DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Branch Dueling Deep Q-Networks for Robotics Applications

Baris Yazici

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Branch Dueling Deep Q-Networks for Robotics Applications

Branchenduell tiefe Q-Netzwerke für Robotikanwendungen

Author:	Baris Yazici
Supervisor:	Prof. Dr. Alois Knoll
Advisor:	Msc. Mahmoud Akl
Submission Date:	06.09.2020

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 06.09.2020 Baris Yazici

Acknowledgments

I want to thank:

- Mahmoud Akl for allowing me to pursue my thesis in this area, and for his wise advice and constructive suggestions.
- Michel Breyer for sharing his precious insights in robotics area.
- My mom, father, and sister for their constant love and support throughout my master's degree

Abstract

Grasping of an object is the most integral behavior for robotics manipulators. Every manipulation task starts with a grasping procedure. A firm grasp will be a must for service and industrial robots of the future. The only way we can achieve a trustworthy grasp and manipulation is to let the robotic manipulators learn by themselves. To this end, we successfully delivered a learning-based grasping policy by combining curriculum learning with action-branching DQN in the reinforcement learning framework. Moreover, the robustness of the trained models is tested in new scenes and with novel objects. We evaluated three algorithms: BDQ, DQN, and SAC. SAC delivered a robust grasp policy only after 2 hours of training in simulation. This model performed 100% on the test data, and 97% on the wooden blocks set, an entirely new set of objects.

Contents

1 Introduction

1.1 Objectives

Manipulation of objects is one of the most natural action of humankind. Humans never stood and planned about how to manipulate an item. It is an instinct for humans to grab objects in specific ways. Even an infant human can easily manipulate different shapes and colors of objects. Manipulation helps us to use tools, gadgets and provide services. From rehabilitation to service robots, tool usage is vital to enable them to achieve their objective. Therefore, manipulation skills will be central for robots of any kind. Example of our daily manipulation tasks are shown in figure 1.1.

For a complicated manipulation task, first of all, grasping the object is essential. We need first firmly to grab a water bottle to drink it. Without a firm grasp, the manipulation process may not be successful. Nowadays, every kind of robot is helping us produce in the industry. The task robots are responsible for expands from the car manufacturing to high precision microchip producing. On all those tasks, robots are incredibly efficient and precise. However, they are repetitive and specialized in one process. If a car company decides to manufacture a new car model, the whole process may need to be changed, and the robot's program needs to be hardcoded from zero. Eventually, we want to avoid this redundant task of programming all from zero each time we change the process.

Our main objective is to enable robotic manipulators to incrementally learn to grasp tools and generalize the learned policy to unseen objects. This procedure promises a robust gripper that can adapt well to unseen objects and environments. Moreover, rather than hardcoding robot's every move, we take another approach, that the robot learns by itself based on the observation and rewards it receives from the simulation environment. We foresee that through learning-based methods, robots will learn to extrapolate their knowledge to unknown objects and processes 1.2.

1.2 Contribution

1. **Testing the BDQ algorithm in grasping setup** - Recently developed BDQ algorithm promises to solve DQN's intractability problem. The novel action branching architecture proved to learn complex humanoid walking, hopper, and walker 2d tasks [2]. Although these tasks are complex and have high action-space dimensionality, they are similar by nature. Our grasping gym benchmark environment demands a novel exploration strategy and poses new challenges that are entirely

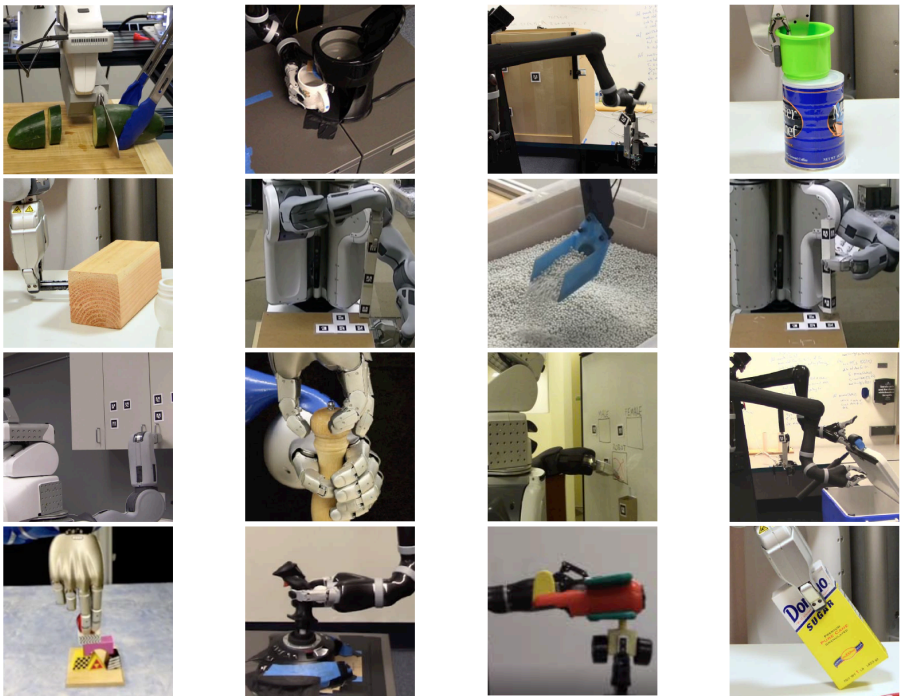


Figure 1.1: Different manipulation skill adopted to robotic manipulators [1]

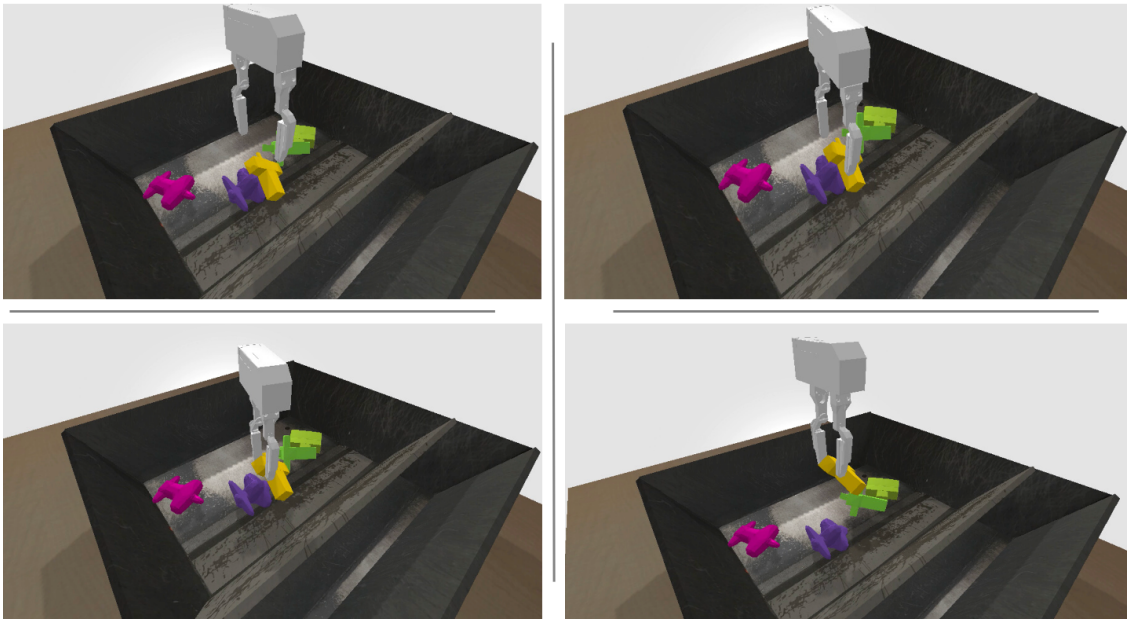


Figure 1.2: Our robot model performs grasping in table scene

different from walking tasks. In this respect, our work provides the first test of the BDQ algorithm in robotic grasping operation.

2. **Grasping Gym benchmark tasks** - Default gym benchmark tasks do not cover the whole spectrum of robotic manipulation. For instance, OpenAI gym provides pick and place, push, reach, and manipulate block tasks; all goal-based [3]. Those tasks aim to solve the problem only by reaching the goal position. In contrast, our gym benchmark environment targets to provide the most generalized grasp policy. We aim not only the best performing model in the current model but the best model to generalize to new objects and scenes. That is why we implemented two scenes: Floor and table. We conduct our training in the floor scene, and in the table scene, we test the robustness of the trained model. We also use an entirely new wooden blocks dataset from Breyer et al.'s work [4]. Indeed, tests with wooden blocks proved to be the toughest of all.

3. **Raw depth perception** - Perception is the most crucial input for an RL agent. We compared the agent's performance with three different observation types: Breyer et al. 's autoencoder implementation as a baseline, novel raw depth, and RGBD perception pipeline. We compare these two new observation types against autoencoder implementation from Breyer et al. We provide a straightforward interface to change between different observation types.

4. **BDQ implementation based on Stable-Baselines** - The original BDQ algorithm is implemented based on OpenAI baselines. Currently, Stable-baselines, a fork of OpenAI baselines, presents more features and support for users. Whenever a user files an issue, they answer it less than a day. Apart from the support, their codebase is simpler to understand and debug [5]. Thus, as soon as, we encountered problems with the original BDQ code, we began the reimplementation based on stable-baselines. Our repository is hosted at Github [1] as an open-source MIT licensed project.

5. **State-of-art grasping score with SAC algorithm** - Although our primary goal is to test the BDQ's performance on high dimensional action-spaces, we observed that SAC performs nearly perfectly in the grasping environment. SAC model surpassed all previously tested algorithms in the grasping research area with 100% test set performance

6. **Minimized the allocated GPU memory** - Smart allocation of GPU memory is a must in the RL setting. Naïve approach of allocating full GPU memory can lead to memory issues and low utilization. We noticed that problem early on with the Breyer et al.'s encoder implementation. The encoder was reserving half of the GPU memory and not utilizing them all. The implementation was complicated because of the disconnection between the Keras interface and Tensorflow. While

[1] `https://bit.ly/31TMzst`

the encoder uses Keras high-level interface of Tensorflow, Stable-baselines algorithms were implemented purely based on Tensorflow. Therefore, the interaction between those applications was broken. We combined them in the same Tensorflow compute graph. Thus, the encoder did not attempt to allocate GPU memory on its own but uses the same memory reserved for the Stable-baselines algorithm. This bug fix led us to utilize more GPU memory, and naturally, more experiments can be run in parallel.

7. **Ablation studies** - We conducted several ablation experiments to check the relative importance of the modules we used. These experiments cover the absence of actuator width information as observation, the lack of input and reward normalization, and the nonexistence of shaped reward function. We suppose the results show the most critical module for the RL algorithm performing on grasping tasks.

8. **The distinction of validation set** - Before our work, the object dataset was only divided into training and test set. The best performing model was evaluated based on the test set's performance, introducing a biased in the selection procedure. The test set should never be touched during the training procedure. Therefore, we introduced the validation set; during training, we test the agent's performance against the validation set periodically. Through this fix, we achieved a more generalizing and robust model.

1.3 Challenges

1. **Reproducibility** - Reproducibility is a known issue on machine learning-based approaches. They are never guaranteed to find the global optimum. They are even never guaranteed to find the same local optimum over different trials. Our grasping simulator is random by nature. In our stochastic environment, reproducibility issues can be overwhelming. We cannot verify the hyperparameters' relative value if the performance changes tremendously over different trials. The situation is similar in the low-level. Stable baselines documentation[2] mention that they cannot even assure the same performance in CPU if the model is trained in GPU. Therefore, we needed to spend more time getting the mean value by running the same model for multiple instances.

2. **Hyperparameter Optimization** - Similar to the reproducibility issue, hyperparameter optimization is a very well-known issue in machine learning. No theoretical proofs are saying how hyperparameters need to be tuned. The only solution is to assign the parameters based on some heuristics intuitively. In our work, we used a hyperparameter optimization library to handle this problem. Some trials took up to 80 trials over a week to deliver the most optimized parameters.

[2] `https://bit.ly/3jwxhQH`

3. **GPU memory optimization** - GPU memory is the most valuable asset for deep learning applications. If one utilizes the GPU efficiently, one can run more experiments in parallel. Thus, we separated most of our time to fix the encoder or algorithm memory leaks. These leaks clog the GPU memory and lead to unexpected training run shutdowns. That is why we put a lot of effort to minimize the GPU memory usage and maximize memory utilization.

4. **Keras and Tensorflow mismatch** - Keras is a high-level, user-friendly interface built on Tensorflow backend. Using both Keras and Tensorflow leads to complicated scenarios, where computation graphs variables can be mixed up. We encountered that the original BDQ algorithm attempted to save the Keras variables used for the autoencoder implementation with the actual Tensorflow computation graph variables. The overall situation caused a malfunction in the trained model's save and load functionality. We solved the issue by adequately namespacing every graph variable and tediously saving them in the BDQ algorithm's save method.

2 Background

2.1 Related Works

Robot manipulation and grasping is an intensively researched area. The wide span of the subject makes it difficult to categorize all different approaches under one umbrella. Nevertheless, analytical and empirical grasping approaches tend to be the most used categorization in the literature [6].

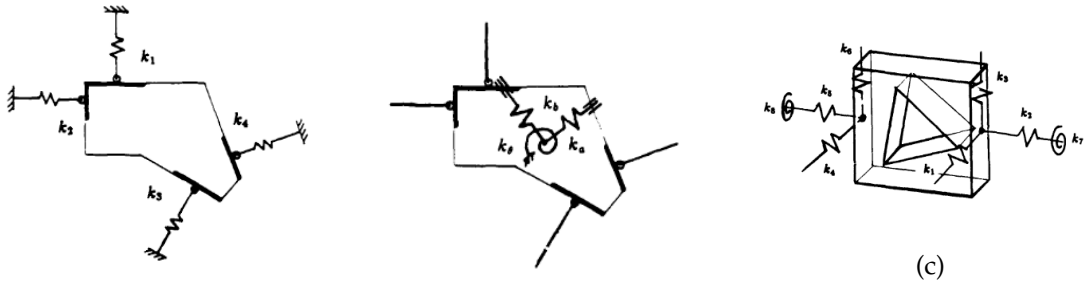
2.1.1 Analytical Grasping Approach

The analytical approach has been introduced first by Nguyen. He tried to solve the grasping problem by defining analytical objectives, such as a successful grasp should achieve force closure and satisfy the stability conditions by closing the object from each direction [7]. Nguyen modeled the hand and the object, to be grasped, to compute the stable force closure grasp analytically. This kind of analytical technique comes with an exhaustive computational burden. Although the proposed algorithm works fast and correct on planar objects, they do not scale to household objects such as mugs or bottles [6]. Moreover, in most robotics setups, full geometrical models of the objects are not available [8].

2.1.2 Empirical Grasping Approach

The empirical grasping approach was introduced to overcome the difficulties faced with the analytical grasping approach. Empirical methods include learning, which is based on sampling and training. Some examples of empirical grasping approaches are imitating a human teacher [9], learning from the handcrafted features [10], and deep reinforcement learning (RL) technique to learn close-loop dynamic visual grasping strategies [11]. Imitating a human teacher can quickly learn the demonstrated training data. However, it has difficulties to scale to the novel objects, which were not seen during the training [6]. Saxena et al.'s technique to learn from the feature effectively generalizes to unseen objects, but it fails to choose the best grasp for a particular task [6].

RL approaches learn a thorough grasp of an object based on a definition of a goal. Thus, the learned grasp of an object is already linked to the task's goal. The flexibility of RL made itself attractive among grasping researchers. One disadvantage of the RL approach is that it requires a considerable amount of data to train. Although recent applications introduced randomization for sim-to-real transfer of the models [12], it still has difficulties to perform reliably in the real-world [13].



(a) (b) (c)

Figure 2.1: Stable force closure examples are represented as virtual springs. Nguyen proved that all force closures could be modified to be a stable grasp candidate[7].

2.2 Learning applications on grasping

Having experts implement handcrafted features and hand-coded controllers is time-consuming and lack generalization. Among all other approaches, machine learning applications on grasping have proven to have a high success rate on novel objects.

2.2.1 Deep Learning for Detecting Grasps candidates

Lenz et al. propose a five parameters model trained by supervised learning from the candidate grasps database. They draw a rectangle box representing the best grasp location with the five-dimensional parameter model, x, y coordinates, width, height, and orientation.

Their cascaded network architecture bypasses the need for handcrafted features. After training, the first layer network learns low-level features and delivers possible naïve grasping candidates, and the second layer chooses the top-ranked grasping candidate 2.2.

Although they achieved up to 90% success rate on grasping lab tools, this method is limited to only parallel plate gripper. It does not generalize to use other gripper types, unless the dataset is entirely updated for these grippers. Besides data collection process is time-consuming and biased.

2.2.2 Dex-net

Another approach for grasping is to learn the grasp robustness factor. Mahlet et al. trained convolutional neural networks to learn the grasps robustness function from 6.7 million point cloud data. They collected the data similarly to Lenz et al. with an analytical grasp metric to evaluate the grasp's quality.

With 98% success rate, their approach indeed proved to be robust. However, the

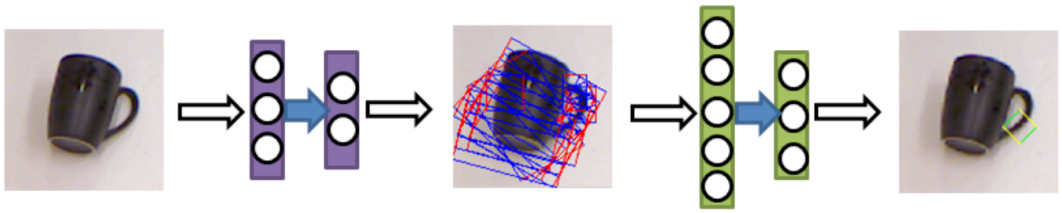


Figure 2.2: Red-blue rectangles represent possible grasp candidate. Green rectangle is the top-ranked grasp rectangle [14]

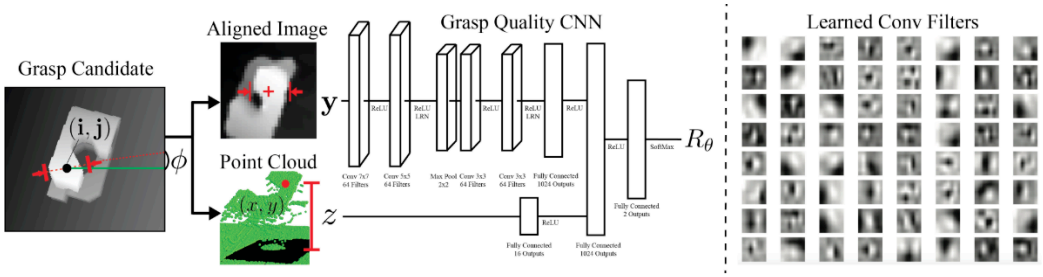


Figure 2.3: Dex-net architecture [14]

designed network only outputs the grasp location, thus a grasp planning algorithm is needed to complete the grasping process. Moreover, data collection can be tedious due to large number of machine-labeled dataset.

