TEL AVIV UNIVERSITY
The Iby and Aladar Fleischman Faculty of Engineering
The Zandman-Slaner School of Graduate Studies

LEARNING TO THROW WITH A HANDFUL OF SAMPLES USING DECISION TRANSFORMERS

A thesis submitted toward the degree of
Master of Science in Engineering

by

Maxim Monastirsky

October 2022

TEL AVIV UNIVERSITY
The Iby and Aladar Fleischman Faculty of Engineering
The Zandman-Slaner School of Graduate Studies

LEARNING TO THROW WITH A HANDFUL OF SAMPLES USING DECISION TRANSFORMERS

A thesis submitted toward the degree of
Master of Science in Engineering

by

Maxim Monastirsky

This research was carried out at Tel Aviv University
in the School of Mechanical Engineering
Faculty of Engineering
under the supervision of Dr. Avishai Sintov

October 2022

Acknowledgments

First of all I would like to acknowledge and give my warmest thanks to Dr. Avishai Sintov, for your support and guidance along the way, for your pursue of the new and unknown. Your enabling approach is essential for true innovation.

I would also like to thank my dearest friends and co-workers: Inbar Ben-David, Osher Azulay, Nadav Kahanowich, Anton Gurevich, Eran Bamani, Itamar Mishani, Alon Mizrahi, Nimrod Curtis and Alon Laron, for being there in hard times and good times, providing different perspectives and a laugh or two.

Thanks to my parents Natalia and Evgeny Monastirsky, for providing me the tools I use today and supporting me along the way. Yael Bar-Eli, my life partner, for always being there. And lastly, I want to thank my cat Tubi, for loving me regardless.

Abstract

Throwing objects by a robot extends its reach and has many industrial applications; providing better efficiency to many tasks such as packaging in warehouses, object transfer, conveyor belt management and recycling. While analytical models can provide efficient performance, they require accurate estimation of system parameters. Reinforcement Learning (RL) algorithms can provide an accurate throwing policy without prior knowledge. However, they require an extensive amount of real world samples which may be time consuming and, most importantly, pose danger. Training in simulation, on the other hand, would most likely result in poor performance on the real robot.

In this work, we explore the use of Decision Transformers (DT) and their ability to transfer from a simulation-based policy into the real-world. Contrary to RL, we re-frame the problem as sequence modelling and train a DT by supervised learning. The DT is trained off-line on data collected from a far-from-reality simulation through random actions without any prior knowledge on how to throw. Then, the DT is fine-tuned on an handful (~ 5) of real throws. Results on various objects show accurate throws reaching an error of approximately 4cm. Also, the DT can extrapolate and accurately throw to goals that are out-of-distribution to the training data. We additionally show that few expert throw samples, and no pre-training in simulation, are sufficient for training an accurate policy.

The work in this thesis was submitted for publication in the *IEEE Robotics and Automation Letters*, September 2022.

Table of Contents

List of Figures

List of Tables

1 Introduction

The ability of a robot to accurately throw an object to a desired target can provide better efficiency to many tasks such as packaging in warehouses, object transfer, conveyor belt management and recycling [33]. By throwing an object, the robot utilizes its dynamics for extending its reach. The robot can place objects in boxes or bins positioned in farther region without the need to physically reach them.

The throwing problem has been addressed in several analytical approaches where system models and parameter tuning are required [41, 39]. In contrary, not much work has been carried out using machine learning approaches. Nevertheless, recent work combined a physics-engine simulation with a regression network trained by real world throws [49]. While the method achieved results outperforming human throws, it requires an extensive amount of real world throws and some prior how to throw. Similarly, Reinforcement Learning (RL) applications for throwing requires a significant amount of real-world samples in order to demonstrate sufficient performance. Consequently, only few demonstrated such capabilities in limited scenarios [20].

RL has been successful in many complex simulated tasks including Atari video games [26] and physics-engine environments [24, 9] where the data is acquired at a lower cost [50]. However, training RL policies on real robots is a tedious and time consuming task [25]. Hence, the lack of extensive RL work on the object-throwing problem can be explained by the logistic requirement for a large amount of real throws and a reset mechanism to facilitate the collection. The robot may be required to work for a very long time. More important, the robot has no prior on how to throw and random actions may pose danger and cause damage. Simulation-based learning, on the other hand, provides a safe and cost-effective way to collect data through interactions with the environment. However, simulations rarely capture reality and the trained policies are usually poorly transferred [29].

Classic online RL algorithms require to continuously apply and update the current pol-

icy on the robot and collect on-policy data. This online setting is usually time consuming and very sample inefficient. Hence, it may be practical in a simulation environment, but many real robotic applications do not have the privilege of applying such online training procedures. On the other hand, offline RL algorithms, such as Deep Q-Learning and Deep Deterministic Policy Gradient (DDPG) [24, 1], require data that is correlated to the distribution of the current policy and are unable to extrapolate to new out-of-distribution scenarios [10].

In recent years, significant advancements in natural language processing and vision have been credited to the development of the Transformer architecture [44, 8]. In particular, an autoregressive model termed Generative Pre-trained Transformer (GPT) [31] is responsible for significant breakthrough in text-to-image [32] and language models [7]. The Transformers were recently taken to the world of RL in the form of *Decision Transformers* (DT) [5]. In DT, RL is considered as sequence modeling problem while completely eliminating the need for bootstrapping for long term credit assignment typically done in temporal-difference learning. Hence, a policy can be learned from offline and off-policy examples. Experiments in simulated environments have shown some capability for out-of-distribution learning [32]. Yet and to the best of the authors' knowledge, DT has not been evaluated on real world robots and sim-to-real transfer.

In this work, we investigate the use of DT for multi-goal object throwing with a robotic arm and its ability to reduce the required number of real-world samples. In particular, we are interested in investigating the sim-to-real ability of DT to generalize from a simulation-based model to the real world. Unlike prior work, no prior knowledge on how to throw nor a particular object release time are given to the simulation. Consequently, the sequence of actions for a throw motion is fully learned and is not of constant length. In addition, the simulation is not required to be tuned to the dynamic parameters of the real system and arbitrary values can be used. By using a simulation, the model is able to explore various motions without any risk. We also observe the augmentation of the simulated trajectories for data efficiency by using the Hindsight Experience Replay (HER) [2]. Then, only a handful of off-line recorded real throws is required in order to fine-tune the model yielding accurate real-world performance. The training data recorded off-line from simulation via random actions is shown to be of a short range distribution. However, the model exhibits ability, both in simulation and real world, to accurately throw to out-of distribution goals.

To summarize, this work shows that a real robotic arm can successfully learn a dynamically complex task by adopting the DT architecture while dramatically improving sample efficiency. Also, no visual perception is used in the process and control is based solely on

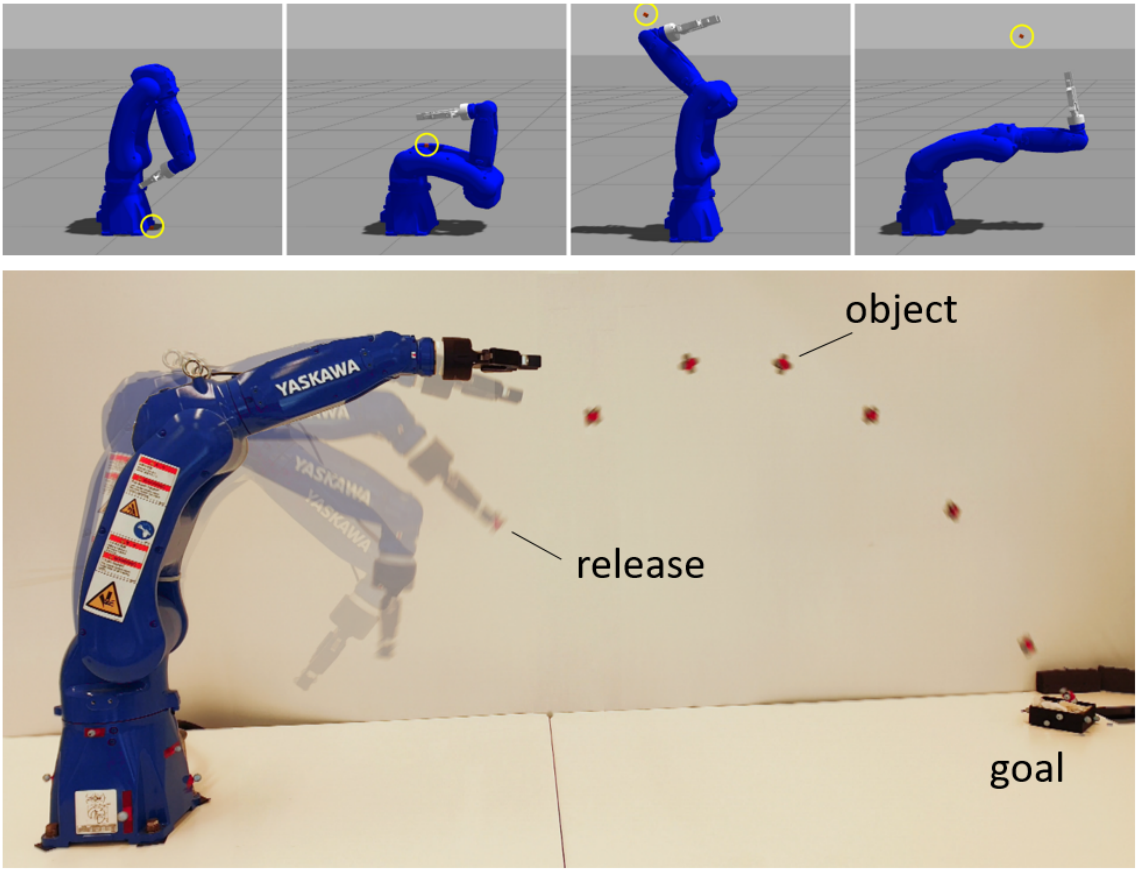

Figure 1.1: (Top row) A simulated robotic arm is used to collect throw trajectories acquired by random motions and arbitrary object release time. Four random throws are shown where the object is marked with a yellow circle for better visibility. (Bottom) A Decision Transformer (DT), trained off-line with the simulated data, is shown to be able to extrapolate and generalize to out-of-distribution goals such that a real robot is able to accurately throw to desired ones.

the joint state of the robotic arm. We also introduce the first integration of HER with DT to exploit arbitrary throws. The results expose the unique abilities of the DT to transfer from an arbitrary simulation and extrapolate using out-of-distribution training data. This has implications to other systems beyond the throwing problem. To the best of the author's knowledge, this is the first implementation and experimental analysis of sim-to-real with DT and using DT for the throwing problem.

