t = \sum_{i=t}^{\infty} \gamma^i R(s_i, a_i, s_{i+1}) \tag{2.3}$$

where γ is a discounting factor that satisfies:

$$0 \leq \gamma \leq 1 \tag{2.4}$$

There are many reasons why a discounting factor is applied. First, it is mathematically convenient to discount rewards. Second, discounting rewards can avoid infinite returns in infinite horizon MDP. Third, it can leverage the uncertainty about the future which is not fully represented.

Value Function

A state value function $V^{\pi}(s)$ is defined as the expected return starting from state s, and then following policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t | s_t = s] \tag{2.5}$$

A state-action value function $Q^{\pi}(s, a)$ is defined as the expected return starting from state s, taking action a, and then following policy π:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[G_t | s_t = s, a_t = a] \tag{2.6}$$

Bellman Equation

Bellman equation writes the value in terms of the immediate reward from the initial choices and the value of the remaining decision problem that results from those initial choices.

For state value function $V^{\pi}(s)$, it can be decomposed as:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[r_t + \gamma V^{\pi}(s_{t+1}) | s = s_t] \tag{2.7}$$

State action value function $Q^{\pi}(s)$ can be decomposed similarly:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[r_t + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) | s = s_t, a = a_t] \tag{2.8}$$

Optimal Value Function

The optimal state value function $V^*(s)$ is the maximum state value function over all policies:

$$V^*(s) = \max_{\pi} V^*(s) \tag{2.9}$$

The optimal state value function $Q^*(s,a)$ is the maximum state action value function over all policies:

$$Q^*(s,a) = \max_{\pi} Q^*(s,a) \tag{2.10}$$

Optimal Policy

Define an ordering over policies:

$$\pi \geq \pi' \text{ if } V^{\pi}(s) \geq V^{\pi'}(s), \forall s \tag{2.11}$$

Theorem 1 *For any Markov Decision Process:*
There exists an optimal policy π^* *that is better than or equal to all other policies,* $\pi^* \geq \pi, \forall \pi$
All optimal policies achieve the optimal state value function, $V^{\pi^*}(s) = V^*(s)$
All optimal policies achieve the optimal state action value function, $Q^{\pi^*}(s,a) = Q^*(s,a)$

An optimal policy can be found by maximising over $Q^*(s,a)$:

$$\pi^*(s) = \text{argmax}_a Q^*(s,a) \tag{2.12}$$

An MDP is solved when the optimal policy or the optimal state action value function is acquired.

2.1.2 Dynamic Programming

When P and R are available, MDP could be solved using dynamic programming [2]. The value function $V^{\pi}(s)$ could be rewritten with P and R as:

$$V^{\pi}(s_t) = \sum_{a_t, s_{t+1}} P(s_{t+1}|s_t, a_t)[R(s_t, a_t, s_{t+1}) + \gamma V^{\pi}(s_{t+1})] \tag{2.13}$$

A greedy policy $\pi(s)$ could be rewritten as:

$$\pi(s_t) = \text{argmax}_a\{\sum_{s_{t+1}} P(s_{t+1}|s_t, a)[R(s_t, a, s_{t+1}) + \gamma V^{\pi}(s_{t+1})]\} \tag{2.14}$$

Value Iteration

In value iteration [1], the value function $V^{\pi}(s)$ is updated iteratively:

$$V_{i+1}(s_t) = \max_a\{\sum_{s_{t+1}} P(s_{t+1}|s_t, a)[R(s_t, a, s_{t+1}) + \gamma V_i(s_{t+1})]\} \tag{2.15}$$

where i is the number of iterations, and V_0 is an initial guess of the value function and is typically 0 for all states. Value iteration iteratively updates the value function for all states in each iteration, until the value function converges to the optimal value function.

Policy Iteration

In policy iteration [5], there are two steps: policy evaluation and policy improvement. The first step is policy evaluation, where the value function $V^{\pi}(s)$ of policy π is evaluated iteratively similar to value iteration:

$$V^{\pi}_{i+1}(s_t) = \sum_{s_{t+1}} P(s_{t+1}|s_t, \pi(s_t))[R(s_t, \pi(s_t), s_{t+1}) + \gamma V^{\pi}_i(s_{t+1})] \tag{2.16}$$

The second step is policy improvement, where the policy is updated with Equation 2.14. The algorithm ends when there is no change in the policy improvement step, indicating the policy is converged to the optimal policy.

2.1.3 Temporal Difference Learning

However, P and R are not always available. When they are not, MDP could be solved via temporal difference learning (TD learning) [20]. In TD learning, if

the agent executes action a_t at state s_t base on policy $\pi(s)$, transitions to state s_{t+1} and receives reward r_{t+1}, the value function $V^\pi(s_t)$ is updated as:

$$\begin{aligned} V^\pi(s_t) &= (1-\alpha)V^\pi(s_t) + \alpha[r_t + \gamma V^\pi(s_{t+1})] \\ &= V^\pi(s_t) + \alpha[(r_t + \gamma V^\pi(s_{t+1})) - V^\pi(s_t)] \end{aligned} \tag{2.17}$$

where $[(r_t + \gamma V^\pi(s_{t+1})) - V^\pi(s_t)]$ is also know as the TD error. TD error is calculated with the difference of the target from the Bellman equation and the current value function output. α is the learning rate (also known as step size) that controls the step of each learning iteration, $0 < \alpha < 1$.

Q Learning

Q learning is a variation of temporal difference learning that updates the state action value function $Q(s, a)$:

$$Q^\pi(s_t, a_t) = Q^\pi(s_t, a_t) + \alpha[(r_t + \gamma \max_{a_{t+1}} Q^\pi(s_{t+1}, a_{t+1})) - Q^\pi(s_t, a_t)] \tag{2.18}$$

The algorithm is shown in Algorithm 1.

Algorithm 1 Q Learning

initialize Q table $Q(s, a)$ arbitrary , and $Q(\text{terminal state}, \cdot) = 0$
for episode=1, M **do**
  for t=1, T **do**
    choose action a_t using policy derived from Q (e.g. ϵ-greedy)
    execute a_t, get (s_{t+1}, r_t)
    $Q(s_t, a_t) = Q(s_t, a_t) + \alpha[(r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t)]$
  end for
end for

Deep Q Learning

Q learning requires a Q table to store state-action values, which means that it can not work if there is an infinite number of states or actions, or the state space or action space is too large. Deep Q Learning, also known as Deep Q Network (DQN) [13, 14] leverages the advantage of deep learning to learn a Q network that takes input of high dimensional image input as a state s and outputs the Q value for each action a, $Q(s, a)$. In order to make deep learning working in the Q learning paradigm, DQN proposes the following techniques:

1. Experience Replay

   Past sequential experiences are highly correlated with each other. To achieve an independent and identically distribution of the data for updating the network model, DQN gathers and stores transitions in a replay buffer while interacting with the environment. In each training step, it randomly samples a minibatch of transitions from the replay buffer, and use those transitions to update the Q network.