2.2.3 Deep Reinforcement Learning for Vision-Based Robotic Grasping: A Simulated Comparative Evaluation of Off-Policy Methods

The sequential decision-making process is inherent in all grasping tasks. This property of grasping motivates us to model it with a reinforcement learning framework. Unlike other works, this approach trains end-to-end policies to automatically learn to grasp without any prior knowledge about the environment or the gripper model. Moreover, their system's end-to-end nature eliminates the need for additional grasp planner, which was a default in previous works. Quillen et al. showed that off-policy RL can increase the robustness of grasping models. Based on their investigation, corrected Monte Carlo and Deep-Q-Learning outperformed the earlier supervised learning approach. They observe that model-free RL approaches allow pre-grasp manipulation to increase the chances of grasp in future actions.

They designed one complete network to concatenate perception input and actions from two different branches 2.6. Although the number of parameters increases drastically with convolutional layers from the perception branch, it makes the system more compact and easier to understand.

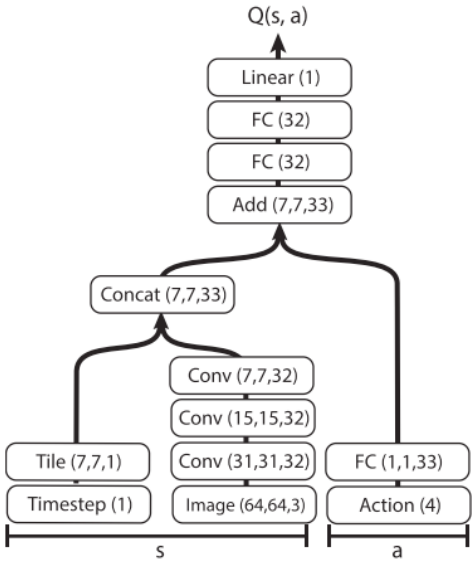


Figure 2.4: Deep Learning network architecture of [15]

2.2.4 Comparing Task Simplifications to Learn Closed-Loop Object Picking Using Deep Reinforcement Learning

Breyer et al.'s learning setup is similar to the Quillen et al. with one robot gripper and randomly spawned objects in the simulation. One significant difference regarding the learning setup is that Breyer et al. generates the robot gripper only until the wrist without the rest of the arm. Thus they do not need to calculate the inverse kinematics equation.

Breyer et al. enhanced the RL-based data-driven grasp approach with curriculum learning and autoencoder for perception. Thanks to the curriculum learning setup, which increases task difficulty based on the agent's current performance, they shortened the agent's time to explore the environment. In other words, curriculum learning guides the agent throughout its learning phase. Besides, with autoencoder in the perception layer, their network tends to spend less time comprehending the observation compared to the Quillen et al. 's approach.

Unlike Quillen et al., they experimented with an on-policy RL algorithm, TRPO. Policy-based algorithms tend to explore better than epsilon greedy based exploration. This property may also contribute to the higher success rate and faster convergence. After training grasping models in simulation, they tested them on a real robot. Although they did not use domain randomization to minimize the reality gap, they achieved up to 78% success rate on real hardware.

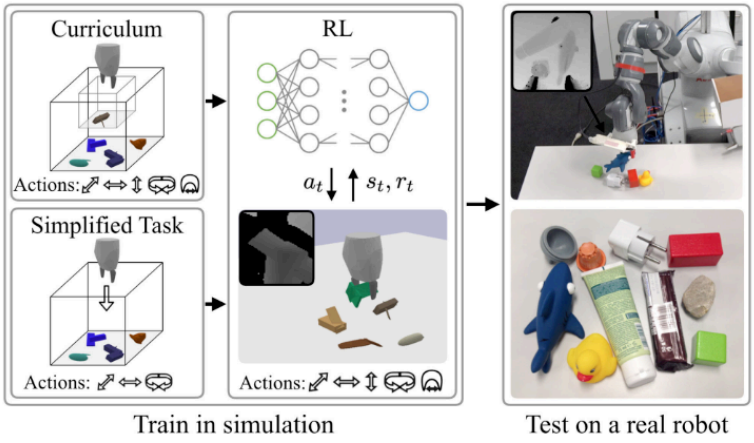


Figure 2.5: Breyer et al. [4]

2.2.5 Solving Rubik's Cube With a Robot Hand

OpenAI et al. present a novel approach for in-hand manipulation problems. Their system attacks Sim2Real discrepancy between simulated models performing on real robots.

Training the Deep RL models on simulation is becoming more and more common. The popularity of simulation increases the demand for enhanced Sim2Real model transfer algorithms. Domain randomization has shown great success in bridging the gap between reality and simulation [16]. More researchers implement machine learning methods that randomize the environment and gripper material parameters. Through this approach, trained models can achieve higher generalization property with robustness to increasing noise at the sensor or environment settings. OpenAI proposes a Meta-Learning method that involves memory like recurrent neural networks to learn the environment's underlying dynamics. Meta-learning and automated domain randomization algorithm combined results in an enhanced model transfer to real-world robots.

2.3 Reinforcement Learning

The simplest examples of learning come from our own lives; we learn to walk, speak the language, or cook. All those activities span our entire life, it influences who we are and the decisions we take in life. We know that living animals such as mammals learn from their social and asocial interactions with the environment [18]. Although we have not yet developed a full-scale theory of animal learning, we have developed computational objectives for machine learning [19]. This computational approach falls into three categories as supervised, unsupervised, and reinforcement learning. In this chapter, we will consider the Reinforcement Learning objectives and problem formulation.

Reinforcement learning provides a systematic approach to maximize the reward by

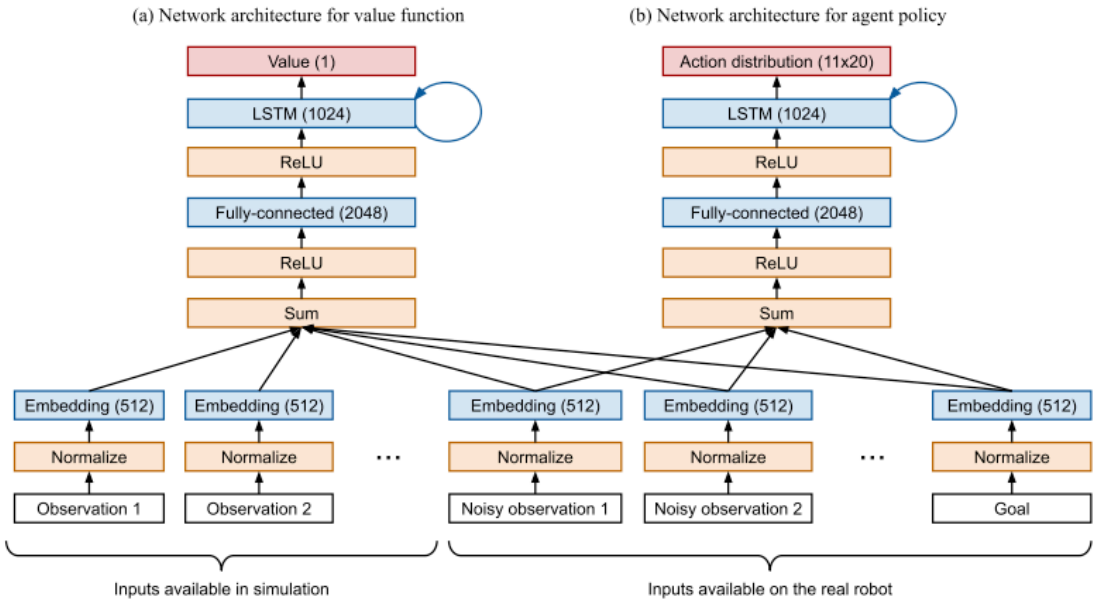


Figure 2.6: OpenAI policy and value network architecture [17]

linking observations to actions. A reinforcement learning agent creates its data by interacting with the environment. Therefore, it is fundamentally different from supervised and unsupervised learning, where the data is already provided [19]. Another critical difference is the inherent goal-oriented approach. The reinforcement learning agent maximizes the rewards to achieve its goal. Whereas, other machine learning approaches lack this goal [19]. For instance, supervised learned software recognizing faces can be used for security reasons to detect criminals or can be easily used to unlock phones. However, a reinforcement learning agent trained to drive a car autonomously can only drive a car. In a sense, reinforcement learning provides us end-to-end learning.

A core feature of reinforcement learning is that it acts on uncertain environments and, in return, receives the observation and reward. Fundamentally, a learning agent collects this experience and tunes its action to increase the expected reward. The expected reward term refers to the end of the horizon. For example, a chess-playing agent can choose to sacrifice the queen in the next move for a checkmate in the move after. In this case, the reward would decrease when the agent loses a queen, but the agent's goal will be satisfied by terminating the game. For a well-defined reinforcement learning system, we can speak of four main components: Policy to decide the actions, a reward to maximize the expected reward in the horizon, and a model of the environment, describing which directions the chess pieces can move. The components of reinforcement learning are formulated based on Markov Decision Process. In the MDP chapter, we will detailly explain RL components. We will explain first Markov Chains, then MDP, in the next chapters [20].

2.3.1 Markov Chain

Markov chain has a strong correlation with the weak law of large numbers. This law has a tremendous significance on stochastic modeling. It states that the average of a large number of experimentations converges to the real value of the probability of a particular task. For example, if one tosses infinite amounts of the coin, the average number of heads should converge to 0.5 [21]. Bernoulli's weak law of large numbers only covers independent events. Markov proved that Bernoulli's law also holds on dependent cases [21]. As the law of large numbers suggests, if one conducts a large number of iterations on a problem, one can infer the transition matrix. This matrix proves to be the only information one needs to compute the next state.

This characteristic defines the famous Markov property; the current state captures all the necessary information to predict the next state. If we extrapolate this example to slightly complex systems, such as weather forecast, we need today's weather report to predict tomorrow's forecast, assuming that we know the transition matrix. Naturally, one can formulate other events with Markov Chain lawnmower, and a random walk is the straightforward examples in the literature[21].

It is also possible to attach rewards to Markov Chain's formulation. In figure 2.7, the transition between states is represented with the arcs. Moreover, each transition has a probability similar to the transition matrix; we defined before. Each state has an immediate reward and a value function. The immediate reward is received directly when the actor moves to that state. The value of a state represents how likely the future actor will end up collecting high rewards.

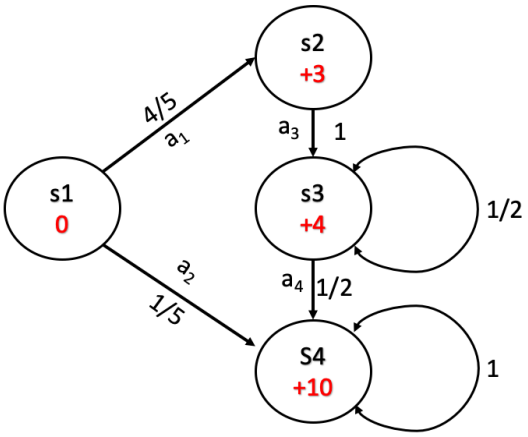


Figure 2.7: Markov chain with transition probabilties and rewards

The value of a state will be later significant to solve Reinforcement Learning problems through Value Iteration methods. They are one of the essential algorithms that led to the initial success of reinforcement learning research.

2.3.2 Markov Decision Process

Markov decision process is a slightly advanced version of the Markov Chain. It includes action on top of the Markov Chain. Reinforcement learning problems are formalized as a Markov Decision Process rather than Markov Chain because RL agents are free to choose from different actions.

MDPs first came into play as part of the optimal control problem by Bellman [22]. Bellman applied dynamic programming methods to solve the MDP problem optimally. However, this methodology was not scalable to larger problems stemming from the curse of dimensionality problem [19].

The fundamental elements of MDP are as follows:

- Agent: The actor takes an action in the environment to learn.
- Environment: The agent interacts with the Environment (Plant).
- Rewards: Environment returns rewards based on the interaction made by the actor.
- State: View of the environment from the eyes of the actor.

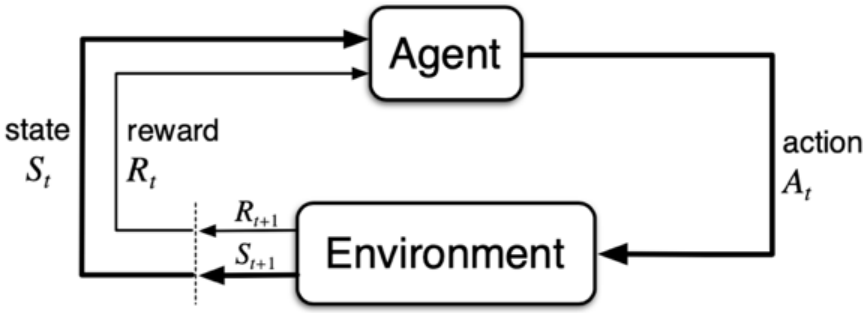


Figure 2.8: MDP structure

The action process follows; the agent acts on the environment at time t, and the environment returns the reward and state of the action at time $t+1$. Based on the state of the environment at the $t+1$ agent makes another action A_{t+1}, which results in R_{t+2} and S_{t+2}.

In every MDP system, agents should be reachable to every other states through a sequence of actions. The transition between states is of great value for the MDP framework. One can compute the transition probability in MDP, given the inner dynamics of the environment [19]. The below equation defines the internal dynamics probability. It tells how probable it is to end up in state s' with reward r by taking action a in state s.

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = d, A_{t-1} = a\} \tag{2.1}$$

One can calculate the transition probability function from the inner dynamics' probability function.

$$p(s'|s,a) = Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in R} p(s', r|s, a) \tag{2.2}$$

Next chapter, we will dive into the definition of the reward and value function. Based on those concepts, we will build the logic on how to solve MDPs optimally.

2.3.3 Reward and Value function

As defined in the Markov Chain section, rewards and value functions are the essence of value iteration algorithms. In this section, we will describe the objective of the RL problem formally. As we mentioned in previous chapters, we want to increase the sum of rewards we achieve at the terminal state. If we define the reward at the final state T as R_T and the reward at the initial state t is R_t, the returns we receive is the sum of rewards in every state.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.3}$$

G_t is the notation of expected return. We will mostly use the discounted version of the expected return calculation. We introduce a discount factor γ to value the rewards that the agent receives now, compare to the rewards in the future.

$$G_t = R_{t+1} + \gamma^1 * R_{t+2} + \gamma^2 * R_{t+3} + ... + \gamma^T - 1 * R_T = \sum_{k=0} \gamma^k R_{t+k+1} \tag{2.4}$$

For instance, if a rational human offered to choose between one million Euros now, versus one million Euros in 50 years, would usually choose one million Euros now. Therefore, our agent also weighs the rewards it receives now, over 50 steps from the current state. In the meantime, we do not want the agent to undervalue the importance of reaching the terminal state through the highest reward sequence. Given the below example (2.9), if we introduce a discount factor of 0, the agent will always try to maximize the immediate reward and take a sequence of s1-a2-s3-a4-s4 and end up with a non-optimal greedy algorithm with $G_t = 20$. But with a discount factor, in this example everything between $0 < \gamma < 1$ works, would find the optimal sequence s1-a1-s2-a3-s4 with $G_t = 25$.

The value function is the expectation of returns, while an agent follows the policy (π). The policy represents the probability distribution of an agent taking action a in state s. The policy is similar to the transition probability matrix in the Markov Chain section. Using policy, we can define the value function of an agent following the policy π.

2.3.4 Q-Learning

Q-learning algorithms have been the core of the RL research for almost 20 years. It combines the idea of TD learning in approximating action-value function. Through this

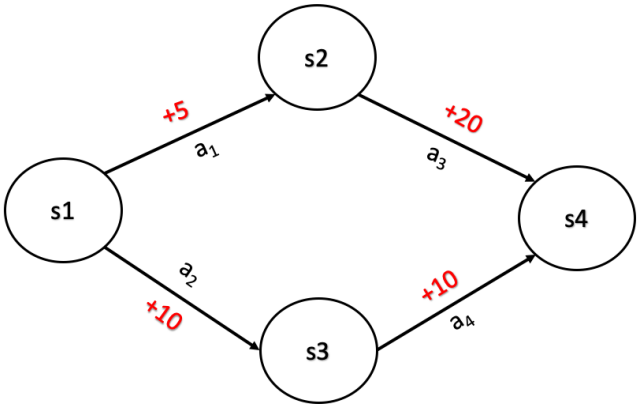


Figure 2.9: Zero discount factor leads to non-optimal solution with s1-a2-s3-a4-s4. Discount factor greater than zero leads finds the optimal solution of s1-a1-s2-a3-s4

approach, the computation converges faster than state-value approximation algorithms. Another strength of the Q-learning algorithm is its off-policy nature. The Q-learning agent can learn from the experiences of different policies. The off-policy nature makes the agent more data-efficient.

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))] \tag{2.5}$$

2.4 Function Approximation

As mentioned before, it is often impossible to find either the optimal policy or the optimal value function due to computational resources. It is often enough to find approximate solutions. Various function approximators are available for us to use. The main categorization follows as; linear and nonlinear function approximators. Their working principle is similar; they both follow a specific policy and sampling process. Based on the experiences, they approximate either policy or the value function.

The application of neural networks provided RL methods a boost. Since the introduction of function approximators, the success of RL applications has increased significantly. Although function approximation tools bring convergency issues in some corner cases, it shows excellent success practically [19].

The notation $V(s)$ changes to $V(s, w)$, if we define value function with weight parameterization. To find suitable weight parameters that represent the value function as general as possible, one needs to update parameters w at each step towards the smallest loss region based on a particular objective description. Firstly, we need to define an objective function and then find a method to tweak our weight parameters towards the objective gradually. There are a couple of possible objectives to learn in the literature; the state-value function, action-value function, or directly the policy. In the case of the

state value function, one can choose the objective function as the mean squared value loss between the estimated state-value function and the target state-value function (2.6). While for policy approximation, expected return represents the objective function 2.9.

Target state-value function can be the Monte-Carlo or TD target. Monte-Carlo's target represents the expected return (G_t), and TD target estimates the bootstrapped state-value function. The main difference between those targets is that Monte Carlo ensured to converge to a global optimum. In contrast, the bootstrapped TD target converges only to a local optimum near the global optimum.

$$VE(w) = \sum_{s \in S} \mu(s)[v_\pi(s) - \hat{v}(s, w)]^2 \tag{2.6}$$

Secondly, we need to optimize for the objective function. Optimization is handled by the stochastic gradient descent method, it is the most used solution to improve the weight parameters based on a defined loss function. 2.7 shows one gradient update step of the stochastic gradient descent method on the weight vector.

$$w_{t+1} = w_t - \frac{1}{2}\alpha \nabla \Big[v_\pi(s) - \hat{v}(s, w)\Big]^2 \tag{2.7}$$

The optimal weight vector needs to find the right balance between strongly representing one state and generalizing to similar unseen states. As a result, our approximator needs to avoid either overfitting or underfitting.

