Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Imperial College London

Department of Information Technology and
Electrical Engineering at ETH Zurich
Departement of Bioengineering at Imperial College
London

Spring 2023

Data-Driven Reinforcement Learning to Model Human Reward-Based Motor Learning in a Real World Task

Semester Project

Dario Bolli
dbolli@ethz.ch

March 23, 2023

Supervisor: Prof. Aldo Faisal, a.faisal@imperial.ac.uk
Professor: Valerio Mante, valerio@ini.uzh.ch

Abstract

In this project, we aim at investigating the underlying neurobehavioural dynamics of sensorimotor learning in real-world settings.
We proceed in a first time to a literature review of the relevant research in the field of motor adaptation and reinforcement learning.
In the second part, we build on top of an existing paradigm to observe learning mechanisms in a real-world task.
We employ a data-driven approach to model the reward-based learning mechanism in a visuomotor adaptation task.
Specifically, we aim at learning a behaviour policy for the task of playing billiards by observing the actions performed by human participants in a real-world setting harnessing the capabilities of offline reinforcement learning.
Our Artificial Agent is able to learn a real-world task policy by observing the actions of a human behaviour policy. This framework further allows the analysis of the learning trends when introducing a visuomotor perturbation.
Finally, we propose avenues for future research building on this paradigm to enhance our understanding of motor-learning mechanisms.

Declaration of Originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
	D.Bolli

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

DarioBolli,
London, *March*23, 2023

Contents

Chapter 1

Introduction

1.1. Motor Learning

Motor learning refers to the process of gaining motor skills and producing more effective movement, through changes occurring in the nervous system following alterations in bodily movements and subsequent observed feedback. This phenomenon is a crucial feature in the development of humans, and occurs across various temporal scales and levels of complexity. Humans acquire basic motor skills like walking or talking over several years and continually adjust to variations in weight, height, and strength throughout their lifetimes. Motor learning enhances the smoothness and precision of movements, enabling humans to gain new skills for a diverse range of contexts, including simple daily tasks, but also more complex movements like learning a new sport or the rehabilitation of motor function after a stroke. Additionally, it enables the calibration of simple movements such as reflexes. Learning a motor skill in real-world scenarios is typically prolonged, complex, and challenging to quantify due to the integration of motor skills across different tasks (e.g. learning to play basketball involves building on previously learned motor skills such as running, throwing, and catching a ball), noise (motor noise, sensor noise, noise in the environment) and the tasks being unconstrained and highly variable. Hence, the isolation of neural processes that result from a specific movement or behaviour is a challenging and intricate task.
In our work we focus on motor adaptation, a subfield of motor learning which refers to behavioural adjustments in response to changes in the environment or in the human body, as opposed to learning new motor skills from scratch.
This paradigm allows to observe the subsequent changes in behaviour and neural activity over a reasonable period of time and in a more controlled setting.

1. Introduction

Research in the field of motor Learning so far has mainly focused on adaptation tasks in highly controlled and simplified laboratory settings that have been designed to isolate and analyse distinct learning features [1].
Two common experiments in the field are force-field adaptation tasks and visuomotor rotation tasks. In force-field adaptation tasks[2], participants execute reaching movements while encountering externally applied forces. Over time, they gradually adjust their movements to compensate for the perturbation forces, ultimately resembling the original unperturbed movements. On the other hand, visuomotor rotation tasks [3],[4] require participants to adapt their movements in response to visual perturbations, aiming to achieve the same outcome as in the non-perturbed state.

Motor learning relies upon the utilisation of feedback signals that provide crucial information to the motor system. These signals serve as the basis for modifying the internal model, which subsequently guides the generation of motor commands and thus the behaviour in future trials. This feedback can take two forms, namely error-based feedback and reward-based feedback [1][5],[6].
The error-based feedback mechanism relies on the detection and correction of sensory-prediction errors to facilitate the acquisition of motor skills. This feedback, which can be provided continuously during the task or upon its completion, allows for the detection and correction of deviations from the objective.
On the other hand, the reward-based mechanism utilises the success or failure of a movement to drive the learning.
Previous research within the motor adaptation framework, particularly in the context of visuomotor adaptation tasks, has demonstrated the co-occurence of those mechanisms.[7] [8].
The neural basis underlying motor adaptation have been primarily identified in the Cortex, specifically the primary motor cortex (M1) and primary somatosensory cortex (S1), as well as in the cerebellum and Basal Ganglia. More precisely, research in the field established that learning from reward is associated with the basal ganglia, whereas learning from error feedback is associated with the cerebellum [1] [9].

The neural signatures of motor learning are significant in the neural beta oscillations (13–30Hz)[10], and specifically in the post-movement beta rebound (PMBR)[11]. The PMBR is a temporarily change in beta oscillations amplitude observed in the sensorimotor cortex following the end of a voluntary movement. This effect is believed to reflect the involvement of the sensorimotor network in motor planning, execution and feedback processing, and has been observed in a variety of motor tasks. In the context of motor learning tasks, different behaviour in the beta activity have been observed for reward-based and error-based motor learning[12][13]. So far, these different neurobehavioural dynamics have been studied in highly restrictive laboratory experiments only, limiting

their generalisation to a real-world scenario.

Although current methods offer valuable insights into the neurobehavioural dynamics and play a vital role in our understanding of motor adaptation, their application is often limited to a narrow range of behaviours, which may not fully reflect the complexity of real-world scenarios. Therefore, there is a need for innovative paradigms that can accurately capture the richness and diversity of real-world settings to provide a more comprehensive and realistic understanding of how the brain processes and prioritises the acquisition of motor skills.

Chapter 2

Previous work

This project serves as a significant milestone in a long-term effort to better understand the neurobehavioural dynamics of motor adaptation in real-world contexts.
To investigate this paradigm in a real-world scenario, S. Haar and A. A. Faisal proposed a motor learning task that reflects real-world challenges in the form of learning to play pool[14].
The game of billiards has well-defined objectives and is readily observable. On top of that, it requires complex skills and involves several sub-tasks such as precision, alignment, ballistic movements, or high-level sequential planning of shots and ball positions. Consequently, billiards serves as an interesting paradigm for investigating motor learning within a real-world, complex task in a practical manner.
During the study, participants received instructions to use a cue stick to execute a pool shot, aiming to pocket a target ball in a specific pocket on the billiard table. The participants had the freedom to move their entire body, engaging in self-paced movements without any imposed restrictions.
The experiment, high-speed camera tracking were employed to monitor the positions of the balls and the cue stick and participants wore a motion tracking "suit" containing wireless Inertial Measurement Unit sensors to capture their body movements. Mobile brain imaging was utilised to monitor EEG activity. This experimental setup enabled participants to execute genuine motor commands and perceive natural somatosensory feedback within a realistic environment.

In the study, the evaluation of learning performance across trials was based on the improvement in the directional error. Subsequently, the researchers analysed the PMBR dynamics, uncovering two distinct groups of participants exhibiting opposing dynamics[15]. These findings indicated the presence of different predominant learning mechanisms for the same motor learning task[16].

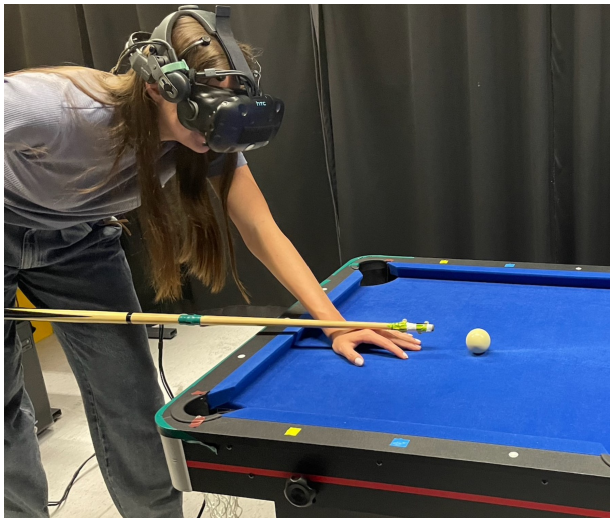

Figure 2.1.: Embodied Virtual Reality in the pool task [18]

The authors demonstrate engaging in real-world motor learning, characterised by fewer constraints on tasks, provide individuals greater flexibility in utilising diverse learning processes. Consequently, this freedom may potentially lead to different learning strategies among individuals.

To allow further investigate those mechanisms, S. Haar and A. A. Faisal devised a method to manipulate the feedback provided to human subjects while preserving a sense of realism by the means of Embodied Virtual Reality (EVR) [17].

To preserve a sense of embodiment within the virtual environment, realistic visual and audio feedback, along with natural haptic feedback through interactions with real-world objects were provided. This setup allows for the manipulation of visual feedback and subsequently the investigation of the two previously introduced feedback learning mechanisms. Significantly, the authors demonstrated similar learning trends in the virtual reality setup and the real-world. This finding not only offers promise for future investigations into learning mechanisms using this paradigm but also highlights its potential application in motor learning skills rehabilitation[19].

Expanding on the EVR pool task paradigm, F. Nardi, M. Ziman, S. Haar, and A. A. Faisal [18] proposed two experiments to isolate and investigate the error-based and reward-based learning mechanisms in a real-world setting. By utilising a Virtual Reality headset, the researchers were able to manipulate the visual feedback provided to the subjects when introducing a visuomotor perturbation, effectively enforcing the use of either one of the specific learning mechanisms.
The perturbation is a 5° counterclockwise rotation of the cueball trajectory in the virtual environment, and thus enforce the participant to modify his shot to pocket the ball (see

fig 2.2). For the error-based feedback group, the trajectories of the balls were hidden

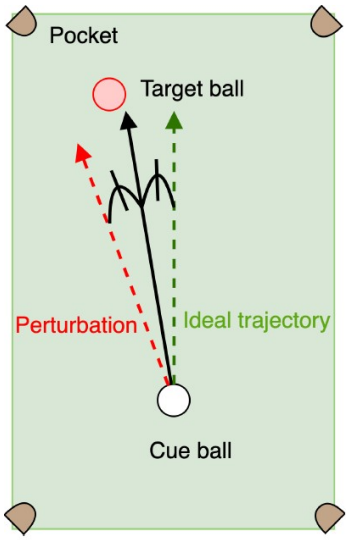

Figure 2.2.: perturbation of the cueball trajectory in the virtual environment

after collision between the cue ball and the target ball, thus constraining the subjects to improve their performance based on the error of the cue ball's trajectory towards the target ball, without receiving any information on the success of the trial.
For the reward-based feedback group, the balls were hidden once the cue ball was hit with the cue stick, and the participants were shown an identical artificial successful trajectory in case of successful outcome. This type of feedback does not allow any estimation on the magnitude of the error, but provides information on the success or failure of the trial.
The results of their study revealed that in a realistic task, successful learning can be achieved through either error-based or reward-based learning mechanisms individually.
The change in angle of the cueball trajectory following the introduction of the perturbation was observed (fig2.3). Two distinct learning patterns emerge from the study: linear adaptation learning in the reward-based setting and exponential learning in the error-based feedback setting. These trends highlight the diverse ways in which the learning process unfolds depending on the type of feedback framework employed.
Within this project, our objective is to explore a potential computational approach relying on a data-driven approach to model the dynamics of the reward-based learning mechanism in a real-world task. Through this investigation, we aim to gain insights into the underlying computational principles governing reward-based learning and its application to real-world scenarios.

Understanding the relationship between these neurological alterations and reward-based learning holds promise for improving PD rehabilitation strategies.

Our work is motivated by an intriguing real-life application, which is to ease the detection and potentially address the rehabilitation of Parkinson's disease (PD) patients.

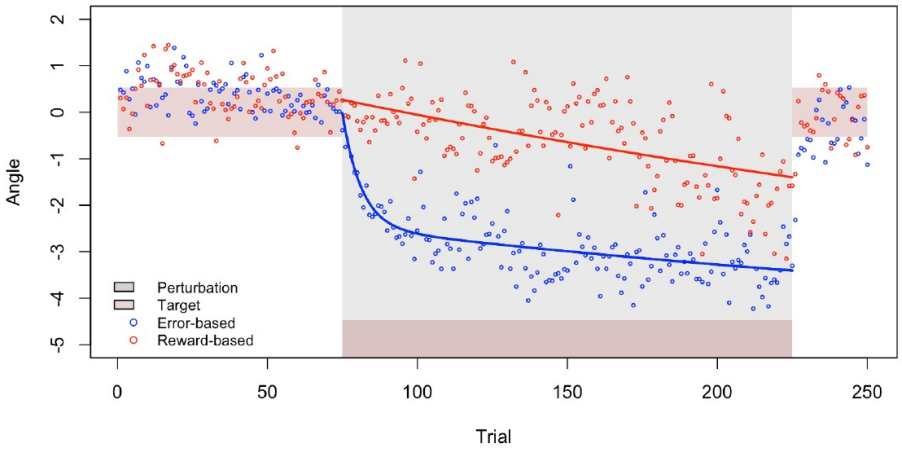

Figure 2.3.: angle adaptation

Individuals affected by PD experience neuronal impairments in the substantia nigra, a region within the basal ganglia [20], where reward-based reinforcement learning occurs [1][21][22].