2. Loss Function

   Like Q learning, DQN also tries to minimize the temporal difference error between the target Q value from the Bellman equation and the current Q value output. The loss function is defined as the mean square error of the temporal difference error. Suppose a Q network $Q(s, a|\phi)$ is parameterized by ϕ, the loss function $L(\phi)$ is defined as:

$$L(\phi) = \mathbb{E}_{s_t, a_t}[\frac{1}{2}(y - Q(s_t, a_t|\phi))^2] \tag{2.19}$$

   where

$$y = \begin{cases} r_t, & \text{if terminated} \\ r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}|\phi), & \text{otherwise} \end{cases} \tag{2.20}$$

3. Target Network

   In Equation 2.20, the target y of an update is calculated by $Q(s, a|\phi)$ itself, which means the algorithm will chase a non-stationary target. A target network $\overline{Q}(s, a|\overline{\phi})$ is introduced to fix this. Target network $\overline{Q}(s, a|\overline{\phi})$ is updated to Q network every certain amount of training steps, and it calculates the target of each training step:

$$y = \begin{cases} r_t, & \text{if terminated} \\ r_t + \gamma \max_{a_{t+1}} \overline{Q}(s_{t+1}, a_{t+1}|\overline{\phi}), & \text{otherwise} \end{cases} \tag{2.21}$$

2.2 Imitation Learning

Reinforcement learning is cheap and straightforward in terms of supervision, but it suffers from high sample complexity. In the robotic domain, action execution

time is extended, even within simulators. This harms the performance of reinforcement learning algorithms because the time expense is often unaffordable.

In imitation learning, supervision through an expert is applied. The expert policy $\pi^e(s)$ provides example trajectories for the agent to imitate, and the goal for the agent is to clone the behavior of the expert such that the resulting state, action trajectory distribution matches the expert trajectory distribution.

Imitation learning is helpful because it often has higher sample efficiency. In Q learning, the target of the update is generated by the bellman equation, which requires the correct target to be back propagated through the terminal state to the initial state, while in imitation learning, the target will be fixed at the expert's behavior.

2.2.1 Imitation Learning via Exhaustive Sampling

The most intuitive approach in imitation learning is treating it as a classification problem: for each state s_t, the agent needs to classify the expert action a_t^e from the action space to execute. To train such a classifier, a dataset of state and expert action pairs is needed. Algorithm 2 shows the most straightforward approach to gather such a dataset and train a policy.

Algorithm 2 Imitation Learning via Exhaustive Sampling

1: initialize dataset $D = \{(s, a)\}$
2: **for** i=1, N **do**
3: sample $s \in S$ uniformly
4: set $D = D \cup (s, \pi^e(s))$
5: **end for**
6: train π using D

However, in Algorithm 2, the distribution of state in D is a uniform distribution over the state space S, so that training an imitator policy might be expensive.

2.2.2 Imitation Learning via Sampling from Expert Policy

An alternative of Algorithm 2 is gathering on policy data using the expert policy π^e. A number of researchers have followed this approach including [17] who call this the *traditional approach*. In this method, the expert policy rolls out each

episode and stores the state-action pairs in D. The whole algorithm is shown in Algorithm 3.

Algorithm 3 Imitation Learning via Sampling from Expert Policy

```
initialize dataset D = {(s, a)}
for episode=1, M do
    initialize initial state s_1
    for t=1,T do
        set a_t = π^e(s_t)
        D = D ∪ (s_t, a_t)
        execute a_t, get s_{t+1}
    end for
end for
train π using D
```

Comparing to Algorithm 2, the agent is trained using a dataset lays in an on-policy distribution. However, the distribution is on-policy for the expert policy, not the learned policy. As a result, the learned policy may cause the agent to visit parts of state space for which it was not trained, resulting in poor performance.

2.2.3 DAGGER

DAGGER[19] is a popular imitation learning algorithm that fixes the on/off policy problem described above by interleaving exploration by the expert policy π^e and the learned policy π. The whole algorithm is shown in Algorithm 4.

DAGGER uses a behavior policy $\pi^b = \beta\pi^e + (1 - \beta)\pi$, which utilizes the expert with a probability of β, and the learned policy π with a probability of $1 - \beta$. β decays from 1 to 0 with a rate λ.

2.2.4 AGGREVATE

AGGREVATE [18] is a value based imitation learning method in which a value function (i.e. $Q(s, a)$) is learned. The value function is then used to select actions. A key question here is how to obtain estimates of value (i.e., discounted reward) to use as targets during learning. AGGREVATE achieves this by rolling out the expert policy and observing costs produced by sample rollouts. As is shown in Algorithm 5, for each iteration i, the inner loop executes the behavior

Algorithm 4 DAGGER

initialize dataset $D = \{(s, a)\}$
initialize policy $\pi(s)$
set $\lambda = [0, 1]$
set $\beta = 1$
for episode=1, M **do**
  set $\pi^b = \beta\pi^e + (1 - \beta)\pi$
  initialize initial state s_1
  for t=1,T **do**
    set $a_t = \pi^b(s_t)$
    $D = D \cup (s_t, a_t)$
    execute a_t, get s_{t+1}
  end for
  train π using D
  $\beta = \lambda\beta$
end for

policy up to time $t - 1$. Then, at state s_t in time step t, it takes an exploratory action a_t and rolls out the remainder of the trajectory using the expert policy. During the rollout, it obtains the value $Q(s, a)$. Once the value function is learned, a policy can be obtained by taking an argmax over the action values from a given state.

Algorithm 5 AGGREVATE

initialize dataset $D = \{(s, a, Q)\}$
initialize policy $\pi(s)$
for i=1, N **do**
  set $\pi^b = \beta\pi^e + (1 - \beta)\pi$
  collect m data as follows:
  for j=1, m **do**
    sample uniformly $t \in \{1, ..., T\}$
    initialize initial state s_1
    execute π^b up to time $t - 1$
    execute some exploration action a_t in current state s_t at time t
    execute π^e from time $t + 1$ to T, observe estimate of cost-to-go $Q(s, a)$
  end for
  gather dateset $D_i = \{(s, a, Q)\}$
  aggregate dataset $D = D \cup D_i$
  train cost-sensitive π using D
end for

Chapter 3

Approach

This section presents the proposed approach. Section 3.1 formulate the problem into an MDP with proper state representation, action representation, and reward shaping. Section 3.2 introduces the five different algorithmic approaches proposed by the thesis. In the end, Section 3.3 presents the neural network architecture used by this thesis.

3.1 Problem Statement

The table top pixel-wise manipulation problem could be formulated as a Markov decision process: in a state s_t at time step t, the agent (robot) need to select an action a_t base on a policy $\pi(s_t)$ and execute that action. The agent then transitions to a new state s_{t+1} and receive an immediate reward $R(s_t, a_t, s_{t+1})$. The target of the agent is to maximizes the cumulative discounted reward $G_t = \sum_{i=t}^{\infty} \gamma^i R(s_i, a_i, s_{i+1})$.