One can also approximate action-value function in the same way as state-value function. The gradient update step of the action-value function is similar to the state-value function as in 2.8

$$w_{t+1} = w_t - \alpha \Big[U_t - \hat{q}(S_t, A_t, w_t)\Big] \nabla \hat{q}(S_t, A_t, w_t) \tag{2.8}$$

U_t can be either TD or Monte Carlo target.

So far, we have only considered the function approximation on the action and state functions. Another promising and popular technique is to approximate the policy function directly. Policy function can lead us to optimal solution in a more direct way than action and state values. Since it directly aims to solve the problem at hand, to find the optimal policy [23]. The objective function of policy gradient methods is simply the expectation of total return under policy π.

$$J(\theta) = E\Big[\sum_{t=0}^{H} R(s_t, u_t)\Big] \tag{2.9}$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{2.10}$$

2.4.1 Neural Networks

Neural networks are the unique cases of function approximators. Although they follow the same procedure as SGD methods, their weight matrix is nonlinearly related to the

approximated $V(S_t, w_{a_t})$. Nonlinear relation brings convergency issues and problems with instabilities. However, the practical success of these kinds of methods shadows the weak theoretical basis.

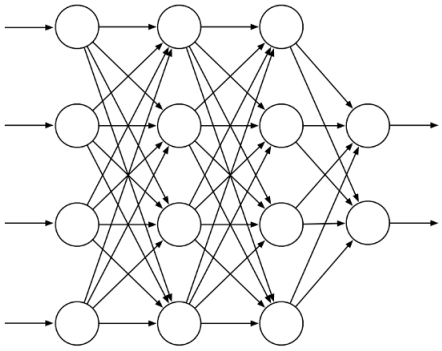


Figure 2.10: Generic neural network with two hidden layers

A generic neural network in 2.10 comprises layers of artificial neurons connected with weight, bias vectors, and activation functions to each other. Each activation function feeds nonlinearity to the general equation.

2.5 Value-Based Reinforcement Learning

Approximation of action-value or state-value function methods falls into the category of value-based RL. These methods are model-free, meaning that they use sampling to solve the optimal solution for the problem [24]. Simultaneously, they are off-policy algorithms, which makes them data efficient but high variance algorithms [23]. These methods only indirectly optimize the policy. Hence they are unstable during the learning phase [19]. Value function-based approaches can overestimate the policy's selected actions, which is a typical problem with Q-learning-based algorithms.

2.5.1 DQN

DQN is the neural network extension of the Q-learning algorithm. It has shown that RL can handle high dimensional nonlinear control problems. DQN optimizes Bellman error by parameterizing the action-value function. Some extensions of DQN introduced experience replay buffer to cope with the high variance problem [25]. [24] inputs randomly collected mini-batches of experiences to the neural network consisting of multiple layers. Thanks to their neural network structure, they overcome the convergency issues.

$$L(\theta) = E_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-)) - Q(s, a; \theta)\right)^2\right] \tag{2.11}$$

2.6 Policy-Based RL

Policy-based solutions directly parameterize the policy to find the optimal policy which returns the highest reward. These approaches proved to be more stable than Value-based counterparts. One reason behind the stability is that actions are sampled from the continuous policy parameterization function, which outputs smooth action probabilities [19]. Similar to Value-based methods, policy-based RL also aims to solve the RL problem only with sampling in a model-free fashion.

2.6.1 Policy Gradient

Policy gradient methods incorporate stochastic gradient ascent on policy objective function. This update optimizes the θ parameters of the policy. Unlike DQN, policy gradient methods perform each update in on-policy fashion. It uses only the experiences taken under the particular policy to update the parameters of this policy [23].

$$\nabla J(\theta) \propto \sum_{s} \mu(s) \sum_{a} q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{2.12}$$

$$= E_\pi \Big[\sum_{a} q_\pi(S_t,a) \nabla \pi(a|S_t,\theta) \Big] \tag{2.13}$$

3 Simulator Choice

Simulator choice was the first important architecture decision we made because it influences the whole software architecture. Simulator lays the groundwork of the robotics environment implementation. Especially for continuous control optimization tasks, accurate simulation plays a significant role in the controller's performance. As Todorov et al. indicated, if the simulation is not accurate enough, an optimization algorithm will find a way to exploit it [26]. We compared three states of art simulators: Mujoco[1], Gazebo[2], and Pybullet[3]. Each simulator comes with advantages and drawbacks. Our decision is based on active support, community, stability, scalability, and ease of use. It is important to note that our analysis includes subjective factors such as ease of use, documentation coverage, and tutorials quality. Although we tried our best to ground our points on facts, subjective criteria include certain personal preferences.

3.1 Mujoco

Mujoco is released in 2015, making it the newest physics engine in our comparison. It stands for Multi-Joint Control. Hence the name, its primary purpose is Robot simulation.

We first considered Mujoco as our main simulator. Our consideration was based on; roboticist mainly using Mujoco at OpenAI and DeepMind [3]. Besides, simulation engine reviews featured Mujoco as the best performing and stable engine [27]. Erez et al. conducted tests that compared physics engines' performance on grasping tasks, where Mujoco by far performed better than the other engines 3.1. Additionally, Mujoco provides an API that lets users develop in C language. C language API is a significant advantage for researchers aim to get as close to the hardware as possible to save time. Another plus is mujoco-py offers a Python API to control Mujoco simulation easily. OpenAI product Mujoco-py[4] is entirely open-source with MIT license.

On the other hand, MuJoCo requires a license to use for research purposes. Personal non-commercial license costs 500$ as of 2020 July. License cost is the main drawback because we prefer complete open-source software. Another factor is the training time; since our models require over 10M time-steps to train, we need to use every millisecond time gain in every step. So, GPU support is highly crucial. Mujoco only supports CPU

[1] http://www.mujoco.org/
[2] http://gazebosim.org/
[3] https://pybullet.org/
[4] https://github.com/openai/mujoco-py

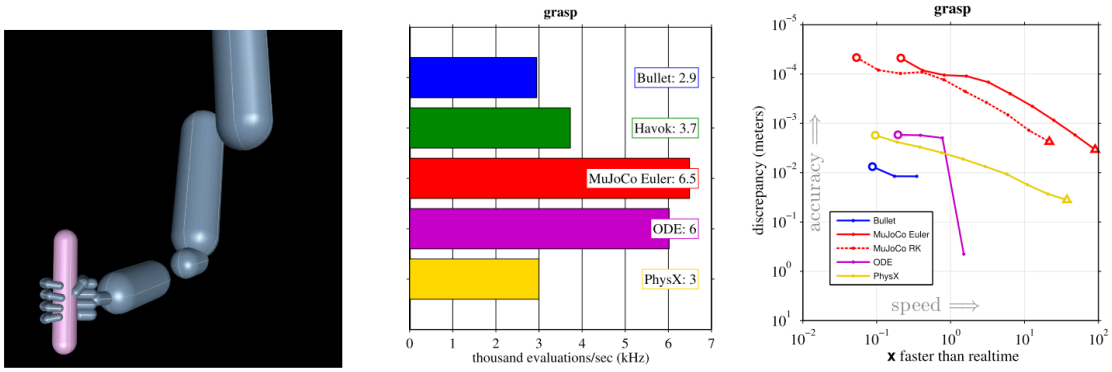


Figure 3.1: physics engine performance on grasping task. Mujoco performs the best [27]

simulation, which has the potential to be slower than GPU. As a side-note, we have not compared the MuJoCo CPU version's speed against Bullet GPU version. MuJoCo developers prepared a performance page where they explain the GPU vs. CPU comparison. They pointed out that if decoupled systems are simulated like particle simulation, GPU has an advantage. Whereas, on simulations with coupled system such as a humanoid robot, CPU gains a slight edge 3.2. They also noted that there is a room for research in the areas besides coupled or decoupled systems[5].

Finally, observing large open-source projects like OpenAI-Roboschool, which initially used MuJoCo, deprecating, and suggesting PyBullet, made us reconsider. We believe that the reason community migrating towards PyBullet is completely open-source codebase.

3.2 Gazebo

The Gazebo is the oldest simulator among Bullet and Mujoco. The development of Gazebo dates back to the 2002 University of Southern California. Later, Willow Garage took over and extended Gazebo to ROS and PR2. Gazebo became the primary simulation engine of the ROS community. Eventually, in 2012 Gazebo became part of OSRF (Open Source Robotics Foundation)[6], a spin-off from Willow Garage.

Technically, Gazebo is a simulation platform, which inherits different physics engines, such as Bullet, ODE[7], Simbody[8], and DART[9]. According to the documentation of Gazebo 11.0, it supports ODE engine default, and other engines can be used if developers compile Gazebo from the source. That means the performance of the overall

[5] http://mujoco.org/performance.html
[6] https://www.openrobotics.org/
[7] https://www.ode.org/
[8] https://simtk.org/
[9] https://dartsim.github.io/

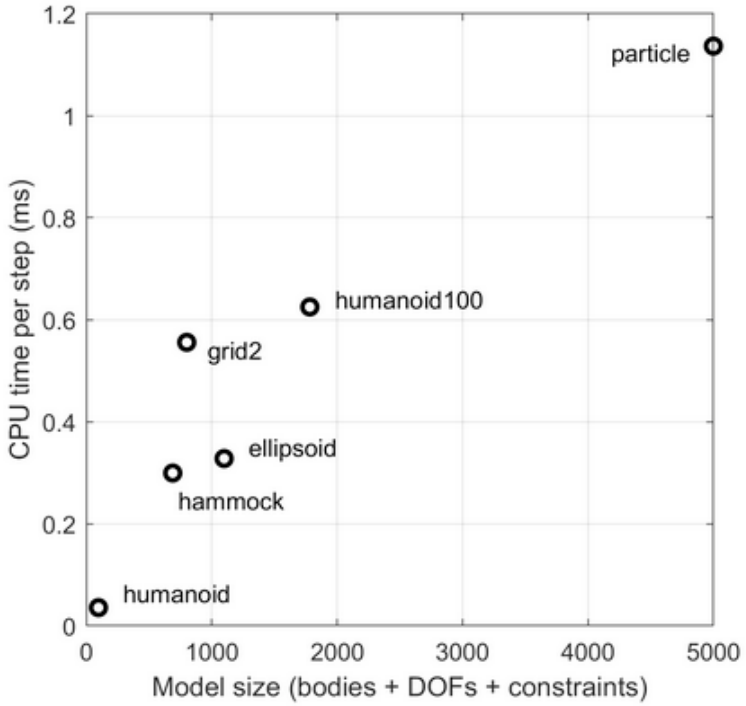


Figure 3.2: CPU time per step comparison of different tasks [27]

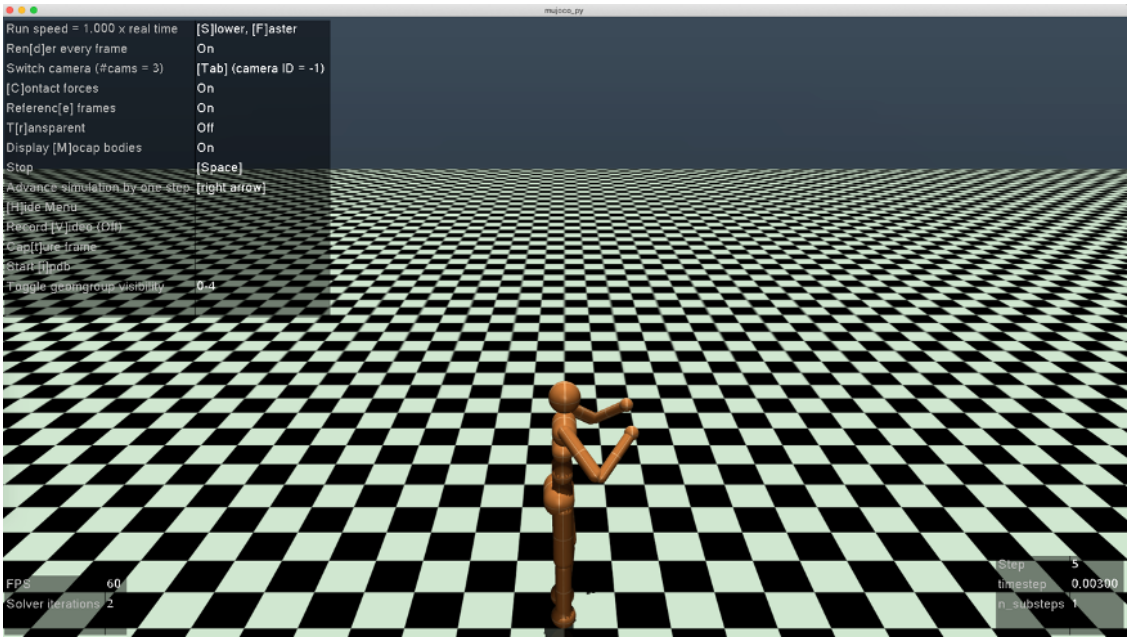
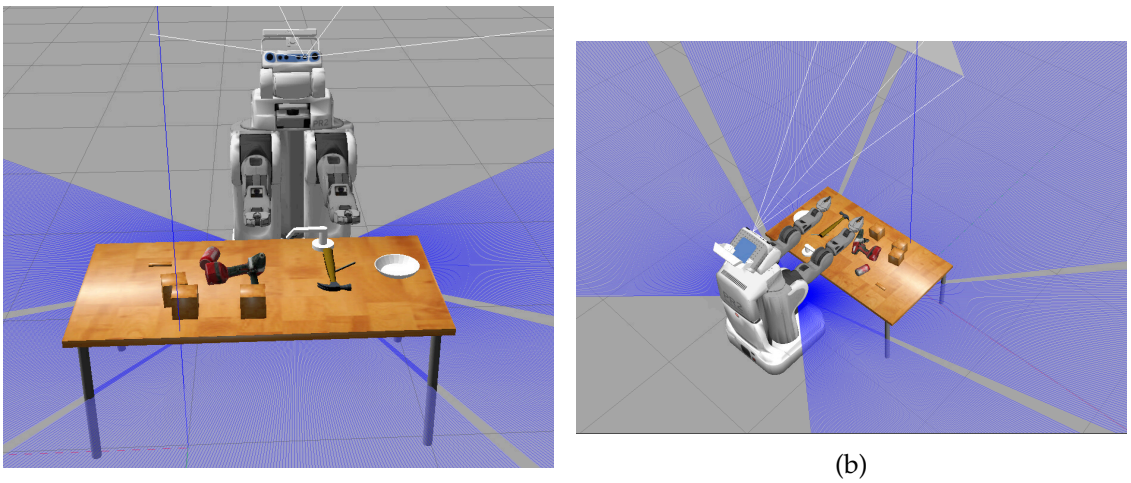


Figure 3.3: Mujoco simulation of a humanoid model. Rendered with MJViewer



(a) (b)

Figure 3.4: Table-top environment in Gazebo simulation with variety of objects and PR2 robot

Gazebo simulation highly depends on those individual physics engine's performances. Another dependency of Gazebo is ROS (Robotics Operating System)[10]. ROS infrastructure handles all communication. Thus, users need to rely on ROS to interact with Gazebo. We believe this assumption is too large. Even if relying on ROS can bring many well-structured tools, learning ROS has a steep curve and can cause a large overhead for simple projects. Nonetheless, developers invested highly on neat and clean documentation to reduce the overhead for new users. We consider the documentation and tutorials as a merit of the open-source project. Correspondingly, being open source contributes hugely to a large community willing to support, answering questions on forums, and submitting pull requests for possible bug fixes.

According to Pitonakova et al., Gazebo has usability issues due to not having a 3D mesh editing option and difficulties of installing dependencies for third party models. They also noted that Gazebo performs reasonably well in large simulation environments, so it could be more convenient to conduct extensive swarm robotics experiments on Gazebo [28].

Based on our experiments in 3.4, we found that Gazebo provides useful models to set up a table-top environment for robotics picking applications quickly. Although we have not performed any grasping experiments on Gazebo, editing the size, position, and orientation of models directly on simulation GUI is a useful feature, lacking both on Pybullet and Mujoco.

[10] `https://www.ros.org/`

3.3 PyBullet

PyBullet is a simulator built on the Bullet physics engine. Bullet's initial release dates back to 2006. Bullet engine was initially game, and graphics focused, but lately, with PyBullet, it has been increasingly popular among roboticists. We decided on using PyBullet for several strong reasons and overall satisfied by the performance and the features it offers.

Firstly, PyBullet recently published their RL resources on Github [11]. These resources provide everything necessary to quickly start experimenting with any choice of robot, environment, and algorithm. Secondly, in RL robotics research, our reference papers are using PyBullet for their experiments [15] [4]. Therefore, it is more convenient to compare results using the same physical engine to avoid simulation differences. Thirdly, it has sizeable open-source support and free to use.

On the other hand, based on Erez et al. performance tasks, which compares Bullet, Mujoco, ODE, and PhysX engines, Bullet is either at the last place or the one above the last position 3.5 figure. Similarly, they mention Bullet Engine's spring-damper system's incompatibility with the standard PD controller design [27].

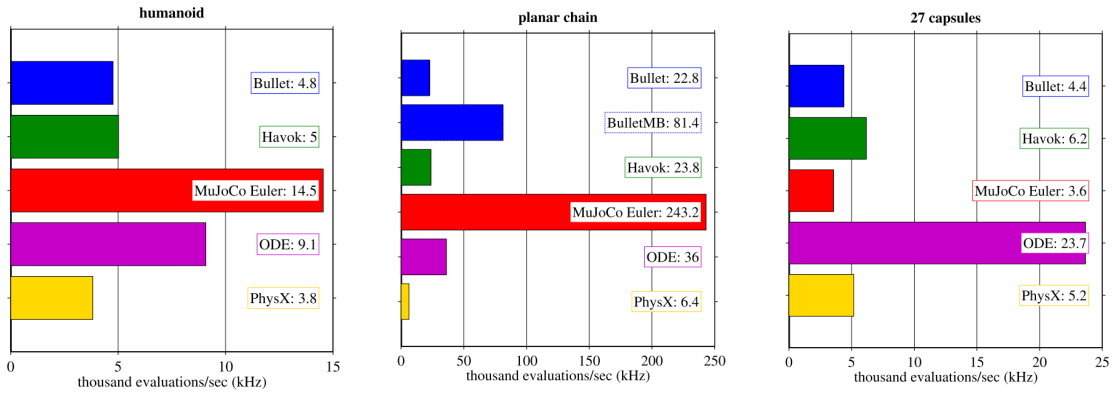
Many open-source robotics learning resources migrating to PyBullet. This migration brings more open-source contributors who are more focused on Reinforcement Learning. For example, open-source contributors recently developed multi-threading GPU support to PyBullet. Also, Stable Baselines[12] creators implemented a RL training package under Bullet's Github repository[13] [5].