2 Related Work

2.1 Throwing with robots

The throwing problem with pure analytical models has been widely addressed [36]. In [11], torque control was proposed in order to throw a ball with an elastic manipulator. A different work provided a method to optimize the shape of an end-effector along with a test-case of planar object throwing [41]. Another work, on the other hand, focused on the parametrization and motion planning of a throwing motion [39]. However, these require knowledge of the dynamic properties of the system which are usually difficult to estimate. As a result, the approaches are not suitable for unstructured environments with various uncertainties and may exhibit low accuracy.

Not much work has studied the use of modern machine-learning approaches for throwing. Early work used motor primitives and meta-parameters learning with RL [20]. Nevertheless, the method requires some prior understanding of a throwing model and dynamic parameters. Later work used a deep neural-network to map an image state observation into a sequence of motor activations [12]. The approach was demonstrated over a ball throwing scenario. In both of these implementations, no actual throwing performance evaluation was provided.

In a more recent study, a robot has learned to rapidly pick-up and throw objects based on image observations [49]. The method consists of predicting the release velocity using a physics-based controller of an ideal ballistic motion. To compensate for the shortcomings of the physics-based controller, the throwing module includes a regression neural-network that predicts a residual on top of the estimated release velocity. Hence, the method is based on a know-how throwing prior and requires a significant amount real-world samples. In contrast, our proposed method does not require any prior on how to throw and only a handful of real throws are needed.

2.2 Reinforcement Learning

Reinforcement Learning (RL) [17] [40] is a type of machine learning that allows agents to automatically learn how to optimize their behavior given a specific goal. It is concerned with how agents ought to take actions in an environment in order to maximize some notion of cumulative reward. This makes it an ideal approach for teaching robots how to perform tasks in the real world, since they can learn directly from their experience instead of being explicitly programmed using extensive hand-coded rules or human supervision. RL is used in robotics to teach robots various tasks in different environments: industrial, domestic and public environments. Examples of such tasks are grasp and manipulation of objects [21], usage of different tools and objects to interact with the environments [38], and allowing robots to navigate and drive [19]. Unfortunately, most RL and robotics tasks require a large amount of real-world data in order to train the models effectively. This can make the data quite expensive as it can be difficult, time-consuming and sometimes dangerous to collect. There is, however, a relatively cheap solution; training an agent in a simulated environment and transferring it to the real world. But, this solution comes with a big caveat, which we will discuss in the next section.

2.3 Sim-to-real

Learning a policy solely from a simulation and deploying it to the real world is considered a hard challenge. Such problem is commonly referred as the *Reality Gap* or *sim-to-real* (simulation to reality). Many approaches have been proposed for bridging the reality gap. Early approach suggested adding noise to the simulation [14]. More recently, domain randomization was proposed where various properties in an existing simulation are constantly changed [50]. Similarly, dynamics randomization was proposed to randomly sample dynamic properties (e.g., robot link mass, damping and friction) in the simulator during training [29]. Consequently, the policy is able to adapt to uncertainties that may emerge when transferring to the real system. Such approach, however, requires full knowledge of the different dynamic parameters in the model, and can be time-consuming since the policy must experience a large variance of dynamic possibilities. [47] showed another approach, by training agents to do the same task in different environments, they were able to extract task-specific states (like driving dynamics) and domain-specific states (like graphical parameters in the simulation). Then using the task-specific states to perform well in new domains. Unfortunately, this requires data from a number of different domains, and in our case of robotic arm, it is not easy as it may require another simulation or another

robotic arm.

All of the above approaches require the formation of a physics-engine based simulation that is sufficiently close to the real system and environment. Closing the reality gap is not easy and collecting real world data is almost always inevitable. In this work, we show that DT can provide an efficient sim-to-real transfer from a simulation with arbitrary dynamic parameters.

2.4 Transformers and attention in Reinforcement Learning and Robotics.

Although Transformers have shown great advancements in language and vision in recent years, yet their impact on RL and Robotics is relatively small. Some work has been done with combining transformers and attention mechanisms in RL for stabilization and efficiency [28, 48, 35]. However, such combination acts only as an additional mechanisms to the existing actor-critic framework. In robotics, on the other hand, transformers have been used in various ways for path-planning [4, 16], imitation [6, 18] and grasping [13]. But, it is important to note that these examples use the transformers as a stand alone architecture in a traditional way, designed for a specific task, and are not part of an RL framework.

Lately, a new approach has been taken by recent works, abandoning the actor-critic framework completely and relying solely on the transformer architecture for the RL problem. A recent study re-framed the RL problem as a time sequence problem and used the Decision Transformer (DT) architecture alone to predict actions [5]. In further work, the transformer was used to model distributions over trajectories followed by beam search as a planning algorithm to find the optimal trajectory [15]. Recent works in text-to-image field [32] and natural language processing [7], showed that a transformer can model a wide distribution of behaviors, enabling better generalization and transfer. This allows the DT to work well in an offline setting, a task that is traditionally challenging due to error propagation and value overestimation [23]. Recent RL work demonstrated this [22]. An agent was taught to play up to 46 Atari games simultaneously at close-to-human performance, using only one DT model, which was trained off-line. Furthermore, in [34], an agent was taught to not only play Atari games, but also to caption images, chat, stack blocks with a real robot arm and much more, using only one model. In [38], a robot was trained to conduct multiple complex tasks from text commands. Following recent work, the DT framework is adopted in this work for further study in the context of real-world object throwing and sim-to-real.

3 Background

In this chapter, we provide a necessary background in Reinforcement Learning (RL), Transformers and Attention. In RL, we present the classic framework for discussions in our work, and provide deeper understanding of the baselines used for comparison. In Transformers, we present the architecture of classic and modern transformers, as they are the core of our algorithm.

3.1 Reinforcement Learning (RL)

A system can be described by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ where $\mathcal{S}$ and $\mathcal{A}$ are the state and action spaces, $\mathcal{P} = P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t-1}, \mathbf{a}_{t-1}, \ldots)$ is the transition probability function of the system, and $\mathcal{R}(\mathbf{s}_t, \mathbf{a}_t)$ is the reward function. A traditional RL algorithm requires to acquire a policy π_θ that produces actions $\pi_\theta(\mathbf{a_t}|\mathbf{s_t})$ that, in turn, produce a trajectory $\{\mathbf{s}_0, \mathbf{a}_0, r_0, \ldots, \mathbf{s}_T, \mathbf{a}_T, r_T\}$, for $\mathbf{a_t} \in \mathcal{A}$, $\mathbf{s_t} \in \mathcal{S}$ and $r_t = \mathcal{R}(\mathbf{s}_t, \mathbf{a}_t)$, that maximizes the expected reward

$$J(\theta) = \mathbb{E}_{\mathbf{a_t} \sim \pi_\theta} \left[\sum_{t=0}^{T} r_t(\mathbf{s}_t, \mathbf{a}_t) \right]. \tag{3.1}$$

A policy π_θ is usually in the form of neural network with parameters θ.

Policy Gradients (PG). One way to find the optimal policy parameters θ^* is to use Gradient Ascent [40, 46] using the objective's gradients $\theta_{t+1} = \theta_t + \nabla J(\theta)$. It can be shown that $\nabla J(\theta)$ is

$$\nabla J(\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{T} \nabla \log \pi_\theta(\mathbf{a_t}|\mathbf{s_t}) \sum_{t'=t}^{T} (r_{t'} - b) \right] \approx \frac{1}{N} \sum_{n=0}^{N} \left(\sum_{t=0}^{T} \nabla \log \pi_\theta(\mathbf{a_t}|\mathbf{s_t}) (Q(\mathbf{s}_t, \mathbf{a}_t) - V(\mathbf{s}_t)) \right). \tag{3.2}$$

It can be shown that a *baseline b* can be subtracted from the reward in order to make the learning process more efficient. An example of simple baseline is the average reward

$b = \frac{1}{M}\sum_{i=0}^{M} r_i$, using this baseline centers the samples around its average reward, and for every gradient step we increase the probability of approximately 50% of the good samples, and decrease the probability of the other bad 50% samples. A baseline of average reward is a good and quick baseline, because it eliminates the bias, but it is not the best, because it does not reduce the variance of the gradients. A better and more common choice for b is the value function, defined as $V(\mathbf{s}_t) = \sum_{t'=t}^{T} \mathbb{E}_\pi [r(\mathbf{s}'_t, \mathbf{a}'_t)|(\mathbf{s}_t)]$. $Q(\mathbf{s}_t, \mathbf{a}_t)$ is the expected future reward $Q(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^{T} \mathbb{E}_\pi [r(\mathbf{s}'_t, \mathbf{a}'_t)|(\mathbf{s}_t, \mathbf{a}_t)]$. The last approximation is conducted through Monte-Carlo, averaging the gradient over N samples.