This work investigates the training of a greedy deterministic policy $\pi(s_t)$ that selects action maximizing a state-action function $Q(s_t, a_t)$

3.1.1 State Representation

This work model each state s_t into a pair of depth image of the scene, I_t, the in-hand image of the previous time step, H_{t-1}, and the holding state of the

gripper c_t:

$$s_t = (I_t, H_{t-1}, c_t) | c_t \in \{0, 1\} \tag{3.1}$$

Where $c = 0$ stands for there is no holding object in the gripper, and $c = 1$ stands for the gripper is holding a object. The scene in I covers 0.3m $\times$ 0.3m table top surface, and the pixel resolution of the depth image I is 90×90, so each pixel represents approximately 0.1cm^2 in the scene. H_t is generated by cropping I_t centered at the picking position.

3.1.2 Action Representation

This work only considers top-down end effector actions. Each action a is parameterized by a primitive action p representing the end effector motion (pick or place in this work), the middle position of the end effector (x, y), and the orientation θ of the end effector along z axis:

$$a = (p, x, y, \theta) | p \in \{\text{pick}, \text{place}\} \tag{3.2}$$

Inspired by [25], This work uses pixel-wise mapping between image of the scene I and (x, y) in action a, each pixel in I is a potential (x, y) position:

$$|x| \times |y| = |I| = 90 \times 90 \tag{3.3}$$

θ is represented by rotating I. Given a fixed (x, y) pair, rotating the end effector by θ degrees along z axis centered at (x, y) in the scene is spatially the same as rotating the scene $-\theta$ degrees along z axis centered at (x, y). In this work, θ is a set of 8 discrete value between 0 and π (from the end effector's perspective, $[\pi, 2\pi]$ is spatially the same as $[0, \pi]$):

$$\theta = i \times \frac{\pi}{8} | i \in [0..7] \tag{3.4}$$

The z position in all actions are decided by heuristics. In primitive actions p, a pick action consists of moving the end effector to the target position and close the gripper, and a place action consists of moving the end effector to the

target position and open the gripper. p is decided by c:

$$p = \begin{cases} \text{pick}, & c = 0 \\ \text{place}, & c = 1 \end{cases} \tag{3.5}$$

A pick action will be executed if the gripper is not holding an object. Otherwise, a place action will be executed.

3.1.3 Reward Shaping

This work applies sparse reward to all tasks.

$$R(s_t, a_t, s_{t+1}) = \begin{cases} 1, & \text{the task is accomplished} \\ 0, & \text{otherwise} \end{cases} \tag{3.6}$$

The agent only gets +1 reward when the task is accomplished.

3.2 Algorithm

3.2.1 DQN

The most intuitive approach is vanilla Deep Q-Network (DQN) [13, 14], in which a Q network takes input of state s_t and outputs the state-action value $q(s_t, a_t)$ for each action a_t. The Q network is trained using Huber loss [6] and the temporal difference error:

$$\begin{aligned} x &= Q(s_t, a_t) \\ y &= \begin{cases} r_t, & \text{if terminated} \\ r_t + \gamma \max_{a_{t+1}} \overline{Q}(s_{t+1}, a_{t+1}), & \text{otherwise} \end{cases} \\ L_Q &= \text{Huber}(x, y) = \begin{cases} \frac{1}{2}(x-y)^2, & \text{for } |x-y| < 1 \\ |x-y| - \frac{1}{2}, & \text{otherwise} \end{cases} \end{aligned} \tag{3.7}$$

where Q is the Q network, $\overline{Q}$ is a target Q network. The full algorithm is presented in Algorithm 6.

The exploration is achieved by adding a Gaussian noise on the output Q

Algorithm 6 DQN

initialize Q network $Q(s, a)$
initialize target Q network $\overline{Q}(s, a)$
initialize replay buffer $D(s, a, s', r)$
anneal ϵ in K episodes
for episode=1, M **do**
  for t=1, T **do**
    $\pi^b = \text{argmax}_a[Q(s_t, a) + \mathcal{N}(0, (c \cdot \epsilon)^2)]$
    execute $a_t = \pi^b(s_t)$, get (s_{t+1}, r_t)
    $D = D \cup (s_t, a_t, s_{t+1}, r_t)$
    sample minibatch (s_t, a_t, s_{t+1}, r_t) from D
    set $x = Q(s_t, a_t)$
    set $y = \begin{cases} r_t, & \text{for terminated } s_{t+1} \\ r_t + \gamma \max_{a_{t+1}} \overline{Q}(s_{t+1}, a_{t+1}), & \text{otherwise} \end{cases}$
    set $L_Q = \text{Huber}(x, y)$
    perform a gradient descent step
    set $\overline{Q} = Q$ after a certain amount of training steps
  end for
end for

map and then take the argmax instead of performing random action because the exploration around the empty area in the scene is meaningless.

3.2.2 DQN with Q* target

Base on Equation 2.5, the value of state s is defined as:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[\sum_{t=0}^{\infty} \gamma^t r_t] \tag{3.8}$$

This work utilizes sparse reward. Given the reward sequence $[r_1, ..., r_T]$ in a successful trajectory:

$$r_t = \begin{cases} 0, & 1 \leq t < T \\ 1, & t = T \end{cases} \tag{3.9}$$

Where T is the termination time step, i.e. the goal is accomplished.

This implies:

$$\begin{aligned} V^{\pi}(s_t) &= \mathbb{E}_{\pi}[\sum_{t=0}^{\infty} \gamma^t \mathbb{I}(t = T)] \\ &= \mathbb{E}_{\pi}[\gamma^{\tau}] \\ V^{*}(s) &= \mathbb{E}_{\pi}[\gamma^{\tau^*}] \end{aligned} \tag{3.10}$$

Where τ is the steps needed to reach the goal from state s_t following policy π, and τ^* is the steps needed to reach the goal from state s_t following the optimal policy π^*.

τ^* could be calculated inside the simulator using heuristics taking account of the target state and the current state. So instead of using temporal difference error to train a Q network, this approach trains the network using the optimal value function $V^*(s)$.

The full algorithm is presented in Algorithm 7

Algorithm 7 DQN with Q*

initialize Q network $Q(s, a)$
initialize replay buffer $D(s, a, s', \tau^*)$
anneal ϵ in K episodes
for episode=1, M **do**
  for t=1, T **do**
    $\pi^b = \text{argmax}_a[Q(s_t, a) + \mathcal{N}(0, (c \cdot \epsilon)^2)]$
    execute $a_t = \pi^b(s_t)$, get (s_{t+1}, τ^*)
    $D = D \cup (s_t, a_t, s_{t+1}, \tau^*)$
    sample minibatch $(s_t, a_t, s_{t+1}, \tau^*)$ from D
    set $x = Q(s_t, a_t)$
    set $y = \gamma^{\tau^*}$
    set $L_Q = \text{Huber}(x, y)$
    perform a gradient descent step
  end for
end for

The advantage of this method comparing to vanilla DQN is that the access of $V^*(s) = \gamma^{\tau^*}$ changes the TD learning problem into a regression problem. In TD learning, the correct training target needs to be back propagated from the terminal state to the initial state, while with $V^*(s)$, the model is fed with the ground truth value from the first training step. This makes the training much

more efficient.