Chapter 3

Theory

Richard Sutton & al [23] suggested that the goal of maximising rewards is sufficient to drive behaviour encompassing the diverse range of abilities observed in the realms of both natural and artificial intelligence.
Generating motor commands that maximise rewards relies on two crucial components, namely an optimal action selector and an optimal state predictor[8]. In our project, we focus on the selection of high-level motor commands that maximise task rewards, assuming an optimal learner. This particular challenge is addressed by the field of reinforcement learning.

3.1. Reinforcement Learning foundations

Reinforcement learning provides a theoretical framework to define the strategies that an agent, such as an animal, a person or an artificial agent, should use to optimise the reward in an environment. It describes an adaptive process that relies on the exploration of the given environment and on simple measures of the outcome such as success or failure.

This theory draws strong inspiration from the fields of behavioural science and neuroscience, and notably the dopamine reward system in the Basal Ganglia ([24], [21], [22]). It has been successfully applied in a variety of applications, including game strategy[25], robotics[26], autonomous vehicles[27], advanced human-like motor-control of humanoid interacting with objects [28] and even to control multiple agents collaborating to play football [29]

Through the use of Markov Decision Processes (MDP) [30], sequence of decisions or actions can be modelled in discrete-time. This allows to formulate the Reinforcement Learning problems within a mathematical framework.

At each time step t, the agent receives some representation of the environment's state, $s_t \epsilon S$, and selects an action, $a_t \epsilon A(s)$, that leads him to a new state s_{t+1}. This transition from state s_t to state s_{t+1} following action a_t gives the agent a reward $r_{t+1} \epsilon R$ at timestep $t+1$.
The rollout of an agent's policy through time provides a sequence of states, actions and rewards called a trajectory. A trajectory is usually represented by a sequence of tuples $\tau = s_t, a_t, r_{t+1}, s_{t+1}, d_{i=1,\dots,N}$, where d is a binary value encoding the final step of a trajectory.

To evaluate the success of an agent in achieving its goal, the sum of potential future rewards accumulated over the learning task, called cumulative reward or return is typically used.

$$g(\tau) = \big[\sum_{t=1}^{\tau} \gamma^{t-1} R_t\big] \tag{3.1}$$

The expected cumulative reward in a given state s for a policy π, called value function, estimates the value of being in a given state s and act according to policy π thereafter.

$$V^{\pi}(s) = \mathbb{E}\big[\sum_{t=1}^{\tau} \gamma^{t-1} R_t | \pi\big] \tag{3.2}$$

The agent's objective is to learn a mapping from the state space to the action space, called policy π, that maximises this value function:

$$\hat{\pi} = \underset{\pi}{\operatorname{argmax}}\ \mathbb{E}\big[\sum_{t=1}^{\tau} \gamma^{t-1} R_t | \pi\big] \tag{3.3}$$

where γ is a discount factor that determines the importance of future rewards, H is the length of the observed sequence of states, actions, rewards, termed trajectory, and $\mathbb{E}$ denotes the expected value over all possible sequences of states, actions, and rewards.
The policy is typically learned by the agent through trial and error, where it takes actions in the environment and receives feedback in the form of rewards.

The Bellman equation allows to estimate the value function as the sum of the immediate reward r and the discounted expected value of the next state $V^{\pi}(s')$:

$$V^{\pi}(s) \approx r + \gamma V^{\pi}(s') \tag{3.4}$$

for a deterministic policy or

$$V^{\pi}(s) \approx r + \gamma \sum_{s' \in S} P^{\pi}_{ss'} V^{\pi}(s') \tag{3.5}$$

$P^{\pi}_{ss'}$ being the transition probability from state s to state s' in the stochastic case. This

approximation can be used to update the estimate of the value function iteratively, according to the Temporal Difference (TD) learning rule:

$$V^{new}(s) = V(s) + \alpha * (r + \gamma V(s') - V(s)) \tag{3.6}$$

(note that we simplify the notation by dropping the π term in the value function), where alpha is a learning rate parameter that controls the step size of the update. This rule allows the agent to gradually improve its estimate of the value of each state based on the rewards it receives and the expected future rewards, and thus better optimise its policy. This learning pattern was found to be very similar to the firing pattern of Dopaminergic (DA) neurons [31] where delayed learning occurs following a reward prediction error.

A variant of the value function is the action-value function $Q^{\pi}(s, a)$, which estimate the expected return from being in a given state s, choosing action and act according to policy π thereafter.
TD-learning update rule can also be applied to $Q(s, a)$ and gives us the Q-learning update:

$$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha * (r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{3.7}$$

A large number of reinforcement learning (RL) algorithm rely on adaptations of the Q-learning update.

In many real-world tasks the state and action spaces are very large, making it infeasible to explore the entire environment, to learn a probability for each action and find the optimal policy or value function. To address this, approximations of the value function and/or policy that can generalise well across the environment are necessary. Linear function approximators and artificial neural networks have proven to be powerful function approximators that can generalise well from a limited number of examples. A successful approach to learning an approximation of the optimal policy is to use the weights θ of an neural network to parametrise the policy. These Policy-based estimation methods allow to estimate a probability distribution over actions, providing a way to handle complex tasks in high-dimensional state space[25].

Following from the Bellman equation 3.4, the Bellman operator is defined as

$$[B^{\pi}V](s) = \mathbb{E}_{a \sim \pi(\cdot|s)}[r(s, a) + \gamma \mathbb{E}_{s'|s,a}[V(s')]] \tag{3.8}$$

for any V, and iterating on the Bellman optimality operator

$$[B^{*}V] = \max_{\pi} [B^{\pi}V] \tag{3.9}$$

has been proven to converge to the true value function. The identical equations can be defined for the Q-value function, and the principle of convergence applies similarly.[30].

Expanding on this concept, Policy iteration employs the Bellman optimality operator to improve the value function, thereby refining policy evaluation. Subsequently, the policy is updated based on the maximisation of the value function V^{π} for each state.
The Actor-Critic method[30], a Deep Reinforcement Learning algorithm, is rooted in the policy iteration framework. It employs artificial neural networks to approximate both the policy and the value function. These approximations are iteratively updated in an alternating manner, allowing the algorithm to learn and refine its understanding of the environment.

The actor model consists in a neural network that seeks to approximate the policy function, aiming to maximise the expected return. Observing transitions $(s, a, r, s') \sim \tau$, the actor updates the estimated policy $\hat{\pi}$ toward actions that maximises the expected Q-function:

$$\hat{\pi}^{k+1} \leftarrow \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}_{s \sim \tau, a \sim \pi^k(a|s)} \left[\hat{Q}^{k+1}(s, a)\right] \tag{3.10}$$

On the other hand, the critic model is tasked with assessing the quality of the actions selected by the actor. It uses another neural network that approximates the Q-function. The Critic iterates on the Bellman operator to estimate the Q-value, allowing to evaluate the policy of the actor model:

$$\hat{Q}^{k+1} \leftarrow \underset{Q}{\operatorname{argmin}}\, \mathbb{E}_{s,a,s' \sim \tau} \left[((r(s, a) + \gamma\, \mathbb{E}_{a' \sim \hat{\pi}^k(a'|s')} \left[\hat{Q}^k(s', a')\right]) - Q(s, a))^2\right] \tag{3.11}$$

The two models are trained iteratively, with updates alternating between the actor and the critic networks. This iterative process continues until the algorithm converges to the optimal policy, improving the actor's ability to select actions and the critic's capacity to evaluate their quality.

Soft Actor Critic (SAC) is an RL algorithm that incorporates a stochastic policy aiming at providing a more accurate representation of the true distribution of actions given a specific state. Additionally, SAC introduces the concept of maximising the entropy of the policy. This encourages the agent to select actions that are more diverse and exploratory, ultimately leading to the development of a more resilient and adaptable policy.

In SAC, a stochastic policy is implemented by having the actor network generate the parameters defining the Gaussian distribution (i.e. mean and standard deviation) of the

actions associated with a given state, from which the specific action is sampled. To maximise policy entropy, SAC introduces an entropy term in the loss function. This term is calculated as the negative logarithm of the probability of the action sampled from the policy distribution.

$$[B^{\pi}Q](s,a) = r(s,a) + \gamma \mathbb{E}_{a' \sim \pi}[Q(s',a') - log(\pi(a',s)] \tag{3.12}$$

where a' is the action sampled following policy π and $-log(\pi(a',s)$ represents the entropy of the policy.

The current State of the Art in deep reinforcement Learning involves the development of models that build upon the Actor-Critic framework. Different improvements allow to address certain limitation such as continuous actions spaces [32], or Q value overestimation by the critic network [33].

3.2. Dataset and Problem Definition

We model the motor commands selection process in our billiards task as an RL agent. We expand on the Isolation of Motor Learning mechanisms experiment[18] introduced in the previous section 2, and utilise the data previously collected in that context.

The experiment was conducted on 40 healthy right-handed subjects. Each participant had to shoot 250 trials, divided in 10 blocks of 25 shots with short breaks in-between to mitigate the effect of fatigue. The initial three blocks serve as a baseline phase, allowing the subjects to familiarise with the movements. During the following six blocks, a perturbation is introduced in the form of a counterclockwise rotation of 5° on the cue ball trajectory. This perturbation requires the subjects to make compensatory adjustments in their movements by aiming at an imaginary target, rotated 5° clockwise with respect to the original target to successfully pocket the ball. Additionally, a group-specific visual feedback is provided through the Embodied Virtual Reality headset. Lastly, there is a wash-out block where the effects of the perturbation are removed.

The cuestick movements were recorded using markers and motion trackers, which were then translated into corresponding motions of a virtual cuestick within the virtual environment. Careful calibration between the real world and the virtual environment allowed a sense of embodiment. The participants experienced haptic feedback through their interaction with a physical cue, billiards table, and cueball. However, the game's physics were computed using the Unity platform [34], and the visual feedback was displayed within an Embodied Virtual Reality setup.

The task can be formulated as a sequence of discrete actions (e.g. position of the cuestick sampled at 90 Hz frequency) and thus formulated as an MDP.
The success of a shot can be determined from the velocity of the cueball immediately after it is struck, which we will refer to as the hitting time. Thereafter, it is assumed that all variables are known and subsequent computations can be performed in a deterministic manner.

Firstly, we determined the hitting time of each trial in the dataset by applying a set of conditions to the positions of the cue stick and cue ball. We then established a designated time window around the hitting time. Through visual examination of multiple trials, we defined a window size of n=30 timesteps, which roughly corresponds to 33 milliseconds (considering a sampling frequency of 90Hz), preceding the hitting time. Additionally, we included m=5 timesteps following the hitting time to observe the velocity of the cue ball immediately after impact.

We establish a Markov Decision Process framework for our task.
The state space is defined as the relevant variables accessible to the human agent, which have an impact on the task's outcome. Specifically, the positions of the balls, cue and corners of the table, the velocity of the cue ball, and the velocity of the cue stick are used to define the agent's state at time t:

- cueballpos: cue ball position (x, y, z)
- cueballvel: cue ball velocity (x, y, z)
- targetballpos: target ball position (x, y, z)
- cornerpos: corner position (x, y, z)
- cueposfront: front position of the cue stick (x, y, z)
- cueposback: back position of the cue stick (x, y, z)
- cuedir: direction of the cue stick (x, y, z)

Since we do not currently possess a dataset containing information pertaining to the human body such as the joint angle and torque for this experiment so far, we observe the actions applied to the cue by the human agent. However, exploring a more complete model of the human body in a subsequent experiment would be an interesting avenue for future work.
With access to the cue velocity, we define our actions as the force and angle of the cue. By considering the change in momentum, we can calculate the impulse force using the difference in cue velocities.

$$F = \Delta V = m * (V_{t+1} - V_t) \tag{3.13}$$

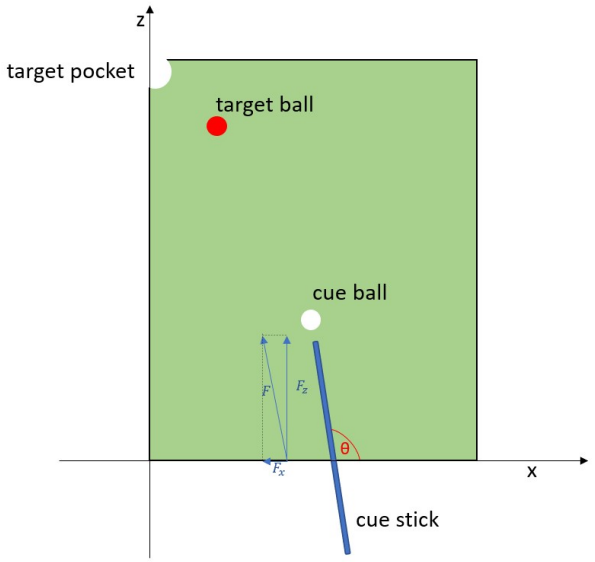

Figure 3.1.: Problem representation

where V is the velocity of the cue stick and F is the impulse force.