In conclusion, PyBullet is the most actively developed physics engine among the three contenders. Thanks to the open-source community, any small bug is reported and solved quickly, missing in Gazebo and Mujoco. We assume more roboticists will migrate to PyBullet in the future. As a result of this, it will become a primary physics engine for robotics research. Despite research papers representing Pybullet engine performance as the worst among three, we have not noticed any significant failure in engine performance in practice.

Simulator	**Gazebo**	**MuJoCo**	**Pybullet**
Designed For	Robotics	Robotics	Graphics/Games/Robotics
License	Open-Source	Closed-Source	Open-Source
API	C++/Python	C	Python
Documentation	4/5	3/5	5/5
Ease of Use	2/5	3/5	5/5

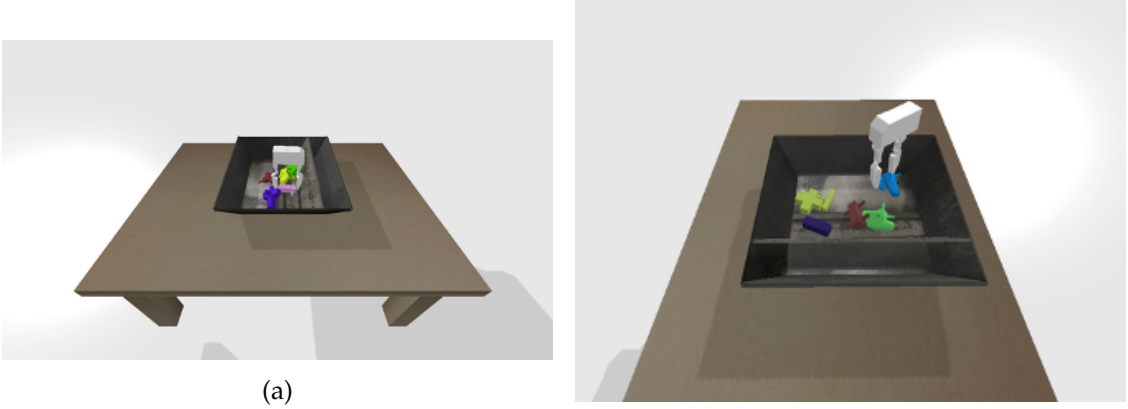
Table 3.1: Comparison of simulators. Finally, we decided on PyBullet

[11] `https://bit.ly/3h1X7uU`
[12] `https://github.com/hill-a/stable-baselines`
[13] `https://bit.ly/3jqU7Kd`



(a) (b) (c)

Figure 3.5: Comparing physics engine performances. Bullet is either at the last place or the one above the last position [27]



(a) (b)

Figure 3.6: Screenshot from our trained hand model in Pybullet

4 Reinforcement Learning Algorithms and Tools

In this chapter, we will dig deeper into Reinforcement Learning algorithms. How the algorithms are implemented, what are their strengths and weaknesses? We will present our state-of-art algorithm, SAC (Soft-Actor-Critic), and the algorithm we want to test on the robotics applications, BDQ (Branched-Dueling Q-learning). We will mention the fundamental differences between those two algorithms, such as their optimization objective functions and different exploration approaches.

After the introduction of algorithms, we are going to present RL environment frameworks and algorithm libraries. We base our custom robotics environment on Gym environment definition. In the same way, we intensively used an open-source RL baseline algorithm library called Stable-Baselines. For the implementation of neural networks, we imported Tensorflow and Keras. Finally, we introduce Optuna as our hyper-parameter optimization tool.

Overall we depend on many different software libraries, which we do not mention here; the interested audience can take a look at our Github page[1].

4.1 Soft Actor Critic (SAC)

SAC algorithm is described as the state-of-art algorithm as of 2020 [5]. Therefore, we chose SAC as our baselines algorithm to compare against the BDQ. SAC is an actor-critic, model-free algorithm. Actor critic refers to the optimization of both policy and value function. Model-free nature stems from the sample based update structure. The success of SAC lies in low sample complexity and robustness to different hyperparameters [29]. SAC achieved this success due to; actor-critic architecture, entropy maximization, and off-policy update structure.

Actor-critic architectures have long been used in RL literature [30]. [29]. The actor-critic's core idea is optimizing the policy and value functions separately but combining them during the policy iteration part of the RL algorithm. The policy function is responsible for the policy evaluation, while the value function is responsible for policy improvement.

$$\pi^{*}_{MaxEnt} = \arg\max_{\pi} \sum_{t} \mathop{\mathbb{E}}_{(s_t,a_t)\sim\rho_\pi} [r(s_t,a_t) + \alpha H(\pi(.|s_t))] \tag{4.1}$$

[1] `https://bit.ly/3k8uSga`

$$H(X) = \mathop{\mathbb{E}}_{X}[I(x)] = -\sum_{x \in X} p(x) \log p(x) \tag{4.2}$$

Off-policy, algorithms are naturally better at working in the sparse data region because they can incorporate past experiences and use state-action-rewards pairs from different policies. Albeit the off-policy approach brings high variance into the learning process, new papers overcome this problem with Polyak-Rupper averaging or adaptively setting the step size of the stochastic gradient descent optimizer [19].

4.1.1 SAC Implementation as Baseline Algorithm

We use the stable-baselines implementation of the SAC algorithm. This implementation uses double soft-Q functions, a soft-state value function to estimate the critic, and a policy network to estimate the actor. Stable-baselines supports automatically learning the entropy coefficient. This feature saves the user time by avoiding hand-tuning on the entropy coefficient term.

Unlike the tabular soft-policy iteration method, where policy evaluation and policy iteration steps follow each other strictly, soft-actor-critic calculates the losses of both critic and actor update the parameters by stochastic gradient descent in one iteration. Although the function approximator implementation of SAC does not directly follow the policy-iteration, it inherently follows the generalized policy iteration schema.

The essential characteristic of SAC is the entropy maximization. The entropy bonus comes into play when calculating the bellman backup of soft-Q-function, soft-state value function, and the policy loss. That means the learned policy must maximize the rewards and entropy at the same time.

The soft Q-function loss is represented in the equation below 4.5, which is the same loss function used by Mnih et al. in 2015, updates the parameters of the Q-network in the direction of the TD target. The only difference in the Q-function update step between SAC and DQN is the inclusion of entropy variables in the state-value network 4.3.

$$V(s_t) = \mathop{\mathbb{E}}_{a_t \sim \pi}[Q(s_t, a_t) - \alpha \log \pi(a_t|s_t)] \tag{4.3}$$

$$\pi_{new} = \arg\min_{\pi' \in \Pi} \left(\pi'(.|s_t) \middle|\middle| \frac{exp(\frac{1}{\alpha} Q^{\pi_{old}}(s_t, .))}{Z^{\pi_{old}}(s_t)} \right) \tag{4.4}$$

$$J_Q(\theta) = \mathop{\mathbb{E}}_{(s_t, a_t) \sim D} \left[\frac{1}{2} (Q_\theta(s_t, a_t) - (r(s_t, a_t) + \gamma \mathop{\mathbb{E}}_{(s_t, a_t) \sim p} [V_{\theta^-}(s_{t+1})]))^2 \right] \tag{4.5}$$

SAC and DQN diverge in the policy improvement step, where DQN updates the policy based-on epsilon-greedy approach, SAC parameterizes the policy with a neural network and updates the parameters in the direction of the minimum loss function

4.6. Haarnoja et al. manipulate the policy definition with a reparameterization trick to allow stochastic descent on the loss function 4.7.

$$J_\pi(\phi) = \mathop{\mathbb{E}}_{s_t \sim D, \epsilon \sim N} \Big[\alpha \log \pi_\phi(f_\phi(\epsilon_t; s_t)|s_t) - Q_\theta(s_t, f_\phi(\epsilon_t; s_t)) \Big] \tag{4.6}$$

$$a_t = f_\phi(\epsilon_t; s_t) \tag{4.7}$$

The soft Actor-Critic algorithm given in 4.1 follows the general guidelines of actor-critic type of algorithms with the introduction of entropy maximation RL into the definitions of Q and policy functions.

Algorithm 1 Soft Actor-Critic

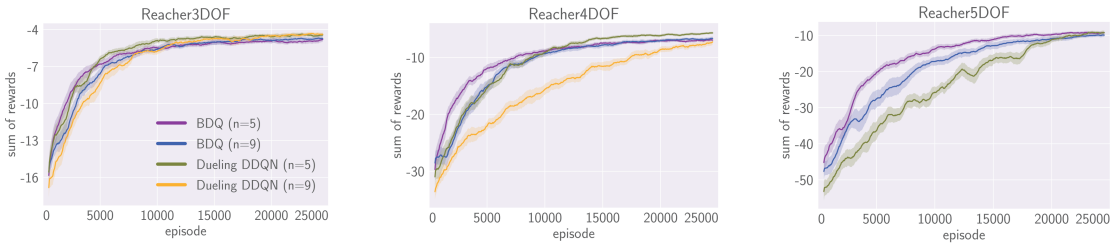
Input: θ_1, θ_2, ϕ ▷ Initial parameters
$\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$ ▷ Initialize target network weights
$\mathcal{D} \leftarrow \emptyset$ ▷ Initialize an empty replay pool
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ ▷ Sample action from the policy
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ ▷ Sample transition from the environment
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$ ▷ Store the transition in the replay pool
  end for
  for each gradient step **do**
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$ ▷ Update the Q-function parameters
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$ ▷ Update policy weights
    $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$ ▷ Adjust temperature
    $\bar{\theta}_i \leftarrow \tau\theta_i + (1-\tau)\bar{\theta}_i$ for $i \in \{1, 2\}$ ▷ Update target network weights
  end for
end for
Output: θ_1, θ_2, ϕ ▷ Optimized parameters

Figure 4.1: SAC pseudocode [29]

4.2 Branching Dueling Q-Network (BDQ)

BDQ is our test algorithm. Tavakoli et al. developed BDQ as a variant of Dueling Double DQN [2]. They aimed to solve the intractability of the DQN algorithm on high dimensional continuous tasks. The key feature of their algorithm is the shared module. The shared module representation allowed the DQN algorithm to cope with the intractability problem. Tavakoli et al. showed that their network could output multi-dimensional actions without convergence issues. They believe the structure's stability is due to the encoded latent representation of the input in the shared module.

Their results stress that, especially in higher dimensional action spaces, BDQ performs better than Double-dueling DQN implementation 4.2. Even compared to proven



(a) (b) (c)

Figure 4.2: BDQ performances compared to Dueling-Double-DQN with increasing action spaces

algorithms, like DDPG, BDQ performs better. Although researchers at Google DeepMind presented in 2016 that the DDPG algorithm outperforms DQN variants at almost every task [31], Tavakoli et al. proved the opposite that BDQ shows better performance than DDPG on the Humanoid walking benchmark task 4.3.

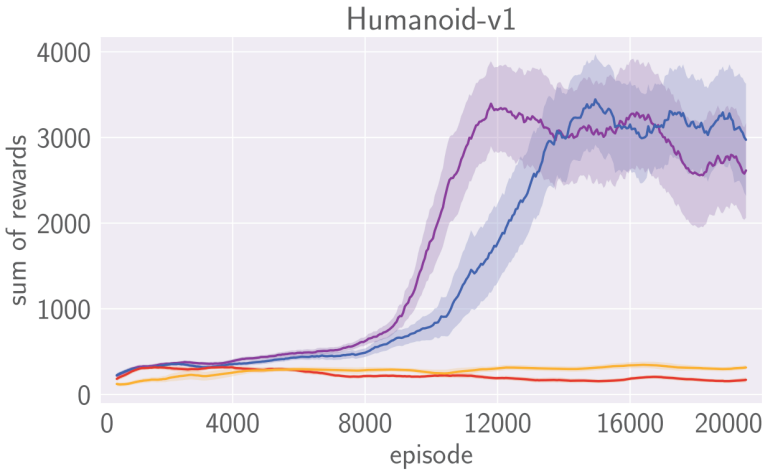


Figure 4.3: Blue and purple lines represent BDQ algorithm with 33 and 17 action padding. Orange line shows DDPG's performance

In this section, we will go through the implementation details of BDQ. The authors of the BDQ article provided their implementation of BDQ on Github. Based on the original code and the insights from the article, we implemented our version of BDQ on stable-baselines codebase. As a result, we avoided the save, load model issues from BDQ original code, and welcomed new features such as callback structure, parameter manipulation, and warm starting.

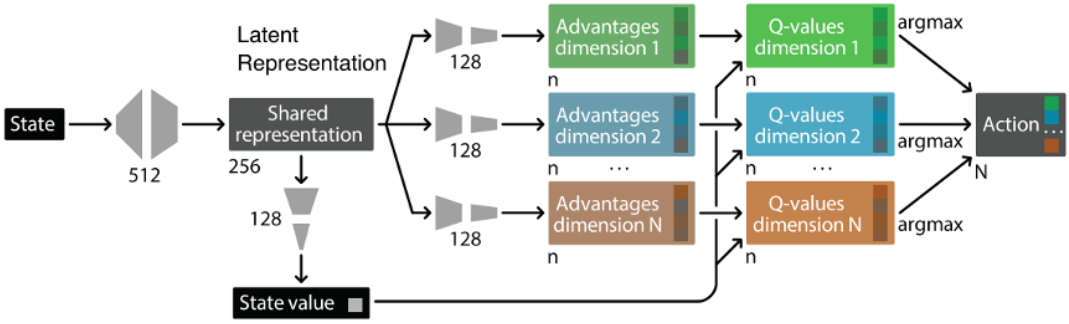


Figure 4.4: BDQ network representation

4.2.1 BDQ Implementation

BDQ adopts new improvements from the DQN algorithm, such as double-q function, dueling architecture, prioritized replay. One crucial design decision is that while advantage functions are unique to every stream of actions, they kept the same state-value estimation for each action branch in 4.4. Later they combined the common state-value estimate and the advantage function in the aggregation layer. This approach helps with the generalization of similar states' actions by reducing the overfitting of action branches. Therefore, they can scale up to more complex and higher dimensional tasks, where Dueling-Double DQN becomes intractable.

Another key difference of BDQ is the aggregation layer; they subtract the mean advantage value from the individual branch's advantage value. Then, sum up the result to calculate the Q-function value of each branch, shown in 4.8. Although the lack of identifiability problem exists, they achieve better results than the theoretically proven max reduction method(4.9).

$$Q_d(s, a_d) = V(s) + \left(A_d(s, a_d) - \frac{1}{n} \sum_{a'_d \in A_d} A_d(s, a'_d)\right) \tag{4.8}$$

$$Q_d(s, a_d) = V(s) + \left(A_d(s, a_d) - \max_{a'_d \in A_d} A_d(s, a'_d)\right) \tag{4.9}$$

TD-target definition of BDQ also differs from DDQN. Where DDQN based approach calculates the individual TD-target for every branch in 4.10, BDQ sets one global target for all actions by taking the mean of the max Q-function variable in the TD-target equation 4.11. In our opinion, a common TD-target for all branches of BDQ underlines the dependency between action branches and forces them to act in collaboration.

$$y_d = r + \gamma Q_d^- \left(s', \arg\max_{a'_d \in A_d} Q_d(s', a'_d)\right) \tag{4.10}$$

$$y = r + \gamma \frac{1}{N} \sum_d Q_d^- \left(s', \arg\max_{a'_d \in A_d} Q_d(s', a'_d)\right) \tag{4.11}$$

Analogous to the TD-target definition, they express loss differently than the DDQN counterpart. BDQ authors calculate the loss first by taking the mean of the squared TD-error over the branches and then taking the expectation of this score in 4.12. Identical to the TD-target calculation, loss definition also helps the action branches to act dependent on each other.

$$L = \mathop{\mathbb{E}}_{(s,a,r,s')\sim D}\Big[\frac{1}{N}\sum_{d}(y_d - Q_d(s,a_d))^2\Big] \tag{4.12}$$

The exploration-exploitation trade-off is still an on-going research in RL. Nevertheless, researchers assume that better-exploring algorithms have an edge over weakly exploring algorithms. SAC with a robust entropy-based exploration approach, considered to be the best exploring algorithm and the state-of-art RL algorithm [29]. BDQ original code offers two different exploration approach, epsilon greedy and Gaussian noise. Since epsilon-greedy usually used with discrete DQN based algorithms, the authors found it inadequate to explore continuous spaces. Instead of epsilon-greedy, they recommended the use of Gaussian noise for better performance.

BDQ follows the same pseudocode of double-DQN(4.5) with dueling and branching network extensions.

Algorithm 1: Double DQN Algorithm.

input : $\mathcal{D}$ – empty replay buffer; θ – initial network parameters, θ^- – copy of θ
input : N_r – replay buffer maximum size; N_b – training batch size; N^- – target network replacement freq.
for *episode* $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  for $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
    Sample next frame x^t from environment $\mathcal{E}$ given (s, a) and receive reward r, and append x^t to $\mathbf{x}$
    if $|\mathbf{x}| > N_f$ **then** delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ **end**
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple (s, a, r, s') to $\mathcal{D}$,
      replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
    Sample a minibatch of N_b tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
    Construct target values, one for each of the N_b tuples:
    Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
    $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
    Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps
  end
end

Figure 4.5: Double-DQN pseudocode [32]

4.3 OpenAI Gym

The gym framework provides a standard definition of a reinforcement learning environment. It assumes that every environment is formalized with the Markov Decision Process (defined in section-2.3.2) [3]. Hence, it follows the structure of the agent taking action and receiving observation and reward as a result of it.

```
ob0 = env.reset() # sample environment state, return first observation
a0 = agent.act(ob0) # agent chooses first action
ob1, rew0, done0, info0 = env.step(a0) # environment returns observation,
# reward, and boolean flag indicating if the episode is complete.
a1 = agent.act(ob1)
ob2, rew1, done1, info1 = env.step(a1)
...
a99 = agent.act(o99)
ob100, rew99, done99, info2 = env.step(a99)
# done99 == True  =>  terminal
```

Figure 4.6: Gym interface to interact with an environment

The code example 4.6 demonstrates the interaction between an agent and the environment. OpenAI designed the Gym framework in a way that it is agent diagnostic. Therefore, it allows the user to try different algorithms on the agent side freely. The environment, on the other hand, should follow the Gym guidelines. A custom gym environment has to override the step, reset, seed, render, and close functions. The step function runs one timestep of the environment and returns a tuple of reward, observation, done, and info. The reset function resets the domain setting to a random or predefined initial state and returns the state's observation. The seed function starts the seeding of the random number generator. Close function is called at the end when the user wishes to quit the environment. Thus, it performs the necessary memory clean-up or resource deallocation.