This approach is similar to AGGREVATE [18], but does not require the rollout of expert policy to get the value target. This makes training a lot faster due to the long action execution time in robotics.

3.2.3 Expert Guided Exploration

In expert guided exploration, the exploration is not guided by a random policy. Instead, it is guided by a expert policy π^e:

$$\pi^b = \epsilon\pi^e + (1-\epsilon)\hat{\pi} \tag{3.11}$$

where π^e is generated by a full state planner inside the simulator.

Since exploration policy is orthogonal to updating, expert guided exploration could be applied to both vanilla DQN and DQN with Q^*. DQN with Q^* and guided exploration is shown in Algorithm 8

Algorithm 8 DQN with Q* and Guided Exploration

initialize Q network $Q(s,a)$
initialize replay buffer $D(s,a,s',\tau^*)$
anneal ϵ in K episodes
for episode=1, M **do**
  for t=1, T **do**
    $\hat{\pi} = \text{argmax}_a(Q(s_t,a))$
    $\pi^b = \epsilon\pi^e + (1-\epsilon)\hat{\pi}$
    execute $a_t = \pi^b(s_t)$, get (s_{t+1},τ^*)
    $D = D \cup (s_t,a_t,s_{t+1},\tau^*)$
    sample minibatch (s_t,a_t,s_{t+1},τ^*) from D
    set $x = Q(s_t,a_t)$
    set $y = \gamma^{\tau^*}$
    set $L_Q = \text{Huber}(x,y)$
    perform a gradient descent step
  end for
end for

Due to the extensive action space in the pixel-wise parameterization action space and the sparse reward in the tasks, it is hard for the agent to gather a successful trajectory. Guided exploration tends to solve that by exploring using

a mixed policy of the learned policy and the expert policy.

3.2.4 Modified DAGGER

DAGGER [19] learns a policy to imitate an expert policy. In the pixel-wise action space framework, this work can apply DAGGER by treating the policy as a classification problem: given a state s_t, the model needs to classify the expert action from the action space. The policy being trained is a greedy policy that executes the action having the highest probability to be the expert action. The full algorithm is presented in Algorithm 9.

Algorithm 9 Modified DAGGER

initialize pseudo-Q network $\tilde{Q}(s, a)$
initialize replay buffer $D(s, a^e)$
anneal ϵ in K episodes
for episode=1, M **do**
  for t=1, T **do**
    $\hat{\pi} = \text{argmax}_a(\tilde{Q}(s_t, a))$
    $\pi^b = \epsilon\pi^e + (1 - \epsilon)\hat{\pi}$
    execute $a_t = \pi^b(s_t)$
    get expert action $a_t^e = \pi^e(s_t)$, add (s_t, a_t^e) to D
    sample minibatch (s_t, a_t^e) from D
    set $p(a_t|s_t) = \frac{e^{\tilde{Q}(s_t, a_t)}}{\sum_a e^{\tilde{Q}(s_t, a)}}$
    set $L = \mathbb{E}[-\log\ p(a_t^e|s_t)]$
    perform a gradient descent step on L
  end for
end for

In the modified DAGGER, this work train a policy by training a pseudo-Q network $\tilde{Q}(s, a)$ that outputs the score for each action in a given state. Note that though the network is outputting the state action score, the score is not the Q value in the Q-learning framework. During training, the scores are sent to a Softmax function to generate the probability of each action being the expert action, and the loss function is the negative log-likelihood of the expert action.

3.2.5 Modified ADET

Lakshminarayanan *et al.* introduced Accelerated DQN with Expert Trajectories (ADET) [11], in which they combine TD loss with a cross-entropy imitation loss:

given state action pair (s_t, a_t) and the expert action a_t^e, define:

$$\begin{aligned} p(a_t|s_t) &= \frac{e^{\tau^{-1}Q(s_t,a_t)}}{\sum_a e^{\tau^{-1}Q(s_t,a)}} \\ L_E &= \mathbb{E}[-\log p(a_t^e|s_t)] \\ L_{\text{ADET}} &= L_Q + wL_E \end{aligned} \tag{3.12}$$

where L_Q could be either from vanilla DQN or DQN with Q^*. The full algorithm of the modified ADET is presented in Algorithm 10

Algorithm 10 Modified ADET

initialize Q network $Q(s, a)$
initialize target Q network $\overline{Q}(s, a)$
initialize replay buffer $D(s, a, s', r)$
anneal ϵ in K episodes
for episode=1, M **do**
  for t=1, T **do**
    $\hat{\pi} = \text{argmax}_a(Q(s_t, a))$
    $\pi^b = \epsilon\pi^e + (1 - \epsilon)\hat{\pi}$
    execute $a_t = \pi^b(s_t)$, get (s_{t+1}, r_t)
    $D = D \cup (s_t, a_t, s_{t+1}, r_t)$
    sample minibatch (s_t, a_t, s_{t+1}, r_t) from D s.t. half of them are expert transitions
    set $L = \begin{cases} L_Q, & \text{for non expert transitions} \\ L_{\text{ADET}}, & \text{for expert transitions} \end{cases}$
    perform a gradient descent step on L
    set $\overline{Q} = Q$ after a certain amount of training steps
  end for
end for

ADET is essentially a combination of DQN and DAGGER. It learns a value function $Q(s, a)$ which is biased to the expert action. It generates a probability distribution on A by applying a Softmax function on Q, and tries to maximize the probability of the expert action a^e.

3.3 Network Architectures

For a primitive action $p \in \{\text{pick}, \text{place}\}$ and a rotation θ, the depth image of the scene I with a resolution of 90×90 is padded with zeros to the size of its diagonal, 128×128, and is rotated by θ degrees. The resulting image I_θ with

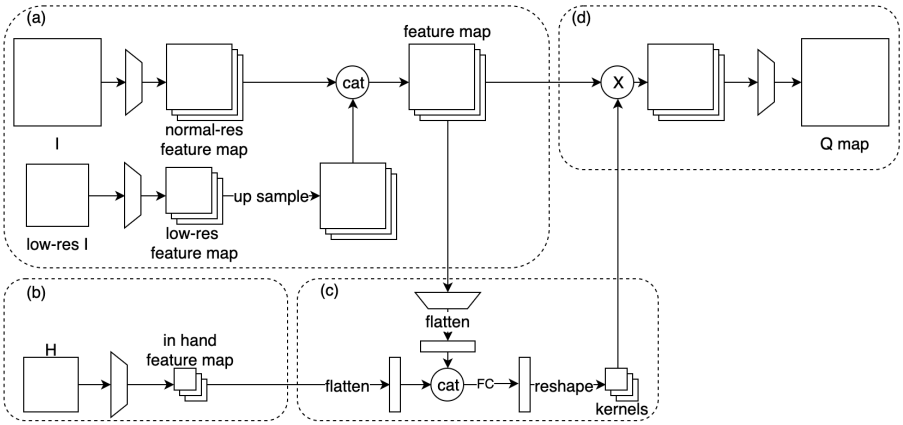


Figure 3.1: The network architecture with four components: (a) a scene feature map encoder, (b) an in hand feature map encoder, (c) a kernel decoder, (d) a Q map generator.

a size of 128×128 and an in-hand image H with a size of 24×24 is the input of the network. The output has the same size of I_θ, which is then rotated $-\theta$ degrees, and the zero paddings are removed to generate a Q value map with size 90×90. The value at the pixel (x, y) in the resulting Q map represents the Q value for the action at (p, x, y, θ). The underlying architecture of the model is the modified Fully Convolutional Neural Network (FCN) [12] from [25]. This work applied the concept of Dynamic Filter Network [8] and the input pyramid [23] to enrich the global features inside the model.