And the angle can be computed based on the position of the cue stick as:

$$\theta = arctan(\frac{cueposfront_x - cueposback_x}{cueposfront_z - cueposback_z}) = arctan(\frac{cuedir_x}{cuedir_z}) \tag{3.14}$$

given that we choose the correct trigonometric quadrant.

- θ: cue stick angle with respect to horizontal x-axis.
- F: magnitude of the impulse force exerted by the human subject on the cue stick.

To gain a deeper understanding of our action space, we analyse the data within a specific time window around the hitting timestep. Through careful visual inspection of multiple trials, we identify a pertinent time window for the shot, which spans n=100 timesteps, approximately equivalent to $\sim$ 1.11 seconds (considering the sampling frequency of our data is 90Hz), preceding the hitting timestep. Additionally, we include m=40 timesteps, roughly corresponding to $\sim$ 0.44 s, after the hitting time step.
We divide the trials across all subjects into two categories based on their outcome and compare the mean and standard deviation of both groups during the observed window around the hitting time:

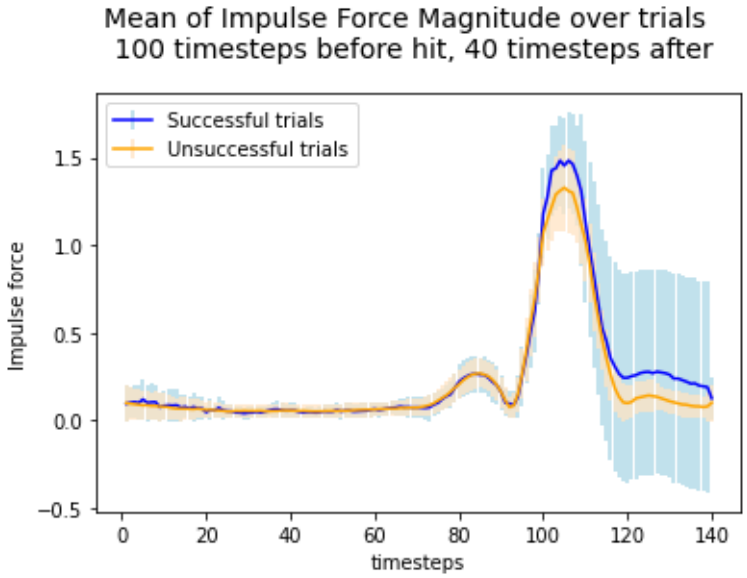

Figure 3.2.: distribution of the force's magnitude given the outcome

We observe minimal discrepancy in the means of the two categories, except for a slight difference immediately after the hitting time, that is however to put in perspective given the significant variance observed after hitting time. Despite this small difference, the force profile holds significance. In fact, we notice a distinct and consistent pattern, indicating that the force profile has been rapidly learned by the human independently from the success of the trial, thus likely resulting from an explicit cognitive process.

We then proceed to a similar analysis for the cue stick angle.

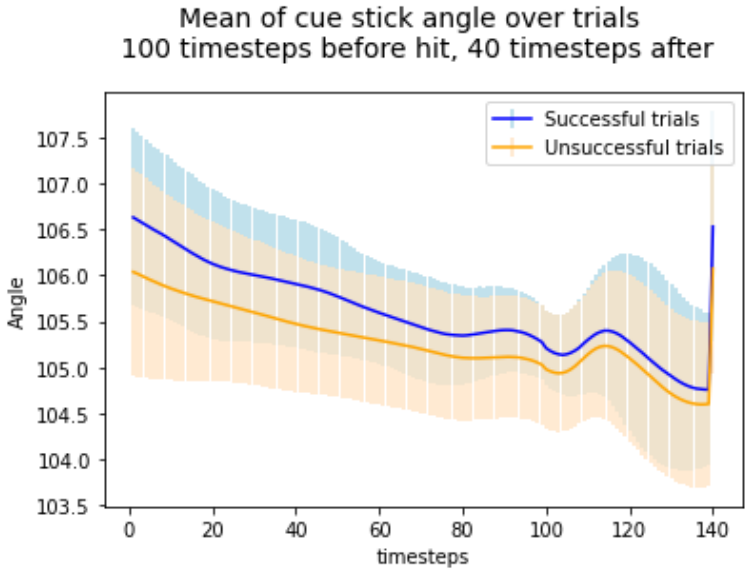

Figure 3.3.: distribution of the cue angle given the outcome

We observe here a more significant difference between the distribution of the cue angle for the two groups. In order to assess this difference, we use the statistical t-test method. The t-statistic measures the difference between the means of the two groups, relative to the variation within each group. If the difference between the means of the two groups is large relative to the variation within each group, the t-statistic will be large, suggesting that the difference is meaningful. The p-value is the probability of observing the values in the data, if there is no difference between the means of the two groups.

For the values at the hitting time, as well as one timestep before and after, we conducted a t-test and observed the following results::

time	t-statistic	p-value
-1	6.830	1.998e-11
0	5.412	9.218e-08
1	4.415	1.220e-05

The analysis reveals a significant disparity between the mean angle of the successful group and the unsuccessful group, which aligns with our intuition.

In the experiment, successful trials are rewarded through artificial trajectory visual feedback, which can be considered as binary reward feedback since it only indicates the success of the trial without providing additional information to the participant.
Therefore, our reward space is sparse, characterised by a positive reinforcement signal

$r = 1$ allocated at the terminal state in case of a successful outcome. In all other states, the reward value is set to zero $r = 0$. Learning from sparse rewards is possible in RL due to the propagation of the Q-function value from the terminal state to the previous state-action pairs, thereby enabling the optimal Q-value estimation for each state-action pair. Nonetheless, learning in environments with sparse rewards poses significant challenges. Therefore, we explored the possibility of incorporating additional potential reward functions to facilitate the learning process while ensuring that they align with plausible reward signals that a human agent could use.
In line with the concept of learning from failure as well, we introduced a negative reward of $r = -1$ when a trial resulted in an unsuccessful outcome. By applying a stronger penalty to unsuccessful trials, our intention is to provide the learning agent with feedback that emphasizes the importance of learning from mistakes and avoiding unsuccessful outcomes in order to improve its performance.
Based on the definition of our state space, action space, and reward function, as well as the dataset we have collected, we create the following transition tuples (s_t, a_t, r_{t+1}, s_{t+1}, d), with $s_t,\ s_{t+1} \in S$, $a_t \in A$, $r_{t+1} \in R$, and $d \in (0, 1)$.
For a trial of N-recorded timesteps, the constructed MDP is:

	state s							action a	
time	cueballpos	cueballvel	targetballpos	cornerpos	cueposfront	cueposback	cuedir	θ	F
0	0.04, 0.85, -0.32	0.0, 0.0, 0.0	-0.21, 0.85, 0.54	-0.31, 0.83, 0.65	0.04, 1.14, -0.32	0.31, 0.99, -1.48	-0.27, -0.15, 1.16	103.3	0.39
...	...	...	...	...	...	...	...	...	...
N	0.04, 0.85, -0.31	-0.28, 0.14, 1.16	-0.21, 0.85, 0.54	-0.31, 0.83, 0.65	0.04, 0.85, -0.31	0.34, 1.01, -1.46	-0.3, -0.16, 1.15	105.2	0.39

	reward r	next state s'							done d
time		cueballpos	cueballvel	targetballpos	cornerpos	cueposfront	cueposback	cuedir	
0	0	0.04, 0.85, -0.32	0.0, 0.0, 0.0	-0.21, 0.85, 0.56	-0.31, 0.83, 0.65	0.05, 1.14, -0.3	0.29, 0.99, -1.45	-0.24, -0.15, 1.15	0
...	...	...	...	...	...	...	...	...	...
N	1	0.04, 0.85, -0.31	-0.28, 0.14, 1.16	-0.21, 0.85, 0.54	-0.31, 0.83, 0.65	0.04, 0.85, -0.31	0.34, 1.01, -1.46	-0.3, -0.16, 1.15	1

Lastly, we apply column-wise min-max normalisation to both the states and actions.

3.3. Off-policy Reinforcement learning

Generating new states, actions, and rewards samples can be very fastidious, expensive or sometimes dangerous in real-world tasks such as in healthcare decision, or when deploying a naive automated robot. Therefore having the ability to learn an optimal policy based on data collected in another way is crucial.
This technique is widely used to pretrain a naive agent in an Off-policy way before deploying it in the real environment.

Off-policy Reinforcement learning introduce the idea of maximising the expected cumulative reward of a target policy not by interacting with the environment using the target policy, but by collecting experiences generated by a behaviour policy.

The estimated policy is derived from indirect interactions with the environment through a behaviour policy, which can assume various forms such as a rudimentary rule-based policy, an alternative reinforcement learning agent, or human subjects, as is the case in our study.

Off-policy RL allows to continually learn from previously collected experiences in a process called experience replay [35], where the behaviour policy interacts with the environment, and stores the observations in a buffer. This process can be repeated in an iterative way, where the behaviour policy is frequently updated through the target policy.
Experience replay allows the agent to be trained by leveraging all of the experiences observed so far, or a subset thereof, leading to a reduced number of required interactions with the environment.

The most common approaches for estimating the expected return under a target distribution given samples generated by a behaviour policy are based on Importance Sampling (IS) [36]

IS gives an unbiased and consistent estimator[37] of the expected value of a function $f(x)$ where x has a true data distribution p, but has been drawn from a sampling distribution q, by computing the likelihood ratio between the probability of x under distribution q and its probability under distribution p.
The expected value of the function f(x), when data x follows a probability distribution p can be estimated through the Monte Carlo sampling method, which consists in taking an average of $f(x)$ over N samples drawn from probability distribution p. By a simple trick, we can approximate the expected value of the function f(x) with samples drawn from another distribution q as follows:

$$\mathop{\mathbb{E}}_{p}[f(x)] = \int f(x)p(x)dx = \int f(x)\frac{p(x)}{q(x)}q(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)\frac{p(x_i)}{q(x_i)}\ ,\ where\ \ x_i \sim q(x) \tag{3.15}$$

The idea behind importance sampling applied to Off-policy evaluation is to adjust the updates of the Q function as if the data were sampled according to the target policy by weighting the updates according to the importance sampling ratio. The importance sampling ratio is calculated by comparing the relative probability of the trajectories occurring under the target policy π_e and behaviour policy π_b.

Hence, the value function $Q^{\pi_e}(s,a)$ given a trajectory $\tau = s_i, a_i, r_i, s'_{ii=1,\dots,N}$ generated by a policy π_b can be rewritten as follows:

$$Q^{\pi_e}(s,a) = IS(\pi_e, \tau, \pi_b) = g(\tau)\ \rho_N^\tau = \big[\sum_{t=1}^{\tau} \gamma^{t-1} R_t|\pi\big]\rho_N^\tau\ ,\ where\ \ \rho_N^\tau = \prod_{i=1}^{N} \frac{\pi_e(a_i|s_i)}{\pi_b(a_i|s_i)} \tag{3.16}$$

One of the most common variants of Importance sampling is Weighted Importance Sampling (WIS) [36], which adds a weight system to importance sampling. This new weight gives more importance to the trajectories that are proportionally the closest to the evaluated policy π_e.

$$WIS(\pi_e, \tau, \pi_b) = \sum_{i=1}^{N} g(\tau_i) w_t^{H_i}\ ,\ where\ \ w_t^{H_i} = \frac{\rho_t^{H_i}}{\sum_{j=1}^{N} \rho_t^{H_j}} \tag{3.17}$$

Off-policy evaluation gives us the mean to learn a policy by observing human subjects trials in a real world task.

3.4. Offline Reinforcement learning

In contrast to the methods described so far, Offline reinforcement learning introduces the idea of learning from a static dataset D of observed transitions (s, a, r, s'). The agent no longer has the ability to interact with the environment and collect additional transitions using the behaviour policy. It allows to learn the behaviour of an agent solely by observing its actions, without any interaction with the environment [38].
It is interesting to note that even if the dataset of observations does not reflect optimal actions, Offline RL offers a powerful way to learn by combining parts of different behaviours observed over time to achieve a goal. This approach mirrors how humans learn, as we are able to combine information from our experiences to determine the best possible behaviour despite having observed only a less optimal behaviour. For example, when learning to play chess, a human may start by making random moves that occasionally lead to success. Through this trial-and-error process, the human can gather valuable experience and combine the most successful moves to develop a good playing strategy.