The term $Q(\mathbf{s}_t, \mathbf{a}_t) - V(\mathbf{s}_t)$ is commonly termed the *Advantage* or *Critic* (The policy π_θ is called the *Actor*), and can be written as

$$A(\mathbf{s}_t, \mathbf{a}_t) = Q(\mathbf{s}_t, \mathbf{a}_t) - V(\mathbf{s}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + \gamma V(\mathbf{s}_{t+1}) - V(\mathbf{s}_t). \tag{3.3}$$

Scalar $\gamma \in [0, 1]$ is the discount factor, prioritizing actions in the near future. From Eq. (3.3), it can be seen that in order to evaluate the Critic, we only need to approximate the Value function. The Value function V_ϕ is also a neural-network with parameters ϕ, and is updated using bootstrapping:

$$V_\phi(\mathbf{s}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma V_\phi(\mathbf{s}_{t+1}). \tag{3.4}$$

Hence, a second objective

$$J(\phi) = \frac{1}{2}(r(\mathbf{s}_t, \mathbf{a}_t) + \gamma V_\phi(\mathbf{s}_{t+1}) - V_\phi(\mathbf{s}_t))^2. \tag{3.5}$$

is to be minimized. This method is guaranteed to converge even when randomly initiated.

Q-Learning and Deterministic Actions. By observing Eq. (3.4), one can similarly learn to approximate the Q-function according to

$$Q_\phi(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_\pi \left[V_\phi(\mathbf{s}_{t+1})\right] \approx r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \max_{\mathbf{a}_{t+1}} Q_\phi(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}). \tag{3.6}$$

and use an implicit policy to get a deterministic action, as follows

$$\mathbf{a}_t = arg\max_{\mathbf{a_t}} Q_\phi(\mathbf{s}_t, \mathbf{a}_t). \tag{3.7}$$

This creates a much easier setting, as there is only one neural-network to approximate and it works well with off-policy samples [45].

Deep Deterministic Policy Gradients (DDPG). In case of discrete action space, Eq. (3.7) can easily yield the optimal action, but in case of continuous action space, this is not an easy task. Lillicrap et al. [24] suggested to approximate the deterministic action $\mathbf{a}_t$ by using a second neural network $\mathbf{a_t} = \mu_\pi(\mathbf{s}_t)$ [24]. The new objective is

$$J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} r_t(\mathbf{s}_t, \mu_\theta(\mathbf{s}_t))\right] \tag{3.8}$$

with gradient

$$\nabla J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \mu_\theta(\mathbf{s})|_{\mathbf{s}=\mathbf{s_t}} \nabla_\mathbf{a} Q(\mathbf{s},\mathbf{a})|_{\mathbf{s}=\mathbf{s_t},\mathbf{a}=\mu_\theta(\mathbf{s}_t)}\right]. \tag{3.9}$$

Eq. (3.9) is derived using the chain rule. The critic objective in this case is

$$J(\phi) = \frac{1}{2}(r(\mathbf{s}_t,\mathbf{a}_t) + \gamma Q_\phi(\mathbf{s}_{t+1}, \mu_\theta(\mathbf{s}_{t+1})) - Q_\phi(\mathbf{s}_t,\mathbf{a}_t))^2, \tag{3.10}$$

with gradient

$$\nabla J(\phi) = (r(\mathbf{s}_t,\mathbf{a}_t) + \gamma Q_\phi(\mathbf{s}_{t+1}, \mu_\theta(\mathbf{s}_{t+1})) - Q_\phi(\mathbf{s}_t,\mathbf{a}_t)). \tag{3.11}$$

We now have two objectives: Eq. (3.8) is the reward expression to be maximized and (3.10) is the critic error to be minimized. Eq. (3.9) and (3.11) can be approximated using Monte-Carlo approximation similar to Eq. (3.2). All of the above methods designed to work in a on-line manner. Meaning, the policy must be constantly applied on the environment in order to collect more trajectories, which in turn are used to update the policy and the Q or Value functions. In our work, we compare our performance to this method, as it is one of the most common approaches for continuous action space problems.

Behaviour Cloning (BC). Contrary to Policy Gradients and Q-Learning methods discussed above, Behaviour Cloning is a rather simpler method designed to learn a policy from an expert [30, 42]. A set of trajectories is collected by an expert. A trajectory is of the form $\tau_i^* = ((\mathbf{s}_{0,i}^*, \mathbf{a}_{0,i}^*), ..., (\mathbf{s}_{T,i}^*, \mathbf{a}_{T,i}^*))$, and a policy $\pi_\theta(\mathbf{a}|\mathbf{s})$ is learned through these trajectories in a supervised, off-line manner by minimizing the loss

$$J(\pi) = \text{Loss}(\mathbf{a}^*, \pi_\theta(\mathbf{a}|\mathbf{s})). \tag{3.12}$$

The loss function is implemented with the Mean Square Error (MSE) function. If an expert is available during training time, the learned policy $\pi_\theta(\mathbf{a}|\mathbf{s})$ can be deployed to

collect more trajectories, and the expert can be used to get feedback by revising the states in the collected trajectory and obtaining the corresponding actions by the expert. But, this is not our case as we don't have en expert in this work. We use this method as additional baseline for comparison, as it is very similar in nature to our work. BC is both supervised learning and off-line.

3.2 Transformers

The Transformer, seen in Fig. 3.1, was introduced by Vaswani et al. [44] to efficiently model sequential data. It consists of an encoder and decoder pair, containing stacked self-attention and cross-attention layers, respectively, with residual connections. Sequential data is the input to the transformer in which each element within it is termed a *token*. Each token is embedded through a linear layer. Furthermore, positional encoding is then added to the embedded token to provide information regarding its position in the sequence. The Transformer inputs m tokens and outputs m tokens preserving input dimensions.

Attention. The embedded tokens are used to obtain the *Queries* (Q), *Keys* (K) and *Values* (V), by passing three copies of the same embedded token through three different linear layers. One can think of the Queries, Keys and Values analogous to retrieval systems. Mapping Queries against Keys of different tokens will be associated with how strongly these tokens are related, and then the value will be returned with correlation to how strong these tokens are related. The Queries, Keys and Values of all the tokens are stacked into vectors, and then used to calculate the *Attention*, as follows

$$Attention(\mathbf{Q},\mathbf{K},\mathbf{V}) = softmax(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})\mathbf{V}. \tag{3.13}$$

The division by $\sqrt{d_k}$ is used for stability of training, Where d_k is the embedding dimension. The attention mechanism allows the transformer to learn the contextual relationships between the tokens. Multi-Head Attention meaning a number of different attentions are calculated in parallel, for every attention-head, queries, keys and values are calculated with different linear layers. Cross-Attention inputs keys and values from the encoder, and queries from the decoder while Self-Attention inputs Keys, Queries and Values from the same source (encoder or a decoder). After a new token is generated, it is added to the Outputs vector, a vector that is inputted to the decoder as seen in Fig. 3.1. A classic example for the input sequence is a sentence to be translated to another language. The decoder will be inputted an empty sequence (As there are no outputs yet), and will output a translated

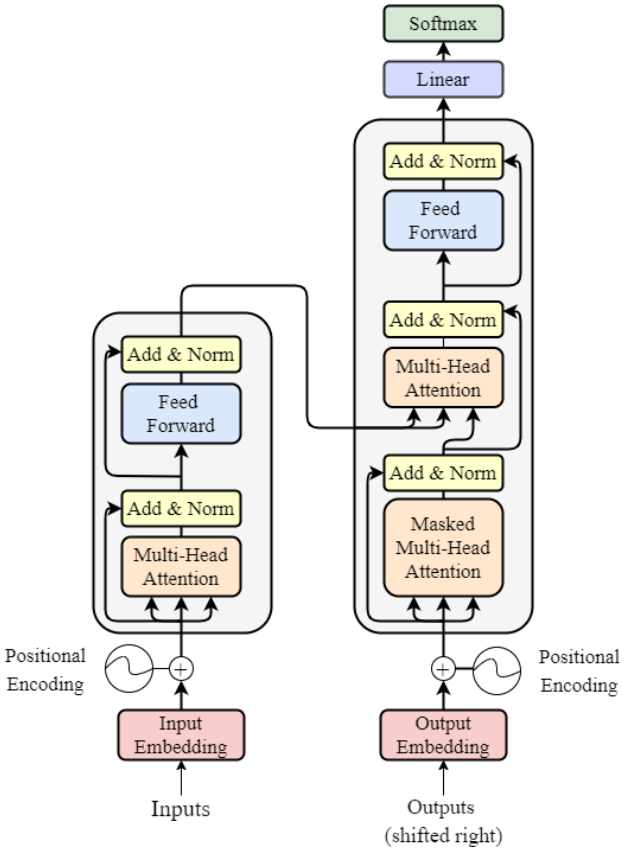

Figure 3.1: Transformer Architecture [44]. The left and right sides are the encoder and decoder, containing self and cross attentions respectively.