A scene feature map encoder (Fig. 3.1 (a)) encodes I_θ with a size of 128×128 into a 32-channel feature map. First, a low-resolution I_θ with a size of 64×64 is generated by downsampling I_θ. Then two groups of convolutional layers are applied to I_θ and low resolution I_θ respectively. They have the same shape but do not share weights. Each group has one $32 \times 11 \times 11$ layer, one $32 \times 5 \times 5$ layer, two $32 \times 3 \times 3$ layers, and one $16 \times 1 \times 1$ layer. All layers have a stride of 1 and zero paddings to keep the image size consistent while passing through each layer. The output feature map of low resolution I_θ is upsampled to 128×128. The concatenation of a 16-channel normal resolution feature map and a 16-channel upsampled low-resolution feature map forms the 32-channel scene feature map.

An in hand feature map encoder (Fig. 3.1 (b)) encodes H with a size of 24×24 into a 64-channel feature map. It has one $32 \times 4 \times 4$ convolutional layer

with a stride of 2, one $64 \times 4 \times 4$ convolutional layer with a stride of 1, and one $64 \times 3 \times 3$ convolutional layer with a stride of 1.

A kernel decoder (Fig. 3.1 (c)) takes input of the feature maps from (a) and (b) to generate a kernel. One $8 \times 1 \times 1$ convolutional layer followed by a 2×2 max pool layer is applied to the scene feature map from (a) to reduce the scale. The output is flattened into a vector and is then concatenated with the flattened vector from the in-hand feature map. Then two fully connected layers with 1024 and 1056 hidden units are applied to the concatenated vector. The output vector is reshaped into a $32 \times 1 \times 1$ kernel ($1065 = 32 \times 32 + 32$). When $p =$ pick, the kernel decoder only takes input of the feature maps from (a).

A Q map generator (Fig. 3.1 (d)) applies convolution operation to the feature map from (a) and the kernel from (c), and applies a $1 \times 1 \times 1$ convolutional layer to generate the Q value map.

Each primitive action (pick, place) has its own network, and the 8 rotations is represented by 8 channels of the input image.

Chapter 4

Experiments

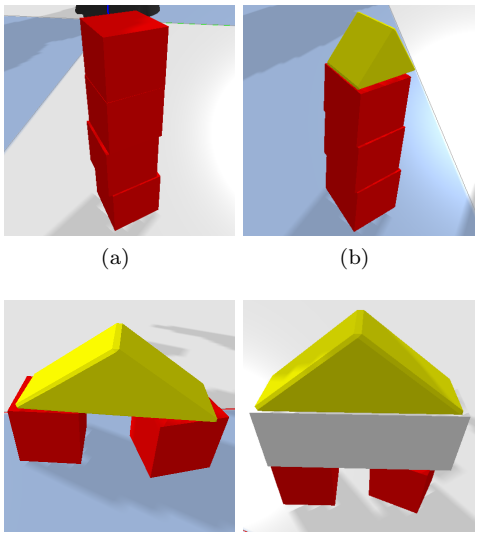
This chapter presents the evaluations of the experiments in this thesis. Section 4.1 describes the four simulated environments used in this thesis. Section 4.3 reports the experiments evaluating different algorithmic approaches. Section 4.4 evaluates the performance of the proposed network architecture compared with different baseline network architectures.

4.1 Environments

This work implemented four environments in Pybullet [3]. As is shown in Fig. 4.1, the *Cube Stacking* environment requires the agent to stack 4 cube blocks on top of each other. The *House Building 1* environment requires the agent to stack 3 cubes first and put a triangle roof on top of the cube stack. The *House Building 2* environment need the agent to put two cubes at the bottom and then place a long triangle roof on top. *House Building 3* requires two cubes at the bottom, a brick in the middle, and a long triangle roof on top.

4.2 Hardware Specifications

This work run experiments on workstations with three different hardware specifications: 1. NVIDIA RTX 2080 TI GPU with Intel CORE i9-9900k CPU; 2. NVIDIA Tesla K20 GPU with Intel Xeon E5-2650 CPU; 3. NVIDIA Tesla K40



(a) (b) (c) (d)

Figure 4.1: The four environments in this work. (a) Cube Stacking. (b) House Building 1. (c) House Building 2. (d) House Building 3.

GPU with Intel Xeon E5-2690 CPU.

4.3 Evaluating Different Approaches

4.3.1 Experimental Setup

In this experiment, this work evaluates the performance of different approaches in all four of the environments. For all experiments, this work runs five simulators in parallel to increase the speed for data gathering. This work train the models on PyTorch [15] using Adam [10] optimizer with a fixed learning rate of 0.5×10^{-5}. In Cube Stacking and House Building 1 environments, ϵ anneals from 0.5 to 0 in 10,000 episodes; in House Building 2 and House Building 3 environments, ϵ anneals from 0.5 to 0 in 20,000 episodes. The future discount factor γ is 0.5. The batch size for all DQN variations is 32, and 16 for other methods. The replay buffer has a size of 100,000 transitions.

This section evaluates the following approaches: 1. DQN; 2. DQN+Q^*; 3. DQN+guided; 4. DQN+Q^*+guided; 5. DAGGER; 6. ADET; 7. ADET+Q^*.

4.3.2 Result

The learning curves are shown in Fig 4.2. The x-axis is the number of episodes; the y-axis is the return per episode (because of the sparse reward shaping, it could also be interpreted into the task success rate). The results are averaged over five runs and plotted with a window of 1000 episodes. The black dot lines indicate the end of exploration.