Listing 4.1: Gym environment example instantiation

```
from gym.envs.registration import register

register(
    id='gripper-env-v0',
    entry_point='manipulation_main.gripperEnv.robot:RobotEnv',
)

env = gym.make('gripper-env-v0', config='config/gripper_grasp.yaml')
```

Gym environment definition also provides a useful helper function gym.make(). After registering the custom environment, one can instantiate the environment with a simple one-liner represented in the code 4.1. It also allows the user to pass an argument to the constructor of the custom environment. In our case, we provide the configuration file as an argument.

4.4 Stable Baselines

While the OpenAI gym framework identifies the environment, Stable Baselines provides the agent part of the RL framework. Stable Baselines is a robust open-source

fork of OpenAI baselines. Yet it is based on OpenAI Baselines; it has surpassed the performance of OpenAI Baselines in many ways. The authors of Stable Baselines have composed a Medium article[2] that describes every additional feature and bug fix over OpenAI baselines. In this section, we will only mention the critical components that are needed for our project.

First and foremost, Stable Baselines has strong support for the state-of-art algorithm SAC. SAC proved to be the best algorithm we tested in our environment. Apart from SAC, it supports ten more algorithms[3]. Stable Baselines maintains a sophisticated save/load structure. Since we aim to warm-start and fine-tune the neural network variables, we should save and load the network variables individually. A use-case in our application is truncating the last soft-max layer of a saved model to change the action-space size. We often encountered problems with OpenAI Baselines save/load structure.

Moreover, Stable Baselines supports input and reward normalization. Later in the results section, we will show that our best performing model is equipped with both input and reward normalization.

Lastly, because of the extra features, fully documented, and tested code-base, we decided to reimplement the BDQ algorithm based on Stable Baselines. Example of how to run a simple training and saving can be seen below in 4.2

Listing 4.2: Example of training and saving BDQ algorithm on gripper-env

```
import gym

from stable_baselines.bdq.policies import MlpActPolicy
from stable_baselines import BDQ

env = gym.make('gripper-env-v0', config='config/gripper_grasp.yaml')

model = BDQ(MlpActPolicy, env, verbose=1)
model.learn(total_timesteps=10000)

obs = env.reset()
for i in range(1000):
    action = model.predict(obs)
    obs, rewards, dones, info = env.step(action)

env.close()
```

[2] https://bit.ly/3ia3GvM
[3] a2c, acer, acktr, ddpg, dqn, gail, her, ppo, trpo, td3

4.5 Machine Learning Framework

RL owes its success partly to a strong function approximator, deep neural networks. Where tabular Q-function only solved games like tic-tac-toe, neural networks integration to Q-function scales to high dimensional complex tasks like Atari games [24]. The growing success of neural networks inspired large companies like Google and Facebook to start their own open-source machine learning frameworks. Among those projects, Tensorflow, Keras, and PyTorch are the most popular ones [33] [34] [35].

Granted that we decided to work with Tensorflow and Keras on different parts of the project, it is worth mentioning that PyTorch is becoming dominant in research [36]. In figure 4.7 demonstrates how Pytorch has risen steeply in terms of mentions in the major conferences. It achieved a staggering comeback from a maximum of 6% mentions to 78.72% mentions in three years.

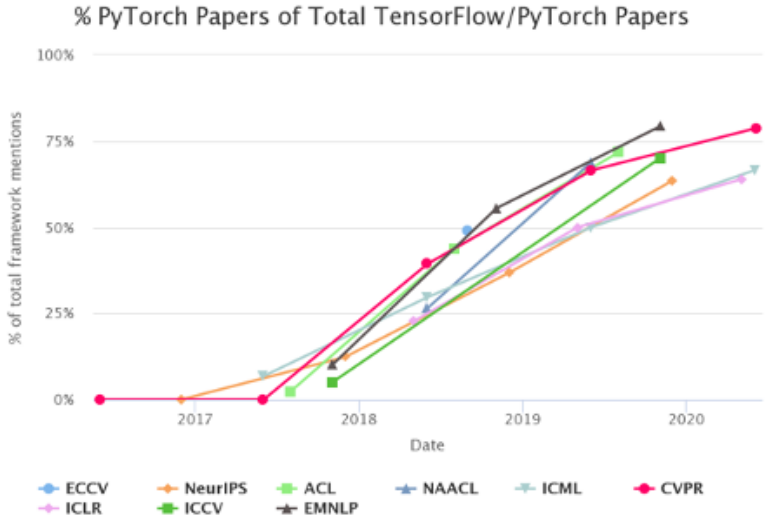


Figure 4.7: PyTorch and Tensorflow comparison based on the mentions in major machine learning conferences [36]

According to the researchers, PyTorch's success is due to simplicity, great API, and performance. Even Tensorflow, in its last edition Eager, delivered a similar API to PyTorch. Though, there have been reports about the weaknesses of Eager in performance and memory. In short, it is unlikely that Tensorflow recovers and catches PyTorch anytime soon.

In conclusion, it is also possible to see the transition from Tensorflow to Pytorch in the RL community. Stable Baselines released the PyTorch version of Baselines algorithms in May 2020 [37]. OpenAI Spinningup, an educational incentive for RL teaching from OpenAI, has moved to PyTorch in February 2020. Furthermore, they now host the master branch with PyTorch implementation [23].

4.6 Optuna - Hyperparameter Optimization Library

As defined in the challenges section, hyperparameter tuning is one of the most challenging parts of Reinforcement Learning. There are some unique guidelines like with higher batch-size, one can afford higher learning-size. However, in general, there is no theoretical foundation that describes hyperparameter selection. In recent years many hyperparameter optimization libraries have been developed to address the issue of parameter selection. Some software designed a tree-structured Parzen estimator like Hyperopt, and some use Gaussian processes like Spearmint and GpyOpt to optimize for the best performing parameters [38]. Optuna outperforms other optimization software with the following points:

- Define-By-Run API
- Better Pruning Strategy
- Easy to Setup

The Define-By-Run interface is first coined in the machine learning context. PyTorch surpassed Tensorflow with the help of its dynamic Define-By-Run API. While in the machine learning framework, Define-By-Run refers to defining the neural network on the go, in hyperparameter optimization, it relates to allowing the user to specify the objective function [38]. Optuna takes care of the dynamic construction of the search space. In other words, the user does not need to enter the search space statically.

Interface design only defines how users can interact with the software but give no hint about the underlying algorithm's performance. For a robust hyperparameter optimization framework, a cost-effective pruning system is a must. Optuna adopts a variant of the Asynchronous Successive Halving algorithm to trigger a background call to the 'should-prune' method [39].

Optuna is an open-source MIT licensed software. Hence, it attracts a variety of users in the ML world to verify the scalability and robustness. Indeed, it proves to be easy-to-use with over 2.8k stars on their Github page[4]. Moreover, we were satisfied with their documentation and thorough tutorials [5].

Overall, we were satisfied with the performance of Optuna. Although we did not try the other optimization framework, the figure in 4.8 represents the high-level comparison to Optuna.

4.7 Hardware Setup

Reinforcement learning in coordination with neural networks uses huge computational resources. As of today, GPUs are the most common and efficient way to train neural

[4] `github.com/optuna/optuna`
[5] `optuna.readthedocs.io/en/stable/`

Framework	API Style	Pruning	Lightweight	Distributed	Dashboard	OSS
SMAC [3]	define-and-run	✗	✓	✗	✗	✓
GPyOpt	define-and-run	✗	✓	✗	✗	✓
Spearmint [2]	define-and-run	✗	✓	✓	✗	✓
Hyperopt [1]	define-and-run	✗	✓	✓	✗	✓
Autotune [4]	define-and-run	✓	✗	✓	✓	✗
Vizier [5]	define-and-run	✓	✗	✓	✓	✗
Katib	define-and-run	✓	✗	✓	✓	✓
Tune [7]	define-and-run	✓	✗	✓	✓	✓
Optuna (this work)	define-by-run	✓	✓	✓	✓	✓

Figure 4.8: Hyperparameter optimization library comparison [38]

networks. As shown in the figure 4.9, GPUs are 11-fold faster than the CPU neural network computation [40]. Thus, a current GPU can decrease the learning time 11 times shorter than a CPU.

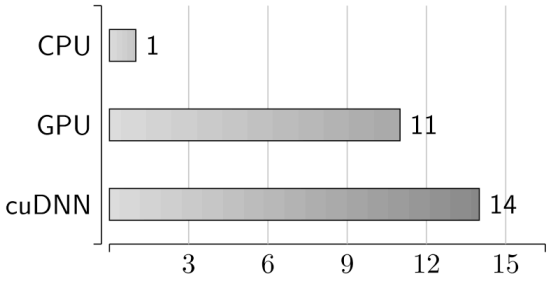


Figure 4.9: Neural network copmutation performance of different hardwares

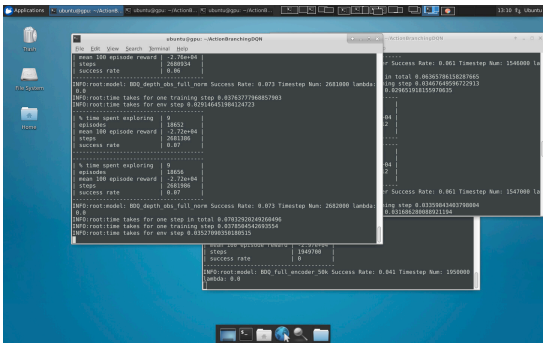
For the duration of the thesis, we gradually increase our access to more extensive learning resources. At the beginning of the project, we used our lab computer at Human-Brain-Project Lab that only had an Nvidia GeForce GTX-670, 31.2gb Ram. Since we did not need many computational resources in the beginning, GeForce GT-670 with 4gb ram was enough for us. As the computational requirements got higher, we migrated to use google cloud services. We received free google cloud credits from Roboy[6] only for two months of usage. Enabling us to train larger networks in parallel with a more current graphics card Nvidia Tesla V100[7]. Tesla V100 is designed for AI research. Thus it is more optimized compared to the GeForce series in terms of neural network training. After the free usage period had ended, we switched to the LRZ cloud framework. At LRZ, we continued to have a Tesla V100 with 368gb and 20VCPUs.

The increase in the compute capability delivered a faster physics simulation. In return, the total training time got faster. Apart from the computational capacities, among the three different platform, LRZ provided the fastest internet connection. Having their server center at Garching, there was no recognizable latency. Although Google cloud

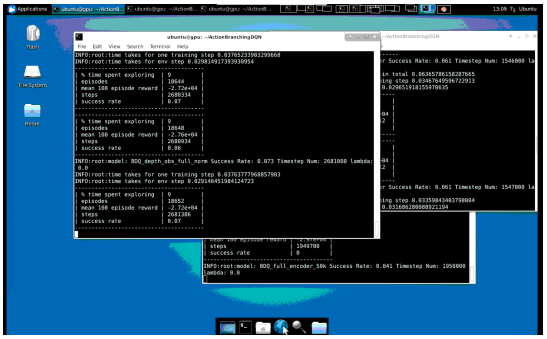
[6] https://roboy.org/
[7] https://www.nvidia.com/en-us/data-center/v100/

platform servers are located in Frankurt, we suffered with a laggy internet connection. Overall LRZ cloud computation framework provides a safe and fast connection. Moreover, we received a quick response to our questions regarding the framework usage.

The remote desktop connection was a necessary tool for our project. Given that we trained neural networks entirely on the cloud, we needed a way to visualize the trained agents. Copying the trained model file back to the local computer is an option. Even though PyBullet supports MacOS, the neural network's forward pass still takes long on consumer hardware. For these reasons, we needed to use remote desktop connections. We tested two different remote desktop software: VNCViewer, and Tiger Remote Desktop. Tiger remote desktop has the edge over VNCViewer in terms of lightweight and user-friendly design. Besides, we often experienced blurred views and unable to change the screen resolution on VNCViewer. We installed the Xfce[8] Unix based desktop environment on the LRZ cloud machine and forwarded the VNC port to localhost to connect with Tiger Remote Desktop. Xfce delivers a lightweight, easy-to-use desktop environment. We were overall satisfied with the performance of the remote desktop environment.



(a) Screenshot of Xfce desktop on LRZ cloud from Tiger remote desktop tool



(b) Fuzzy view problem from VNCViewer remote desktop tool

Figure 4.10: Remote desktop connection to LRZ cloud computing

[8] https://www.xfce.org/

5 Curriculum Learning

Like most of the terms, ML research borrowed 'Curriculum Learning' from cognitive science. Curriculum learning draws a parallel from human intelligence to machine intelligence. As humans learn through meaningfully ordered, sophisticated education systems, machines profit from guided learning patterns [41]. Elman demonstrates how starting small in both human and machine context results in enhanced learning [42]. He designed a recurrent neural network to learn the grammatical structure of a language. At the beginning of the training, this model had limited and restrictive capacity; later on, it gained more resources to exploit. Akin to a toddler, who has limited cognitive ability but manages to learn a language fluently. Elman's approach proved to understand a grammatical structure of a language, which, without the curriculum approach, would be impossible to comprehend. In general, either in human or machine settings, curriculum learning promises faster convergence and better generalization [41].

Curriculum learning also unlocks new possibilities in robot learning. Indeed, Sanger showed inverse dynamics problem on robotics manipulators could be solved with the integration of Trajectory Extension Learning [43]. During training, he commands an approximate solution for the next value of parameters as guidance [43]. Karpathy et al. evaluated a two-level curriculum structure to learn the motor skills of the acrobat. He showed that curriculum learning could enhance the generalization of learned skills for underactuated control, depicted in 5.1. Consequently, before our work, Breyer et al. have proved that the curriculum learning approach can facilitate a robotics manipulator's learning on object grasping tasks. They found the curriculum integrated trials the most successful [4].

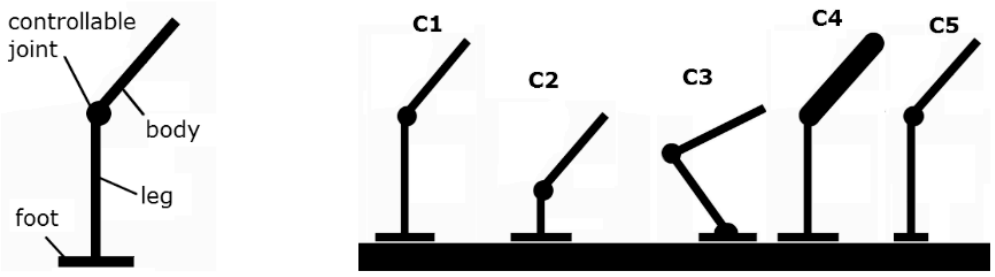


Figure 5.1: Acrobot learned motor skills through two-level curriculum strategy [44]

Either on supervised learning or reinforcement learning, curriculum integration proved useful to ease the learning process. Even though curriculum strategy brings a high number of hyperparameters and prior bias from expert knowledge, we found curriculum

learning the most critical part of our work. As we represent in the future work section, we believe the automatic learning of curriculum parameters will bring more attention to the curriculum-based techniques.

6 Experimental Setup

In this section, we will explain different types of experimental environments, reward, and observation structures.

6.1 Problem Setup

We base our environment setting on Breyer et al.'s table cleaning grasping environment. In the simulation environment, a gripper is spawned until the wrist and attempts to grasp randomly drawn objects from the table or the floor. The gripper is deprived of an arm and a base. Accordingly, the computation of inverse kinematics is ignored.

The observed state originates from the RGBD camera mounted on the gripper. The gripper is position controlled with continuous input between -1 to 1. Based on the task description, an action is represented either by [dx, dy, dz, ϕ, gripper open/close] or [dx, dy, ϕ].

Episode either terminates by the timeout signal after 150 timesteps or the success signal. Success signal is triggered when the gripper carries an object to a predefined height defined by the curriculum strategy.

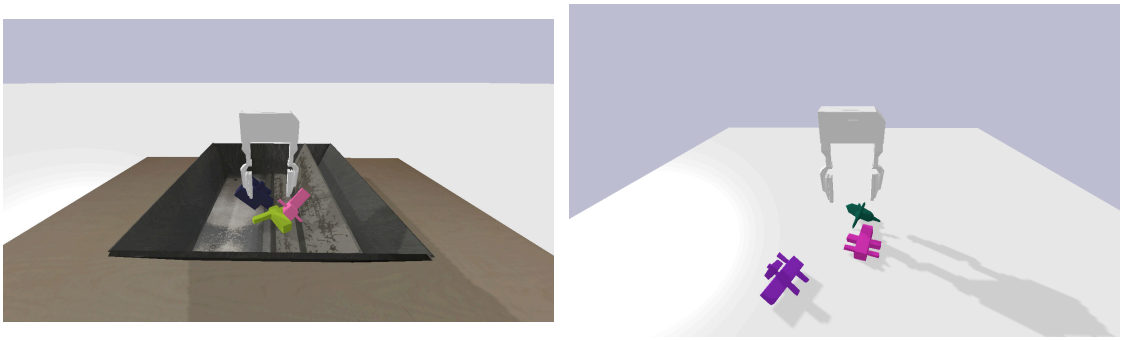
We conduct our experiments on two different simulation scenes: Floor (7.10a) and table with a tray (7.10c). Floor scene represents the Breyer et al.'s setup, and the table with a tray scene is same as the Quillen et al.'s configuration. We investigated the transfer performance of the model trained in the floor scene in the table setup.

Different task descriptions will be explained in the next section. Our assumptions regarding the environment are:

1. Gravity is compensated on the gripper, but not the objects.
2. The gripper can only take relative action to its position. In every timestep, relative translation and rotation are limited to maximum of 3cm and 15cm, respectively.
3. Other than applied actions, no other external force acts on the gripper.

6.2 Robotics Environment Descriptions

We have two testbed environments: Simplified task description and the full task description. We primarily use the simplified environment to test and prune the algorithms quickly. If the model solves the simplified environment, we move forward to try it in



(a) Table Scene

(b) Floor Scene

Figure 6.1: Table and floor scenes

the full environment. Those two descriptions deviate in terms of action, observation, reward, and curriculum definition 6.1, 6.2.

The simplified environment has fewer action dimensions to control compared to the full environment. A total of three-dimensional action controls the x, y coordinates translation, and yaw rotation. Hence, every timestep it receives a fixed size of downward z-axis movement. Similar to Quillen et al. and Breyer et al., our gripper also automatically closes when it reaches a certain height threshold.

The full environment, on the other hand, has full control over all dimensions. The gripper receives five-dimensional action to control cartesian coordinates x, y, z, yaw rotation, and the gripper open/close in the full environment.

Observation from the environment differs as well, based on the task description. In the simplified scenario agent only receives the hundred-dimensional encoded depth image. Whereas, in the full scenario, the agent gets the actuator width and the hundred-dimensional encoder output.

We used a custom shaped reward function for the environment setting. In the simplified environment, a sparse reward definition was used.