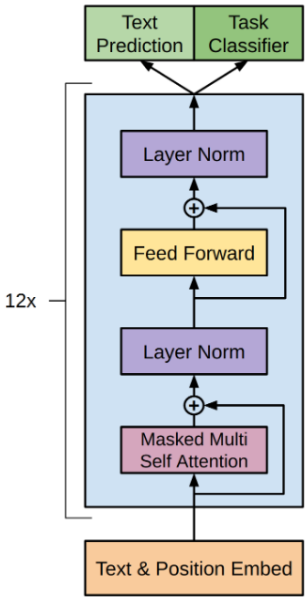

Figure 3.2: Generative Pre-trained Transformer (GPT) Architecture.

word at a time. Every time a translated word is outputted, it will be added to the Output sequence.

Generative Pre-trained Transformer (GPT). A GPT architecture [31] later introduced some changes to the original Transformer by utilizing only the decoder part of the transformer, stacking a number of decoders together, getting rid of the cross-attention mechanism as well. In addition, the GPT is applying causal masking forcing the Transformer to take into account only previous tokens in the sequence instead of the whole sequence. The outputted token of the GPT is added the the end of the input sequence. Thus, enabling autoregressive generation.

4 Method Overview

4.1 Problem Formulation

State. An n degrees-of-freedom robot is given as illustrated in Figure 4.1. Let $\mathbf{s} \in \mathcal{S}$ be the state of the robot where $\mathcal{S} \subset \mathbb{R}^{n+1}$. As such, a state $\mathbf{s} = (\theta_1, \ldots, \theta_n, \theta_{gr})$ is comprised of robot actuator angles $\theta_1, \ldots, \theta_n$ and the binary state of the gripper $\theta_{gr} \in \{0, 1\}$ indicating closed (1) or open (0). A Markov Decision Process (MDP) is defined by

$$P(\mathbf{s}_{t+1}|\mathbf{s}_t) = P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{s}_{t-1}, ..., \mathbf{s}_0) \tag{4.1}$$

where P is a probability distribution function. In other words, a state $\mathbf{s}_t$ captures all the relevant past information. In our case, a state $\mathbf{s}_t$ only includes current positions while missing velocity and acceleration information, making the process not MDP. Hence, determining the next state $\mathbf{s}_{t+1}$ may require a sequence of past states. We use this form of state due to the fact that the Decision Transformer (DT) inherently accesses all the past tokens (rewards-to-go, states and actions) in a window of size w, allowing the DT to evaluate on its own motor velocities and accelerations.

Goal. The aim of the robotic arm is to throw a grasped object to a desired goal $\mathbf{x}_{goal} \in \mathbb{R}^2$. Goal $\mathbf{x}_{goal} = (x_g, y_g)$ is a position on some horizontal plane in the vicinity of the robot with respect to its base. Consequently, the throwing distance is given by $d_g = \|\mathbf{x}_{goal}\|$ and is an input to the DT. For safety reasons, the throwing direction is determined analytically by $\phi = \arctan 2(x_g, y_g)$.

Action. Let $\mathbf{a}_t \in \mathcal{A}$ be an action of the system at time t where $\mathcal{A} \subset \mathbb{R}^{n+1}$. Hence, an action is composed of actuator velocities $\omega_1, \ldots, \omega_n$ for the arm, and opening or closing command $a_{gr} \in [0, 1]$ to the gripper. The gripper is initialized while closed on the object, i.e., $a_{gr} = 1$. Once condition $a_{gr} \leq \tau$ is satisfied for some pre-defined threshold $\tau > 0$, the gripper opens.

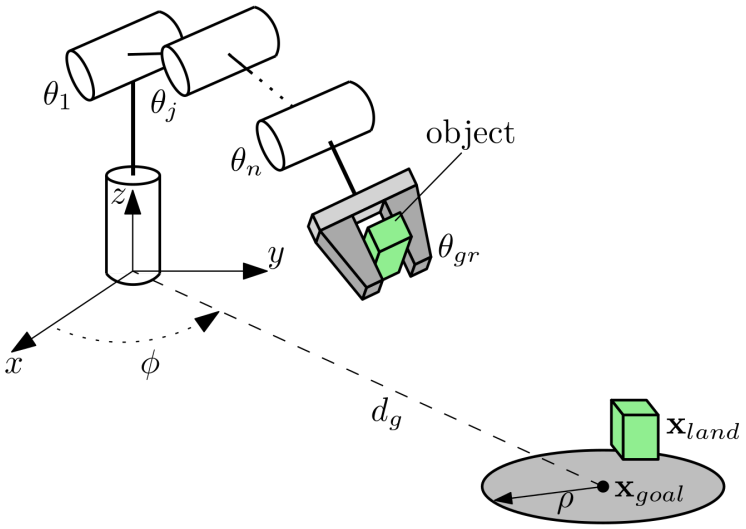

Figure 4.1: Illustration of the robot with the object and goal.

Reward. A sparse reward function is defined for a robot throw in the form:

$$\mathcal{R}(\mathbf{s}_t, \mathbf{a}_t) = \begin{cases} 1, & \theta_{gr} = 0 \wedge \|\mathbf{x}_{land} - \mathbf{x}_{goal}\| \leq \rho, \\ -1, & \theta_{gr} = 0 \wedge \|\mathbf{x}_{land} - \mathbf{x}_{goal}\| > \rho, \\ 0, & \text{otherwise}, \end{cases} \tag{4.2}$$

where $\rho > 0$ and $\mathbf{x}_{land}$ is the actual landing position of the object. A throw is considered successful if the first landing point $\mathbf{x}_{land}$ is inside a circle of radius ρ around the goal position. The robot is penalized if the object did not land in the circle. We note that non-sparse rewards did not provide sufficient results in preliminary work and as indicated in [2].

The system at any given time can be described by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ where $\mathcal{P} = P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t-1}, \mathbf{a}_{t-1}, \ldots)$ is the transition probability function of the system. A traditional RL algorithm requires to acquire a trajectory $\{\mathbf{s}_0, \mathbf{a}_0, r_0, \ldots, \mathbf{s}_T, \mathbf{a}_T, r_T\}$, for $r_t = \mathcal{R}(\mathbf{s}_t, \mathbf{a}_t)$, that maximizes the expected reward $\mathbb{E}\left[\sum_{t=0}^{T} r_t\right]$. In DT, on the other hand, pre-recorded roll-outs are used for off-line training. Hence, one can only find a trajectory that produces a desired reward as discussed next.

4.2 Data Collection and Augmentation

Training data is collected from a simulated environment where the kinematics of the robotic arm are modeled. However, dynamic parameters, such as link inertia and mass,

are chosen arbitrarily and may be very different from the values of the real robot. We assume that no prior motion of how to throw exists. Hence, temporally correlated noise is injected into the actuators of the simulated arm resulting in efficient exploration through random motion. To generate such noise, the Ornstein-Uhlenbeck process [43] is used in the form

$$x_{k+1} = (\mu - x_k)\theta\Delta t + \sigma\varepsilon_k\sqrt{\Delta t} \tag{4.3}$$

where x_k is the noise at time-step k, Δt is the sample time, ε_k is a normal noise $\varepsilon_k \sim \mathcal{N}(0,1)$ and, θ, μ and σ are process parameters. At the beginning of each throw trajectory, a random goal is selected. In addition, a random time-step is chosen for when the gripper will open and release the object.

In this work, we explore the benefits of data augmentation based on the Hindsight Experience Replay (HER) [2]. Given a trajectory where the object landed at position $\mathbf{x}_{land}$, K_{her} samples are generated with the same trajectory while their corresponding goals are randomly picked inside a circle of radius 2ρ around the landing spot $\mathbf{x}_{land}$. For $K_{her} = 0$, we always pick the landing spot as a goal. For $K_{her} > 0$, goals with distance $[0,\rho]$ and $(\rho, 2\rho]$ from $\mathbf{x}_{land}$ are considered success and miss, respectively. This is done in order to create a balanced dataset of successful and unsuccessful throws, to diversify the variance of rewards in the training and to give a clearer insight to the DT regarding the boundary between successes and misses.

With the above, a recorded trajectory will be of the form

$$\tau_i = \{\mathbf{s}_0, \mathbf{a}_0, r_0 \ldots, \mathbf{s}_{T_i}, \mathbf{a}_{T_i}, r_{T_i}\} \tag{4.4}$$

where T_i is the length of the motion and $\mathbf{s}_{T_i}$ is the state in which the object was released. Trajectory τ_i is accompanied with the goal $\mathbf{x}_{goal}$ determined in the augmentation. The rewards included in the trajectory are updated based on (4.2) and indicate whether the goal was successfully reached. Finally, a dataset of M trajectories $\mathcal{P} = \{\tau_1, \ldots, \tau_M\}$ is obtained for generating DT trajectories and training as discussed next.