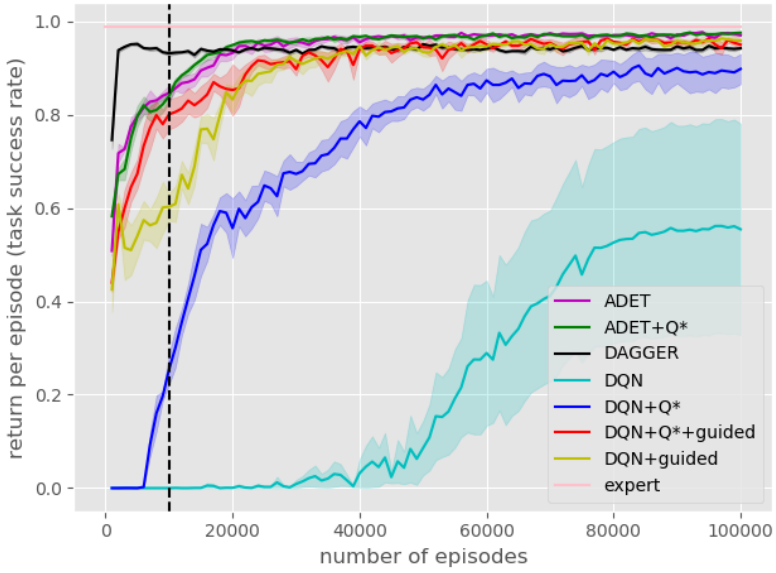
Vanilla DQN (cyan) only learns in Cube Stacking and House Building 2 environments. Those two environments are easier compared to the other two environments. In Cube Stacking, all blocks have the same shape, and the requirement for the sequence of actions is lower than other environments. The task can be accomplished as long as the agent can place a cube on the highest stack. In House Building 2, the minimal steps required for finishing the task is smaller than other environments (4 compared to 6).

DQN+Q^* (blue) learns a policy in three of the four environments. It fails to learn policy in House Building 3 due to the complexity of that environment. Comparing to vanilla DQN (cyan), using an optimal function to generate the update target fastens the speed of convergence and increases the performance in terms of average return. One outlier is the comparison between DQN+Q^* and DQN in House Building 2, where DQN+Q^* learns faster than DQN, but DQN ends up having a higher average return.

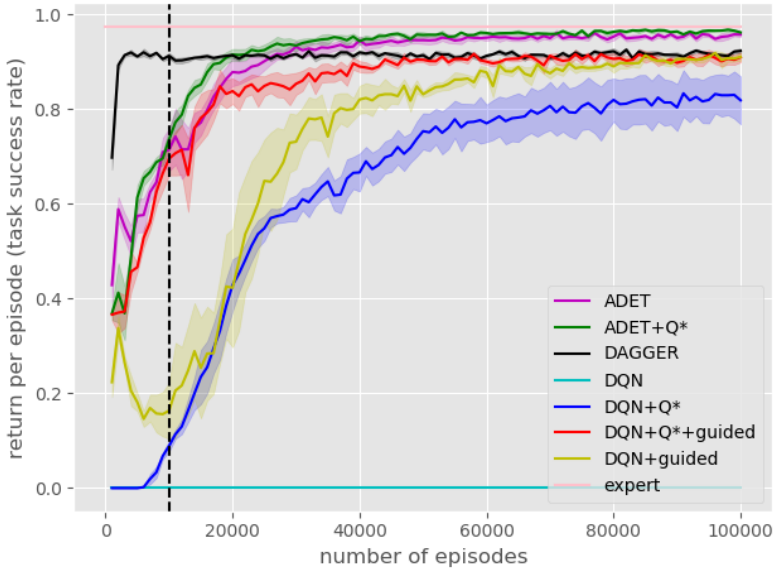
Expert guided DQN variations (yellow and red) always work better than the corresponding versions without the guidance (comparing yellow to cyan and red to blue). This indicates that the use of an expert policy for exploration can dramatically increase the efficiency of exploration.

DAGGER (black) has the fastest speed of convergence in three environments (Cube Stacking, House Building 1, House Building 3). ADET variations (purple and green) tend to be the best performing algorithm. ADET+Q^* (purple) works the best and approaches the expert (pink) in three out of the four environments, except for House Building 3, in which DAGGER (black) slightly outperforms ADET+Q^*.

In all value-based algorithms, the use of Q^* significantly improves the performance of all algorithms (comparing green over pink, blue over cyan, and red over yellow) in all environments except for DQN+Q^* versus DQN in House Building 2 described above.

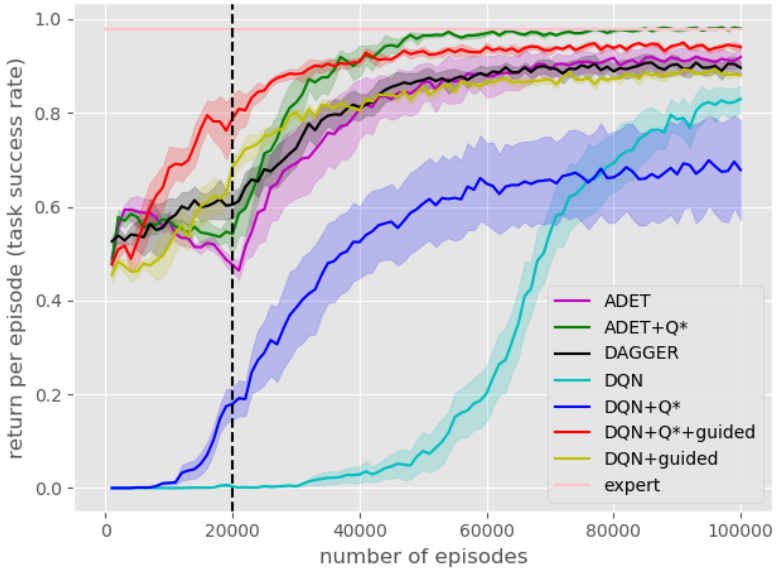


(a)

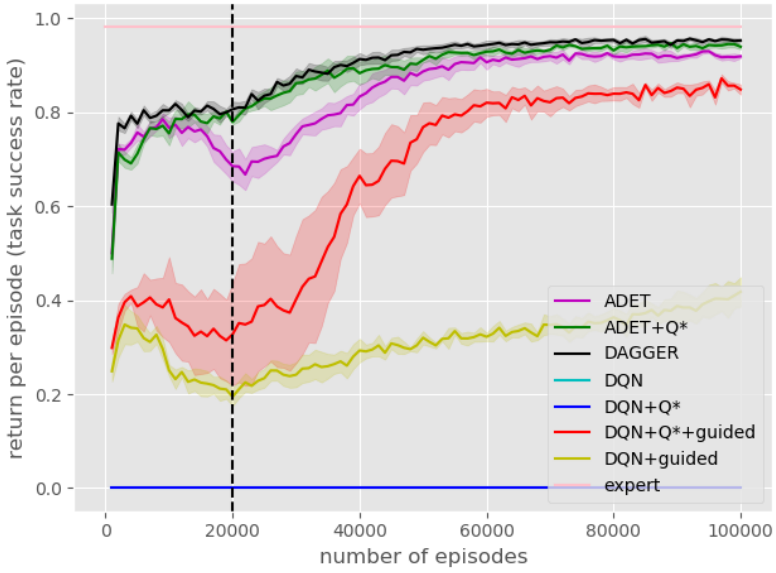


(b)

Figure 4.2: Learning curve of different approaches in different environments. (a): Cube Stacking. (b): House Building 1.