The absence of additional on-policy data for training poses a significant challenge where the estimated target policy π_e becomes highly sensitive to the distribution of the data collected by the behaviour agent. In real-world tasks that involve complex continuous

state-action spaces, the behaviour agent cannot collect samples from the entire environment and is restricted to a limited amount of trajectory samples to estimate the action-state value function. In areas where insufficient samples were collected, the Q-function may think that an unseen action,known as out-of-distribution (OOD), has a high return and will query it. However, the agent has no way of knowing if this action is indeed the best one as it can not explore it, and as a result, its Q-function distribution may deviate from the true distribution. This can lead to overestimation of the Q value function, as shown in the example below:

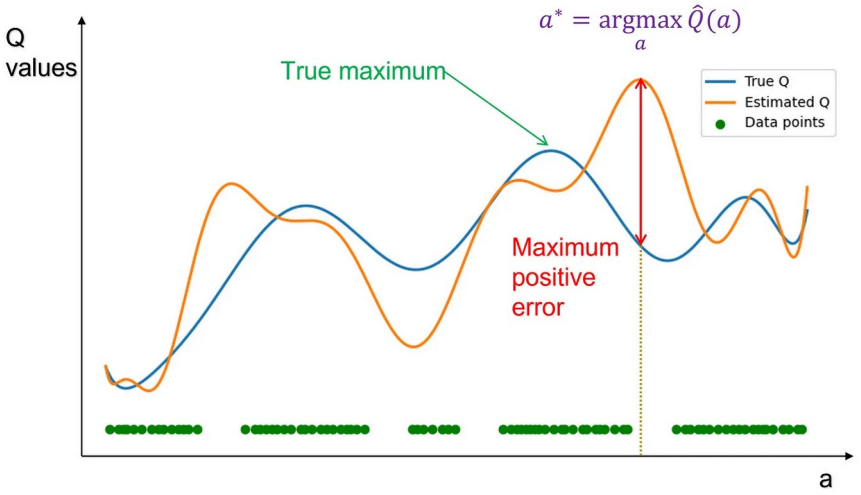

Figure 3.4.: Illustration of distributional shift [39]

This phenomenon, called distributional shift, is only affected by out-of-distribution actions, as the Bellman equation never queries the Q-function on unseen states.
Distributional shift was not an issue in traditional online RL because the online agent, following an epsilon-greedy policy, would collect samples at the estimated maximum of the Q-function and thus realise if the estimated maximum Q-value is off.

Two main intuitions were proposed to address this problem.

The first one is to only allow the policy to take actions in the training data distribution, such that data supports viable predictions.
A simple method following this approach is to add a regulariser term to the policy update in terms of the MSE distance to the behaviour policy that was used to collect the data.

The first model we investigate is an extension of the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm[33]. TD3 is an enhanced Actor Critic algorithm designed to address the common issue of Q value function overestimation. It achieves this

by incorporating two critic networks that select the lower Q value estimation, introducing a delay in updating the actor network, and adding noise to the actions selected by the actor network during Q value evaluation.

To enhance training stability, TD3 also incorporates the use of target networks. These target networks temporarily fix the Q value targets to prevent constant adaptation to a moving target. Periodically updated copies of the actor and critic networks, namely target networks, help maintain moderate deviations from previous network values, preventing divergence and ensuring a more stable convergence. This approach significantly improves the algorithm's performance overall.

Expanding on the TD3 algorithm, TD3 Behaviour Cloning [40] propose to add a behaviour cloning term to regularise the policy loss 3.10, thus constraining how much the learned policy can deviate from the behaviour policy:

$$\pi = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}_{(s,a)\sim D}\left[\lambda\, Q(s,\pi(s)) - (\pi(s)-a)^2\right] \tag{3.18}$$

Another method to address the overestimation of the Q value function that arises due to the query of OOD samples is to adopt a conservative approach when estimating the Q-values. Underestimating the value of unseen outcomes allows the estimated value or performance of the policy that executes unseen behaviours to be small. Conservative Q learning [41] propose to estimate a lower bound of the true value function and derive the expected policy from it.
The authors propose to modify the Q-function update by introducing an additional minimisation of the Q function under specific distribution.
The update of the Q-function in online RL algorithm typically never queries Q-values at unobserved states, but queries Q-values at unseen actions.
Hence they define a distribution of state-action pair $\mu(s,a)$ as $\mu(s,a) = d^{\pi_b}\mu(a|s)$. The marginalisation of the actions, allows to minimise the Q value function under a distribution of actions conditioned on the states. Minimising the Q values over the distribution $\mu(a|s)$ allows to minimise high Q-values on OOD actions is then added to the Bellman update, .
To tighten the lower bound and counterbalance the minimisation term, a maximisation of the Q-values estimates under the data distribution $\hat{\pi}_b(a|s)$ is also introduced in the Bellman update. This allows to minimise high Q-values on OOD actions while maximising Q-value on the actions in the dataset distribution.
The following modifications (in red) are added to the original TD error minimisation of the Q value function update 3.11:

$$\hat{Q}^{k+1} \leftarrow \operatorname*{argmin}_{Q} \alpha \left(\mathbb{E}_{s \sim D, a \sim \mu(a|s)} \left[Q(s,a)\right] - \mathbb{E}_{s \sim D, a \sim \hat{\pi}_b(a|s)} \left[Q(s,a)\right]\right) + \frac{1}{2} \mathbb{E}_{s,a \sim D} \left[(\hat{B}^{\pi} \hat{Q}^{k}(s,a) - Q(s,a))^2\right] \tag{3.19}$$

Where the lower bound is obtained for $\mu = \pi$.

In the context of an Offline RL agent, as the estimated policy $\hat{\pi}^k$ is the one that maximises the current Q-function, the authors propose to choose $\mu(a|s)$ such that it maximises the current estimated Q-function. This gives the following optimisation problem:

$$\min_{Q} \max_{\mu} \alpha \left(\mathbb{E}_{s \sim D, a \sim \mu(a|s)} \left[Q(s,a)\right] - \mathbb{E}_{s \sim D, a \sim \pi_b(a|s)} \left[Q(s,a)\right]\right) + \frac{1}{2} \mathbb{E}_{s,a \sim D} \left[(\hat{B}^{\pi} \hat{Q}^{k}(s,a)] - Q(s,a))^2\right] + R(\mu) \tag{3.20}$$

with $R(\mu)$ a regulariser.
In practice, we choose the regulariser $R(\mu) = H(\mu)$ to be the entropy of the distribution μ. This gives the following update of the Q-function:

$$\min_{Q} \mathbb{E}_{s \sim D} \left[-log \sum_{a} exp(Q(s,a)) - \mathbb{E}_{s \sim D, a \sim \pi_b(a|s)} \left[Q(s,a)\right]\right] + \frac{1}{2} \mathbb{E}_{s,a \sim D} \left[\hat{B}^{\pi} \hat{Q}^{k}(s,a)]) - Q(s,a))^2\right] \tag{3.21}$$

This Q-function update can be used on top of existing Deep reinforcement learning methods such as Actor-Critics, by simply replacing the Q-function update with the CQL Q-function update 3.21.

Chapter 4

Architecture of the Reward-based Model

4.1. TD3-BC implementation

The TD3-BC algorithm builds upon the traditional TD3 algorithm and introduces a slight modification in the policy update, specifically in the form of a behaviour cloning term3.18.
We based our implementation on Scott Fujimoto's PyTorch implementation [42] that we tailored to our framework.
The actor and the two critic networks are defined as Multi-Layer Perceptrons (MLPs) (see appendix A.2 and A.1). Several combinations of number of hidden layers [2,4] and hidden units [32,512] are investigated.
The computation of the update for the critic networks is done according to the following loss:

```
    # Compute targets
    target_Q1, target_Q2 = self.critic_target(new_states, next_action)
    target_Q = torch.min(target_Q1, target_Q2)
    target_Q = rewards + not_terminals * self.discount * target_Q
    # Get current Q estimates
    current_Q1, current_Q2 = self.critic(states, actions)
    # Compute critic loss
    critic_loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(
    current_Q2, target_Q)
```

Listing 4.1: Implementation of TD3 Critic loss

To enhance training stability, the critic networks undergo updates at each epoch, whereas the actor network is updated once every n=2 epochs, according to the following loss:

```
    Q = self.critic.Q1(states, pi)
    lmbda = self.alpha/Q.abs().mean()

```

```
4     actor_loss = -lmbda * Q.mean() + F.mse_loss(pi, actions)
```

Listing 4.2: Implementation of TD3 Actor loss

In addition to the maximisation of the Q-value function, the policy update also involves the minimisation of the MSE loss between loss between the current actions provided by the actor network and the actions generated by the behaviour policy. This approach aims to encourage the policy to select actions that closely align with those taken by the behavior policy.3.18.
Refer to appendix A.3 for a more comprehensive understanding of the implementation,or to the complete code [43].

4.2. Conservative Q-learning implementation

For the Conservative Q-learning algorithm, we follow a pytorch implementation[44] of the CQL modification [41] built on top of the Soft Actor-Critic algorithm, and adapt it to our environment.

Once agin, the actor and the two critic networks are defined as MLPs (see appendix A.4 and A.5). Several combinations of number of hidden layers [2,4] and hidden units [32,512] are investigated.
The actor outputs a stochastic action sampled from the Gaussian distribution estimated by the network, and the log probability of the distributionA.4.

The first step of the update is to compute the Q target, adding the entropy term introduced in 3.12, and using two critic networks to mitigate the overestimation of the Q-values.
$Q_{target}(s,a) \leftarrow r(s,a) + \gamma E_{a' \sim \pi}[Q^{\theta}_{target}(s',a') - \alpha \log \pi_{\phi}(a'|s')]$, where:

- $Q_{target}(s,a)$ is the target Q-value for the current state-action pair.
- $r(s,a)$ is the immediate reward obtained from taking action a in state s.
- γ is the discount factor that controls the trade-off between immediate and future rewards.
- $Q^{\theta}_{target}(s',a')$ is the minimum estimated Q-value function for the next state-action pair between the two target critics networks θ^1_{target} and θ^2_{target}.
- α is the temperature parameter that controls the level of entropy regularization.
- $\pi_{\phi}(a'|s')$ is the target policy for the next state s', parameterized by ϕ.

- $E_{a' \sim \pi}$ denotes the expectation over the next action a' sampled from the action distribution $\pi(a'|s')$.

```
    # Compute Q targets for next states
    Q_target1_next = self.critic1_target(next_states, next_action)
    Q_target2_next = self.critic2_target(next_states, next_action)
    Q_target_next = torch.min(Q_target1_next, Q_target2_next) - self.
    alpha * new_log_pi
    # Compute Q targets for current states
    Q_targets = rewards + not_dones * self.gamma *Q_target_next
```

Listing 4.3: Implementation of CQL Q targets

The loss of the actor is computed as:

```
    actions_pred, log_pis = self.actor.evaluate(states)
    q1 = self.critic1(states, actions_pred)
    q2 = self.critic2(states, actions_pred)
    min_Q = torch.min(q1,q2)
    actor_loss = ((self.alpha * log_pis - min_Q )).mean()
```

Listing 4.4: Actor loss implementation

The loss of the critic networks is composed of two parts. The first is a Mean-Squared Error loss to the moving target networks, which ensures a more stable convergence toward the optimal Q-value through regular updates of the target networks.

```
    # Compute the Q-values estimates of the current state-action pair.
    q1 = self.critic1(states, actions)
    q2 = self.critic2(states, actions)

    critic1_loss = F.mse_loss(q1.squeeze(), Q_targets)
    critic2_loss = F.mse_loss(q2.squeeze(), Q_targets)
```

Listing 4.5: Implementation of Critic loss

The second part is the CQL update as described in 3.21, which is added to the critic loss.
The first term, $log \sum exp(Q(s,a))$ 3.21, is approximated by randomly sampling N=10 actions at every state s from a uniform distribution and 10 actions from the current policy $\pi(a|s)$. The true Q function is then estimated from those samples using Importance sampling 3.16:
$log \sum exp(Q(s,a)) \approx log \frac{1}{2N} \sum_{a_i \sim Unif(a)}^{N} [\frac{exp(Q(s,a_i))}{Unif(a)}] + \frac{1}{2N} \sum_{a_i \sim \pi(a|s)}^{N} [\frac{exp(Q(s,a_i))}{\pi(a_i|s}]$

The CQL loss is computed and added to the critic loss.

```
    cat_q1 = torch.cat([random_values1, current_pi_values1,
    next_pi_values1], 1)
    cat_q2 = torch.cat([random_values2, current_pi_values2,
    next_pi_values2], 1)

    #CQL modification loss (to be added to critic loss
```

```
    cql1_scaled_loss = ((torch.logsumexp(cat_q1).mean() * self.cql_weight
    ) - q1.mean()) * self.cql_weight
    cql2_scaled_loss = ((torch.logsumexp(cat_q2).mean() * self.cql_weight
    ) - q2.mean()) * self.cql_weight

    total_c1_loss = critic1_loss + cql1_scaled_loss
    total_c2_loss = critic2_loss + cql2_scaled_loss
```

Listing 4.6: Implementation of CQL loss

Where the logarithm of the exponential sum is applied to the concatenated Q values for the random actions, the actions from the current policy, and the next actions sampled with the current policy for a given state s.

Refer to appendix A.7 for a more comprehensive understanding of the implementation,or to the complete code [43].