4.3 Decision Transformers

DT Architecture. The DT, based on the GPT architecture as disccused in Chapter 3, is illustrated in Figure 4.2. It is inputted with w last time-steps yielding a total of $3w$ tokens $\{\hat{R}_0, \mathbf{s}_0, \mathbf{a}_0, \ldots, \hat{R}_w, \mathbf{s}_w, \mathbf{a}_w\}$ where $\hat{R}_i$ is the *reward-to-go* [5]. Token embedding is performed with a linear layer for each modality. After the embedding and similar to

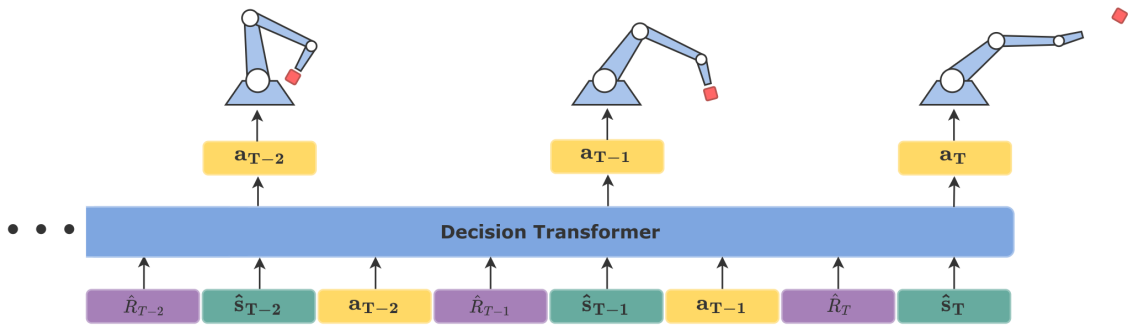

Figure 4.2: Illustration of the Decision Transformer pipeline. In a given timestep, the DT is inputted with previous rewards-to-go, states, actions and current reward-to-go and state. It then predicts the next action to be taken in order to achieve the desired reward. After the predicted action is executed by the robot, we observe the next state and calculate the new reward-to-go based on the received reward.

positional encoding, a time-embedding is added to the embedded tokens and further fed into the DT. During training, a full recorded trajectory τ is inputted into the DT. The DT predicts action $\mathbf{a}_t$ for each timestep t in the trajectory based solely on tokens in the same or previous timesteps. For each predicted action $\tilde{\mathbf{a}}_t$, a loss is calculated comparing between the predicted action $\tilde{\mathbf{a}}_t$ and the actual action $\mathbf{a}_t$ from the recorded trajectory. The auxiliary loss is the sum of the temporal losses and is given by

$$\mathcal{L}_{DT} = \sum_{t=1}^{T} \text{loss}(\tilde{\mathbf{a}}_t, \mathbf{a}_t). \tag{4.5}$$

The DT network is trained by back-propagation with a set of pre-recorded trajectories to minimize $\mathcal{L}_{DT}$.

During inference and at time t, the DT predicts the next action $\tilde{\mathbf{a}}_{t+1}$ based on all previous tokens $\{\hat{R}_t, \mathbf{s}_t, \mathbf{a}_t, \hat{R}_{t-1}, \mathbf{s}_{t-1}, \mathbf{a}_{t-1}, \ldots\}$. Action $\mathbf{a}_{t+1}$ is then exerted on the robot followed by observing the next state $\mathbf{s}_{t+1}$ and updating the reward-to-go $\hat{R}_{t+1}$. This process is repeated until reaching the goal.

DT Trajectories. As mentioned before, DT does not maximize expected reward but instead produces a sequence of actions that should yield a specifically desired reward. The desired reward is inputted into the DT, and must be updated on the fly once any reward is awarded. Therefore, we define the *reward-to-go* token at time t as

$$\hat{R}_t = \sum_{t'=t}^{T} r_{t'}. \tag{4.6}$$

Furthermore and since this is a multi-goal problem, the DT must receive the desired goal for which to generate a sequence of actions. Hence, the desired goal distance d_g is concatenated to the state, creating a state-goal token $\hat{\mathbf{s}}_t$ defined as

$$\hat{\mathbf{s}}_t = (\mathbf{s}_t \| d_g), \tag{4.7}$$

where $\|$ denotes concatenation. The last token will be the action as defined before. With the above said, each trajectory $\tau_i \in \mathcal{P}$ is reformulated to a DT trajectory consisting of rewards-to-go, state-goals and actions in the form

$$\tau_{DT_i} = (\hat{R}_0, \hat{\mathbf{s}}_0, \mathbf{a}_0, ..., \hat{R}_{T_i}, \hat{\mathbf{s}}_{T_i}, \mathbf{a}_{T_i}). \tag{4.8}$$

The DT training set is now $\mathcal{P}_{DT} = \{\tau_{DT_1}, \ldots, \tau_{DT_M}\}$. In this form, the DT can autoregressively train and generate new actions on the fly.

4.4 Sim2Real

Sim2Real Adaptation. The strength of training the DT in simulation is the ability to learn how to throw without any prior while avoiding physical risks. Such process over a real robot is very dangerous and cannot be done. Nevertheless, the robot learns the general motions in simulations while the reality gap prevents direct transfer to the real robot. In order to fully transfer our model to the real robot, the simulation-trained DT is fine-tuned with a small number of throws collected from the real robot. To do so, we exert the simulation-trained DT on the real robot in order to conduct real throws to random goals. For each throw, we multiply the actions $\{\mathbf{a}_0, \ldots, \mathbf{a}_T\}$ with a random gain $\alpha \sim \mathcal{U}(1, \alpha_{max})$. Multiplying the actions by α assists in exploring the real robot domain and in bridging the reality gap. We further augment the collected data as described in Section 4.2. The new throw samples are used to re-train the DT model and refine its weights for it to adapt to the new domain.

Simulation Tuning. While the above sim2real method is sufficient for transferring a simulation trained model to the real robot, tuning the simulation prior to collecting data can reduce the required number of real throws to be collected. Therefore, Baysian optimization is used to further reduce the reality gap [27, 37].

First, a set of M state trajectories $\mathcal{T}_{real}$ is collected from the real robot by applying sequences of actions $\mathcal{A}$. The gap between the $\mathcal{T}_{real}$ and a set $\mathcal{T}_{sim}$ recorded with the same

action sequences A is defined to be

$$\mathcal{L}_{gap} = \sum_{i=1}^{M} \text{MSE}(\tau_i^{sim}, \tau_i^{real}) \tag{4.9}$$

where $\tau_i^{sim} \in \mathcal{T}_{sim}$ and $\tau_i^{real} \in \mathcal{T}_{real}$ for $i = 1, ..., M$. If $\mathcal{L}_{gap}$ is minimal, the simulation can be said to best describe actual robot. Therefore, we search for the PID control gains of the simulated robot actuators that yield similar behaviour to the real robot. In other words, Baysian optimization is used to solve

$$\mathbf{p} = \underset{\mathbf{p}}{\text{argmin}}\, \mathcal{L}_{gap}, \tag{4.10}$$

where $\mathbf{p}$ is the vector of PID control gains for all actuators. At each iteration, the sequence of actions A is exerted on the simulated robot with the instantaneous control gains $\mathbf{p}$ to acquire updated $\mathcal{T}_{sim}$ and $\mathcal{L}_{gap}$. The optimization process repeatedly updates $\mathbf{p}$ until reaching the minimal gap value.

5 Experiments

5.1 Setup

Robotic Hardware. The proposed approach is experimented with a six-degrees-of-freedom Yaskawa Motoman GP8 industrial arm equipped with a Robotiq 2f-85 parallel gripper (Figure 1.1, bottom). Needless to say, training a policy entirely on such powerful robot without any prior of how to throw is extremely dangerous. Without loss of generality and for safety during testing, the throw motion is bounded to a plane perpendicular to the horizontal floor and, thus, the DT learns only to move the second, third and fifth joints. The direction of throw is, therefore, determined analytically according to (x_g, y_g) by ϕ and set by the first joint which is perpendicular to the floor. Similarly, the remaining joints are set constant to zero. Due to the dynamic specifications of the robotic arm, the throwing distance is bounded to $d_g \in [50, 200]$ centimeters. Furthermore, the gripper opens and releases the object when value a_{gr} of the current action is below a threshold τ. The value for τ is chosen to be the mean of all the gripper actions in the training dataset acquired in simulation. The communication frequency with the arm is 10Hz while the maximum trajectory length is set to 1 second yielding an upper bound of $T_i \leq 10$ timesteps for a trajectory. Furthermore, we allow the DT to access all previous steps at any given time along the trajectory. The system is controlled using the Robot Operating System (ROS). Videos of the experiments can be seen in the supplementary material. We couldn't control the velocities of the motors in the arm directly, as it was never designed for this, so we used a Taylor's approximation for position of every motor:

$$\theta_t = \theta_{t-1} + \omega_t / f. \tag{5.1}$$

Where θ_t is the motor angle and ω_t is the predicted motor velocity.

Throwing Evaluation. Fine-tuning and initial evaluation is done by throwing a cube of size $1.5 \times 1.5 \times 1.5$ *cm* with mass of 15 *g*. Only for evaluation of the throwing

performance in the real world, a motion capture system with eight OptiTrack Prime 41 cameras was used. Markers were attached on the base of the robot, on a target plate and on the thrown object. In this way, the robotic arm can automatically detect the goal $\mathbf{x}_{goal}$ relative to itself and throw the object to the target. The object landing position $\mathbf{x}_{land}$ is also detected by the cameras. A throw error is calculated by $e = \|\mathbf{x}_{goal} - \mathbf{x}_{land}\|$. We define a test procedure in which an object is thrown to 30 uniformly distributed pre-defined goals. The test accuracy is the mean $\frac{1}{30}\sum_{i=1}^{30} e_i$ of all throws.