(c)



(d)

Figure 4.2: Learning curve of different approaches in different environments. (c) House Building 2. (d) House Building 3.

4.4 Evaluating Different Network Architectures

In this experiment, this thesis evaluates the performance of different network architectures.

4.4.1 Baseline Network Architectures

FCN with Expanded In Hand Feature Map

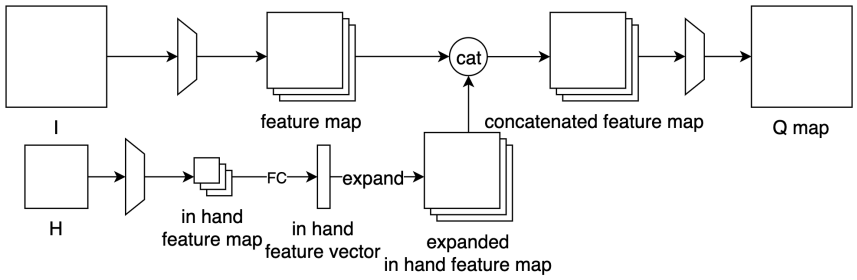
The first baseline architecture is Fully Convolutional Neural Network (FCN) with expanded in hand feature map (Fig. 4.3a). In this architecture, no dynamic filter or input pyramid is applied. The in-hand feature map is flattened into an in hand feature vector, and a fully connected layer with 16 output units compress it into a vector of 16. That vector is expanded into an expanded in hand feature map with size $16 \times 128 \times 128$ where each channel of the map contains the same value as the corresponding value in the in-hand feature vector. This expanded in hand feature map is concatenated with the scene feature map, and a convolutional layer generates the Q map.

FCN with Expanded In Hand and Global Feature Map

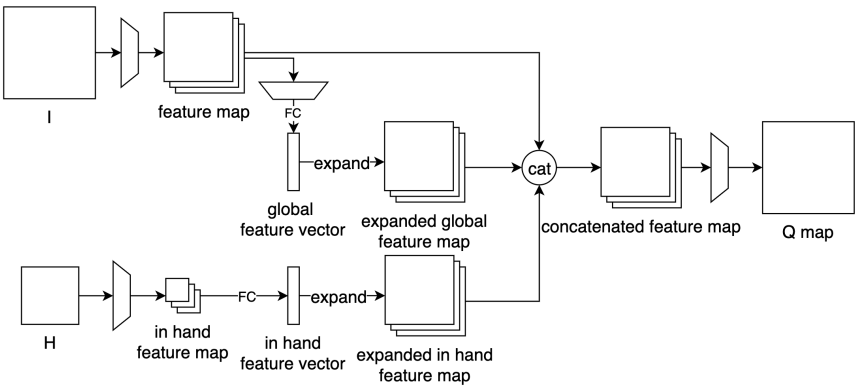
This baseline architecture (Fig. 4.3b) is similar to the one introduced in Section 4.4.1, but the scene feature map is also flattened and compressed into a global feature vector of 16, and expanded into an expanded global feature map with size $16 \times 128 \times 128$. The scene feature map, expanded in hand feature map, and the expanded global feature map are concatenated together, and a convolutional layer generates the Q map.

FCN with Dynamic Filter

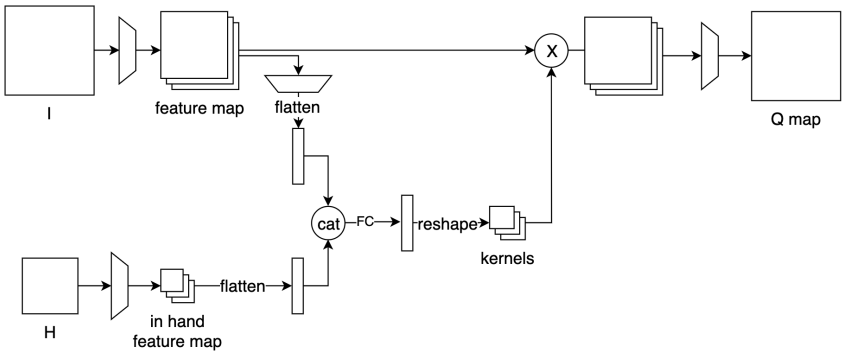
This baseline architecture (Fig. 4.3c) replaces the vector expanding being used in the previous two baseline architectures with a dynamic filter [8]. The dynamic filter is generated from a feature vector from the domain feature map and the in-hand feature map. This architecture is similar to the proposed architecture (Fig. 3.1), but without the input pyramid part with the low resolution I.



(a)



(b)



(c)

Figure 4.3: Three baseline network architectures. (a) FCN with expanded in hand feature map. (b) FCN with expanded in hand and global feature map. (c) FCN with Dynamic Filter

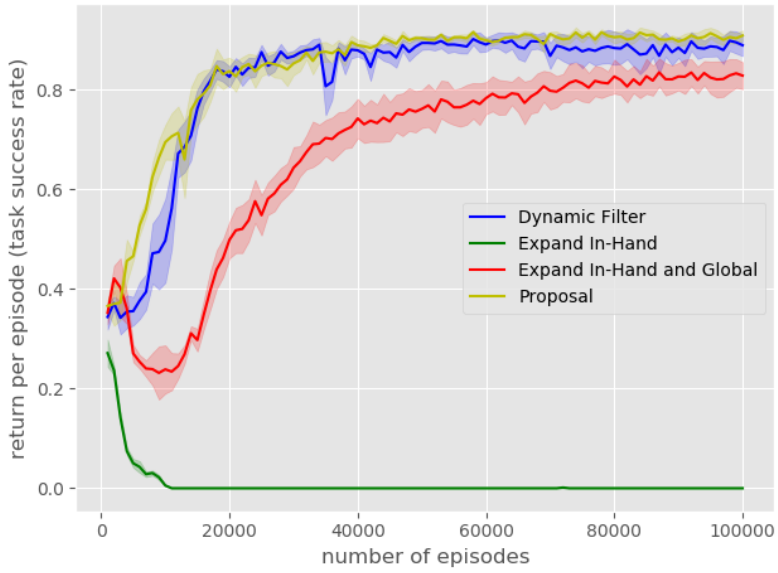
4.4.2 Result

The different architectures are evaluated using DQN+Q^*+guided algorithm in House Building 1 and House Building 3 environments. The resulting learning curves are presented in Fig. 4.4.