Chapter 5

Results

5.1. TD3-BC results

In a first time, we consider a simplified problem, in a 2-dimensional environment, disregarding the y-axis component of every position, and looked at a window of $n = 3$ timesteps before and $m = 2$ timesteps after hitting time. In this experimental setup, we trained and fine-tuned the TD3-BC agent over 500,000 epochs, and monitored several metrics. First, let's examine the losses used in the updates of the actor and critic networks (fig5.1). We observe that the agent rapidly learns effective actions and demon-

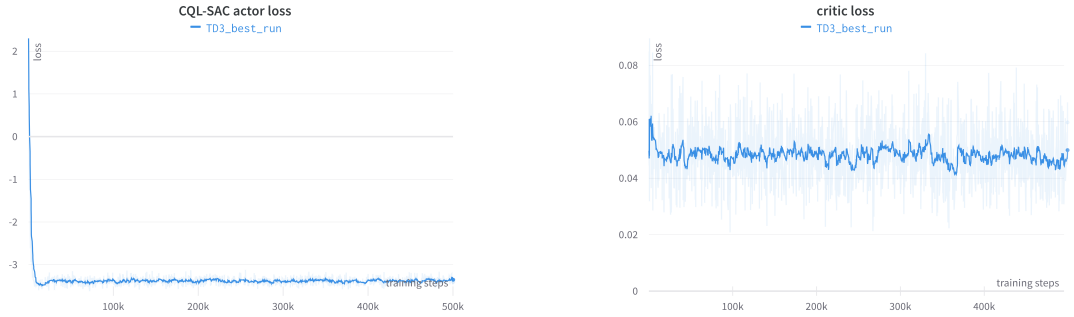

Figure 5.1.: Loss of the Actor and Critic network for a 5 timesteps window

strates a swift understanding of the environment.
Furthermore, we conduct a comparison between the actions chosen by our agent and those performed by the human. We sample an initial state from a separate test set and rollout the policy during the subsequent steps. Specifically, at the hitting time, we examine the following set of actions (fig 5.2). The observed actions provide evidence that

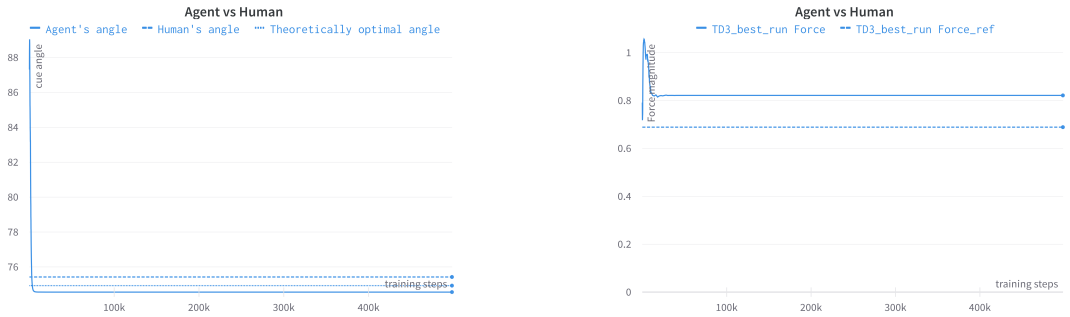

Figure 5.2.: TD3-BC Agent vs Human actions comparison at hitting time for a 5 timesteps window

the agent effectively addressed the task and even surpassed the human performance. Notably, the agent made adjustments to the angle that brought it closer to the theoretically optimal angle, indicating its ability to optimise the action selection.

Next, we analyse our problem from a three-dimensional perspective, taking into account the free movement of the cue in space. To focus on the relevant time frame, we consider a window of $n = 3$ timesteps before and $m = 5$ timesteps after hitting time. Once again, we trained and fine-tuned the TD3-BC agent over 500,000 epochs, and monitored several metrics (fig 5.3). Here, we notice unusual patterns in the loss values. This is a typical

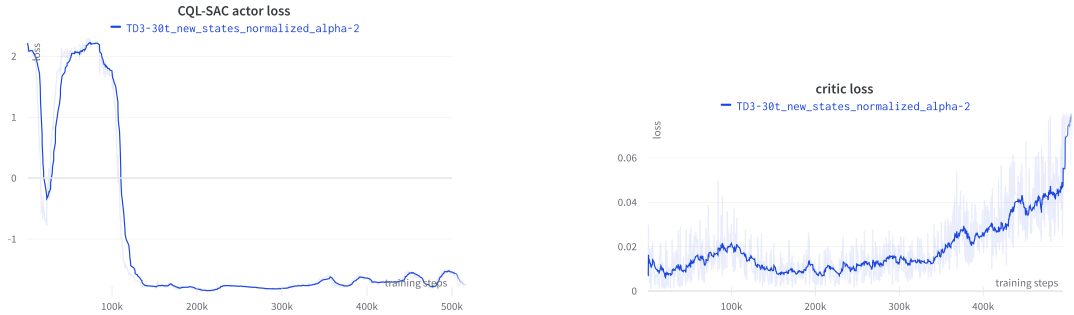

Figure 5.3.: Loss of the Actor and Critic network for a 35 timesteps window

challenge encountered during the training of Actor-Critic models, particularly when they struggle to understand and adapt to the environment. However, those losses are not necessarily a good indicator of the agent's performance. Once more, we examine the actions selected by our agent and the actions executed by the human participant (fig 5.4). These findings suggest that the agent encounters challenges in learning and successfully accomplishing the task in this environment.
To address this issue, we experimented with an alternative offline reinforcement learning

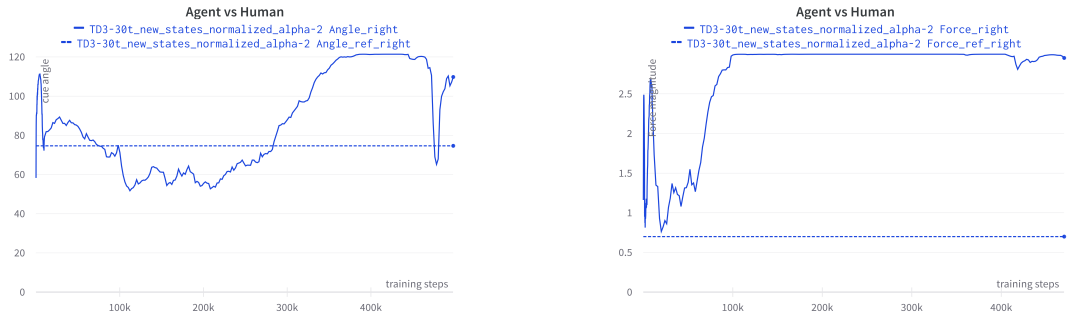

Figure 5.4.: TD3-BC Agent vs Human actions comparison at hitting time for 30

algorithm.

5.2. CQL-SAC results

Following the previously described implementation 4.2, we trained and fine-tuned the CQL-SAC agent for a duration of 500,000 epochs while closely monitoring multiple metrics.
Before delving into the metrics, it is essential to gain a deeper understanding of the significance of these losses.
The actor loss is a metric that quantifies the value of the predicted actions of the actor network according to the Q function estimated by the critic networks, plus an additional entropy regularisation term 4.4. Its minimisation encourages the generation of actions that are more likely to result in higher rewards, as estimated by the critic networks. Conversely, the critic loss measures the discrepancy between the Q-value estimate of the critic network and a moving target, which is periodically updated based on the modified CQL TD-target 3.21. Minimising the critic loss leads to a more accurate estimation of the Q-value by the critic networks.

When training Actor-Critic models, a common problem is the potential for the two networks to compete with each other and update in a non-cooperative manner. Specifically, the critic networks can become overly confident in their Q-value estimates, resulting in low critic loss, minimal updates, and limited exploration of the Q-function. This issue is particularly pronounced at the beginning of training when the Q-value estimates may be inaccurate, leading to sub-optimal policy choices by the actor. Consequently, the actor loss can diverge, as observed in our agent before tuning. To mitigate this issue, extensive parameters tuning may be required.
Thus, after conducting hyperparameters tuning, we identified the optimal settings for our model. Specifically, we found that a batch size of 256 transitions, a learning rate of $1e-5$ for both the critics and the actor networks with a decay of 0.1 every 150,000

episodes, and a tau parameter of 0.001 for the target networks update produced the best results.
We observe the following losses (fig 5.5). It is noteworthy that the loss curve of the

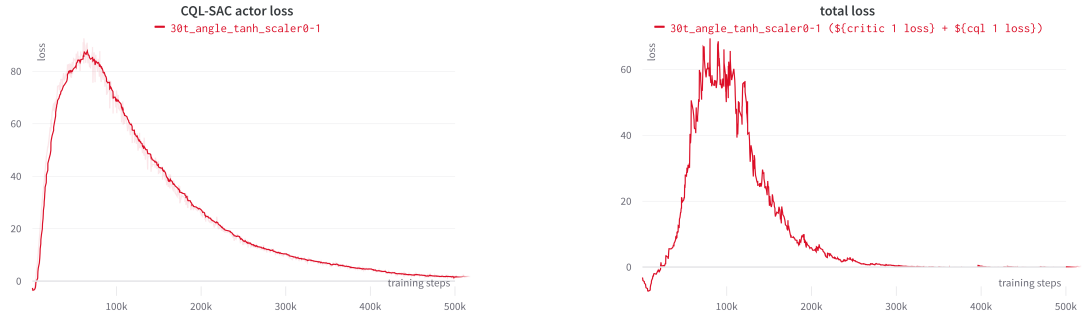

Figure 5.5.: Loss of the Actor and Critic network for a 35 timesteps window

second critic network follows a similar trend as the loss curve of the first critic network (see appendix for critic networks loss curves A.1).
During the initial phase, the actor network's actions are guided by a belief that they are optimal, due to the imprecise critic model. However, after approximately 10,000 epochs, we notice an improvement in the estimated Q-values of the actor's actions by the critic, and subsequently, a decline in both the critic loss and the actor loss. This indicates that the model has begun to learn a meaningful policy.
To further confirm that, we compare the actions selected by our agent to the actions performed by the human participant (fig 5.6). This analysis unveiled that the actions of

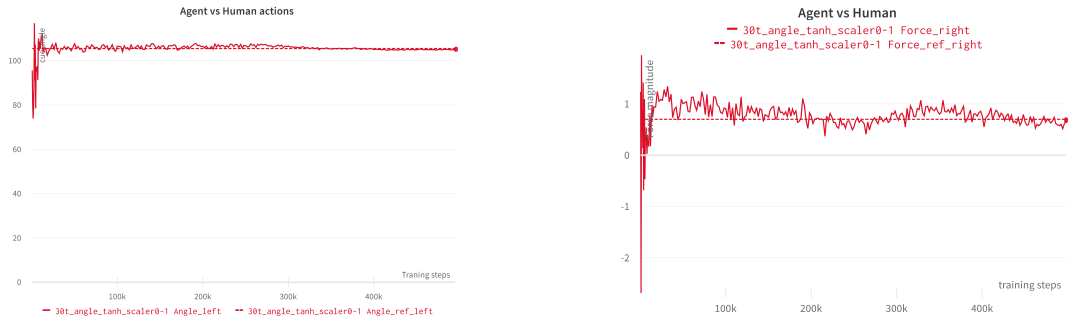

Figure 5.6.: CQL Agent vs Human actions comparison at hitting time for a 35 timesteps window

the CQL agent exhibit higher variability compared to the actions of the TD3-BC agent in the 5-timesteps window simplified scenario. However, the CQL agent demonstrates the ability to learn and successfully complete the intricate task.

A typical way to evaluate a Reinforcement Learning agent's performance is to calculate the average reward obtained by rolling out the learned policy in the environment. However, in Offline Reinforcement Learning, where direct interaction with the environment is not possible, a portion of the dataset can be reserved solely for evaluation purposes, as is commonly done in supervised learning. To this end, we divided our dataset into training and test sets at an 80%-20% ratio.

The agent's performance was then assessed by rolling out the policy on a trajectory of length n using the equations of motion to update the states at each timestep based on the actions predicted by the agent, starting from the initial states in the test set. This computation is done until hitting time. From then on, the physics dynamics are too complex to compute for the agent's evaluation and it would assume hypothesis on the physics of the game.
Therefore, we approximate a measure of the success, by defining a range of actions that lead to successful trials in human participants. The success function is sparse and is defined as 1 for a successful trial, where a trial is deemed successful if the angle of the cue stick at hitting time t is within the defined range 104.1158,105.957 or hard range 104.726, 105.291 and the force is above a certain threshold value. We observe the following success metric (fig 5.7).

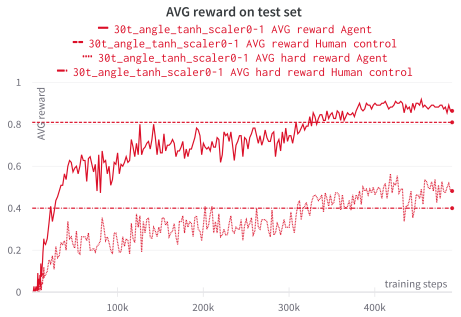

Figure 5.7.: Average success of the agent over the test trials, vs Average success of the Human over the test trials

The agent exhibits similar performances compared to human subjects, with an average success rate of roughly 40%.