Simulation. The same robot was modeled in the ROS-Gazebo physics engine. The dynamic properties of the simulated arm were chosen arbitrary while in the same scale of the real one. Throw data was collected with an object of the same size and mass as for the real system. Noise parameters were set to $\theta = 0.02$, $\mu = 0$ and $\sigma = 0.2$. In addition, data was collected with a success radius of $\rho = 0.5$ *cm*. With these conditions, random trajectories were generated and recorded for training the DT. Since no prior of how to throw was given to the simulation, the recorded trajectories are out-of-distribution to feasible throws. Examples of these random non-feasible throws used for training are seen in Figure 1.1 (top row). To achieve a physical grasp in the simulation we used a gazebo plugin that simulated grasping by connecting the object to the gripper upon closure, and disconnecting when the gripper is opened. This is not ideal and introduced some noise to the simulation that was not present in the real robot setup, reducing the performance in the simulation.

5.2 Model Training

The acquired simulation data is used to train a DT model as described in Section 4. The model architecture and hyper-parameters were optimized to yield the best simulation scores. In particular, the DT architecture, based on GPT, was chosen to be with embedded dimension of 128, 1 hidden layer, 1 attention head and a ReLU activation function. Actions are predicted by including an additional linear layer at the DT output. A Tanh and Sigmoid activation functions are applied to the actuator velocities $\omega_1, \ldots, \omega_n$ and gripper command a_{gr}, respectively, from the outputted action vector $\mathbf{a}_t$. This DT model yields a total of 210,058 trainable parameters. For the DT model to predict actions, the loss function to be minimized in training was defined to be the sum of the Mean Square Error (MSE) on actuator velocities and Binary Cross Entropy (BCE) on the gripper action. The model was trained using the AdamW optimizer with a learning rate of 10^{-4}, weight decay of 10^{-4}, dropout of 0.1 and a linear warm-up for the first 10^4 gradient steps. The model was trained for 100 epochs, where each epoch consists of 100 optimization steps on mini-batches of

128 trajectories sampled uniformly from the simulated dataset. In order to get the very best model, we evaluated our model in the simulation every iteration by conducting 20 throws with goals evenly distributed inside the throwing range, and saved a snapshot of the best performing model, considering average distance from target. While training we noticed that our model was producing actions, in which the last element that was responsible for the releasing timing of the object, was prone to overfit to specific timesteps. In order to avoid this, we added random timestep translation to the trajectory, forcing the model to take actions based on current and previous states and actions and not blindly based on timesteps.

5.3 Simulation Results

We first analyze the performance of the DT in the simulation environment.

Data quantity and augmentation. We begin by studying the performance of the DT with regards to the amount of simulated data and augmentation parameter K_{her}. A set of 1,000 random trajectories was collected in simulation as described above. The DT was trained multiple times with an increasing number M of trajectories. The baseline approach is training the DT directly with the raw data without including HER and augmentation. Hence, the DT is trained with trajectories labeled by randomly generated goals and rewards given accordingly for success or miss of these goals (0.5% probability of a success). Furthermore, for a specific number of trajectories, training was performed over four different HER augmentation parameters $K_{her} = \{0, 1, 3, 5\}$. For $K_{her} = 0$, the actual landing position $\mathbf{x}_{land}$ was set as the goal $\mathbf{x}_{goal}$ of the corresponding trajectory. With data augmentation and when $K_{her} > 0$, K_{her} additional trajectories were generated with sampled goals in the vicinity of $\mathbf{x}_{land}$ as described in Section 4.2.

Figure 5.2 presents throw accuracy results with regards to the number of training trajectories and K_{her}. First, the model reaches saturation with above 500 throws. In addition, HER augmentation is shown to be significant in increasing the accuracy. Nevertheless, augmentation with $K_{her} > 0$ (i.e. adding more trajectories beyond the modified one) exhibits no significant improvement over accuracy. Moreover, various values for the goal radius in the range $\rho \in [0.5, 20]$ *cm* were tested while not providing significant change in performance.

Data Sparsity. We next analyze the ability of the DT to generalize and extrapolate to out-of-distribution goals. Figure 5.3 shows the distribution of 500 collected random trajectories (in black) used for training with respect to the throwing distance d_g and when

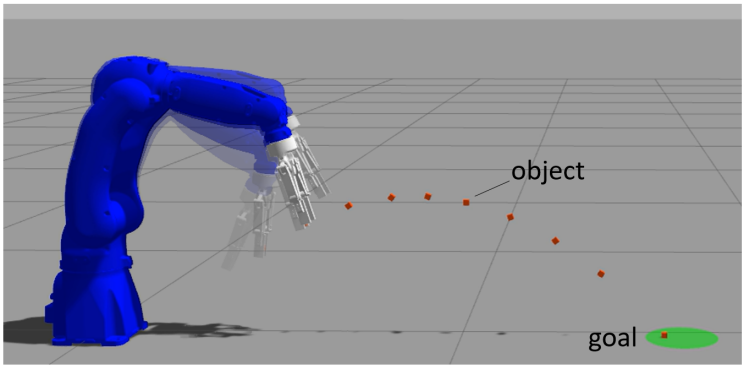

Figure 5.1: Deployment example of the trained DT on the simulated robot for throwing the object to a desired target.

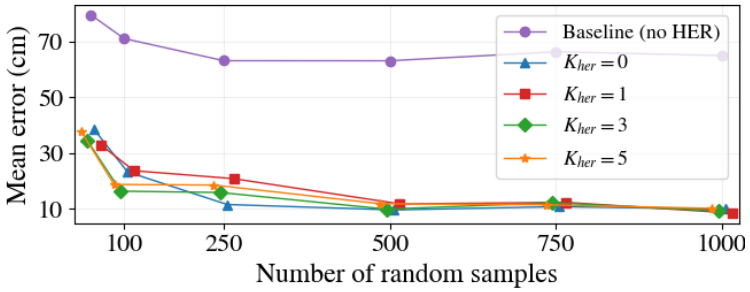

Figure 5.2: Throw accuracy with regards to the number of training trajectories and HER augmentation parameter K_{her}.

$K_{her} = 0$. Examples of such throws are seen in Figure 1.1 (top row). Only 14% of the random throws resulted in the object landing within our desired working range $d_g \in [50, 200]$. On the other hand, Figure 5.3 also shows (in red) the distribution of 1,000 successful throws (i.e., hit within 1cm radius) to the desired range $d_g \in [50, 200]$ while using the trained DT policy. Hence and during test time, the DT manages to hit out-of-distribution goals within this range with high success rate. Figure 5.1 shows an example of a successful simulated throw where the goal is out-of-distribution to the training trajectories.

Baseline Comparison. We compare our results to DDPG [24] and Behavioral Cloning (BC) [42], also discussed in Chapter 3. DDPG is a common Temporal-Difference learning algorithm for continuous control. Similar to DT, BC is a supervised learning algorithm. The objective of the comparison is to analyze whether common RL approaches, i.e., DDPG and BC, can match the throw accuracy and data efficiency of the DT. In addition, we are interested in the learning stability in terms of loss convergence. A similar setup and training process were performed for all methods. All methods were trained and tested in simulation.

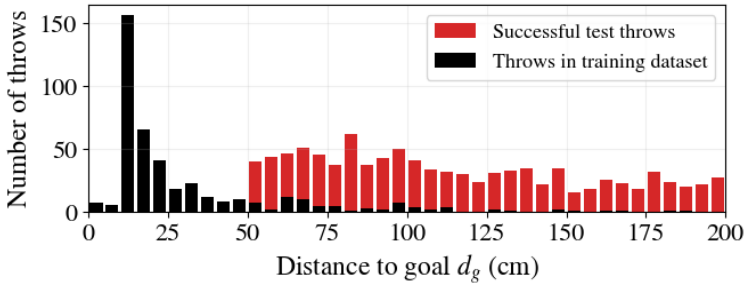

Figure 5.3: Distribution of throws in the training dataset (black) compared to successful test throws (red). Despite the sparsity of throws for $d_g > 50cm$ in the training dataset, DT manages to make accurate throws to out-of-distribution goals.

The hyper-parameters of the models were optimized to converge and reach minimal loss. For DDPG, we used the work of Marcin Andrychowicz et al. [3] to help us optimize the hyper-parameters of the algorithm until it converged. The actor was formed by a Multi-Layer Perceptron (MLP) with two layers and 64 neurons, each. Similarly, the critic is formed of two layers and 256 neurons, each. A buffer of 10,000 trajectories was used and trained for 100 gradient steps for every 40 trajectories executed. Other ratios of gradient steps per trajectories collected caused the algorithm to not converge. The learning parameters are as described in Section 5.2. Similarly, data augmentation is as described in Section 4.2. Different K_{her} values were tested while the best convergence was achieved with $K_{her} = 7$. Furthermore, the optimal radius value for the reward (4.2) is $\rho = 2cm$. BC is implemented similar to DT with an MLP of three layers comprising $\{64, 128, 256\}$ neurons. The current and ten past states are inputted to the MLP which, in turn, outputs the next action.