Curriculum strategy guides the agent to a more challenging environment based on the agent's success rate. In our case, curriculum switch happens both on simplified and full environment definition at a 70% success rate. This switch triggers an increase of difficulty on the environment's curriculum parameters such as max object count, object spawned area, and terminating object height. More detailed information regarding the curriculum parameters is given in the below table in 6.2

6.3 Robot Model

As mentioned in the problem setup section, we only use a gripper without the arm to increase computation speed. This setup obliviates the need for inverse kinematics calculation for each joint of a whole robotics arm. Although we deviate from the real-world setting by excluding the arm, Breyer et al. showed that the model learned without the

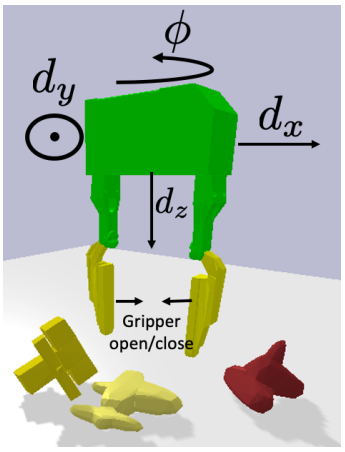


Figure 6.2: Full environment freedom of movement

Env. Description	Action Space	Observation	Reward	Discount Factor
Full	5	Encoder + Actuator Width or RGBD Depth	Shaped Reward	0.99
Simplified	3	Encoder	Binary Reward	1.0

Table 6.1: Different paramters of Simplified and Full environment definitions

Env. Description	Curr Steps	Curr Success Thresh.	Robot Height	Max Objects	Object Area
Full	8	0.7	[0.15, 0.25]	[3, 5]	[0.01, 0.1]
Simplified	4	0.7	[0.17, 0.27]	[3, 5]	[0.01, 0.1]

Table 6.2: Curriculum Parameters of Simplified and Full environment descriptions

(a) ABB Yumi gripper model

(b) WSG50 gripper model

Figure 6.3: Real images of gripper models

entire kinematic chain can still perform successfully on the real-world robot [4]. This success is possible because of the small working setup comprising only 10cm to 10cm area and the limited translation and yaw rotation motions, 3cm and 15cm.

We noticed that in Breyer et al. setting, because of the narrow gripper width of the robot model (ABB Yumi), it could not perform at its maximum potential 6.4a. Thus, we selected a gripper (WSG50) that has a broader gripper opening 6.4b. As a result, we could use the object models without scaling.

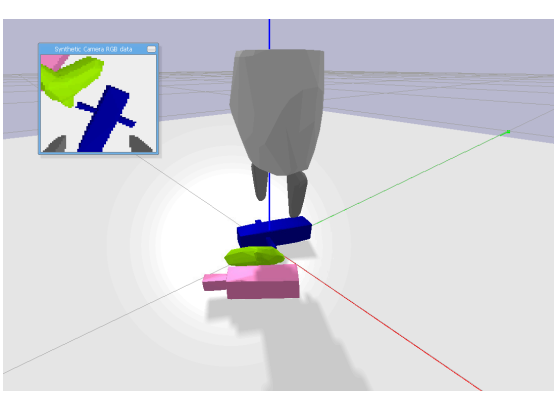
6.4 Object Database

Generally, particular object shape and color usage bring a certain bias to the learned grasp model. Therefore, the object database should be as diverse as possible to encourage the model to perform well on novel objects. Indeed, the generalization of all different shapes and colors of the objects is our primary concern. Since we measure the success rate of an RL agent on the test set, a model that overfits the training dataset would perform poorly on the test set.

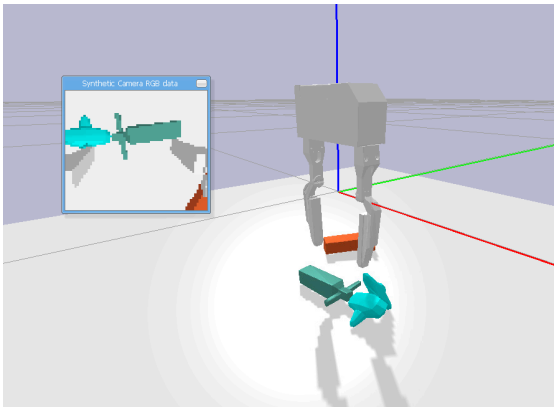
We use the random object database (6.5a) from pybullet-data [1] and the wooden-block datasets (6.5b) from Breyer et al. for our experiments. We trained only on the random object database. Wooden blocks dataset was only used to test the robustness of the trained model.

Unlike, Quillen et al. and Breyer et al. which divided their dataset only into test and training, we used a validation set in addition to them. We believe the inclusion of the validation set contributes to the generalization of the trained model. In total, test and validation sets have 150 objects each, and training sets consist of 700 objects.

[1] `https://bit.ly/3ijyCJM`

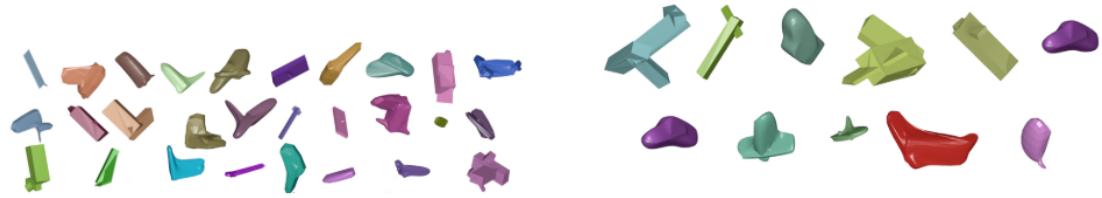


(a) ABB Yumi gripper used in Breyer et al.

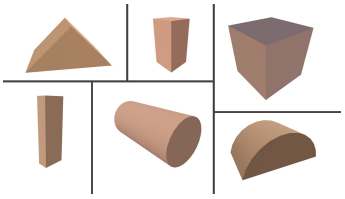


(b) WSG50 gripper used in our work

Figure 6.4: Comparison of robot models



(a) Random object dataset



(b) Wooden block dataset from Breyer et al. [4]

Figure 6.5: Different object datasets are used in our work

6.5 Data Acquisition

We conduct all our experiments solely in the Bullet simulation engine. RL training needs an extreme amount of data and training time. The shortest training time required for our experiments is not shorter than a day. In this respect, simulation provides a low-cost, robust, and scalable stream of data [17]. Although low-quality sensor measurements pose a threat to simulation-based learning techniques, domain-randomization or continuous training strategies seem to mitigate the disadvantages of simulation. Akin to our work Breyer et al. trained a table cleaning gripper, which trained solely in simulation and performed with a 78% success rate on the real robot [4]. On the other end of the spectrum, Kalashnikov et al. collected all the robot training data in four full months and 800 robot hours. They underlined the reality and quality of real-world collected data [11].

The data acquisition process can be dealt with with many different approaches. In our case, the most convenient choice was to lead the research entirely on simulation. We will explain more on the sim-to-real transfer and reality gap in the future work chapter.

6.6 Observation

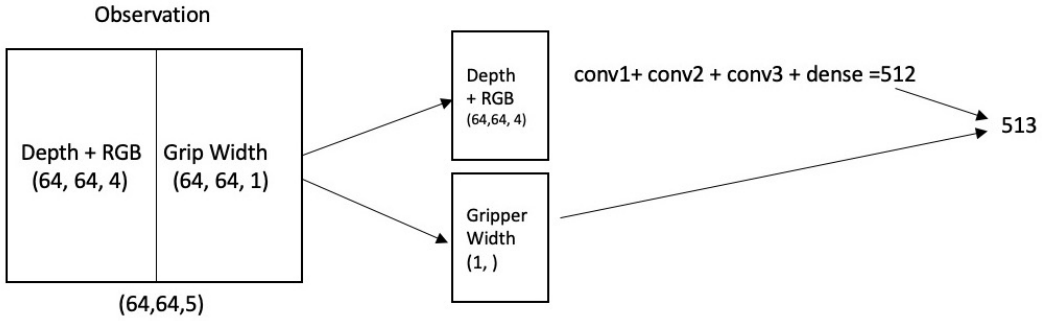
The environment processes the observation data and serves it to the agent after each timestep. The observation here refers to the agent's state in the Markov Decision Process. We implemented three different perception pipelines: Autoencoder, raw depth, and RGBD. The autoencoder setup is the same as Breyer et al., observation consists of the encoded RGBD sensor output. We use the autoencoder's latent space to encode the large RGBD sensor output with size 64x64x3 to 100x1. We collect 18000 training images and 2000 test images on a random agent working in the simplified environment description. We slightly change the Breyer et al.'s data collection parameters to get closer image shots of the objects.

We also implemented a perception layer which allows the direct computation of RGBD sensor output. This approach uses a similar perception layer setup as Mnih et al. We implemented two different types of non-encoded perception layers: RGBD sensor output with four separate channels and the depth sensor output with one channel.

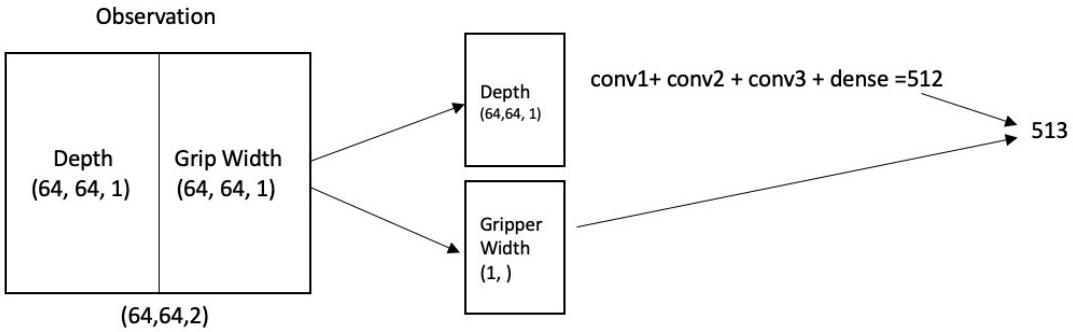
Processing the raw sensor observations demands more delicate handling. Typically, we need to input the sensor output (RGBD or only depth) to the convolutional neural network. However, the inclusion of the actuator width makes the processing more complicated. We ought to add the actuator width information to the sensor output without changing the sensor output's shape. In other words, we cannot flatten the sensor output and add the one-dimensional actuator width information. We need to pad the one-dimensional actuator width information into a three-dimensional array to comply with the sensor output (64x64x1 or 64x64x4). In the environment side, we truncate the padded actuator width information into the sensor output. Therefore,

the actuator width information allocates a dedicated channel at the end of the whole environment observation.

We parse the sensor output channels from the actuator width information again and input it to the agent's observation processing neural network. This neural network consists of three convolutional neural layers and a fully connected layer at the end. After processing the sensor data, we get a one-dimensional array with 512 elements, the hidden layer size of the last fully connected layer. Then, we concatenate the unpadded actuator width information to the 512 elements fully connected layer output. In the end, we squeeze the observation into a one-dimensional array with 513 elements. This resulting array will be further processed in the agent's neural network that approximates either the Q-values, policy, or both. The non-encoded observation process is explained with diagrams below 6.6.



(a) RGBD output combined with gripper width. Overall shape (64,64,5)



(b) Depth sensor output combined with gripper width. Overall shape (64,64,2)

Figure 6.6: Non-encoded observation processing layers

6.7 Reward Structure

As mentioned in the environment description section, we use different reward definitions for simplified and full environments. Simplified environment definition uses sparse reward, meaning when success, the agent receives 1 otherwise 0. The full environment definition handles the learning with a more sophisticated shaped reward function. Shaped reward function awards both the terminal state and sub-goals such as grasping and lifting the object, at the same time it penalizes every spent time before the terminal state. Rewards and time penalty are defined below.

$$\begin{aligned} \text{Terminal Reward: } r_t &= 10000 \\ \text{Grasping Reward: } r_g &= 100 \\ \Delta\text{h Scale Coefficient: } c &= 1000 \\ \text{Time Penalty: } r_{tp} &= 200 \end{aligned}$$

The shape reward function guides the robot to the terminal state by defining sub-goals. Otherwise, the agent may need to spend way too much time to explore the terminal state. In other words, reward function sets up curriculum-like learning objectives to facilitate the learning. The reward function is given in equation 6.1.

$$r = (\text{grasp detected}).(r_g + c.\Delta h) - r_{tp} \tag{6.1}$$

One needs to pay attention that each state except the terminal state, the step reward should be negative. This condition guarantees the maximum reward the agent receives is linked to reaching the terminal state as soon as possible. If a step reward is positive, the agent will try to stay at that state as long as it can to increase its reward, before going to the terminal. Thus, it contradicts our training goal: the best performing agent collects the objects as soon as possible. The timestep penalty should always be greater than the sum of grasp reward and lifting rewards to avoid the contradictory reward situation. This condition is formalized with the given inequality below. Note that the time penalty is a positive integer.

$$r_{tp} > r_g + c\Delta h \tag{6.2}$$

$$\text{Shaped Reward: } \begin{cases} \text{Grasp} : \begin{cases} \text{Non-Terminal} : r_g + c.\Delta h - r_{tp} \\ \text{Terminal} : r_t - r_{tp} \end{cases} \\ \text{No Grasp} : \begin{cases} \text{Non-Terminal} : -r_{tp} \\ \text{Terminal} = \text{Timeout} : -r_{tp} \end{cases} \end{cases} \tag{6.3}$$

6.8 Actions

There are two parts to the action definition. One part defines an interface with OpenAI Gym to let interact with the agent. The other part is the environment, receiving the action and processing it further, such as cropping it and scaling it. An example of an action space definition is given below 6.1.

Listing 6.1: OpenAI gym action space definition

```
if not self._discrete:
    self.action_space = gym.spaces.Box(-1.,
                                       1., shape=(3,), dtype=np.float32)
else:
    self.action_space = gym.spaces.Discrete(self.num_actions_pad*3)
```

It defines the action type and the shape of the action. This definition is later processed by the RL algorithm to output an action based on what the environment expects. Our environment takes only continuous actions. However, we linearly discretized the environment to comply with the DQN algorithm's discrete action space. Based on the discretization padding given in the configuration file, we define the size of the discrete action space. For instance, if the discretization padding is three for each action dimension, and if the simplified environment is set, we would have nine actions. Since the simplified environment has three actions dimension and three actions padding for each action makes up in a total of nine actions. The below diagram presents the linear action discretization 6.7.

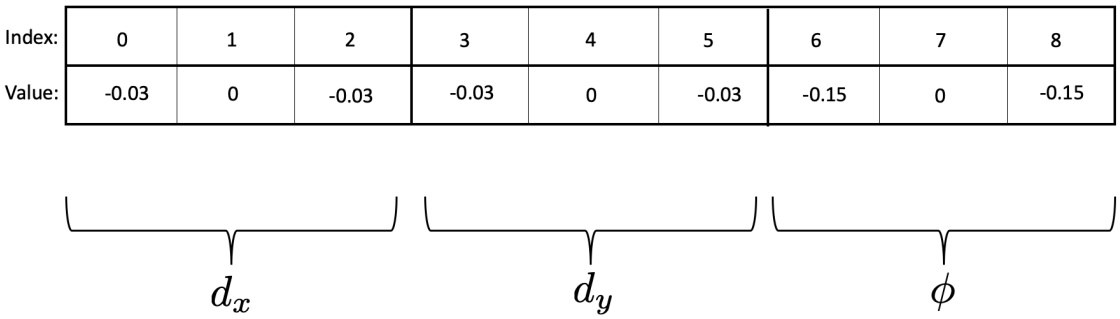


Figure 6.7: Linear action discretization representation. Padding one action dimenstion is 3 and total number of action dimension is 3

As mentioned in Stable-Baselines documentation [2] continuous actions algorithms work best when the action space is defined symmetric and between -1 and 1. Therefore, we need to rescale the actions in the environment side to fit the maximum action limits. This problem does not appear in the discrete action space because we already receive

[2] https://bit.ly/2PYe5hG

the action index from the DQN algorithm and can easily assign it to the respective values.

6.9 Training Structure

As described in the hardware setup section, we train on LRZ compute cloud mainly. Given that we cannot monitor the status of training always, we are obliged to implement a robust training structure to overcome the processes' unexpected shutdown. Also, in general, machine learning by nature is not always improving throughout the training process. The model performance might well get worse towards the end of the training. One way to avoid this problem is to implement checkpoints and evaluation callbacks to inspect the model performance regularly. Another way to achieve robust training is to monitor the training process and log the results to a CSV file. This way, we can overview the training process and judge where to end the training. We primarily monitor the training loss, success rate, and reward.

With the help of evaluation callback, we validate the model performance on a different instance of the gripper environment with the validation object set. We regularly save the best performing model based on average reward over ten episodes. This process assures that we will always have the best-generalized model. If the training unexpectedly ends or the loss increases, we would have a model to judge the performance. We repeat the training process evaluation every 50000 timesteps. Besides, we save the agent model every 25000 timesteps to assure a minimum loss in case of a sudden stop of the training process. The monitoring procedure is taken care of by the modified version of the Stable Baselines monitor wrapper. We extended the wrapper class[3] to save the timestep, success rate, and curriculum lambda in addition to reward and episode information.

Listing 6.2: Integration of monitor wrapper helper function

```
Monitor(gym.make('gripper-env-v0', config=config), "log_file")
```

Moreover, we implemented a parameterized training structure to allow users to try different configurations quickly. We parameterized input and reward normalization and time-feature wrapper.

[3] https://bit.ly/2CtoEX1

7 Expertimental Results

This section presents the experimental results of three different off-policy RL algorithms, BDQ, DQN, and SAC. Our initial goal by experimenting is to investigate the BDQ algorithm's performance compared to its predecessor DQN and current state-of-art algorithm SAC on robotics grasping tasks. We want to understand how sensitive is the BDQ algorithm to hyperparameters and how well it explores the given problem. Moreover, most importantly, does the BDQ algorithm scale to high-dimensional action spaces? We aim to gather valuable insights into which kind of RL algorithm performs best on robotics grasping operation.

We tested our BDQ algorithm implementation, based on a Stable Baselines project [1], in different scenes. All models are trained in the floor scene with a random-urdf object dataset. The RL agents' evaluation is based on the best performing model on the validation set in the floor scene. We examine the robustness of the models in the table scene.

Evaluation runs take hundred episodes long. The sequence, place, and shape of the objects are the same throughout different evaluation runs for different trials. In other words, all models are tested on identical conditions. An episode is considered successful when the gripper lifts an object to a predefined height.

We demonstrate the simplified environment's result and observe the differences between BDQ and DQN and compare it against SAC. Later, we test the same algorithms in the full environment description. We discern the scaling of the algorithms to a larger action space. Moreover, we looked at different perception pipelines' performance by varying the SAC algorithms input type from the encoder to RGBD or raw depth input.

We check the importance of each module on the algorithm's success rate in the ablation studies section. Ablation studies deliver the relative importance of curriculum strategy, normalization, actuator width, and reward to algorithm's overall performance. Finally, we provide specific failure cases for each algorithm.