To further observe our agent's behaviour, we employed a 2D visualisation technique to monitor the states of the cue stick, cue ball, target ball, and corner at each timestep. Using an animation method, we were able to visualise the rollout of our agent's policy for a trajectory (fig 5.8).

At hitting time, the green reference stick denoting the human's behaviour indicates an unsuccessful trial with an angle of 108.927°, while the blue cue stick of the Agent, starting

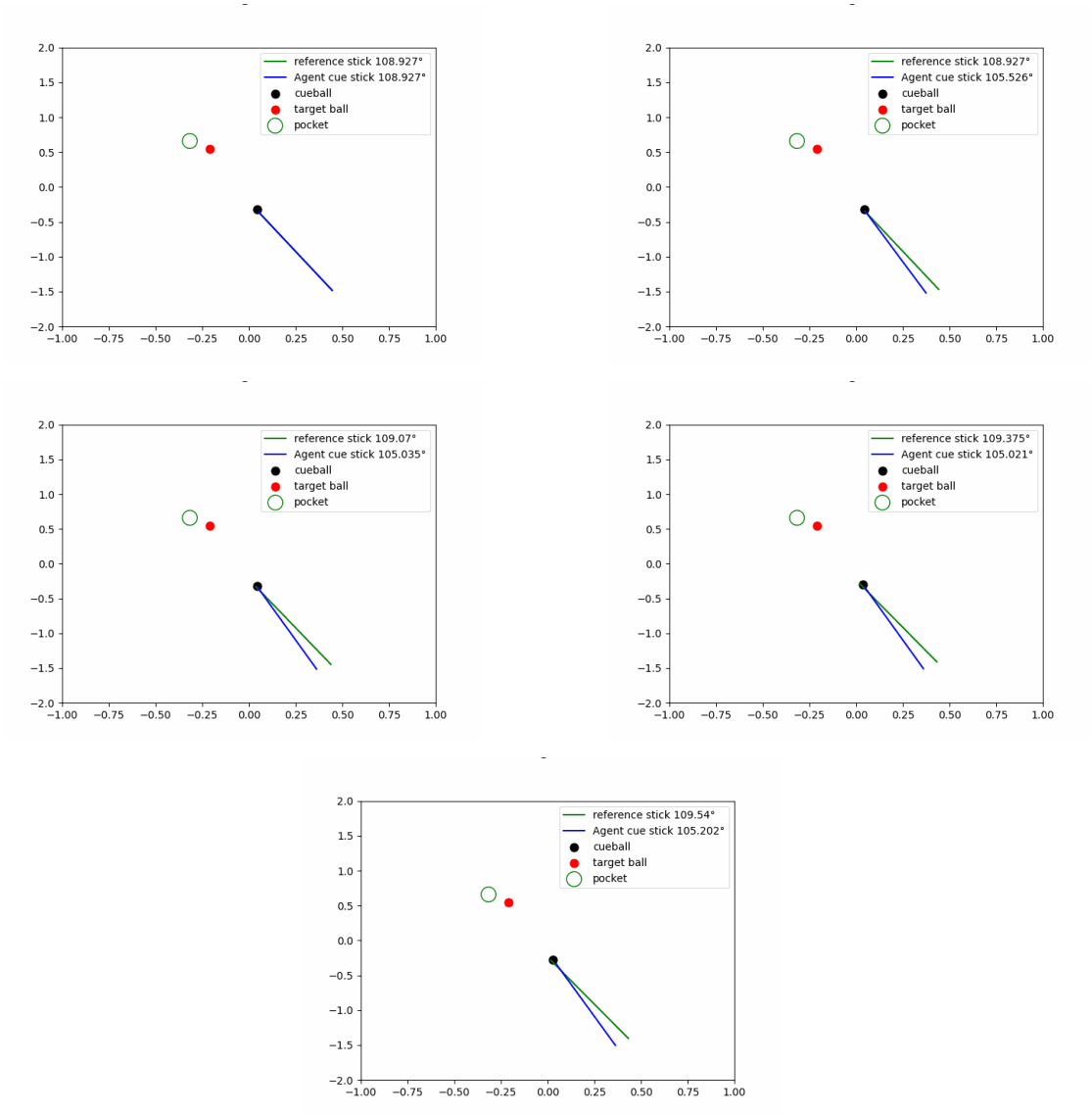

Figure 5.8.: Rollout of the learned policy on 5 timesteps, given an initial state

from the same suboptimal state, adapts to an angle of 105.021° at hitting time, which is within the successful range previously defined (i.e., $104.763 - 105.328$).

The metrics obtained provide valuable insights into the behaviour of our agent and demonstrate its learning capabilities. However, to further evaluate its performance, we aim to conduct a more comprehensive assessment. To achieve this, we made modifications to the Unity simulation environment used in the experimental setup, to enable us to input the actions selected by our agent instead of the recorded movements of the

participant(fig 5.9). By rolling out a trial under identical circumstances as experienced by the human subject during the experiment, we can gain a more direct comparison of the agent's performance (fig 5.10).

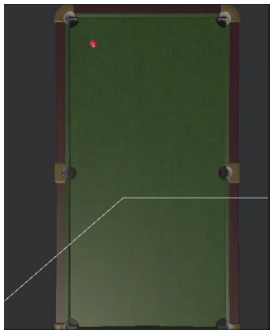

Figure 5.9.: Evaluation of Agent in the Unity environment used for the experiments

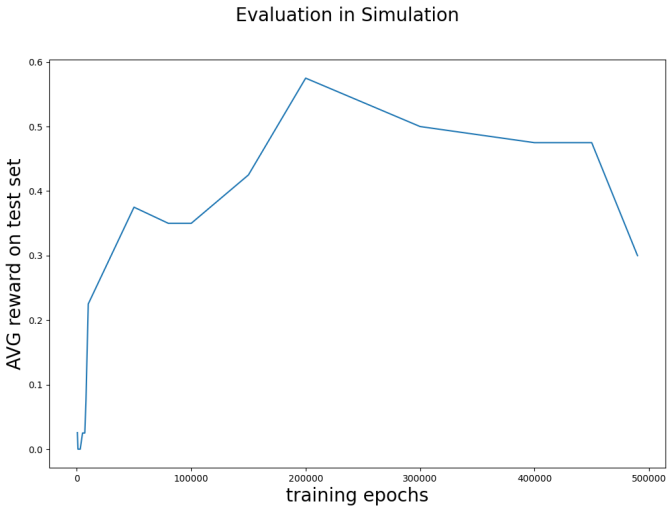

Figure 5.10.: Avg reward observed by our agent in the environment

Due to the extensive evaluation time required in the environment, we conducted a reduced-point evaluation. From the results, it is evident that our agent successfully learns effective actions and reaches a plateau in performance after approximately 200,000 training steps. The utilisation of this environment facilitated the observation of the cueball trajectory angle, enabling a meaningful future comparison with the learning trends observed in real-life scenarios. This comparison will provide valuable insights and enhance our understanding of the learning processes in different contexts and how it relates to a theoretical optimal learner (fig 5.11).

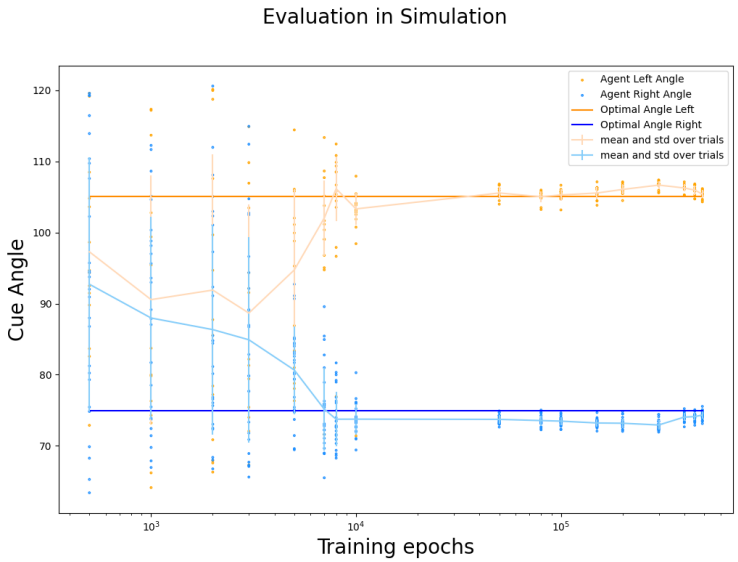

Figure 5.11.: cueball trajectory angle

Chapter 6

Conclusion and Future Work

In this project, we conducted a comprehensive review of relevant research in the motor learning and Reinforcement Learning field and proposed a framework for applying State of the art Offline RL to learn a real-world policy obtained from the observation of human subjects playing pool. This paradigm allows the exploration of the reward-based feedback mechanism in a visuomotor adaptation task to come in a future work.

To successfully apply reinforcement learning in complex real-world scenarios, an agent must extract an efficient representation of the environment from high-dimensional sensory inputs. Furthermore, the agent must use the representation to generalise from past experiences. Interestingly, humans and other animals appear to tackle this challenge by integrating reinforcement learning with hierarchical sensory processing systems [45].

The complex behaviour exhibited by humans when playing pool, a very specific movement, going back and forth with one fixed hand, helps increase precision and reduce variability in the movement.
To replicate such behavior, several additional components need to be incorporated into our existing framework. Indeed, humans' motor commands need to consider the presence of noise during motor execution and learn to establish a mapping between a given command and its corresponding physical outcome. Furthermore, an exact representation of the states is not available to the human, who need to calibrate and estimate it through the integration of sensory feedback.
Having a fixed point (e.g. fixed hand) and repeating a few times a simple back and forth movements helps calibrate the model, reduces variability and enhances the precision.
On top of that, constraints imposed by the human body, including the limited range of motion in the wrist, elbow, and shoulder joints, as well as the constrained reaching distance from the body introduce additional complexities to the task at hand.

As those constraint are not met, our Agent is able to derive a more optimal solution, and ultimately to solve the task in one timestep only. It learns a high-level commands objective, disregarding the subsequent low-level execution and constraints imposed by the body in the real world.
The subsequent phase of our work involves the introduction of perturbations to the training dataset, aimed at assessing the adaptive capacity of the trained agent, as well as determining whether it exhibits the characteristic double-exponential learning curves commonly observed in human learning[1] [18].
In terms of future research, it would be intriguing to explore the implementation of a joint model of the human body, similar to the work presented in [28]. This study developed a realistic humanoid character controller within the Mujoco framework, capable of performing tasks involving object interactions, and learning from both human demonstration and reinforcement learning techniques. Additionally, a promising direction to extend our research is the exploration of a hierarchical approach to action selection, incorporating error-based feedback mechanisms as outlined in [46]. This approach holds great potential for enhancing our understanding of motor learning processes and improving the performance of our agents in complex task environments.

Bibliography

[1] J. W. Krakauer, A. M. Hadjiosif, J. Xu, A. L. Wong, and A. M. Haith, "Motor learning," 2019. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/30873583/

[2] R. Shadmehr and F. Mussa-Ivaldi, "Adaptive representation of dynamics during learning of a motor task," 1994. [Online]. Available: https://www.jneurosci.org/content/14/5/3208

[3] J. W. Krakauer, Z. M. Pine, M. F. Ghilardi, and C. Ghez, "Learning of visuomotor transformations for vectorial planning of reaching trajectories," 2000. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/11102502/

[4] J. A. Taylor, J. W. Krakauer, and R. B. Ivry, "Explicit and implicit contributions to learning in a sensorimotor adaptation task," 2014. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/24553942//

[5] J. W. Krakauer and P. Mazzoni, "Human sensorimotor learning: adaptation, skill, and beyond," 2011. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/21764294/

[6] A. M. Haith and J. W. Krakauer, "Model-based and model-free mechanisms of human motor learning," 2013. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3570165/

[7] S. Ueharam, F. Mawase, and P. Celnik, "Learning similar actions by reinforcement or sensory-prediction errors rely on distinct physiological mechanisms," 2018. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6887949/

[8] J. Izawa and R. Shadmehr, "Learning from sensory and reward prediction errors during motor adaptation," 2011. [Online]. Available: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1002012

[9] K. Doya, "Complementary roles of basal ganglia and cerebellum in learning and motor control," 2000. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/11240282/

[10] A. K. Roopun, S. J. Middleton, M. O. Cunningham, F. E. N. LeBeau, A. Bibbig, M. A. Whittington, and R. D. Traub, "A beta2-frequency (20-30 hz) oscillation in nonsynaptic networks of somatosensory cortex," 2006. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/17030821/

[11] H. Tan, N. Jenkinson, and P. Brown, "Dynamic neural correlates of motor error monitoring and adaptation during trial-to-trial learning," 2014. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/24741058/

[12] F. Torrecillos, J. Alayrangues, B. E. Kilavik, and N. Malfait, "Distinct modulations in sensorimotor postmovement and foreperiod -band activities related to error salience processing and sensorimotor adaptation," 2015. [Online]. Available: https://www.jneurosci.org/content/35/37/12753