Table 5.1 presents the comparative results between the three methods along with another baseline of sole random throws to arbitrary goals without any policy. We note that in the test, goals are set in out-of-distribution distances. Hence and in terms of accuracy, DT outperforms with the best accuracy and exhibits the ability to extrapolate to farther distances. On the other hand, DDPG and BC provide poor results while better than random throws. While DDPG has better results than BC, it requires a large amount of training data. The learning stability was also evaluated by measuring the variance of the evaluation score across the epochs. Results show that stability of DT is lower by an order of magnitude than DDPG. In addition to training stability, we found the DDPG algorithm highly sensitive to changes in hyper-parameters, easily resulting in divergence or bad performance.

Table 5.1: Baseline comparison

Method	**Number of throws**	**Mean error** (cm)	**Stability** (cm^2)
Random throws	500	121±136	-
BC	500 Random	39.3±32.8	6
DDPG	6,360 On-policy	17.8±14.0	3,020
DT	500 Random	8.2±6.4	67

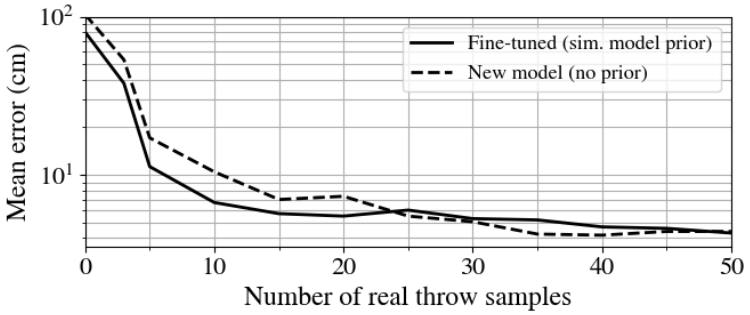

Figure 5.4: Number of real throws required for fine-tuning the DT model.

5.4 Real Robot Results

Fine-tuning with Real Throws. When transferring the pre-trained DT model to the real robot, the yielded mean error for test throws is 80cm. As described in Section 4.4, the simulation model acts as a prior and real throws are required in order to fine-tune it. Hence, a dataset of real throws was collected by generating a set of random goals and attempting to throw to them using the pre-trained model and when $\alpha_{max} = 3.0$. After applying HER, the prior DT model is refined. We next analyze the required number of real throws to reach accurate performance. Figure 5.4 presents the mean throw error with regards to the number of real throws. For comparison, we also train a new DT model without prior training in simulation and with only the real throws. Hence, the model can be seen as trained with only expert data. Results show significant accuracy improvement with only a handful of real throws. For instance, fine-tuning with only 5 and 10 throws reduces error to less than 11cm or 6cm, respectively. With more throws, accuracy keeps improving while improvement rate declines. The error with 50 throws is 4.3cm. Results with only expert throws are similar and show ability to train a new model with only few good samples.

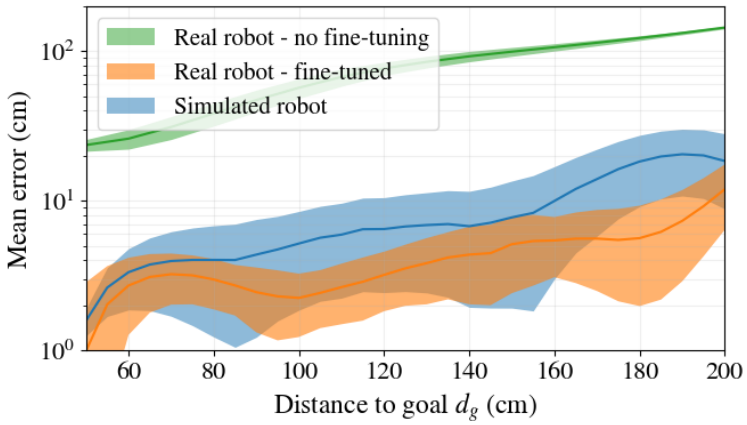

Figure 5.5: Throw accuracy with regards to the distance of the goal d_g for (blue) simulated robot, (green) real robot with non fine-tuned DT and (orange) real robot with fine-tuned DT.

Real Robot Performance. The best performing model from the previous section is chosen for further analysis. Figure 5.5 shows throw accuracy for the simulated and real robots with regards to the distance to the goal d_g. First, the real robot performs very poorly with a non fine-tuned DT emphasizing the significant reality gap. However, with only few real throw samples for fine-tuning, the DT model achieves better accuracy than the simulation with a mean error of 4.3cm. For both domains, higher errors are apparent for targets farther than 170cm as the throw is more complex. On the other hand, throwing to a 15×15 *cm* target box positioned in distance of up to 180cm would yield approximately 100% success rate.

Generalization To Other Objects. We next test the ability of the DT to generalize to other objects without additional fine-tuning. Along with the cuboid, we test an additional six objects, seen in Figure 5.6, including: squeeze ball, cylinder, box, sand ball, copper coil and pencil. Each object is tested for success rate in throwing to short (80 cm), medium (130 cm) and long (180 cm) distance goals. For each goal, the object is thrown 10 times. A success is defined to be hitting a rectangular plate of size 15×15 *cm*. Table 5.2 summarizes the success rate for the throws along with physical properties of the objects. For better understanding, we make a distinction between throws where the initial grasp is on or off the Center-of-Mass (CM) of the object. Off-set to the CM is randomly placed in each throw. When grasping the object on its CM, the model generalizes well and the success rate is high for all objects. However, throws with non-CM grasps to medium and long distances have lower success rate. While having some ability to hit the goal, the model was not trained to compensate for the elongation of the object yielding bias in the throw. For such compensation feature, the model must have a mean to measure the location of

Figure 5.6: Seven test objects including (a) cuboid (with reflective markers, (b) squeeze ball, (c) cylinder, (d) box (e) sand ball, (f) copper coil and (g) pencil.

Table 5.2: Throwing success rate out of 10 throws for seven test objects.

Object	**Mass**	**Dimensions**	CM grasp			Non-CM grasp		
			Success Rate for dist.			**Success Rate** for dist.		
	(g)	(cm)	80cm	130cm	180cm	80cm	130cm	180cm
(a) Cuboid	15	$1.5 \times 1.5 \times 1.5$	100%	100%	100%	-	-	-
(b) Squeeze ball	25	radius 3	100%	100%	100%	100%	50%	50%
(c) Cylinder	50	radi. 2, heig. 8	100%	100%	100%	100%	60%	60%
(d) Box	120	$3 \times 6 \times 9$	100%	100%	100%	100%	40%	40%
(e) Sand ball	60	radius 2.5	100%	100%	100%	100%	0%	0%
(f) Copper coil	386	radi. 2.2, heig. 6.5	100%	100%	90%	90%	40%	0%
(g) Pencil	6	radi. 0.4, len. 18	100%	100%	100%	90%	50%	40%

the CM, which is not available in this work. We leave this to future work.

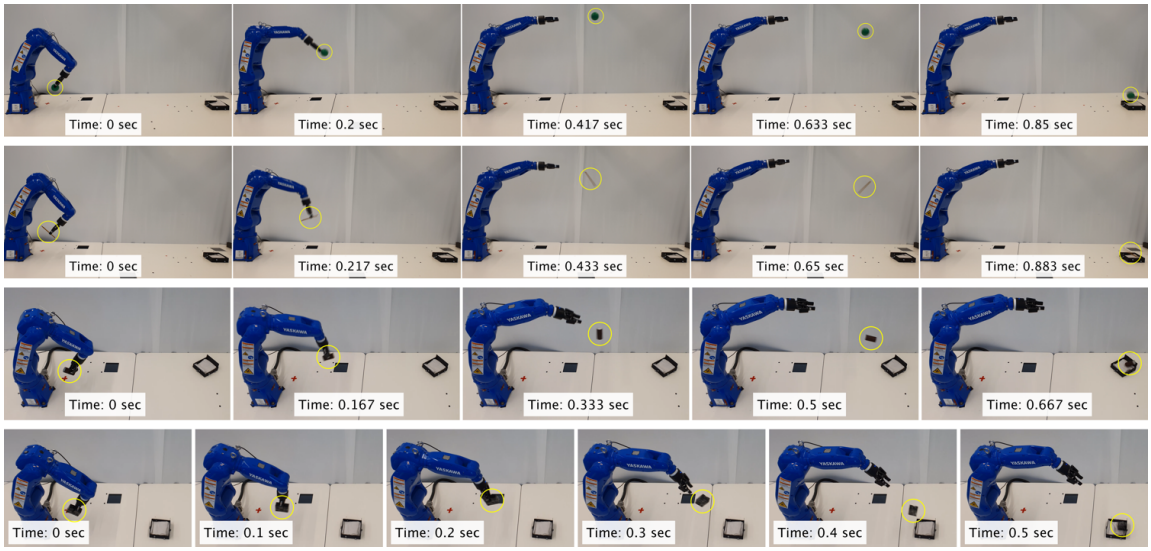

Figure 5.7: Snapshots of four throws (from top to bottom): squeeze ball ($d_g = 180cm$), pencil ($d_g = 180cm$), cylinder ($d_g = 140cm$) and box ($d_g = 90cm$). Objects are marked with a yellow circle for better visualization.