7.1 Simplified Environment Results

All three algorithms successfully converged to a decent grasping policy in the simplified environment. BDQ and DQN were trained with varying action dimension padding 2, 4, 8, 16, and 33. We observed how the neural network size affects the success rate. Two different network sizes were tried on BDQ; a large network with two hidden layers

[1] `https://github.com/BarisYazici/bdq_sb/tree/master/stable_baselines/bdq`

in the shared module with 512 and 256 neurons each and 128 each for state and advantage function estimators, and a small network with two hidden layers in the shared representation with 64 neurons each and 32 neurons each for the state and advantage function estimators. Furthermore, we compared the BDQ performance with two different buffer sizes, fifty thousand and one million. We did not vary the DQN algorithm's hyperparameters. The default buffer size of fifty thousand was used for all trials of DQN. As for the SAC algorithm, we only changed the input from the encoder to depth in the simplified environment.

All BDQ trials converged the same success rate during training except the big network variant 7.2b. BDQ with four pads achieved the best test success rate in the floor environment, with 94%. BDQ with 16 pads came second with a 93% success rate. However, they switched sides in the table scene, where BDQ 16 pads achieved 91%, BDQ four pads stuck at an 89% success rate. We noticed the significant performance difference between the big neural network and the small one in the figure 7.1a. The small network achieves far more notable results, where the big network failed to learn the simplified environment.

DQN algorithm succeeded in learning the simplified environment. However, its top performance lagged significantly behind the BDQ algorithm with a 74% success in the floor scene and a 63% success in table scene. We observed the deteriorated results with an increase in action space dimension. The results are depicted in the figure 7.2a.

SAC algorithm achieved the best training results converged to the highest success rate. It achieved a 97% success rate in the floor scene and an 87% success rate in the table scene. Interestingly, BDQ 16 pads surpassed SAC's performance with a 91% success in the table scene. However, as shown in figure 7.3a, SAC converged to a higher success rate than the BDQ during training.

	BDQ Simplified Scenario			
	Floor Scene		**Table Scene**	
Models	**Training Set (%)**	**Test Set (%)**	**Training Set (%)**	**Test Set (%)**
BDQ_33pads_big	34	29	13	12
BDQ_33pads_small	90	82	86	79
BDQ_16pads_small	89	93	90	91
BDQ_8pads_small	91	90	77	72
BDQ_4_pads	96	94	92	89

Table 7.1: BDQ algorithm's result in the simplified environment

7.2 Full Environment Results

In the full environment description, we generally evaluated the performance of variants of the SAC algorithm. We tested with different perception pipeline, different reward

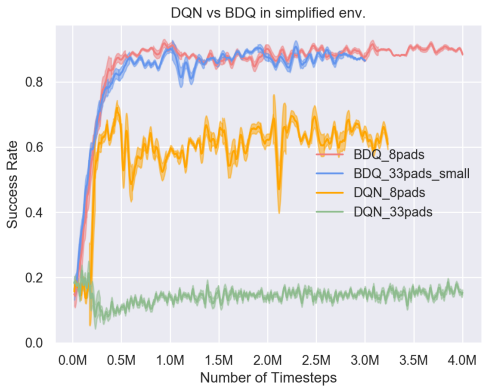
	DQN Simplified Scenario			
	Floor Scene		Table Scene	
Models	**Training Set (%)**	**Test Set (%)**	**Training Set (%)**	**Test Set (%)**
DQN_33pads	6	6	6	7
DQN_16pads	12	6	9	14
DQN_8pads	75	65	38	40
DQN_4pads	75	73	1	1
DQN_2pads	76	74	55	63

Table 7.2: DQN algorithm's result in the simplified environment

	SAC Simplified Scenario			
	Floor Scene		Table Scene	
Models	**Training Set (%)**	**Test Set (%)**	**Training Set (%)**	**Test Set (%)**
SAC_encoder	98	97	84	87
SAC_depth	91	96	28	24

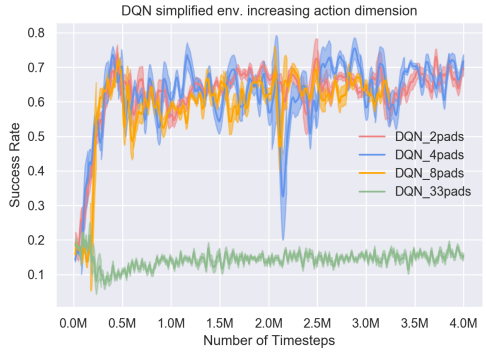
Table 7.3: SAC algorithm's result in the simplified environment

(a) BDQ performance with big and small network sizes



(b) BDQ vs DQN performance with increasing action space. Shaded area shows the standard deviation

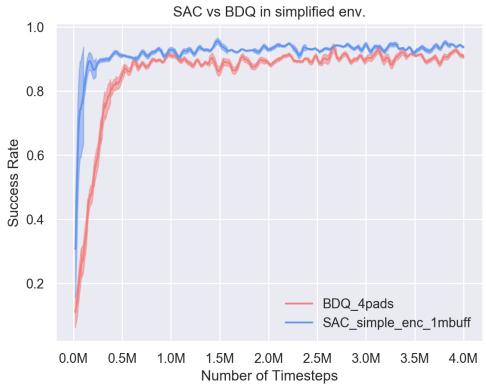
Figure 7.1: BDQ network size comparison and its performance against DQN with increasing action space dimension in simplified environment



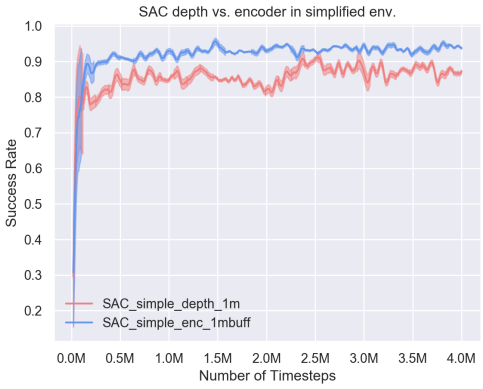
(a) DQN performance with increasing action space dimension.

(b) BDQ performance with increasing action space dimension.

Figure 7.2: BDQ and DQN performances in simplified environment with increasing action space dimension. Shaded areas show the standard deviation



(a) Best BDQ model compared to the best SAC model in simplified environment



(b) SAC depth compared to encoder perception in simplified environment

Figure 7.3: SAC results in simplified environment

definition, and buffer size. Contrary to the simplified environment, we verified the algorithm's robustness on the wooden block object database.

We tested four variations of the SAC model in the full environment: encoder with fifty thousand buffer size, encoder with one million buffer size, depth with one million buffer size, and RGBD with one million buffer size. Among these models, the encoder one million buffer achieved the best transfer result. It performed with 99% and 82% with random objects and wooden blocks in the table scene 7.4. Nevertheless, training plots demonstrate that the encoder training process has more variance than the depth counterpart. Moreover, depth perception converged to a higher success rate than encoder perception during training, as shown in the plot 7.4a.

Like the simplified environment results, we also found that in the full environment, one million buffer size performs better and accumulates less variance than the fifty thousand buffer. As represented in the plot 7.4b, the fifty thousand buffer version had more variance and converged to only a 90% training success rate. In comparison, the one million buffer size version accumulated less variance and converged around a 97% success rate.

Unexpectedly, depth perception also performed better than the RGBD. RGBD perception delivered the worst results among perception types. It converged to a 91% test success rate even in the floor scene. In the same scene, depth perception achieved a 100% test success rate. RGBD plot in figure 7.6 also shows sudden changes, a sign of high variance. Nevertheless, it still converged to a 99% success rate.

Unfortunately, BDQ never delivered a working policy for the full environment. Its learning plot 7.5 also shows the severe decay to a 0% success rate.

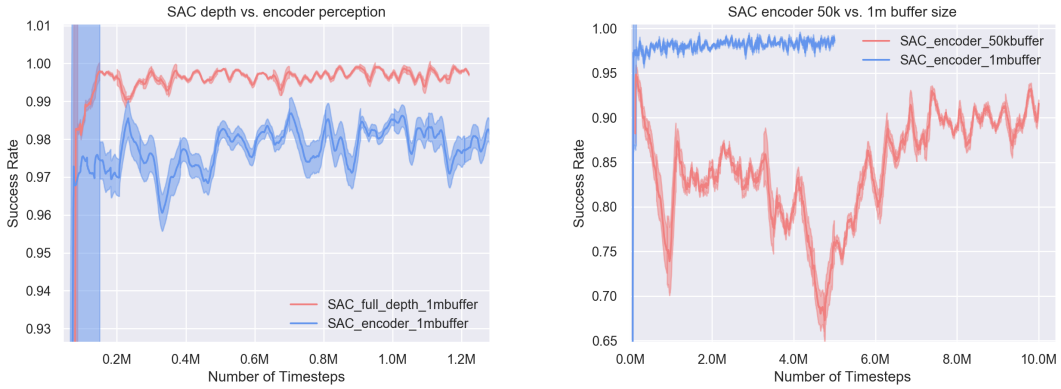
	SAC Full Environment			
	Floor Scene		**Table Scene**	
Models	**Random Obj. (%)**	**Wooden Obj. (%)**	**Random Obj. (%)**	**Wooden Obj. (%)**
SAC_encoder_50k	65	62	63	59
SAC_encoder_1m	100	95	99	82
SAC_depth	100	95	95	23
SAC_rgbd	91	95	46	17
SAC_depth_sparse	99	97	100	72
SAC_depth_no_curr	100	97	97	64
SAC_depth_no_act	95	75	84	24

Table 7.4: SAC Full environment results

7.3 Ablation Studies

In ablation studies, we investigated the individual importance of the following modules: curriculum strategy, input and reward normalization, actuator-width information, and

	SAC Full Environment				
	Floor Scene		Table Scene		
Models	**Random Objects (%)**	**Wooden Blocks (%)**	**Random Objects (%)**	**KUKA Robot Random Objects (%)**	**W**
SAC_encoder_50k	65	62	63	31	5
SAC_encoder_1m	100	95	99	87%	8
SAC_depth	100	95	95	97%	2
SAC_rgbd	91	95	46	38%	1
SAC_depth_no_curr	100	97	97	90%	6
SAC_depth_sparse	99	97	100	28%	7
SAC_depth_no_act	95	75	84	12	2



(a) SAC performance depth versus encoder perception

(b) SAC performance with encoder perception 50k vs 1m buffer

Figure 7.4: SAC performance comparison of perception pipelines and buffer size

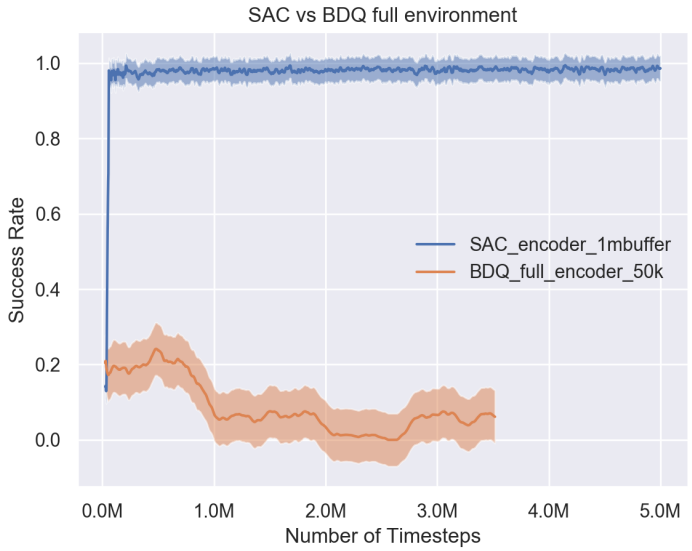


Figure 7.5: SAC compared to BDQ in full environment

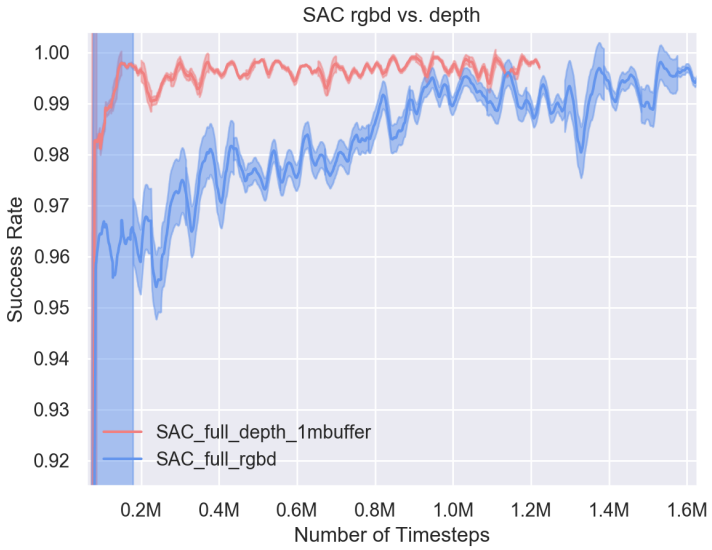


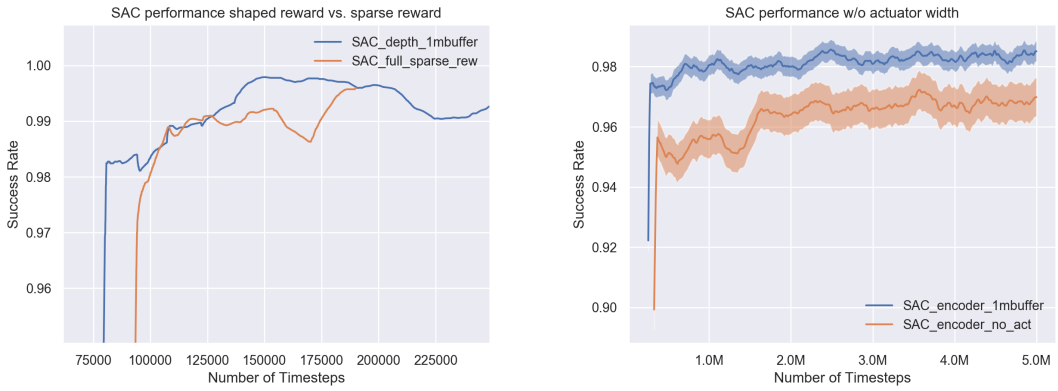
Figure 7.6: SAC performance RGBD vs. Depth. Shaded region represents the standard deviation.

the shaped reward. All ablation study experiments were carried with the SAC algorithm with depth perception and one million experience replay buffer size. All training trials took place in the full environment floor scene. We shared the results of ablation studies in the same table as the SAC full environment 7.4.

Among all ablation variants, only no normalization trial did not provide a working grasp policy. The rest either converged to a lower success rate or converged relatively slower than the baseline.

Sparse reward and no curriculum learning converged 20 thousand timesteps and 400 thousand timesteps after the baselines. Interestingly, sparse reward SAC agent performed the best in the table scene, on random objects with a 100% success rate. Both sparse and no curriculum learning trials performed better than the baseline SAC model on wooden blocks in the table scene. Sparse reward and no curriculum learning achieved 72% and 64%, while baseline reached 23%.

No actuator width observation converged to a slightly worse success rate than the baseline. It achieved a 97% success rate, while the baseline converged to 99%. An extra observation related to the environment proved to be useful.



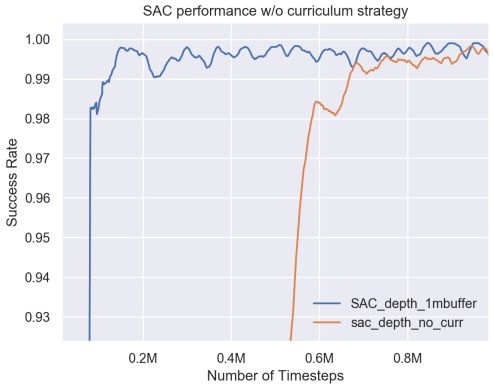
(a) SAC performance with shaped reward vs. sparse reward

(b) SAC performance with encoder perception without actuator width observation

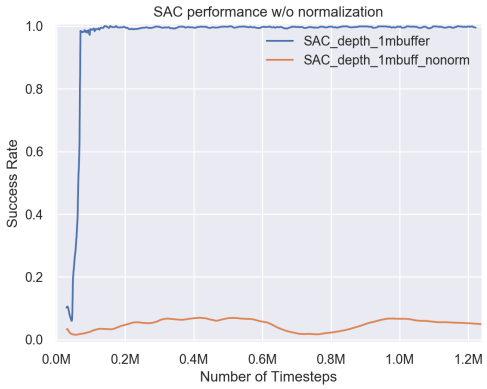
Figure 7.7: Ablation of reward function and actuator width observation

7.4 Failure Modes

1. **Misinterpretation of depth** - One of the most common failure cases of the encoder perception pipeline is the misinterpretation of the depth. As shown in the figures at 7.9, the gripper orients itself correctly and position just a bit higher than the targeted object. After positioning, it attempts to grasp the object but fails because of the height difference between the object and gripper. This scenario occurs more frequently in the test dataset than the training set. The agent can overcome this failure case by understanding that the encoder is yielding wrong information,



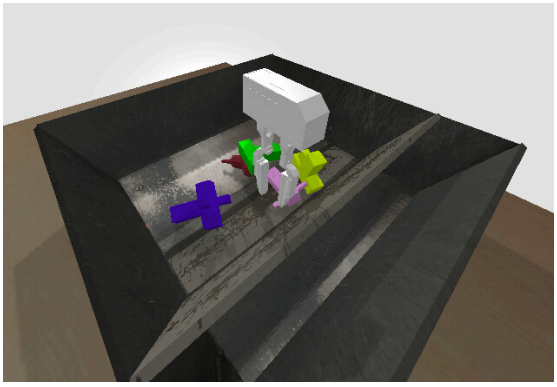
(a) SAC performance withuot curriculum strategy



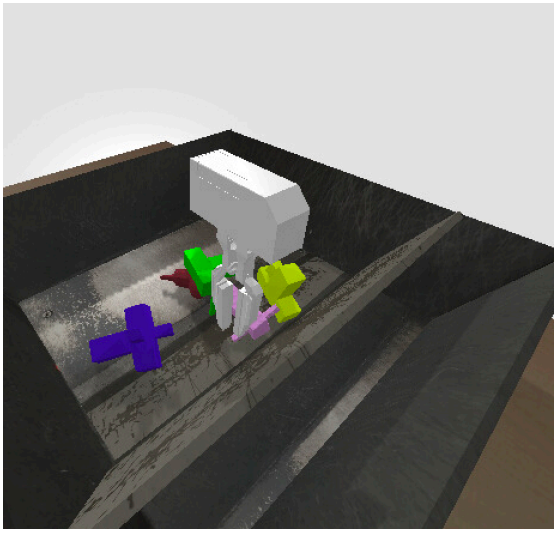
(b) SAC performance in full environment without normalization compared to the performance of the baseline SAC

Figure 7.8: Ablation of curriculum learning and normalization of input and reward

and it needs to go further down to grasp the object. However, since it does not know the test objects, it tries the standard grasp policy but fails because of the discrepancy between encoded depth and real depth. That is the reason why the encoder converged to a lower success rate than the depth.