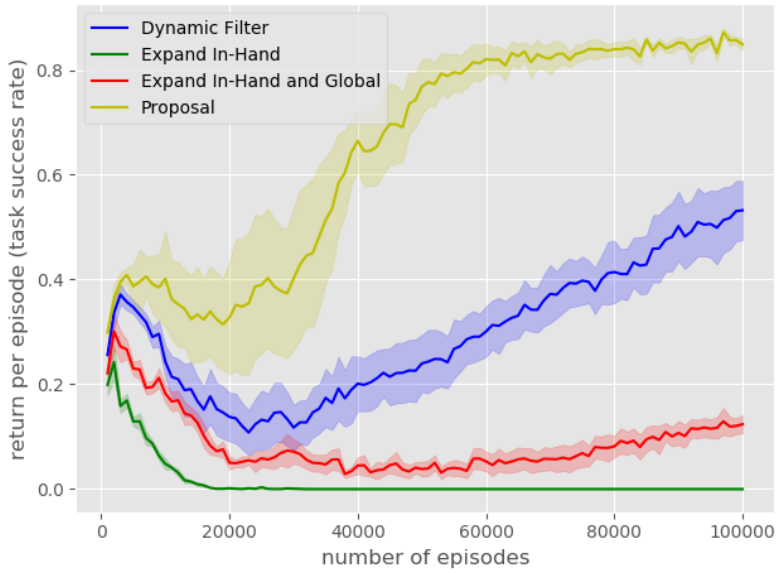
FCN with Expanded In Hand Feature Map (green) works the worst. The major drawback of this architecture is that it can not capture the global knowledge inside the scene since the convolutional filters can only observe the local area around each pixel, which makes it impossible for this network to generate correct Q values. FCN with Expanded In Hand and Global Feature Map (red) solves this by adding an expanded global feature map, but stacking those feature maps together is not an efficient way to solve this problem, because for an expanded feature map with a size of $C \times H \times W$, there are only C features repeated for $H \times W$ times.

Compared to expanding vectors to a feature map, the dynamic filter network (blue) can dynamically generate a convolutional filter with the feature vector from both the global feature of the scene and the in-hand feature from the in-hand image. This filter efficiently carries the necessary global knowledge and in-hand knowledge and applies them to the feature map through a convolutional computation.

In addition to the dynamic filter network, another component of the proposed architecture (yellow), input pyramid, enriches the capability of the model by applying a set of kernels to a low-resolution image. This is especially helpful in House Building 2 and House Building 3 because those environments require the agent to put a block nearby another one. To correctly predict the value of such action, the agent needs intermediate knowledge beyond the local knowledge provided by the static filters and the global knowledge provided by the dynamic filters.



(a)



(b)

Figure 4.4: Learning curve of different network architectures in two environments. (a): House Building 1. (b) House Building 3.

Chapter 5

Conclusion

This chapter summarizes and discusses the main findings of this thesis and outlines the direction of the future work.

5.1 Contribution

This thesis explores the use of imitation learning algorithms in the pixel-wise robotic end effector action parameterization paradigm. An optimal value function $V^*(s)$ and an expert policy $\pi^e(s)$ is implemented in the simulator to facilitate different imitation learning algorithms. Specifically, this work evaluates the following algorithms:

Algorithm	Requires $V^*(s)$	Requires $\pi^e(s)$
DQN		
DQN+Q^*	✓	
DQN+guided		✓
DQN+Q^*+guided	✓	✓
DAGGER		✓
ADET		✓
ADET+Q^*	✓	✓

Table 5.1: Different Algorithms and Their Requirements

The experimental results suggest that when an optimal value function $V^*(s)$ is available, DQN+Q^* can be applied to increase the performance of vanilla DQN. When an expert policy $\pi^*(s)$ is available, DAGGER and ADET can be applied, and they both have stable performance. When both $V^*(s)$ and $\pi^*(s)$ is available, ADET+Q^* tends to be the most reliable approach.

This thesis additionally proposes a network architecture for pixel-wise action parameterization based on Fully Convolutional Neural Network, Dynamic Filter Network, and Input Pyramid. The performance of this network architecture is evaluated comparing to three baseline architectures, and the experimental results show the advantage of this architecture.

5.2 Limitation and Future Work

The first limitation is that this thesis assumes unlimited access to an expert policy. Though the number of episodes for accessing expert policy is limited, within those episodes there is no limitation. However, it is not always possible to code an expert policy $\pi^e(s)$ providing unlimited access, especially in a real-world scenario. One solution is to pre-populate the replay buffer with some expert trajectories and disable the access of the expert policy in the rest of the training process.

The second limitation is the lack of real-world experiments. The evaluation of the thesis is purely in a simulator. The next step of this work will be evaluating the learned policy in real robots.

Additionally, there are some other potential directions for future work. First, DQfD [4] is another strong imitation learning algorithm, and it can potentially be adapted with this work. Second, other methods can be possibly applied to this work to further bootstrap the training process, including curriculum learning and hierarchical learning.

Bibliography

[1] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[2] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[3] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. *GitHub repository*, 2016.

[4] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. Deep q-learning from demonstrations. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[5] Ronald A Howard. Dynamic programming and markov processes. 1960.

[6] Peter J. Huber. Robust estimation of a location parameter. *The Annals of Mathematical Statistics*, 35(1):73–101, 1964.

[7] Stephen James, Andrew J Davison, and Edward Johns. Transferring end-to-end visuomotor control from simulation to real world for a multi-stage task. *arXiv preprint arXiv:1707.02267*, 2017.

[8] Xu Jia, Bert De Brabandere, Tinne Tuytelaars, and Luc V Gool. Dynamic filter networks. In *Advances in Neural Information Processing Systems*, pages 667–675, 2016.

[9] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, et al. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *arXiv preprint arXiv:1806.10293*, 2018.

[10] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[11] Aravind S Lakshminarayanan, Sherjil Ozair, and Yoshua Bengio. Reinforcement learning with few expert demonstrations. In *NIPS Workshop on Deep Learning for Action and Interaction*, volume 2016, 2016.

[12] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[13] 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.

[14] 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, 2015.

[15] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[16] Eric Rohmer, Surya PN Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326. IEEE, 2013.

[17] Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 661–668, 2010.

[18] Stephane Ross and J Andrew Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014.

[19] 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, 2011.

[20] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 2. MIT press Cambridge, 1998.

[21] Ulrich Viereck, Xingchao Peng, Kate Saenko, and Robert Platt. Adapting control policies from simulation to reality using a pairwise loss. *arXiv preprint arXiv:1807.10413*, 2018.

[22] Bohan Wu, Iretiayo Akinola, Jacob Varley, and Peter Allen. Mat: Multi-fingered adaptive tactile grasping via deep reinforcement learning, 2019.

[23] Tianfan Xue, Jiajun Wu, Katherine Bouman, and Bill Freeman. Visual dynamics: Probabilistic future frame synthesis via cross convolutional networks. In *Advances in neural information processing systems*, pages 91–99, 2016.

[24] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. *arXiv preprint arXiv:1903.11239*, 2019.

[25] Andy Zeng, Shuran Song, Stefan Welker, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Learning synergies between pushing and grasping with self-supervised deep reinforcement learning. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4238–4245. IEEE, 2018.

[26] Andy Zeng, Shuran Song, Kuan-Ting Yu, Elliott Donlon, Francois R Hogan, Maria Bauza, Daolin Ma, Orion Taylor, Melody Liu, Eudald Romo, et al. Robotic pick-and-place of novel objects in clutter with multi-affordance grasping and cross-domain image matching. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2018.