6 Conclusions and Future Work

In this work, we have proposed a data-efficient framework for object throwing with DT. A policy is trained off-line using data recorded in simulation through randomized actions without any prior on how to throw. In addition, the simulation consists of arbitrary physical parameters with a slight pre-tuning to the real robot. Then, the DT is fine-tuned with only several real throw examples. In particular, a set of 5-10 throws is sufficient to provide throw accuracy of less than 10cm. Furthermore, experiments on a set of different object yielded high success rate. However, when grasping an object off its CM, the success rate declines. Future work may consider addressing this by including visual feedback or a Force/Torque sensor that can embed grasp off-sets from object CM.

To the best of the author's knowledge, this is the first implementation and experimental analysis of sim-to-real with DT and using DT for the throwing problem. Achieving accurate throws with a real robot in a completely model-free manner, with very low number of random samples while utilizing HER, and even as low as 10-20 of good samples. We believe this framework could further be extended to many other complex dynamic and kinematic tasks, and reduce the effort that is required to transfer from simulated environment to a real robot.

References

[1] Rishabh Agarwal, Dale Schuurmans, and Mohammad Norouzi. An optimistic perspective on offline reinforcement learning. In *International Conference on Machine Learning*, pages 104–114, 2020.

[2] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

[3] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. What matters in on-policy reinforcement learning? a large-scale empirical study. *arXiv preprint arXiv:2006.05990*, 2020.

[4] Devendra Singh Chaplot, Deepak Pathak, and Jitendra Malik. Differentiable spatial planning using transformers. In *International Conference on Machine Learning*, pages 1484–1495. PMLR, 2021.

[5] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

[6] Sudeep Dasari and Abhinav Gupta. Transformers for one-shot visual imitation. *arXiv preprint arXiv:2011.05970*, 2020.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[8] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[9] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *Inter. conference on machine learning*, pages 1329–1338, 2016.

[10] Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. In *International conference on machine learning*, pages 2052–2062. PMLR, 2019.

[11] Yizhi Gai, Yukinori Kobayashi, Yohei Hoshino, and Takanori Emaru. Motion control of a ball throwing robot with a flexible robotic arm. *Inter. Journal of Computer and Information Eng.*, 7(7):937–945, 2013.

[12] Ali Ghadirzadeh, Atsuto Maki, Danica Kragic, and Mårten Björkman. Deep predictive policy training using reinforcement learning. In *IEEE/RSJ Inter. Conf. on Intelligent Robots and Systems*, pages 2351–2358, 2017.

[13] Yunhai Han, Rahul Batra, Nathan Boyd, Tuo Zhao, Yu She, Seth Hutchinson, and Ye Zhao. Learning generalizable vision-tactile robotic grasping strategy for deformable objects via transformer. *arXiv preprint arXiv:2112.06374*, 2021.

[14] Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *European Conference on Artificial Life*, pages 704–720, 1995.

[15] Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem. *Advances in neural information processing systems*, 34:1273–1286, 2021.

[16] Jacob J Johnson, Linjun Li, Ahmed H Qureshi, and Michael C Yip. Motion planning transformers: One model to plan them all. *arXiv preprint arXiv:2106.02791*, 2021.

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

[18] Heecheol Kim, Yoshiyuki Ohmura, and Yasuo Kuniyoshi. Transformer-based deep imitation learning for dual-arm robot manipulation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8965–8972. IEEE, 2021.

[19] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[20] Jens Kober, Erhan Oztop, and Jan Peters. Reinforcement learning to adjust robot movements to new situations. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[21] Oliver Kroemer, Scott Niekum, and George Konidaris. A review of robot learning for manipulation: Challenges, representations, and algorithms. *The Journal of Machine Learning Research*, 22(1):1395–1476, 2021.

[22] Kuang-Huei Lee, Ofir Nachum, Mengjiao Yang, Lisa Lee, Daniel Freeman, Winnie Xu, Sergio Guadarrama, Ian Fischer, Eric Jang, Henryk Michalewski, et al. Multi-game decision transformers. *arXiv preprint arXiv:2205.15241*, 2022.

[23] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.

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

[25] Vincent Lim, Huang Huang, Lawrence Yunliang Chen, Jonathan Wang, Jeffrey Ichnowski, Daniel Seita, Michael Laskey, and Ken Goldberg. Planar robot casting with real2sim2real self-supervised learning. *arXiv preprint arXiv:2111.04814*, 2021.

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

[27] Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.

[28] Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing transformers for reinforcement learning. In *International conference on machine learning*, pages 7487–7498. PMLR, 2020.

[29] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *IEEE Inter. conf. on Robotics and Automation*, pages 3803–3810, 2018.

[30] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems*, 1, 1988.

[31] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[32] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pages 8821–8831, 2021.

[33] Fredy Raptopoulos, Maria Koskinopoulou, and Michail Maniadakis. Robotic pick-and-toss facilitates urban waste sorting. In *IEEE Inter. Conference on Automation Science and Engineering*, pages 1149–1154, 2020.

[34] Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, et al. A generalist agent. *arXiv preprint arXiv:2205.06175*, 2022.

[35] Sam Ritter, Ryan Faulkner, Laurent Sartran, Adam Santoro, Matt Botvinick, and David Raposo. Rapid task-solving in novel environments. *arXiv preprint arXiv:2006.03662*, 2020.

[36] Taku Senoo, Akio Namiki, and Masatoshi Ishikawa. High-speed throwing motion based on kinetic chain approach. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3206–3211, 2008.

[37] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[38] Mohit Shridhar, Lucas Manuelli, and Dieter Fox. Perceiver-actor: A multi-task transformer for robotic manipulation. *arXiv preprint arXiv:2209.05451*, 2022.

[39] Avishai Sintov and Amir Shapiro. A stochastic dynamic motion planning algorithm for object-throwing. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2475–2480, 2015.

[40] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[41] Orion Taylor and Alberto Rodriguez. Optimal shape and motion planning for dynamic planar manipulation. *Autonomous Robots*, 43(2):327–344, 2019.

[42] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, page 4950–4957, 2018.

[43] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

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

[45] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3): 279–292, 1992.

[46] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[47] Jinwei Xing, Takashi Nagata, Kexin Chen, Xinyun Zou, Emre Neftci, and Jeffrey L Krichmar. Domain adaptation in reinforcement learning via latent unified state representation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 10452–10459, 2021.

[48] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, et al. Relational deep reinforcement learning. *arXiv preprint arXiv:1806.01830*, 2018.

[49] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. *IEEE Transactions on Robotics*, 36(4):1307–1319, 2020.

[50] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. In *IEEE Symposium Series on Computational Intelligence*, pages 737–744, 2020.

תקציר

זריקת חפצים באמצעות זרועות רובוטיות מגדילה את טווח העבודה של הרובוט וכוללת הרבה שימושים תעשייתיים; מיון, אריזה, התנהלות בפסי ייצור, מחזור ועוד. בעוד שמודלים אנליטיים יכולים לספק ביצועים מספקים, הם מצריכים שיערוך מדויק של פרמטרי המערכת. אלגוריתמי 'למידה מחיזוקים' (Reinforcement Learning) יכולים לספק מדיניות (Policy) זריקה ללא שום ידע מקדים, אבל, הם מצריכים כמויות גדולות של מידע מהעולם האמיתי (בשונה ממידע מסימולציה), דבר שלוקח הרבה מאוד זמן וכמובן, מהווה סכנה. מצד שני, אימון בסימולציה ככל הנראה יביא לביצועים נמוכים כאשר נשתמש ברובוט האמיתי.

בעבודה זו אנו חוקרים את השימוש בארכיטקטורת ה-Decision Transformers או בקיצור DT, והיכולת שלה לעבור ממדיניות מבוססת סימולציה לעולם האמיתי. בניגוד לאלגוריתמי למידה מחיזוקים מודרניים, אנו מנסחים את הבעיה שלנו כסדרה זמנית, ומאמנים DT באמצעות למידה בהשגחה (Supervised Learning). ה-DT מאומן ב-off-line, על מידע שנאסף מסימולציה שרחוקה מאוד מהמציאות, באמצעות פעולות רנדומליות לחלוטין ללא שום ידע מקדים של איך לזרוק. לאחר מכן ה-DT עובר כוונון עדין (Fine-Tune) באמצעות מספר בודד (כ-5) של זריקות אמיתיות.

זריקות של חפצים שונים מגיעות לדיוקים גבוהים עם שגיאה בסדר גודל של 4 ס"מ. בנוסף על כך, ה-DT יודע לבצע אקסטרפולציה ולזרוק בצורה מדויקת למרחקים שנמצאים מחוץ להתפלגות המרחקים שיש בדוגמאות הזריקה. להוסיף על כך, אנו מראים שמספר זריקות בודדות של *מומחה*, וללא אימון מקדים כלל בסימולציה, מספיקות כדי לאמן מדיניות מדויקת.

אוניברסיטת תל-אביב

הפקולטה להנדסה ע''ש איבי ואלדר פליישמן

בית הספר לתארים מתקדמים ע''ש זנדמן-סליינר

למידת זריקה מתוך דוגמאות בודדות באמצעות שימוש ב-Decision Transformers

חיבור זה הוגש כעבודת מחקר לקראת התואר "מוסמך אוניברסיטה" בהנדסה מכנית

על ידי

מקסים מונסטירסקי

העבודה נעשתה בבית הספר להנדסה מכנית

בהנחיית ד"ר אבישי סינטוב

תשרי תשפ"ג