(a) Gripper positions itself above the pink object



(b) Gripper attempts to grasp the pink object unsuccessfully

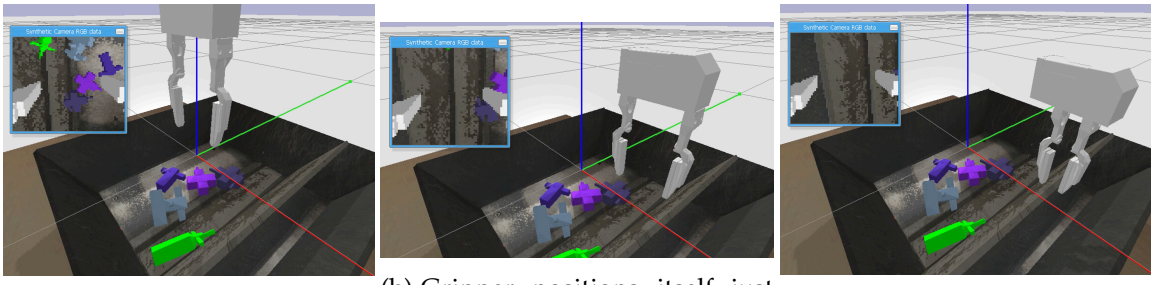
Figure 7.9: Depth misinterpretation of pink object in the full environment

2. **Grasping the tray's edge** - RGBD or Depth observation types perceive the tray's edge as a graspable object. This misjudgment accounts for up to 50% of the failed

grasp attempts in the table scene 7.4 7.10. We realized that the failure percentage increases when we start the gripper from a higher start point.

Both observation types tackled the height problem that appeared in the encoder problem. The online nature of these observation types corrected itself when a depth discrepancy occurs. However, it seems that these online observation methods encode the graspable objects' features very similar to other household objects. This characteristic can be interpreted both positively and negatively. It is positive because the perception layer extrapolates the graspable features correctly from the trained objects and applies it to the tray object. It can be viewed negatively because the tray is not our target object; eventually, the gripper should differentiate between the targeted objects and the extra tray object.

This failure mode can be solved by training the trained model for a short time in the table scene. This procedure should suffice for the agent to understand that tray is not the targeted object.



(a) Gripper starts the movement to the tray's edge

(b) Gripper positions itself just above the tray's edge

(c) Gripper grasps the tray's edge

Figure 7.10: Sequence of movement to the tray's edge

8 Conclusion & Future Work

8.1 Conclusion

Experiment results successfully demonstrated that the BDQ algorithm could scale to high dimensional action spaces without losing performance [2]. Whereas, DQN suffered from the increased action space dimension 7.1b. Apart from scaling, BDQ showed increased robustness than DQN to changing the environment. The best BDQ model achieved a 91% success rate in the simplified environment in the table scene. This performance proves that BDQ smoothly transfers to a different scene where it was not trained. Therefore, we conclude that BDQ can generalize well to different scenes and objects. Generalization is not the case for the DQN model; its performance deteriorates in different scenes. DQN results table (7.2) shows that DQN with four pads of discretization stayed at a 1% success rate in the table scene. In the original floor scene where the training was conducted, it achieved a 76% success rate.

We tested the BDQ algorithm with two different network sizes. The small network achieved significantly better performance than the bigger network. Smaller BDQ network with 33 action pads performed with 82% success rate, whereas the bigger network with the same configuration showed poor performance with 29%.

Although high variance and hyperparameter sensitiveness exist, both Q-function based methods successfully converged to a decent policy for object grasping in the simplified task. On the other hand, both algorithms were incapable of learning the full environment description. We believe this is due to the naïve exploration strategy and limits of value-based methods. Maximum entropy RL inherently explores better than epsilon-greedy or Gaussian noise exploration techniques. Also, the actor-critic methods have higher success potential than value-based methods [31].

Baseline SAC algorithm performed slightly better than the best BDQ model, with a 97% success rate in the floor scene. However, we noticed an apparent decay in SAC's performance in the table scene with an 87% success rate. Generally, SAC with depth perception performed poorly in the table scene. These results explain that BDQ can generalize better to new scenes.

The most noteworthy results come from the SAC algorithm. To the best of our knowledge, the SAC algorithm surpassed all previously tested algorithms in robotics grasping the research in simulation. Our best SAC model converged to a 100% success rate in a cluttered environment. While prior works QT-Opt, Breyer et al. only achieved 96% and 98% success rate [11], [15].

SAC algorithm showed unprecedented robustness to varying scenes and objects. It successfully grasped 97% of wooden blocks, which is an entirely different dataset than

the training dataset. Moreover, it converges to its top performance only after hundred thousand timesteps, about 2-3 hours of training time 7.4a. In contrast, Breyer et al.'s TRPO algorithm converged to its peak performance at around six million-time steps. Three main factors account for this improved performance: off-policy updates, normalization, and perception layer.

Firstly, the SAC algorithm utilizes off-policy updates for both policy and value function estimation. On the contrary, TRPO is an on-policy algorithm that only uses the current batch of experiences for actor-critic updates. Off-policy algorithms usually achieve greater data efficiency than on-policy RL algorithms.

Secondly, in our experiments, we normalized both the observation and reward. Normalization largely accounts for the success of the SAC in the full environment. We found reward normalization the most vital part of our experiments, especially when the shaped reward is used. Ablation studies showed that SAC fails to learn a working grasping policy without normalization 7.8b.

Thirdly, Breyer et al.'s sophisticated autoencoder perception layer provides faster convergence. However, the more straightforward raw depth observation converges to a better success rate with improved robustness 7.4a. We believe the difference between encoder and depth lies in the depth interpretation loss. Autoencoders compress the observation onto a latent-space. This compression causes the agent to misinterpret the depth of the objects. Indeed, in the failure modes section 7.4, we noted down the repetitive failure of grasping an object because of wrong depth interpretation. On the other hand, the depth perception layer is an online method; therefore, it corrects its network weights when a wrong interpretation occurs. The online perception layer also contributes to the end-to-end nature of the RL algorithm. Where our depth perception layer's weights are updated to deliver better-grasping policy, autoencoder's weights are immutable throughout learning.

Another important take off from the ablation studies is the role of curriculum strategy and shaped reward function. We tested in the full environment both with sparse reward function and without curriculum strategy and compared it against the baseline SAC. We noticed that both curriculum and shaped reward speed up the convergence. However, the location of the converged performance does not change immensely. For instance, baseline SAC performed with a 100% success rate on random objects dataset in the floor scene, where without curriculum and shaped reward, it performed 100% and 99%, respectively. We see a similar situation in the table scene, baseline converges to 95% success rate, while without curriculum or shaped reward converge to 97% and 100%, respectively. These results show us that a robust algorithm with a strong perception pipeline and normalization ought to learn without additional modifications. The results also show a strong correlation with the theoretical basis of RL described in Introduction to Reinforcement Learning book from Sutton [19]. Sutton comments that reward engineering gives no gain over binary reward. Although we note down the redundancy of additional modifications, both reward engineering and curriculum learning still useful for speeding up the learning.

As represented in 7.7b without actuator width information, the grasping success rate converged to a lower point. It is essential to note the importance of actuator width observation. The agent evaluates the grasp quality through this information. If the robot cannot close its gripper or the gripper closes fully, then the agent deducts that it is probably not a good grasp. Thus we can conclude that extra observation regarding the environment increases the peak success rate of an agent.

8.2 Future Work

8.2.1 Transfer to Hardware

In this report, we only tested RL models in simulation environments. Transferring the simulation models to hardware poses new challenges such as sensor noise or robot calibration. Despite the hardware challenges, some prior works achieved the transfer of grasping policy from simulation to hardware. The most notable example comes from Breyer et al.; they applied their best performing simulation model on hardware and delivered a 78% success rate without any fine-tuning or domain randomization [4]. Klashnikov et al. achieved up to an 88% success rate in real word grasping evaluation [11]. Our state-of-art SAC model surpassed the result of both prior works in simulation, a natural future extension would be to attempt the sim-2-real transfer.

8.2.2 Soft Entropy Maximization RL extension of BDQ

BDQ performed head-to-head compared to the state of the art SAC algorithm in the simplified environment. However, it performed poorly in the full environment setting. The reason behind this failure is insufficient exploration capability. We believe the advantage of the SAC algorithm over BDQ is the maximum entropy RL setting, which provides noble exploration strategies. BDQ algorithm can be extended to adopt maximum entropy RL definition to deliver better exploration strategies.

8.2.3 Multi-Agent Robotic Grasping System

Industrial robots often work multiple at a given problem. Either welding robots or the montage robots work dual or more. Soon the household robots will also work multiple under challenging scenarios. Therefore, an extension to a multi-agent RL framework can bring significantly more degrees of freedom to a robot.

8.2.4 Automatic Learning of Curriculum Parameters

As mentioned in the conclusion section, the curriculum strategy provides a faster convergence to the top performance. At the same time, curriculum strategy brings new hyperparameters to be tuned. Hyperparameter tuning can be time-consuming and complicated. Therefore, we can treat the curriculum parameters as another objective

variables to be solved for the optimum values during training. This approach could produce a more remarkable performance without losing time for extra hyperparameter tuning.

8.2.5 Extension to Soft Objects

In this work, we only considered rigid objects grasping. However, in our daily life, we grasp and manipulate mostly soft or deformable objects such as a plastic water bottle, cloth manipulation, cable insertion, and toy manipulation. The manipulation of those objects differs from their rigid counterparts. They practically demand a more robust algorithm to account for the unexpected changes on the object's surface. With the extension of soft objects, grasp policy would be more realistic.

List of Figures

List of Tables

Bibliography

[1] O. Kroemer, S. Niekum, and G. Konidaris. "A Review of Robot Learning for Manipulation: Challenges, Representations, and Algorithms". In: (2019). arXiv: 1907.03146. URL: http://arxiv.org/abs/1907.03146.

[2] A. Tavakoli, F. Pardo, and P. Kormushev. "Action branching architectures for deep reinforcement learning". In: *32nd AAAI Conference on Artificial Intelligence, AAAI 2018* (2018), pp. 4131–4138. arXiv: 1711.08946.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[4] M. Breyer, F. Furrer, T. Novkovic, R. Siegwart, and J. Nieto. "Comparing Task Simplifications to Learn Closed-Loop Object Picking Using Deep Reinforcement Learning". In: *IEEE Robotics and Automation Letters* 4.2 (Mar. 2018), pp. 1549–1556. ISSN: 23773766. DOI: 10.1109/LRA.2019.2896467. arXiv: 1803.04996. URL: http://arxiv.org/abs/1803.04996.

[5] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. *Stable Baselines*. https://github.com/hill-a/stable-baselines. 2018.

[6] A. Sahbani, S. El-Khoury, and P. Bidaud. "An overview of 3D object grasp synthesis algorithms". In: *Robotics and Autonomous Systems* 60.3 (2012), pp. 326–336. ISSN: 09218890. DOI: 10.1016/j.robot.2011.07.016. URL: http://dx.doi.org/10.1016/j.robot.2011.07.016.

[7] V.-d. Nguyen. "Constructing Stable Grasps in 3D". In: *Fortune* (1987), pp. 234–239.

[8] P. Schmidt, N. Vahrenkamp, M. Wachter, and T. Asfour. "Grasping of Unknown Objects Using Deep Convolutional Neural Networks Based on Depth Images". In: *Proceedings - IEEE International Conference on Robotics and Automation* (2018), pp. 6831–6838. ISSN: 10504729. DOI: 10.1109/ICRA.2018.8463204.

[9] S. Ekvall and D. Kragic. "Interactive grasp learning based on human demonstration". In: *Proceedings - IEEE International Conference on Robotics and Automation* 2004.4 (2004), pp. 3519–3524. ISSN: 10504729. DOI: 10.1109/robot.2004.1308798.

[10] A. Saxena, J. Driemeyer, and A. Y. Ng. "Robotic grasping of novel objects using vision". In: *International Journal of Robotics Research* 27.2 (2008), pp. 157–173. ISSN: 02783649. DOI: 10.1177/0278364907087172.

[11] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine. "QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation". In: CoRL (2018), pp. 1–23. arXiv: 1806.10293. URL: http://arxiv.org/abs/1806.10293.

[12] O. A. M. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba. "Learning dexterous in-hand manipulation". In: *International Journal of Robotics Research* 39.1 (2020), pp. 3–20. ISSN: 17413176. DOI: 10.1177/0278364919887447. arXiv: 1808.00177.

[13] S. Caldera, A. Rassau, and D. Chai. "Review of deep learning methods in robotic grasp detection". In: *Multimodal Technologies and Interaction* 2.3 (2018). ISSN: 24144088. DOI: 10.3390/mti2030057.

[14] I. Lenz, H. Lee, and A. Saxena. "Deep Learning for Detecting Robotic Grasps". In: (Jan. 2013). arXiv: 1301.3592. URL: http://arxiv.org/abs/1301.3592.

[15] D. Quillen, E. Jang, O. Nachum, C. Finn, J. Ibarz, and S. Levine. "Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods". In: *Proceedings - IEEE International Conference on Robotics and Automation* (2018), pp. 6284–6291. ISSN: 10504729. DOI: 10.1109/ICRA.2018.8461039. arXiv: 1802.10264.

[16] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. "Domain randomization for transferring deep neural networks from simulation to the real world". In: *IEEE International Conference on Intelligent Robots and Systems* 2017-Septe (2017), pp. 23–30. ISSN: 21530866. DOI: 10.1109/IROS.2017.8202133. arXiv: 1703.06907.

[17] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang. "Solving Rubik's Cube with a Robot Hand". In: *arXiv preprint* (2019).

[18] E. L. Thorndike. "Animal Intelligence: Experimental Studies." In: (1911).

[19] R. S. Sutton and A. G. Barto. *Reinforcement Learning, Second Edition: An Introduction - Complete Draft*. 2018, pp. 1–3. ISBN: 9780262039246.

[20] T. Lozano-Pérez and L. Kaelbling. "6.825 Techniques in Artificial Intelligence (SMA 5504)". Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. Fall 2002. URL: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-825-techniques-in-artificial-intelligence-sma-5504-fall-2002/index.htm#.

[21] P. A. Gagniuc. *From Theory to Implementation and Experimentation*. Wiley, 2017. ISBN: 9781119387572.

[22] R. Bellman. "Dynamic programming and stochastic control processes". In: *Information and Control* 1.3 (1958), pp. 228–239. ISSN: 00199958. DOI: 10.1016/S0019-9958(58)80003-0.

[23] J. Achiam. "Spinning Up in Deep Reinforcement Learning". In: (2018).

[24] V. Mnih, D. Silver, and M. Riedmiller. "Deep Q Network (Google)". In: (), pp. 1–9.

[25] L.-J. Lin. "Reinforcement learning for robots using neural networks". In: *PhD Thesis* (1993), p. 160. URL: https://search.proquest.com/docview/303995826?accountid=12063%7B%5C%%7D0Ahttp://fg2fy8yh7d.search.serialssolutions.com/directLink?%7B%5C&%7Datitle=Reinforcement+learning+for+robots+using+neural+networks%7B%5C&%7Dauthor=Lin%7B%5C%%7D2C+Long-Ji%7B%5C&%7Dissn=%7B%5C&%7Dtitle=Reinforcement+learning+for+robots+us.

[26] E. Todorov, T. Erez, and Y. Tassa. "MuJoCo: A physics engine for model-based control". In: *IEEE International Conference on Intelligent Robots and Systems* (2012), pp. 5026–5033. ISSN: 21530858. DOI: 10.1109/IROS.2012.6386109.

[27] T. Erez, Y. Tassa, and E. Todorov. "Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX". In: *Proceedings - IEEE International Conference on Robotics and Automation* 2015-June.June (2015), pp. 4397–4404. ISSN: 10504729. DOI: 10.1109/ICRA.2015.7139807.

[28] L. Pitonakova, M. Giuliani, A. Pipe, and A. Winfield. "Feature and performance comparison of the V-REP, Gazebo and ARGoS robot simulators". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10965 LNAI (2018), pp. 357–368. ISSN: 16113349. DOI: 10.1007/978-3-319-96728-8_30.

[29] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *35th International Conference on Machine Learning, ICML 2018* 5 (2018), pp. 2976–2989. arXiv: arXiv:1801.01290v2.

[30] V. R. Konda and J. N. Tsitsiklis. "Actor-critic algorithms". In: *Advances in Neural Information Processing Systems* (2000), pp. 1008–1014. ISSN: 10495258.

[31] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. "Continuous control with deep reinforcement learning". In: *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings* (2016). arXiv: 1509.02971.

[32] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Frcitas. "Dueling Network Architectures for Deep Reinforcement Learning". In: *33rd International Conference on Machine Learning, ICML 2016* 4.9 (2016), pp. 2939–2947. arXiv: 1511.06581.

[33] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: A system for large-scale machine learning". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016* (2016), pp. 265–283. arXiv: 1605.08695.

[34] F. Chollet et al. *Keras*. https://keras.io. 2015.

[35] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala. "PyTorch Distributed: Experiences on Accelerating Data Parallel Training". In: (2020). arXiv: 2006.15704. URL: http://arxiv.org/abs/2006.15704.

[36] H. He. "The State of Machine Learning Frameworks in 2019". In: *The Gradient* (2019).

[37] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann. *Stable Baselines3*. https://github.com/DLR-RM/stable-baselines3. 2019.

[38] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: (2019).

[39] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, M. Hardt, B. Recht, and A. Talwalkar. "A System for Massively Parallel Hyperparameter Tuning". In: (2018). arXiv: 1810.05934. URL: http://arxiv.org/abs/1810.05934.

[40] D. Schlegel. "Deep Machine Learning on GPUs". In: *Seminar Talk-Deep Machine Learning on Gpus* (2015), p. 1. URL: http://yann.lecun.com/exdb/mnist/.

[41] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. "Curriculum learning". In: *ACM International Conference Proceeding Series* 382.January 2009 (2009). DOI: 10.1145/1553374.1553380.

[42] L. J. Elman. "Learning and development in neural networks - the importance of starting small". In: *Cognition* 38.2 (1993), pp. 71–99. ISSN: 00100277. DOI: 10.1016/S0010-0277(02)00106-3.

[43] T. D. Sanger. "Gradually Increasing Task Difficulty". In: *IEEE Transactions on Robotics* 10.3 (1994).

[44] A. Karpathy and M. Van De Panne. "Curriculum learning for motor skills". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2012. ISBN: 9783642303524. DOI: 10.1007/978-3-642-30353-1_31.