[13] C. Kranczioch, S. Athanassiou, S. Shen, G. Gao, and A. Sterr, "Short term learning of a visually guided power-grip task is associated with dynamic changes in eeg oscillatory activity," 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1388245708001569

[14] S. Haar, C. M. van Assel, and A. Faisal, "Motor learning in real-world pool billiards," 2020. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/33208785/

[15] S. Haar and A. Faisal, "Neural biomarkers of multiple motor-learning mechanisms in a real-world task," 2020. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fnhum.2020.00354/full

[16] ——, "Brain activity reveals multiple motor-learning mechanisms in a real-world task," 2020. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/32982707/

[17] S. Haar, G. Sundar, and A. Faisal, "Embodied virtual reality for the study of real-world motor learning," 2020. [Online]. Available: https://www.biorxiv.org/content/10.1101/2020.03.19.998476v1

[18] F. Nardi, M. Ziman, S. Haar, and A. Faisal, "Isolating motor learning mechanisms in embodied virtual reality," 2022. [Online]. Available: https://www.researchgate.net/publication/362707312_Isolating_Motor_Learning_Mechanisms_in_Embodied_Virtual_Reality

[19] M. K. Holden, "Virtual environments for motor rehabilitation: review," 2005. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/15971970/

[20] A. Galvan, A. Devergnas, and T. Wichmann, "Alterations in neuronal activity in basal ganglia-thalamocortical circuits in the parkinsonian state," 2015. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4318426/

[21] P. N. Tobler, C. D. Fiorillo, and W. Schultz, "Adaptive coding of reward value by dopamine neurons," 2005. [Online]. Available: https://www.science.org/doi/10.1126/science.1105370

[22] J. C. Houk, J. L. Adams, and A. G. Barto, "A model of how the basal ganglia generate and use neural signals that predict reinforcement," 1994. [Online]. Available: https://ieeexplore.ieee.org/document/6287670

[23] D. Silver, S. Singh, D. Precup, and R. S. Sutton, "Reward is enough," 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370221000862

[24] W. Schultz, P. Dayan, and P. R. Montague, "neural substrate of prediction and reward," 1997. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/9054347/

[25] V. Mnih1, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, and G. O. . al, "Human-level control through deep reinforcement learning," 2015. [Online]. Available: https://www.nature.com/articles/nature14236

[26] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, "How to train your robot with deep reinforcement learning – lessons we've learned," 2021. [Online]. Available: https://arxiv.org/abs/2102.02915

[27] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," 2017. [Online]. Available: https://arxiv.org/abs/1704.02532

[28] J. Merel, S. Tunyasuvunakool, Arunahuja, Y. Tassa, L. Hasenclever, V. Pham, T. Erez, G. Wayne, and N. Heess, "Catch carry: Reusable neural controllers for vision-guided whole-body tasks," 2020. [Online]. Available: https://arxiv.org/abs/1911.06636

[29] S. Liu, G. Lever, Z. Wang, J. Merel, S. Tunyasuvunakool, and N. H. . al, "From motor control to team play in simulated humanoid football," 2022. [Online]. Available: https://arxiv.org/abs/2105.12196

[30] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," 2018. [Online]. Available: http://www.incompleteideas.net/book/the-book-2nd.html

[31] R. E. Suri, "Td models of reward predictive responses in dopamine neurons," 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608002000461

[32] T. P. Lillicrap, J. J. Hunt, N. H. Alexander Pritzel, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2018. [Online]. Available: https://arxiv.org/abs/1509.02971

[33] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018. [Online]. Available: https://arxiv.org/abs/1802.09477

[34] "Unity real time development platform," https://unity.com/.

[35] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney, "Revisiting fundamentals of experience replay," 2020. [Online]. Available: https://arxiv.org/abs/2007.06700

[36] J. P. Hanna, S. Niekum, and P. Stone, "Importance sampling policy evaluation with an estimated behavior policy," 2019. [Online]. Available: https://arxiv.org/abs/1806.01347

[37] D. P. K. et Reuven Y. Rubinstein, "Simulation and the monte carlo method," 2017.

[38] S. Levine, "Understanding the world through action," 2021. [Online]. Available: https://arxiv.org/abs/2110.12543

[39] "Mediumillustration of the distributional shift," https://medium.com/@athanasios.kapoutsis/the-monster-of-distribution-shift-in-offline-rl-and-how-to-pacify-it-4ea9a5db043.

[40] S. Fujimoto and S. S. Gu, "A minimalist approach to offline reinforcement learning," 2021. [Online]. Available: https://arxiv.org/abs/2106.06860

[41] A. Kumara, A. Zhou, G. Tucker, and S. Levine, "Conservative q-learning for offline reinforcement learning," https://github.com/BY571/CQL, 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/0d2b2061826a5df3221116a5085a6052-Paper.pdf

[42] "Td3-bc github repository," https://github.com/sfujim/TD3_BC.

[43] "github repository of the project," https://github.com/dario-bolli/reward-based_motor-learning.

[44] "Cql-sac github repository," https://github.com/BY571/CQL.

[45] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," 1980. [Online]. Available: https://link.springer.com/article/10.1007/bf00344251

[46] J. Baladron, J. Vitay, T. Fietzek, and F. H. Hamker, "The contribution of the basal ganglia and cerebellum to motor learning: A neurocomputational approach," 2023. [Online]. Available: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1011024

[47] F. A. dos Santos Mendes, J. E. Pompeu, A. M. Lobo, K. G. da Silva, T. de Paula Oliveira, A. P. Zomignani, and M. E. P. Piemonte, "Motor learning, retention and transfer after virtual-reality-based training in parkinson's disease-effect of motor and cognitive demands of games: a longitudinal, controlled clinical study," 2012. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/22898578/

[48] T. Ikegami, J. R. Flanagan, and D. M. Wolpert, "Reach adaption to a visuomotor gain with terminal error feedback involves reinforcement learning," 2022. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/35648778/

[49] P. Vassiliadis, G. Derosiere, C. Dubuc, A. Lete, F. Crevecoeur, F. C. Hummel, , and J. Duque, "Reward boosts reinforcement-based motor learning," 2021. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8319366/

[50] Z. Z. J. B. Nian Si, Fan Zhang, "Distributionally robust batch contextual bandits," 2022. [Online]. Available: https://arxiv.org/abs/2006.05630

[51] J. Wang, R. Gao, and H. Zha, "Reliable off-policy evaluation for reinforcement learning," 2022. [Online]. Available: https://arxiv.org/abs/2011.04102

[52] M. Uehara, J. Huang, and N. Jiang, "Minimax weight and q-function learning for off-policy evaluation," 2020. [Online]. Available: https://arxiv.org/abs/1910.12809

[53] Q. Liu, L. Li, Z. Tang, and D. Zhou, "Breaking the curse of horizon: Infinite-horizon off-policy estimation," 2018. [Online]. Available: https://arxiv.org/abs/1810.12429

[54] A. Kumara, J. Fu, G. Tucker, and S. Levine, "Stabilizing off-policy q-learning via bootstrapping error reduction," 2019. [Online]. Available: https://arxiv.org/abs/1906.00949

[55] 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," 2018. [Online]. Available: https://arxiv.org/abs/1806.10293

[56] J. Fu, A. Kumar, O. Nachum, G. Tucker, and S. Levine, "D4rl: Datasets for deep data-driven reinforcement learning," 2021. [Online]. Available: https://arxiv.org/abs/2004.07219

[57] M. A. S. Laith Alhussein, "Motor planning under uncertainty," 2021. [Online]. Available: https://elifesciences.org/articles/67019

[58] A. S. Therrien and A. L. Wong, "Mechanisms of human motor learning do not function independently," 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fnhum.2021.785992/full

[59] "Open ai benchmark for the half cheetah environment," https://spinningup.openai.com/en/latest/spinningup/bench.html#halfcheetah-pytorch-versions.

[60] "Deep Mindalpha star, mastering the strategy game starcraft," https://www.deepmind.com/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii.

Appendix A

Appendix

A.1. TD3-BC implementation

```
    class Critic(nn.Module):
  def __init__(self, state_dim, action_dim, hidden_dim=256, hidden_dim2
    =256):
    super(Critic, self).__init__()
    # Q1 architecture
    self.l1 = nn.Linear(state_dim + action_dim, hidden_dim)
    self.l2 = nn.Linear(hidden_dim,hidden_dim2)
    self.l3 = nn.Linear(hidden_dim2,hidden_dim)
    self.l4 = nn.Linear(hidden_dim, 1)

    # Q2 architecture
    self.l5 = nn.Linear(state_dim + action_dim, hidden_dim)
    self.l6 = nn.Linear(hidden_dim,hidden_dim2)
    self.l7 = nn.Linear(hidden_dim2,hidden_dim)
    self.l8 = nn.Linear(hidden_dim, 1)
    def forward(self, state, action):
    sa = torch.cat([state, action], 1)
    q1 = F.relu(self.l1(sa))
    q1 = F.relu(self.l2(q1))
    q1 = F.relu(self.l3(q1))
    q1 = self.l4(q1)

    q2 = F.relu(self.l5(sa))
    q2 = F.relu(self.l6(q2))
    q2 = F.relu(self.l7(q2))
    q2 = self.l8(q2)
    return q1, q2
```

Listing A.1: TD3 Critic network

```
    class Critic(nn.Module):
  def __init__(self, state_dim, action_dim, hidden_dim=256, hidden_dim2
    =256):
    super(Critic, self).__init__()
    # Q1 architecture
    self.l1 = nn.Linear(state_dim + action_dim, hidden_dim)
    self.l2 = nn.Linear(hidden_dim,hidden_dim2)
    self.l3 = nn.Linear(hidden_dim2,hidden_dim)
    self.l4 = nn.Linear(hidden_dim, 1)

    # Q2 architecture
    self.l5 = nn.Linear(state_dim + action_dim, hidden_dim)
    self.l6 = nn.Linear(hidden_dim,hidden_dim2)
    self.l7 = nn.Linear(hidden_dim2,hidden_dim)
    self.l8 = nn.Linear(hidden_dim, 1)
    def forward(self, state, action):
    sa = torch.cat([state, action], 1)
    q1 = F.relu(self.l1(sa))
    q1 = F.relu(self.l2(q1))
    q1 = F.relu(self.l3(q1))
    q1 = self.l4(q1)

    q2 = F.relu(self.l5(sa))
    q2 = F.relu(self.l6(q2))
    q2 = F.relu(self.l7(q2))
    q2 = self.l8(q2)
    return q1, q2
```

Listing A.2: TD3 Actor network

```
    def train(self, experience, batch_size=1):
    self.total_it += 1
    actor_loss = torch.tensor(0.0, device = device)
    critic_loss = torch.tensor(0.0, device = device)
    # Sample replay buffer
    states, actions, new_states, rewards, terminals = experience
    not_terminals = ~terminals
    with torch.no_grad():
      # Select action according to policy and add clipped noise
      noise = (
        torch.randn_like(actions) * self.policy_noise
      ).clamp(-self.noise_clip, self.noise_clip)
      next_action = (
        self.actor_target(new_states) + noise
      ).clamp(-self.max_action, self.max_action)
      # Compute the target Q value
      target_Q1, target_Q2 = self.critic_target(new_states, next_action)
      target_Q = torch.min(target_Q1, target_Q2)
      target_Q = rewards + not_terminals * self.discount * target_Q.
    squeeze()

    # Get current Q estimates
    current_Q1, current_Q2 = self.critic(states, actions)
    # Compute critic loss
```

```
    critic_loss = F.mse_loss(current_Q1.squeeze(), target_Q) + F.mse_loss
(current_Q2.squeeze(), target_Q)
    # Optimize the critic
    self.critic_optimizer.zero_grad()
    critic_loss.backward()
    self.critic_optimizer.step()

    # Delayed policy updates
    if self.total_it % self.policy_freq == 0:

      # Compute actor loss
      pi = self.actor(states)
      Q = self.critic.Q1(states, pi)
      lmbda = self.alpha/Q.abs().mean().detach()

      actor_loss = -lmbda * Q.mean() + F.mse_loss(pi, actions)
      # Optimize the actor
      self.actor_optimizer.zero_grad()
      actor_loss.backward()
      self.actor_optimizer.step()

      # Update the frozen target models
      for param, target_param in zip(self.critic.parameters(), self.
critic_target.parameters()):
          target_param.data.copy_(self.tau * param.data + (1 - self.tau) *
target_param.data)

      for param, target_param in zip(self.actor.parameters(), self.
actor_target.parameters()):
          target_param.data.copy_(self.tau * param.data + (1 - self.tau) *
target_param.data)
    self.actor_scheduler.step()
    self.critic_scheduler.step()
    return  critic_loss.cpu().detach(), actor_loss.cpu().detach()
```

Listing A.3: Implementation of TD3 Behaviour Cloning

A.2. CQL-SAC implementation

```
    class Actor_CQL(nn.Module):
    def __init__(self, state_size, action_size, max_actions, min_actions,
     hidden_dim=256, hidden_dim2=256):
        super(Actor_CQL, self).__init__()
        self.log_std_min = log_std_min
        self.log_std_max = log_std_max

        self.fc1 = nn.Linear(state_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, hidden_size)
        self.mu = nn.Linear(hidden_size, action_size)
        self.log_std_linear = nn.Linear(hidden_size, action_size)
```

```
    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        mu = torch.tanh(self.mu(x))
        log_std = self.log_std_linear(x)
        log_std = torch.clamp(log_std, self.log_std_min, self.log_std_max)
        return mu, log_std
    def evaluate(self, state, epsilon=1e-6):
        mu, log_std = self.forward(state)
        std = log_std.exp()
        dist = Normal(mu, std)
        e = dist.rsample().to(state.device)
        action = torch.tanh(e)
        log_prob = (dist.log_prob(e) - torch.log(1 - action.pow(2) + epsilon)).sum(1, keepdim=True)
        return action, log_prob
```

Listing A.4: CQL Actor network

```
    class Critic_CQL(nn.Module):
    def __init__(self, state_size, action_size, hidden_size=256, hidden_size2=256):
        super(Critic_CQL, self).__init__()
        self.fc1 = nn.Linear(state_size+action_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, hidden_size)
        self.fc5 = nn.Linear(hidden_size, 1)

    def forward(self, state, action):
        x = torch.cat((state, action), dim=-1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x=self.fc5(x)
        return x
```

Listing A.5: CQL Critic network

The temperature parameter alpha weighting the entropy regularisation is automatically tuned given a desired minimum target entropy:

```
    actions_pred, log_pis = self.actor.evaluate(states)
    alpha_loss = - (self.log_alpha.exp() * (log_pis + self.target_entropy)).mean()
    self.alpha = self.log_alpha.exp()
```

Listing A.6: Implementation of alpha weight optimization

```
def train(self, experiences, batch_size=1):
        """Updates actor, critics and entropy_alpha parameters using experience tuples (s, a, r, s', done).
        """
```

```
        self.epoch += 1
        states, actions, next_states, rewards, not_dones = experiences

        actor_loss = torch.tensor(0.0, device = device)
        alpha_loss = torch.tensor(0.0, device = device)
        critic1_loss = torch.tensor(0.0, device = device)
        critic2_loss = torch.tensor(0.0, device = device)
        cql1_scaled_loss = torch.tensor(0.0, device = device)
        cql2_scaled_loss = torch.tensor(0.0, device = device)


        # Compute alpha loss
        actions_pred, log_pis = self.actor.evaluate(states)
        alpha_loss = - (self.log_alpha.exp() * (log_pis.cpu() + self.
    target_entropy).detach().cpu()).mean()
        self.alpha_optimizer.zero_grad()
        alpha_loss.backward()
        self.alpha_optimizer.step()
        self.alpha = self.log_alpha.exp().detach()

        # ---------------------------- update critic
    ---------------------------- #

        with torch.no_grad():
            # Get predicted next-state actions for the current policy
            next_action, new_log_pi = self.actor.evaluate(next_states)
            #Get Q-values of the next actions taken by the current policy
     using the target networks
            Q_target1_next = self.critic1_target(next_states, next_action
    )
            Q_target2_next = self.critic2_target(next_states, next_action
    )

            #Define the target Q-value as the minimum Q-value between the
     two target critic networks
            Q_target_next = torch.min(Q_target1_next, Q_target2_next) -
    self.alpha.to(self.device) * new_log_pi
            # Compute Q targets for current states
            Q_targets = rewards + not_dones * self.gamma * Q_target_next.
    squeeze()

        # Compute the Q-values estimates of the current state-action pair
    .
        q1 = self.critic1(states, actions)
        q2 = self.critic2(states, actions)

        critic1_loss = F.mse_loss(q1.squeeze(), Q_targets)
        critic2_loss = F.mse_loss(q2.squeeze(), Q_targets)

        # CQL modification, num_repeat=10 following to CQL paper
        #sample random actions
        random_actions = torch.FloatTensor(q1.shape[0] * 10, actions.
    shape[-1]).uniform_(0, 1).to(self.device)
```

```
47          num_repeat = int (random_actions.shape[0] / states.shape[0])
48          temp_states = states.unsqueeze(1).repeat(1, num_repeat, 1).view(
    states.shape[0] * num_repeat, states.shape[1])
49          temp_next_states = next_states.unsqueeze(1).repeat(1, num_repeat,
     1).view(next_states.shape[0] * num_repeat, next_states.shape[1])
50
51          current_pi_values1, current_pi_values2  = self.
    _compute_policy_values(temp_states, temp_states)
52          next_pi_values1, next_pi_values2 = self._compute_policy_values(
    temp_next_states, temp_states)
53
54          random_values1 = self._compute_random_values(temp_states,
    random_actions, self.critic1).reshape(states.shape[0], num_repeat, 1)
55          random_values2 = self._compute_random_values(temp_states,
    random_actions, self.critic2).reshape(states.shape[0], num_repeat, 1)
56
57          current_pi_values1 = current_pi_values1.reshape(states.shape[0],
    num_repeat, 1)
58          current_pi_values2 = current_pi_values2.reshape(states.shape[0],
    num_repeat, 1)
59
60          next_pi_values1 = next_pi_values1.reshape(states.shape[0],
    num_repeat, 1)
61          next_pi_values2 = next_pi_values2.reshape(states.shape[0],
    num_repeat, 1)
62
63          cat_q1 = torch.cat([random_values1, current_pi_values1,
    next_pi_values1], 1)
64          cat_q2 = torch.cat([random_values2, current_pi_values2,
    next_pi_values2], 1)
65
66          assert cat_q1.shape == (states.shape[0], 3 * num_repeat, 1), f"
    cat_q1 instead has shape: {cat_q1.shape}"
67          assert cat_q2.shape == (states.shape[0], 3 * num_repeat, 1), f"
    cat_q2 instead has shape: {cat_q2.shape}"
68
69          #CQL modification loss (to be added to critic loss
70          cql1_scaled_loss = ((torch.logsumexp(cat_q1 / self.temp, dim=1).
    mean() * self.cql_weight * self.temp) - q1.mean()) * self.cql_weight
71          cql2_scaled_loss = ((torch.logsumexp(cat_q2 / self.temp, dim=1).
    mean() * self.cql_weight * self.temp) - q2.mean()) * self.cql_weight
72
73          total_c1_loss = critic1_loss + cql1_scaled_loss
74          total_c2_loss = critic2_loss + cql2_scaled_loss
75
76
77          # Update critics
78          # critic 1
79          self.critic1_optimizer.zero_grad()
80          total_c1_loss.backward(retain_graph=True)
81          clip_grad_norm_(self.critic1.parameters(), self.clip_grad_param)
82          self.critic1_optimizer.step()
83
```

```
84          # critic 2
85          self.critic2_optimizer.zero_grad()
86          total_c2_loss.backward()
87          clip_grad_norm_(self.critic2.parameters(), self.clip_grad_param)
88          self.critic2_optimizer.step()
89
90          # ---------------------------- update actor
    ---------------------------- #
91          current_alpha = copy.deepcopy(self.alpha)
92          actor_loss, log_pis = self.calc_policy_loss(states, current_alpha
    )
93          self.actor_optimizer.zero_grad()
94          actor_loss.backward()
95          self.actor_optimizer.step()
96          # ----------------------- update target networks
    ----------------------- #
97          self.soft_update(self.critic1, self.critic1_target)
98          self.soft_update(self.critic2, self.critic2_target)
```

Listing A.7: Implementation of CQL modifications in the Soft Actor Critic model

```
1  def select_action(self, state, eval=False):
2      """Returns actions for given state according to the  current policy.
       """
3
4      with torch.no_grad():
5          if eval:
6              action = self.actor.get_det_action(state)
7          else:
8              action = self.actor.select_action(state)
9      return action.cpu()
10
11 def calc_policy_loss(self, states, alpha):
12     actions_pred, log_pis = self.actor.evaluate(states)
13     q1 = self.critic1(states, actions_pred.squeeze(0))
14     q2 = self.critic2(states, actions_pred.squeeze(0))
15     min_Q = torch.min(q1,q2).cpu()
16     actor_loss = ((alpha * log_pis.cpu() - min_Q )).mean()
17     return actor_loss, log_pis
18
19 def _compute_policy_values(self, obs_pi, obs_q):
20     with torch.no_grad():
21         actions_pred, log_pis = self.actor.evaluate(obs_pi)
22
23         qs1 = self.critic1(obs_q, actions_pred)
24         qs2 = self.critic2(obs_q, actions_pred)
25
26     return qs1 - log_pis.detach(), qs2 - log_pis.detach()
27
28 def _compute_random_values(self, obs, actions, critic):
29     random_values = critic(obs, actions)
30     random_log_probs = math.log(0.5 ** self.action_size)
```

```
    return random_values - random_log_probs
```

Listing A.8: Util functions

```
    class Actor_CQL(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, hidden_size=256,
    hidden_size2=256, init_w=3e-3, log_std_min=-20, log_std_max=2):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor_CQL, self).__init__()
        self.log_std_min = log_std_min
        self.log_std_max = log_std_max

        self.fc1 = nn.Linear(state_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)  #, hidden_size2)
        self.fc3 = nn.Linear(hidden_size, hidden_size)
        self.mu = nn.Linear(hidden_size, action_size)
        self.log_std_linear = nn.Linear(hidden_size, action_size)
        #self.dropout = nn.Dropout(0.1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(*hidden_init(self.fc3))
        self.mu.weight.data.uniform_(-3e-3, 3e-3)
        self.log_std_linear.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        #x = self.dropout(x)
        x = F.relu(self.fc2(x))
        #x = self.dropout(x)
        x = F.relu(self.fc3(x))
        #x = self.dropout(x)
        mu = self.mu(x)
        log_std = self.log_std_linear(x)
        log_std = torch.clamp(log_std, self.log_std_min, self.log_std_max
    )
        return mu, log_std

    def evaluate(self, state, epsilon=1e-6):

        mu, log_std = self.forward(state)
        std = log_std.exp()
```

```
        dist = Normal(mu, std)
        e = dist.rsample().to(state.device)
        action = torch.tanh(e)
        log_prob = (dist.log_prob(e) - torch.log(1 - action.pow(2) +
epsilon)).sum(1, keepdim=True)

        return action, log_prob


    def select_action(self, state):
        """
        returns the action based on a squashed gaussian policy. That
means the samples are obtained according to:
        a(s,e)= tanh(mu(s)+sigma(s)+e)
        """
        mu, log_std = self.forward(state)
        std = log_std.exp()
        dist = Normal(mu, std)
        e = dist.rsample().to(state.device)
        action = torch.tanh(e)
        return action#.detach().cpu()

    def get_det_action(self, state):
        mu, log_std = self.forward(state)
        return torch.tanh(mu).detach().cpu()
```

Listing A.9: Util functions

```
    class Critic_CQL(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, hidden_size=256,
hidden_size2=512, seed=1):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            hidden_size (int): Number of nodes in the network layers
        """
        super(Critic_CQL, self).__init__()
        #self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size+action_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)  #, hidden_size2)
        self.fc3 = nn.Linear(hidden_size, hidden_size)
        self.fc4 = nn.Linear(hidden_size, 1)
        #self.dropout = nn.Dropout(0.1)

        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
```

```
26          self.fc3.weight.data.uniform_(*hidden_init(self.fc3))
27          self.fc4.weight.data.uniform_(-3e-3, 3e-3)
28
29      def forward(self, state, action):
30          """Build a critic (value) network that maps (state, action) pairs
        -> Q-values."""
31          x = torch.cat((state, action), dim=-1)
32          x = F.relu(self.fc1(x))
33          #x = self.dropout(x)
34          x = F.relu(self.fc2(x))
35          #x = self.dropout(x)
36          x = F.relu(self.fc3(x))
37          #x = self.dropout(x)
38          return self.fc4(x)
```

Listing A.10: Util functions

A.3. CQL-SAC metrics

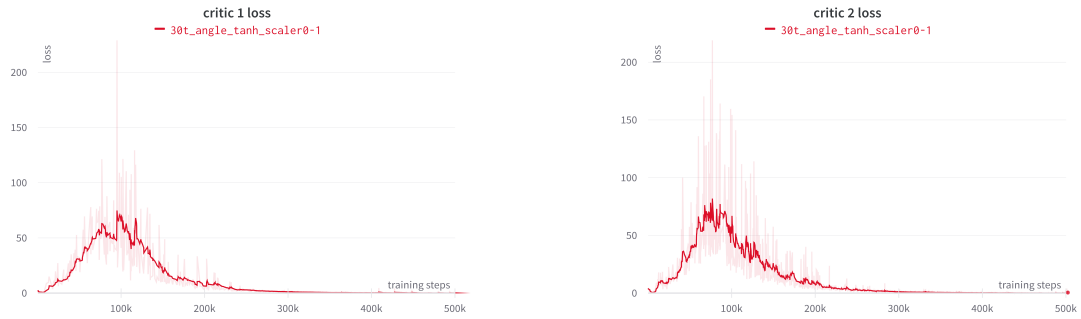

Figure A.1.: Loss of the Critic network for a 35 timesteps window